



WORLD TOOLKIT®

REFERENCE MANUAL

RELEASE 9

ENGINEERING ANIMATION, INC.
SENSE8® PRODUCT LINE
100 Shoreline Highway, Suite 282
Mill Valley, CA 94941



This Reference Manual copyright © 1991 - 1999 by Engineering Animation, Inc. All rights reserved. No part of it may be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior written consent of Engineering Animation, Inc.

SENSE8, WorldToolKit, and World Up are registered trademarks of Engineering Animation, Inc. World2World is a trademark of Engineering Animation, Inc. Sound technology provided by DiamondWare, Ltd. Portions Copyright 1994-1999 DiamondWare, Ltd. All rights reserved. Other brand and product names are trademarks or registered trademarks of their respective holders.

WorldToolKit is based in part on the work of the Independent JPEG Group.

Current version: **April 1999**

ENGINEERING ANIMATION, INC.

SENSE8 Product Line

100 Shoreline Highway, Suite 282

Mill Valley, CA 94941 USA

Telephone: 415/339-3200

Facsimile: 415/339-3201

Web site: www.sense8.com

Acknowledgements to the WorldToolKit team: Leon Franzen, Hans Kessock, Dave Hinkle, Sumanth Ravulakollu and Mukund Bhakta.

This book was printed in the United States of America.

Contents in Brief

Introduction to WorldToolKit	1-1
The Universe	2-1
Object/Property/Event Architecture	3-1
Scene Graphs	4-1
Movable Nodes	5-1
Geometries	6-1
Polygons	7-1
Materials	8-1
3D Text	9-1
Textures	10-1
Tasks	11-1
Lights	12-1
Sensors	13-1
Paths	14-1
Motion Links	15-1
Viewpoints	16-1
Windows	17-1
Adding User Interface (UI) Objects	18-1
Drawing Functions	19-1
Sound	20-1
Client-Server Networking (Via the World2World Servers)	21-1
Multicast Networking	22-1
Serial Ports	23-1
Portability	24-1
Math Library	25-1
C++ Programming	26-1
Frequently Asked Questions	A-1
Environment Variables	B-1
Defined Constants	C-1
Error Messages and Warnings	D-1
Writing a Sensor Driver	E-1
WTK Neutral File Format	F-1
Transitioning From Version 2.1 To Release 6/7/8/9	G-1

Transitioning From Release 6 To Release 7/8/9.....	H-1
Third-party Software.....	I-1
Sources of Components.....	J-1
The WTK Users' Group.....	K-1
Technical Support	L-1
Glossary	M-1

Contents

1: Introduction to WorldToolKit	1-1
Welcome	1-1
What is WTK?	1-1
Scene Graph Architecture	1-2
What WTK Does	1-2
Overview of the WTK Classes	1-3
Naming Conventions	1-4
Additional Features	1-5
WTK Documentation	1-7
Special Interest Group	1-11
Basic System Configuration	1-12
Input Sensors Supported	1-12
Extending a System for Virtual Reality	1-12
A Sample WTK Application	1-13
Important WTK Functions	1-14
Universe	1-15
Geometry	1-15
Polygon	1-16
Sensor	1-16
Light	1-17
Viewpoint	1-17
Path	1-17
Window	1-18
Scene Graph	1-18
Drawing	1-19
User Interface	1-19
Sound	1-19
2: The Universe	2-1
Introduction	2-1
Universe Construction and Destruction	2-1
Simulation Management	2-5
The Universe Action Function	2-11
The Universe's Objects	2-12

Global Rendering Parameters	2-18
Rendering Options	2-18
Other Global Functions	2-20
Performance and Timer Functions	2-22
Universe Options	2-24
Resource Files	2-28
The Resource Hierarchy	2-29
Choosing an Appropriate Resource File	2-29
WTK Parameters Specified in a Resource File	2-30
Telling WTK to Use Resource Information	2-32
Modes of Stereoscopic Viewing	2-34
Field Sequential Mode	2-34
Over/Under Mode	2-35
Interlaced Mode	2-36
3: Object/Property/Event Architecture	3-1
Overview	3-1
Supported Types and Supplied Properties	3-2
WTnode Properties	3-3
WTviewpoint Properties	3-4
WTwindow Properties	3-4
WTsensor Properties	3-5
WTpath Properties	3-6
WTbase Objects and Functions	3-7
WTbase Functions for WTbase Objects	3-7
WTbase Functions for the Supported WTK Object Types	3-10
Properties	3-14
Events	3-23
Time	3-27
4: Scene Graphs	4-1
Introduction	4-2
The Scene	4-2
Elements Of A Scene	4-2
The Viewpoint	4-3
The Scene Graph	4-3
Why WTK Uses the Scene Graph Structure	4-4
Scene Graph Concepts in Detail	4-5
The Node	4-5
The Scene Graph Hierarchy	4-6
Viewing your Scene Graph	4-8
How WTK Draws the Scene Graph	4-9

Why the Ordering of Children is Important	4-15
State Accumulation and State Propagation	4-17
State Encapsulation	4-21
Other Node Types	4-25
Building a Scene Graph	4-29
How to Create the Scene Graph Tree	4-29
Building a Composite Object in the Scene – Composite Transformations	4-30
WTK Scene Graph Functions	4-39
Constructing Node Types	4-39
Constructing Light Nodes	4-43
Constructing Geometry Nodes	4-44
Constructing Movable Nodes	4-45
Constructing Fog Nodes	4-45
Loading a File into a Scene Graph	4-45
Saving a Scene Graph	4-48
Node Property Functions	4-48
Geometrical Property Functions	4-51
LOD Node Functions	4-55
Separator Node Functions	4-56
Switch Node Functions	4-57
Transform Node Functions	4-58
URL for Anchor and Inline Nodes	4-62
Anchor Node Functions	4-63
Inline Node Functions	4-63
Fog Node Functions	4-64
Open GL Callback Node Functions	4-67
Bounding Boxes	4-72
Scene Graph Assembly	4-74
Utility Functions	4-76
Scene Graph Structure Inquiry	4-76
Scene Graph Traversal	4-78
Additional Topics Related to the Scene Graph	4-79
Node Paths	4-79
Intersection Testing	4-85
Picking Polygons	4-91
Sensor Attachment	4-92
5: Movable Nodes	5-1
Introduction	5-1
What Makes Up a Movable Node?	5-1
Movable Node Creation Functions	5-3

Geometry and Light Movable Node Creation	5-3
Group Movable Node Creation	5-4
Movable Nodes Compared to ‘Regular’ Nodes	5-6
Movable Node Position and Orientation	5-7
Movable Node Hierarchies	5-9
Movable Node Instancing	5-13
6: Geometries	6-1
Introduction	6-1
Modeling Considerations	6-2
File Formats Supported by WTK	6-2
WTK VRML 1.0 Limitations	6-4
Exporting a File in the VRML Format	6-4
Notes on the Autodesk 3DStudio Mesh reader	6-5
Notes on the MultiGen OpenFlight File Reader	6-5
Subfaces in MultiGen/ModelGen	6-7
Constructing a World with Multiple Objects	6-7
Vertex Normals and Gouraud Shading	6-8
Vertex Colors and Radiosity	6-9
Back Face Rejection	6-10
Overlapping Polygons	6-12
Roundoff and Scaling	6-13
Creating Predefined Geometries	6-14
Creating Custom Geometries	6-21
Other Geometry Functions	6-26
Geometry Properties	6-28
Materials used with Geometries	6-30
Geometry Polygons and Vertices	6-32
Geometry Modification	6-37
Geometry Optimization	6-39
Creating Reflection Mapped Optimized Geometries	6-41
Vertex-level Geometry Editing	6-42
7: Polygons	7-1
Introduction	7-1
Polygon Attributes	7-2
Polygon ID’s	7-6
Geometry that Contains a Polygon	7-7
Polygon Access	7-8
Vertex Access	7-8
Dynamic Polygon Creation	7-10
Deleting Polygons	7-12

Polygon Intersection Testing	7-13
8: Materials	8-1
Introduction	8-1
Material Properties	8-1
Calculations Made to Determine Color	8-3
About “In” and “Out” Vectors	8-5
Using Material Tables	8-5
Material Table Functions	8-7
Example: Adding Shininess to a Multi-colored Geometry	8-10
Material Table Entry Functions	8-14
Advanced Topics	8-17
How WTK Deals With Out-Of-Range Indices	8-17
Using Material Index Table Entries	8-18
Using Materials Tables With Geometries	8-18
Notes on Specific File Formats	8-19
OpenGL Compatibility	8-19
9: 3D Text	9-1
Creating Three-dimensional Text in WTK	9-1
NFF 3D Font Files	9-5
10: Textures	10-1
Introduction	10-2
Supported Texture File Formats	10-3
Applying Textures	10-4
How WTK Applies a Texture to a Polygon	10-5
Texture Size	10-7
Texture Naming Conventions	10-8
Transparent Textures	10-8
Applying Textures with Explicit uv Values	10-13
Animating Textures	10-18
Assigning Textures in 3D File Formats	10-22
Deleting Textures	10-23
Changing Texture Properties	10-23
Filtering Textures	10-24
Setting the Default Texture Filter	10-25
Manipulating Textures	10-27
Texture Rotation, Scaling, and Other Operations	10-27
Manipulating Texture uv Values Directly	10-32
Screen Loading	10-33

11: Tasks	11-1
Introduction	11-1
Creation and Deletion Functions	11-2
Other WTask Functions	11-5
12: Lights	12-1
Introduction	12-1
Light Nodes	12-1
Light Node Attributes	12-2
Calculating Color	12-3
Determining Intensity	12-3
Creating Shadows	12-4
Using Light Files	12-4
Performance	12-4
Constructing Light Nodes	12-5
Light Properties	12-12
13: Sensors	13-1
Introduction to the Sensor Class	13-2
Sensor Lag and Frame-rate	13-5
Sensor Construction and Destruction	13-5
Accessing Sensor State	13-11
Rotating Sensor Input	13-16
Geometry Motion Reference Frames	13-19
Constraining Sensor Input	13-21
Using Different Baud Rates	13-22
Sensor Name	13-23
User-specifiable Sensor Data	13-23
Custom Sensor Drivers	13-24
The Mouse	13-26
Ascension Bird	13-39
Streaming-Mode Flock of Birds Driver	13-44
Ascension Extended Range Bird	13-51
CIS Graphics Geometry Ball, Jr.	13-53
Fakespace BOOM Devices	13-55
Fakespace Pinch Glove System	13-59
Fifth Dimension Technologies' 5DT Glove	13-63
Gameport Joystick	13-67
Limitations	13-67
Installing the joystick driver under NT	13-67
Configuring and calibrating the joystick	13-67
Creating a Gameport Joystick Sensor Object	13-68

Logitech 3D Mouse (Red Baron)	13-73
Logitech Head Tracker	13-77
Logitech Space Control Mouse (Magellan)	13-81
Polhemus ISOTRAK	13-85
Polhemus ISOTRAK II	13-88
Polhemus InsideTRAK	13-90
Polhemus FASTRAK	13-92
Precision Navigation Wayfinder-VR	13-96
Spacetec IMC Spaceball	13-100
Spacetec IMC Spaceball SpaceController	13-104
StereoGraphics CrystalEyes and CrystalEyesVR LCD Shutter Glasses ..	13-108
ThrustMaster Formula T2 Steering Console	13-111
ThrustMaster Serial Joystick	13-113
VictorMaxx Technologies' CyberMaxx2 HMD	13-119
Virtual i-O i-glasses!	13-121
Virtual Technologies CyberGlove	13-123
Initializing the CyberGlove	13-124
Calibrating the CyberGlove	13-126
Creating a Graphical Hand Model for CyberGlove	13-127
Setting the Visibility of the Hand Model	13-130
Accessing Hand Model Objects	13-130
Accessing the CyberGlove Bend Angle Data	13-132
Defined Constants for the CyberGlove Hand Model	13-134
For Windows NT Users:	13-135
14: Paths	14-1
Introduction	14-1
Path Construction and Destruction	14-2
Functions	14-4
Path Management	14-8
Loading and Saving Paths	14-11
Path File Format	14-12
Recording and Playback	14-13
Path Element Management	14-24
The WTPathelement Class	14-24
Path Editing	14-27
Path Name	14-29
User-specifiable Path Data	14-29
15: Motion Links	15-1
Introduction	15-1
Motion Link Sources and Targets	15-1

Reference Frames	15-2
Constraints	15-3
Motion Link Functions	15-3
Constraints on Motion links	15-9
16: Viewpoints	16-1
Introduction	16-1
Basic Viewpoint Management	16-3
Linking a Sensor to a Viewpoint	16-6
Accessing Viewpoint Position and Orientation	16-8
Using a Specified Reference Frame	16-14
Viewpoint Aspect Ratio	16-18
Stereo Viewing	16-19
Coordinate Transformations	16-24
Viewpoint Name	16-25
User-specifiable Viewpoint Data	16-25
Viewpoint Intersection Test	16-26
17: Windows	17-1
Introduction	17-1
Window Construction and Destruction	17-2
Accessing Universe's Windows	17-8
Associating Scene Graphs with Windows	17-8
Window Size and Placement	17-10
Windows and Viewpoints	17-11
Zooming the Window Viewpoint	17-13
Window-projection Functions	17-14
Other Window-projection Functions	17-17
Picking and Ray Casting	17-20
Window-rendering Properties	17-22
Window Name	17-27
User-specifiable Window Data	17-28
System-specific Window ID	17-29
Viewports	17-30
18: Adding User Interface (UI) Objects	18-1
Creating a UI Application	18-2
User Interface Objects	18-13
Forms	18-13
File-selection Boxes	18-13
Message Boxes	18-15
Text-input Dialogs	18-15

Checkbuttons	18-16
Labels	18-17
Pushbuttons	18-18
Radioboxes	18-18
Scales	18-19
Scrolled Lists	18-21
Scrolled Text	18-23
Text Fields	18-24
Menus	18-24
Tool Bars	18-28
User Interface Object's Utility Functions	18-29
Accessing the Scale Factors	18-29
Accessing the Text for Text UI Objects	18-29
Accessing the Position of a Selection (Scrolled Lists and Radioboxes) ..	18-31
Accessing the Number of Items (Scrolled Lists and Radioboxes)	18-32
Accessing Text of Scrolled List Items	18-32
Inserting or Deleting Items (Scrolled Lists)	18-33
Accessing Status of UI Objects	18-34
Accessing State of UI Objects (Menu Items and Checkbuttons)	18-35
Accessing the Position of UI objects	18-36
Extending The UI Functionality of Your Application	18-37
Controlling the WorldToolkit Simulation Loop	18-37
Miscellaneous Functions	18-40
19: Drawing Functions	19-1
User-defined Drawing Functions	19-1
2D Drawing	19-1
Pre-defined 2D Drawing Functions	19-1
3D Drawing	19-8
Pre-defined 3D Drawing Functions	19-8
20: Sound	20-1
Introduction	20-1
Supported Devices	20-1
Device-level Functionality	20-3
CRE Device Parameters	20-7
Device-level Spatializing Functions	20-9
Sound-level Functionality	20-10
Sound-level Spatializing Functions	20-17
21: Client-Server Networking (Via the World2World Servers)	21-1

Introduction	21-1
Sharing Properties	21-2
Locked Properties	21-3
Persistent Properties	21-3
Update Frequencies	21-3
Time Sensitive Properties	21-4
WTbase – Working with Unsupported Object Types	21-5
Property Sharing Functions	21-5
Sharegroups	21-11
Locked Sharegroups	21-12
Registered Interest	21-13
Persistent Sharegroups	21-14
Sharegroup Functions	21-15
Network Connections	21-22
Synchronous and Asynchronous Connections	21-22
Update Rates	21-23
Connection Callbacks	21-23
Connection Functions	21-26
Enumeration	21-34
Example of an Enumeration Tree	21-34
WorldToolKit and World Up Compatible Properties	21-38
22: Multicast Networking	22-1
Introduction to Networking in WTK	22-1
How the Transport Layer Works	22-2
How the Protocol Layer Works	22-3
How the WorldToolKit Layer Works	22-3
How the Application Layer Works	22-4
Sample Transaction	22-4
Local Machine	22-4
Remote Machines	22-5
Message Latency	22-5
Byte Ordering	22-6
Network Functions	22-7
23: Serial Ports	23-1
Introduction to the Serial Port Class	23-1
Serial Port Construction and Destruction	23-1
Reading and Writing to a Serial Port Object	23-3
User-specifiable Serial Data	23-4
Platform Specific Functions	23-5

24: Portability	24-1
Providing for Portability	24-1
Reading the Keyboard	24-1
Reading File Directories	24-4
Messages and Errors	24-5
Waiting	24-8
Memory Allocation	24-9
25: Math Library	25-1
Introduction	25-1
WTK Math Conventions	25-2
WTP2: 2D Vectors	25-4
WTP3: 3D Vectors	25-5
WTq: Quaternions	25-12
WTPq: Coordinate Frame Structure	25-19
WTm3: 3D Matrices	25-21
WTm4: 4D Matrices	25-22
Conversion Functions	25-25
Floating-point Comparisons	25-33
Reference-frame Math Utilities	25-34
26: C++ Programming	26-1
Introduction	26-1
Class Diagrams	26-2
Classes and their Methods	26-4
Prototypes for Global functions	26-5
World2World Client C++ Applications	26-5
WtBase Classes	26-6
Stand-alone Classes	26-33
Math Classes	26-39
Defines	26-45
Appendix A: Frequently Asked Questions	A-1
Introduction	A-1
What Is The Difference Between WTnode_load And WTgeometrynode_load?	
A-3	
What Is The Difference Between WTmovnode_load and WTnode_load? ...	A-4
How Do I Display Multiple Instances Of An Object?	A-5
How Do I Pick The Frontmost Polygon At A Specific Point In A Specific Win-	
dow?	A-6
Can WTK Detect Keyboard Events?	A-8
How Can I Detect Button Events Using the “Misc Data” Functions?	A-10

How Do I Use Material Tables for Colors?	A-11
How Do I Get Transparencies In A Texture?	A-12
How Do I Dynamically Change The Appearance Of A Geometry?	A-13
How Do I Create Special Effects: Clouds, Missile Trails, Exhaust and Explosions	A-13
Gas Clouds	A-13
Missile plumes	A-14
Spaceship exhaust	A-14
Explosions	A-15
How Do I Load Lights As Movables?	A-15
How Do I Make An Object Follow A Light?	A-16
How Do I Make An Object Follow The Viewpoint?	A-16
How Do I Recursively “Walk” Down The Scene Graph?	A-19
How Do I Get A Pointer To A Node Using Its Name?	A-20
How Do I Associate A Task With a Particular Object?	A-21
How Do I Handle Portals In This Release?	A-22
How Do I Test For Intersections Between The Viewpoint And The Universe? .	A-24
How Do I Test For Objects Intersecting With Other Objects In The Universe? .	A-25
How Do I Get The Rendered Position Of An Object?	A-25
How Do I Create A Simple Animation Using Switch Nodes?	A-26
How Can I Optimize Performance Using LOD Nodes?	A-29
What Is Terrain Following?	A-31
How Do I Keep An Object Perpendicular To The Viewpoint Direction At All Times?	A-33
How Do I Change The Event Order?	A-34
How Do I Integrate A WTK Rendering Window With A Host-Specific Window?	A-35
Orienting Sensors Differently	A-36
How Do I Use Orientation-Tracking Sensors (On A Head-Mount-Display) That Are Not Positioned Along The Central Axis Of The HMD?	A-36
Example Code	A-37
How Do I Measure Performance On My Machine?	A-38
On UNIX Platforms, How Do I Get A Pointer To The Display That WTK Is Using?	A-38
How do I use Boston Dynamic's DiGuy with WTK (or any other BDI character set)?	A-39
Appendix B: Environment Variables	B-1
WTKCODES	B-1
WTIMAGES	B-2

WTMODELS	B-2
WTKZBUFFERSIZE	B-3
WTKALPHATEST	B-3
WTKMAXTEXTSIZE	B-4
WTKSQRTTEX	B-4
WTKPROXY	B-4
WTKALPHAENABLE	B-5
WTBIRDDELAY	B-5
WTKLS	B-5
WTKNOSTEREO	B-6
WTKMULTISAMPLE	B-6
WTKCPU	B-6
WTKDISPLAY	B-7
WTKSHMEM	B-8
Appendix C: Defined Constants	C-1
Constraint Constants	C-1
Display Constants	C-2
Drawing Constants	C-2
Event Order Constants	C-3
Eye Constants	C-3
Filetype Constants	C-3
Frames of Reference Constants	C-4
Keyboard Constants	C-4
Light Type Constants	C-5
Material Table Property Constants	C-5
Mathematical Constants	C-6
Message Constants	C-6
Motion Link Source and Target Constants	C-7
Node Constants	C-7
Option Constants	C-9
Path Constants	C-9
Projection Type Constants	C-10
Rendering Constants	C-10
Sensor Constants	C-11
Serial Port Constants	C-17
Sound Constants	C-18
Sound Device Constants	C-19
Texture Constants	C-20
User Interface Constants	C-20
Window Constants	C-21
Other Constants	C-21

Appendix D: Error Messages and Warnings	D-1
Error Messages	D-1
Warnings	D-5
Appendix E: Writing a Sensor Driver	E-1
Overview	E-2
WTK Math Conventions	E-2
Sensor Records Must Be Relative	E-2
Constraining Sensor Records	E-3
Scaling Sensor Records	E-3
Talking to the Serial Port	E-4
Include Files	E-4
Driver Functions	E-5
Example 1: Update Function for the Mouse	E-8
Example 2: Driver for the Geometry Ball Jr.	E-10
Example 3: Update Function for Absolute Device (Pseudocode)	E-15
Appendix F: WTK Neutral File Format	F-1
The NFF Format	F-1
The BFF Format (Binary NFF)	F-1
NFF Syntax	F-2
NFF Header	F-2
NFF Objects	F-3
NFF Materials	F-4
NFF Vertices	F-4
NFF Polygons	F-6
NFF Format Extensions	F-8
Automatic Normal Generation	F-8
NFF Version History, Backward Compatibility	F-9
A Sample NFF File	F-10
Appendix G: Transitioning From Version 2.1 To Release 6/7/8/9	G-1
Introduction	G-1
Paradigms of this Release	G-2
The Scene Graph	G-2
Instancing	G-5
Materials	G-5
Lights	G-6
Special Effects (Fog)	G-6
3D Sound	G-7
Multiple Windows	G-7

User-Interface (UI) Objects	G-7
Motion Links	G-7
Switches and Level of Detail Nodes	G-8
Replaced Features	G-8
Mapping WTK V2.1 Functions To This Release	G-9
Details on Mapping WTK V2.1 Functions to This Release	G-22
Loading In Objects	G-22
Changes in Reading/Writing NFF Files	G-24
Attaching Objects To One Another	G-25
Handling Of Lights In This Release	G-26
Moving from WTxx_addsensor to Motion Links	G-27
Rotating A Movable About Its Midpoint	G-28
Changing Vertex Positions	G-28
Differences in Applying Tasks	G-29
Positioning And Moving Objects In Your Scene: WTobject and WTgeometry	
G-30	
Picking	G-31
Animation	G-31
The Lack of WTgroup_* Functions	G-32
Pivot Points And Handles	G-32
Coordinate Frames	G-34
New Functions to Facilitate Incorporation of WTK V2.1 Applications into the	
R6/R7/R8/R9 Paradigm	G-34
Scene Graphs and Nodes	G-34
Material Colors	G-35
Appendix H: Transitioning From Release 6 To Release 7/8/9	H-1
Changed Functions from Release 6 to	
Release 7/8/9	H-1
WTK User-Interface (UI) Functions	H-1
Transformations	H-4
Appendix I: Third-party Software	I-1
Image Conversion (SGI)	I-2
Image Conversion (Windows 32-bit Platforms)	I-2
Model Conversion	I-3
3D Modelers	I-3
Appendix J: Sources of Components	J-1
Input Devices	J-1
Output Devices	J-2
Video Accelerators	J-3

Appendix K: The WTK Users' Group	K-1
Participating in SIG-WTK	K-1
Communicating with SIG-WTK	K-2
SIG-WTK:Email Archives	K-2
Appendix L: Technical Support	L-1
U.S. Technical Support	L-1
Non-US Technical Support	L-2
SIG-WTK Users' Group	L-2
Appendix M: Glossary	M-1
Index.....	Index-1

Introduction to WorldToolKit

Welcome

Welcome to WorldToolKit (WTK), an advanced cross-platform development environment for high-performance, real-time 3D graphics applications. WTK has the function library and end-user productivity tools you need to create, manage, and commercialize your applications. With the high-level application programmer's interface (API), you can quickly prototype, develop, and configure your applications as required.

From writing custom sensor drivers to rapidly developing virtual reality applications, WTK offers an intuitive set of functions that provide a wide range of functionality. This chapter introduces you to the WTK application development environment, highlights the major concepts and features in this release, and reviews the basic hardware and software components of a WTK development system.

What is WTK?

Simply stated, you build your virtual world by writing code to call WTK functions. WTK is a library of over 1000 functions written in C that enable you to rapidly develop new virtual reality applications. One function call can do the work of hundreds of lines of C code, dramatically shortening development time.

WorldToolKit is so named because your applications can resemble virtual worlds, where objects can have real-world properties and behavior. You control these worlds with a variety of input sensors, from a simple mouse to "six degrees of freedom" input devices. Users can experience these worlds with a computer display (which acts as a movable window into a world) or by using a position-tracked, head-mounted, stereoscopic display.

WTK is structured in an object-oriented way, although it does not use inheritance or dynamic binding. WTK functions are object-oriented in their naming convention, and are

organized into over 20 classes. These classes include the Universe (which manages the simulation and contains all other objects), Geometries, Nodes, Viewpoints, Windows, Lights, Sensors, Paths, Motion Links, and others. (See *Overview of the WTK Classes* on page 1-3.) Functions are included for things such as device instancing, display setup, collision detection, loading geometry from a file, dynamic geometry creation, specifying object behavior, manipulating textures, and controlling rendering.

Scene Graph Architecture

The architecture of this release of WTK incorporates the power of scene hierarchies. With WTK you can build a simulation by assembling nodes into a hierarchical *scene graph*, which dictates how the simulation is rendered and allows all of the efficiencies of a state-preserving, stack-oriented rendering architecture. Each node of the scene graph (or scene graphs) represents part of the simulation.

This efficient visual database representation provides increased performance, control, and flexibility through features such as hierarchical object culling, efficient use of transform information, Level of Detail switching, object grouping, VRML compatibility, and the ability to load in models and data from the Internet. With the scene graph approach, you can create a light, and specify the light's location in the scene graph such that it only effects the geometry you choose.

While providing the expressiveness and flexibility of constructing the scene graph for your visual database node-by-node, WTK also contains functions that let you create scene graphs by loading in files that contain scene graph descriptions. For example, loading a VRML file from the Internet into your scene graph requires just a single function call. WTK also provides functions for easily modifying and reconfiguring scene graphs.

What WTK Does

WTK manages the tasks of rendering, reading input sensors, importing geometries, and a wide range of simulation functions. You are left free to concentrate on developing the details of your 3D applications.

At the core of an application written using WTK is a simulation loop that reads input sensors, updates objects, and renders a new view of your scene onto the display. WTK is designed to be used in real-time applications such as simulations, where frame rates on the

order of 5 to 30 frames per second are maintained. WTK's main loop and event dispatching mechanisms are similar to those of a conventional window manager, but WTK applications differ in that they are intended for use in situations where the user's viewpoint or objects in the universe are continuously changing.

WTK incorporates the philosophy of OpenVR™, which means it is portable across platforms, including SGI, Sun, DEC, Intel, and Evans and Sutherland. WTK is optimized to leverage the power of each hardware platform it supports, enabling your applications to use the “fast path” through whatever graphics acceleration system you are using.

WTK supports a wide variety of input and output devices, and allows you to incorporate existing C code (such as device drivers, file readers, and drawing routines) into your WTK application.

Overview of the WTK Classes

WTK is structured in an object-oriented way. Most WTK functions are object-oriented in their naming conventions and are grouped into the following classes:

- **Universe** is the “container” of all WTK objects such as geometries, nodes, viewpoints, sensors, etc. While you can have multiple scene graphs and simulations, there is only one universe. You can temporarily add or remove geometries and nodes from being considered by the simulation manager. You can also define the sequence of events in the simulation.
- **Geometries** are graphical objects that are visible in a simulation, such as a block, sphere, cylinder, and 3D text. You can dynamically create geometries or import them from other sources. Once you create a geometry, you need to create a corresponding (geometry) node so that it can be included in a scene graph.
- **Nodes** are the building blocks from which scene graphs are constructed. Node types other than geometry nodes, such as light nodes, fog nodes, transform nodes, level-of-detail (LOD) nodes, and switch nodes are not visible, though they can affect the appearance of geometry nodes.
- **Polygons** can be dynamically created and texture-mapped using various sources of image data. You can render polygons in either wireframe, smooth-shaded or textured modes.
- **Vertices** can be dynamically created or read from a file. You can also associate vertices with vertex normals for gouraud shading.

- **Lights** can be dynamically created or loaded from a file.
- **Viewpoints** define the position and orientation in a virtual world from which all of the geometries in a simulation are projected to the screen and rendered. WTK supports one or more viewpoints. You can also control a viewpoint's position and orientation by attaching sensors to it.
- **Windows** display your scene. A WTK application can have multiple windows into the same virtual world and/or multiple windows into different virtual worlds.
- **Sensors** can be connected to transform nodes, viewpoints, movable nodes, etc., to manipulate object motion. Multiple sensor objects are supported.
- **Path** objects allow geometric or viewpoints to follow predefined paths. You can dynamically create, interpolate, record, and play paths.
- **Tasks** can be used to assign behaviors (such as movement, change in appearance) to individual objects.
- **Motion Links** connect a source of position and orientation information with a target that moves to correspond with that changing set of information. For example, you can have a motion link between a sensor and a viewpoint.
- **Sound** objects can be loaded, associated with 3D objects in the scene, and played.
- **User Interface** elements can be created for both X/Motif and Microsoft Windows environments.
- **Networking** capabilities enable you to build applications that can asynchronously communicate over an Ethernet between several PC and UNIX workstations. This allows distributed simulations to be created where a mixture of PCs and UNIX workstations support a single simulation.
- **Serial Port** functions simplify the task of communicating over serial ports.

Naming Conventions

Naming conventions for WTK functions are such that each class of object has a *typedef* (type definition) defining an object of that type. For instance, *WTsensor* is a sensor object, and *WTserial* is a serial port object. Objects are always dealt with through pointers. In fact, the internal state of WTK objects is not accessible except through WTK function calls provided for this purpose. Objects in WTK are “opaque,” enforcing data abstraction. The state of any object must be accessed through “set” and “get” access functions defined in the WTK library.

All functions acting on a given class have, by convention, a name that begins with the class name. In addition, all classes accessible by the user have an object constructor whose name ends in *_new*, which returns a new object of the given class, and an object destructor ending in *_delete*, which accepts and destroys an object of the given class.

For instance, the function:

```
WTviewpoint *WTviewpoint_new();
```

creates a new viewpoint object and returns a pointer to that object, as in:

```
newview = WTviewpoint_new();
```

This new viewpoint could subsequently be destroyed by the call:

```
WTviewpoint_delete(newview);
```

Most functions expect a pointer to an object of their class as the first argument. This is the object to which the function is directed. To copy a viewpoint, you would call the function *WTviewpoint_copy*, which takes a pointer to an already-existing viewpoint and returns a pointer to a newly-created copy of that viewpoint:

```
WTviewpoint *old_viewpoint, *new_viewpoint;  
new_viewpoint = WTviewpoint_copy(old_viewpoint);
```

The universe object is special in that there is only one universe at any given time. For this reason, universe functions do not require a universe pointer as the first argument.

Additional Features

SOUND

WTK provides a cross-platform API for creating 3D and stereo sound. On Windows 32-bit systems, WTK supports Windows-compatible sound cards, DiamondWare sound, and Crystal River Engineering products. On Silicon Graphics Workstations, WTK supports the SGI system audio and Visual Synthesis 3D sound products. See *Appendix H, Third Party Software*, and the SENSE8 web site at <http://www.sense8.com> for the latest information on third party sound device support.

WTK's sound API provides support for 3D spatialization of sounds, doppler shifts, volume and roll-off controls, and other effects. It supports output to a variety of devices including headphones, surround sound, and stereo systems.

USER-INTERFACE OBJECTS

You can add a user interface (UI) to your simulations by using WTK's cross-platform user-interface objects. These objects let you quickly and easily create a (2D) graphical user interface. These UI objects have been designed in both Motif and Windows styles, to match the native operating system. The UI object types provided include: toolbars, bitmaps, menus, message boxes, text boxes, file-request dialogs, and others. When you recompile your simulation on another platform, the UI objects automatically change to match the new operating system. For example, if you develop an application using toolbars for X-Windows, and then recompile it in Windows, your simulation will use Windows style toolbars.

MULTIPIPE/MULTI-PROCESSOR SUPPORT

A multipipe/multi-processor version of WTK is also available. It provides support for rendering to multiple graphics pipes or screens and utilizes the additional power available on multi-processor systems. This is useful for creating high-resolution stereo displays for Computer-Assisted Virtual Environment's (CAVEs).

VRML SUPPORT

WTK supports the reading and writing of VRML 1.0 files.

OTHER FEATURES

Other features of this release include the following:

- **Materials and Translucency** - Complete control of coloring geometries, including specular highlights. WTK takes full advantage of the features available with OpenGL.
- **Task Objects** - You can specify the behavior of any geometry, node, or C structure by assigning tasks to it.

- **Performance Optimizations for Rendering** - Support for triangle stripping, state sorting, etc.
- **Atmospheric Effects** - Support for special effects, such as fog, haze, and cloud layers.
- **Constraints** - Available on the translations and rotations of your geometry or other scene graph components.
- **Textures from Memory** - For video and playback onto object surfaces.
- **Orthographic Projections** - Useful for plan views or anytime a perspective projection is not desired.
- **Cross-Platform 2D Drawing Calls** - Support for geometrical shapes, lines, bitmaps, etc.
- **Support for Many Sensors** - See the table on page 13-3 for a list of the WTK supported sensors.
- **Support for 3D Text** - Capability of creating 3D text in your virtual world.
- **Support for Many File Formats** - Supports WRL, FLT, DXF, NFF, OBJ, 3DS, BFF, SLP, and GEO file formats.
- **C++ Wrappers** - Provides the choice of programming in either C or C++.

WTK Documentation

The available sources of documentation for WTK include the following:

REFERENCE MANUAL

The Reference Manual describes the core functionality of WTK. This reference manual has 23 chapters and 12 appendices:

Chapter 1, *Introduction to WorldToolKit*, provides an overview of the WTK application development system, introduces key concepts pertaining to WTK's object-oriented nature, and reviews the basic hardware and software components of a WTK development system.

Chapter 2, *The Universe*, introduces the universe class and describes many of the key functions for interacting with and managing your simulation.

Chapter 3, *Object/Property/Event Architecture*, describes the new Object/Property/Event programming paradigm that has been introduced with WTK Release 8.

Chapter 4, *Scene Graphs*, describes how scene graphs are created and describes the various kinds of nodes used to construct a scene graph.

Chapter 5, *Movable Nodes*, describes the concept and basic structure of movable nodes, and how they are created, positioned and built into hierarchies.

Chapter 6, *Geometries*, introduces the concept of geometries, and provides file format and instancing information. Functions are provided to create predefined geometries, copy existing geometries, add materials to geometries, etc.

Chapter 7, *Polygons*, discusses the polygonal surfaces that geometrically describe an object. Functions for polygon construction, querying, and intersection-testing with other graphical entities are also presented.

Chapter 8, *Materials*, introduces material tables and their functions, including setting values in the material table and creating new material tables.

Chapter 9, *3D Text*, shows how to create 3D text in your simulation. 3D text objects are special forms of graphical objects.

Chapter 10, *Textures*, describes the textures that can be applied to the surfaces of graphical objects, and the functions to apply, manipulate, and animate them.

Chapter 11, *Tasks*, discusses the way tasks are assigned to a geometry (or other object) to provide movement, change its appearance, detect intersections with other geometries, etc.

Chapter 12, *Lights*, describes the WTK functions used to manage lighting conditions in the graphical environment.

Chapter 13, *Sensors*, provides information about the WTK sensor functions, using the data from sensors, and using various manufacturers' hardware with your simulation.

Chapter 14, *Paths*, introduces the concept of a path, which is a sequence of position and orientation information. Functions are described for creating paths, editing them, and using them to guide the viewpoint or other objects.

Chapter 15, *Motion Links*, introduces the concept of linking sources and targets of position and orientation information with a motion link. Functions are provided to link targets to sensors or paths.

Chapter 16, *Viewpoints*, introduces the WTK “viewpoint” object, which defines how your simulation is projected onto your display device. Functions are described for viewpoint construction, movement, coordination with sensor input data, and stereo viewing.

Chapter 17, *Windows*, shows how to create windows, associate viewpoints with windows, and set or change the characteristics of a window.

Chapter 18, *Adding User Interface (UI) Objects*, describes how to add a cross-platform graphical user interface to your simulations.

Chapter 19, *Drawing Functions*, provides information on 2D and 3D drawing functions supported by WTK.

Chapter 20, *Sound*, introduces spatialized and regular sound support for a variety of hardware platforms. Options and functions give you the ability to control how, when, and where sound is included in your simulation.

Chapter 21, *Client-Server Networking (Via the World2World Servers)*, describes how to create multi-user client-server applications for use with Sense8’s World2World server product.

Chapter 22, *Multicast Networking*, describes how you can create applications that asynchronously communicate over an Ethernet network.

Chapter 23, *Serial Ports*, describes the class of functions that simplifies the task of communicating over serial ports.

Chapter 24, *Portability*, discusses issues associated with constructing platform-independent WTK applications, and describes functions for using the keyboard, working with files and directories, and handling messages or errors.

Chapter 25, *Math Library*, provides a description of the WTK math functions for managing position and orientation data.

Appendix A, *Frequently Asked Questions*, provides answers to some common questions on how to use many of WTK’s powerful features.

Appendix B, *Environment Variables*, describes the environment variables that you can use to customize WTK's operation on your computer.

Appendix C, *Defined Constants*, lists WTK's constants.

Appendix D, *Error Messages and Warnings*, reviews the error messages and warnings, and how to suppress or redirect them.

Appendix E, *Writing a Sensor Driver*, introduces the functions available to simplify the task of writing a custom sensor driver. Sample sensor driver programs are also given in this chapter.

Appendix F, *WTK Neutral File Format*, describes WTK's generic ASCII and binary formats for describing polygonal geometry, and gives sample NFF files.

Appendix G, *Transitioning From Version 2.1 To Release 6/7/8/9*, provides key information to smooth your transition from WTK V2.1 to this Release 6/7/8/9.

Appendix H, *Transitioning From Release 6 To Release 7/8/9*, lists the functions that have changed from Release 6 to Release 7/8/9, and describes what you need to do if your application uses these functions.

Appendix I, *Third-party Software*, includes a list of other software products and their vendors that you may find useful.

Appendix J, *Sources of Components*, includes a list of hardware products and their vendors that you may find useful.

Appendix K, *The WTK Users' Group*, gives you contact information for the WTK Users' Group.

Appendix L, *Technical Support*, gives technical support contact information for WTK.

Appendix M, *Glossary*, provides definitions for many important WTK terms and concepts.

THE WTK INSTALLATION AND HARDWARE GUIDES

System-specific aspects of WTK are described in the appropriate WTK Installation and Hardware Guide. There is a version of the Installation and Hardware Guide for most platforms on which WTK runs. Throughout this Reference Manual you're referred to this

Installation and Hardware Guide whenever there are system-specific considerations for a particular subject.

THE WTK QUICK REFERENCE GUIDE

An alphabetical summary of all WTK functions, macros, and constants is given in the WTK Quick Reference Guide. This Quick Reference Guide is available in PDF format on the WTK product CD. A hardcopy of the Quick Reference Guide is **NOT** shipped with the WTK product.

ONLINE DOCUMENTATION

An online version of this manual and the (platform-specific) Installation and Hardware Guide is installed with WTK in the portable document format (PDF). PDF is a cross-platform file format that you can read with an Adobe Acrobat reader. This reader was installed during WTK installation unless you chose not to install it.

These online documents are identical with the printed documents but allow you to use a search feature to quickly find WTK functions and other valuable reference information. (TIP: While you are viewing a document in the PDF reader, click the second icon on the toolbar to display the bookmarks. Then, click a bookmark to go to any chapter.) For more information on using the PDF reader, see the Adobe Acrobat help file.

ADDITIONAL SOURCES

See the Readme file that was installed with WTK for last minute information or reported problems. You can also find the latest product information by accessing the SENSE8 web site (<http://www.sense8.com>).

Special Interest Group

WTK users are invited to join the WTK User's Group (SIG-WTK). The WTK User's Group has been organized by WTK customers with assistance from EAI/SENSE8. SIG-WTK provides a worldwide electronic forum for the discussion of shared interests.

To subscribe or unsubscribe to SIG-WTK, e-mail your request to:
sig-wtk-request@sense8.com with the text *subscribe* or *unsubscribe* as the body of the message.

Basic System Configuration

A basic WTK development system includes several key hardware and software components. These components are listed as follows. Additional system-dependent components that may not be necessary with your system are also listed [in square brackets].

- Host computer(s)
- [Hardware graphics accelerator board]
- WTK library
- C compiler
- [3-D modeling program]
- [Bit-map editing software]

It is also suggested that you have a mouse and at least one 3D/6D input sensor such as the Spacetec IMC Spaceball.

Input Sensors Supported

WTK supports a wide range of 3D and 6D input sensors, both desktop devices and devices that can be worn on the body to sense position and orientation. Routines to read supported sensor types are part of the WTK library. (See *Introduction to the Sensor Class* on page 13-2 for a list of the sensors that WTK supports.) Not all devices are supported on all systems, so please check your Hardware Guide to see which devices are supported on your system.

Extending a System for Virtual Reality

To extend the basic system configuration for a virtual reality interactive display, additional hardware components are required. The following list assumes that you have the software

and hardware listed under *Basic System Configuration* on page 1-12, including a 3D/6D input sensor:

- A stereoscopic head-mounted display or stereo projection system.
- Video signal conversion, typically from the RGB signals of the graphics device to the NTSC video inputs on the head-mounted display.
- One or more position tracking devices (to track the head position and orientation and/or other body gestures).

These components are system-dependent, so talk to your local distributors or directly to SENSE8 when configuring your system.

A Sample WTK Application

A WTK application is C source code that includes WTK function calls. These applications may be as simple or as complex as you like. Because WTK includes many “high-level” function calls, you can prototype an application with a few lines of code, and then extend it based on the demands of your application.

The following example is a simple, but complete WTK application; here’s what it does:

- Creates a new universe (with an empty scene graph).
- Loads a graphical model of a planet into the scene graph.
- Attaches a mouse device to the viewpoint.
- Assigns a behavior (task) to the planet, causing it to spin about its axis.
- Allows the user to fly around the planet using the mouse.

```
/* simple.c Usage: Use the mouse buttons to fly around a spinning planet. */
```

```
#include "wt.h"
```

```
void spin(WTnode *);
```

```
#define Y_AXIS 1
```

```
void main(int argc, char *argv[])
```

```
{
    WTnode *root;
    WTnode *planet;
    WTsensord *sensor;    /* the Mouse */
    WTviewpoint *view;    /* the Viewpoint */

    WTuniverse_new(WTDISPLAY_DEFAULT, WTWINDOW_DEFAULT);
    root = WTuniverse_getrootnodes();
    planet = WTmovnode_load(root, "PLANET.NFF", 1.0);
    sensor = WTmouse_new();
    view = WTuniverse_getviewpoints();
    WTviewpoint_addsensor(view, sensor);
    WTtask_new(planet, spin, 1.0);
    WTuniverse_ready();
    WTuniverse_go();      /* Starts simulation */
    WTuniverse_delete();  /* All done */
}

void spin(WTnode *planet)
{
    WTmovnode_axisrotation(planet, Y_AXIS, 0.1);
}
```

Important WTK Functions

Some of the most widely used WorldToolKit functions are briefly described here; the actual WTK functions are described in their respective chapters. They're also listed in the index. Also see your *Quick Reference Guide*, which includes all the WTK functions, macros, and constants, with a brief description of each one.

Note: Most classes use **new** to create an object, **add** to add it to the simulation, **remove** to remove it from the simulation, and **delete** to delete it from memory. All **set** functions have a corresponding **get** function.

Universe

<i>WTuniverse_framerate</i>	Returns the current simulation framerate.
<i>WTuniverse_go</i>	Starts the simulation.
<i>WTuniverse_setactions</i>	Defines actions that take place within the simulation event loop.
<i>WTuniverse_setbgrgb</i>	Sets the background color of the universe.
<i>WTuniverse_seteventorder</i>	Changes the processing order of the events in the simulation.

Geometry

<i>WTgeometry_newvertex</i>	Adds a vertex to a geometry object.
<i>WTgeometry_begin</i>	Creates a new geometry object for vertex addition.
<i>WTgeometry_beginedit</i>	Informs WTK that you are going to edit a geometry.
<i>WTgeometry_close</i>	Finishes the definition of the geometry.
<i>WTgeometry_deletetexture</i>	Removes all textures from the geometry.
<i>WTgeometry_endedit</i>	Informs WTK that you are finished editing a geometry.
<i>WTgeometry_getvertices</i>	Gets the first vertex of a geometry.
<i>WTgeometry_save</i>	Saves a geometry to a file.
<i>WTgeometry_settexture</i>	Globally applies a texture to a geometry object.
<i>WTgeometry_settextureuv</i>	Drapes a texture over the geometry.
<i>WTgeometry_setvertexnormal</i>	Sets the vertex normal.
<i>WTgeometry_setvertexmatid</i>	Sets the material of a vertex.
<i>WTgeometry_stretch</i>	Stretches the geometry along the X, Y, or Z axis.

Polygon

<i>Wtpoly_addvertex</i>	Adds a vertex to a polygon under construction.
<i>Wtpoly_begin</i>	Adds an empty polygon to a geometry.
<i>Wtpoly_close</i>	Finishes the definition of a polygon.
<i>Wtpoly_intersectpolygon</i>	Tests for intersection between two polygons.
<i>Wtpoly_deletetexture</i>	Removes a texture from a polygon.
<i>Wtpoly_rotatetexture</i>	Rotates a texture on a polygon.
<i>Wtpoly_scaletexture</i>	Scales and automatically tiles a texture.
<i>Wtpoly_setmatid</i>	Sets the material of a polygon.
<i>Wtpoly_settexture</i>	Applies a texture to a polygon.
<i>Wtpoly_settexturestyle</i>	Sets the shading and transparency values of a texture.
<i>Wtpoly_settextureuv</i>	Maps texture onto a polygon in a user-specified way.
<i>Wtpoly_setuv</i>	Changes the way a texture is mapped to a polygon's vertices.
<i>Wtpoly_stretchtexture</i>	Stretches a polygon's texture.
<i>Wtpoly_translatetexture</i>	Shifts a texture on a polygon by a pixel amount.

Sensor

<i>Wtsensor_getmiscdata</i>	Retrieves button-press information from the sensor object.
<i>Wtsensor_setconstraints</i>	Constrains values read in from a sensor.
<i>Wtsensor_setsensitivity</i>	Sets the sensitivity value of a sensor.
<i>Wtsensor_setrecord</i>	Stores the current relative position and orientation record.
<i>Wtsensor_setupdatefn</i>	Changes a sensor's update function.

Light

<i>WTlightnode_load</i>	Reads a formatted file to create spot, point, directed, and/or ambient lights.
<i>WTlightnode_newdirected</i>	Creates a directed light.
<i>WTlightnode_newpoint</i>	Creates a point light.
<i>WTlightnode_newspot</i>	Creates a spot light.
<i>WTlightnode_newambient</i>	Creates an ambient light.
<i>WTlightnode_setambient</i>	Sets the ambient light color.
<i>WTlightnode_setangle</i>	Sets the half-angle of a spot light's cone.
<i>WTlightnode_setattenuation</i>	Sets the attenuation of point and spot lights.
<i>WTlightnode_setdirection</i>	Sets the direction of a light.
<i>WTlightnode_setposition</i>	Sets the location of a light.

Viewpoint

<i>WTviewpoint_addsensor</i>	Attaches a sensor to control the viewpoint.
<i>WTviewpoint_moveto</i>	Moves the viewpoint to a particular location and orientation.
<i>WTviewpoint_setaspect</i>	Adjusts the aspect ratio of the image.
<i>WTviewpoint_setconvergence</i>	Adjusts the convergence of the image.
<i>WTviewpoint_setparallax</i>	Adjusts the parallax of the image.

Path

<i>WTpath_copy</i>	Copies a path.
<i>WTpath_interpolate</i>	Performs a Bezier, B-spline, or Linear interpolation of a path.
<i>WTpath_play</i>	Begins the playback of a path.

<i>WTpath_record</i>	Begins recording the viewpoint position and stores the information in a path.
<i>WTpath_setmode</i>	Sets the path's playback mode.

Window

<i>WTwindow_setbgrgb</i>	Sets the background color of a window.
<i>WTwindow_setprojection</i>	Sets symmetric, asymmetric, or general window projections.
<i>WTwindow_setviewangle</i>	Sets the window's horizontal view angle.
<i>WTwindow_setyonvalue</i>	Sets the window's yon clipping value.
<i>WTwindow_zoomviewpoint</i>	Moves the window's viewpoint so the entire scene is in view.

Scene Graph

<i>WTnode_load</i>	Loads a data file into the scene graph.
<i>WTnode_addchild</i>	Makes a node a child of the specified parent node.
<i>WTnode_getchild</i>	Retrieves the specified child of the specified parent node.
<i>WTnode_getparent</i>	Retrieves the specified parent of the specified child node.
<i>WTnode_numchildren</i>	Returns the number of children for a specified node.
<i>WTnode_remove</i>	Removes a node from all of its parent nodes.
<i>WTnode_boundingbox</i>	Highlights a node of the scene graph with a bounding box that is visible in the simulation.
<i>WTnode_getextents</i>	Obtains the extents of the scene graph subtree.
<i>WTnode_intersectnode</i>	Tests for intersection between parts of the scene graph.

Drawing

<i>WTwindow_draw2Dcircle</i>	Draws a 2D circle.
<i>WTwindow_draw2Drectangle</i>	Draws a 2D rectangle.
<i>WTwindow_draw2Dpoint</i>	Draws a 2D point.
<i>WTwindow_draw2Dline</i>	Draws a 2D line.
<i>WTwindow_draw3Dpoints</i>	Draws a 3D point.
<i>WTwindow_draw3Dlines</i>	Draws a 3D line.

User Interface

<i>WTuimessagebox_new</i>	Creates a message box.
<i>WTuiscrolledtext_new</i>	Creates a scrollable text box.
<i>WTuimenubar_new</i>	Creates a menu bar.
<i>WTui_dimitem</i>	Grays out a menu item.

Sound

<i>WTsound_load</i>	Loads a sound from a file.
<i>WTsound_play</i>	Plays the sound.
<i>WTsound_setnodepath</i>	Assigns the sound to a source.

The Universe

Introduction

The universe is the “container” of all WTK objects. These objects can include geometries, sensors, lights, viewpoints, serial ports, paths, or other object types. Once you create these objects, they are automatically maintained by the WTK simulation manager (see *Simulation Management* on page 2-5). While you can have multiple scene graphs (see the next chapter, *Scene Graphs*) in your universe simulation, there is only one WTK universe. As a result, unlike the methods for other WTK objects, universe methods do not require a pointer as the first argument.

This chapter describes WTK’s functions for constructing (or destroying) a universe, managing a simulation, specifying universe rendering styles, calculating an application’s performance, setting global parameters for WTK, and using a resource file to set parameters for your universe.

Universe Construction and Destruction

In a WTK application, you create the universe using the function *WTuniverse_new*. *WTuniverse_new* must be the first WTK call in your main program and must be called only once in an application. This function initializes the universe’s state and initializes the graphics device, configuring it for the output device with which the virtual world is to be viewed.

The universe is deleted using the *WTuniverse_delete* function. This function frees all of the objects in the universe, including those that have been removed from the simulation with the remove function appropriate for that object type, such as *WTnode_remove*. The *WTuniverse_delete* function also cleans up and closes the graphics hardware or WTK display.

WTuniverse_new

```
void WTuniverse_new(  
    int display_config,  
    int window_config);
```

This function initializes the universe's state and the graphics device used to view the simulation. The graphics device used to view the simulation may be a stereo head-mounted display, stereo shutter glasses, or (for a monoscopic view) simply your computer monitor. Other than for functions whose names begin with *WTinit_* (for example, *WTinit_defaults*), *WTuniverse_new* must be the first WTK call in your main program and must be called only once in an application.

Note: If using WTK's UI functionality, make the call to *WTuniverse_new* after the call to *WTui_init*, which is used for creating the top-level application shell.

WTuniverse_new also creates a viewpoint for the universe. In WTK there can be many viewpoints. The viewpoint created by *WTuniverse_new* is by default the viewpoint through which the simulation is displayed.

See also *WTviewpoint_new* on page 16-3 and *WTwindow_setviewpoint* on page 17-11.

The *display_config* parameter specifies the number of windows which are displayed. For information on stereoscopic viewing, see page 2-34.

The possible values for *display_config* are:

<i>WTDISPLAY_DEFAULT</i>	Creates a single window.
<i>WTDISPLAY_NOWINDOW</i>	No windows will be created. Sometimes useful when creating a GUI using Motif, MFC, or WTK's UI functions.
<i>WTDISPLAY_CRYSTALEYES</i>	For CrystalEyes glasses. Creates a single full screen stereo window which has no border.
<i>WTDISPLAY_STEREO</i>	Creates two windows. Should only be used by legacy code.
<i>WTDISPLAY_NEEDSTENCIL</i>	This constant can be combined with <i>WTDISPLAY_DEFAULT</i> or <i>WTDISPLAY_NOWINDOW</i> by using the bitwise OR operator (<code> </code>), to request the use of the stencil buffer on systems which contain

stencil buffer hardware. It should only be used if you want to use your system's stencil buffer hardware to obtain interlaced stereo in your window(s) or if your application uses the stencil buffer hardware for application specific purposes.

The values for *window_config* specify the characteristics of the window or windows created by *WTuniverse_new*. With *window_config*, you can specify host-system specific window parameters. Your Hardware Guide also describes how to configure your system if an NTSC (television) signal is required by your display device. (Many head-mounted displays require an NTSC signal.) For information on stereoscopic viewing, see page 2-34.

These are the possible values for *window_config*:

WTWINDOW_DEFAULT

Creates a window with no special attributes. The window has a border unless *WTWINDOW_NOBORDER* is used in combination with this constant (via the bitwise OR operator).

WTWINDOW_STEREO

Creates a stereo window on systems that have hardware support for stereo. On systems without hardware stereo support, this option will create 2 images in the window (one on the top with the left eye view, the other on the bottom with the right eye view). On Windows platforms, if this option is selected and the *WTDISPLAY_NEEDSTENCIL* option is selected in the *display_config* parameter, the behavior you will obtain is that of *WTWINDOW_STEREOVSPLIT*.

WTWINDOW_STEREOVSPLIT

This constant can be combined with the *WTWINDOW_STEREO* option by using the bitwise OR operator (`|`), to create 2 images in the window (one on the top with the left eye view, the other on the bottom with the right eye view) even if your system has hardware stereo support. In essence, this option will cause WTK to disable your system's stereo

	hardware and to create a “vertically split” stereo window instead.
<i>WTWINDOW_RBSTEREO</i>	Creates a window with red/blue stereo.
<i>WTWINDOW_INTERLACEEVENODD</i>	Creates an interlaced stereo window whose even numbered scanlines correspond to the left eye view and whose odd numbered scanlines correspond to the right eye view. This option requires that the <i>WTDISPLAY_NEEDSTENCIL</i> option be selected in the <i>display_config</i> parameter.
<i>WTWINDOW_INTERLACEODDEVEN</i>	Creates an interlaced stereo window whose odd numbered scanlines correspond to the left eye view and whose even numbered scanlines correspond to the right eye view. This option requires that the <i>WTDISPLAY_NEEDSTENCIL</i> option be selected in the <i>display_config</i> parameter..
<i>WTWINDOW_NOBORDER</i>	This constant can be combined with any of the above listed options by using the bitwise OR operator (), to create a window without a border.
<i>WTWINDOW_SCREENn</i>	Where n is a number from 0 to 8. In the multi-pipe/multi-processor version of WTK, this constant can be combined with any of the above listed options by using the bitwise OR operator (), to specify which screen the window is to be placed on.

If the *window_config* parameter is set to any of the stereo options (*WTWINDOW_STEREO*, *WTWINDOW_RBSTEREO*, *WTWINDOW_INTERLACEEVENODD*, or *WTWINDOW_INTERLACEODDEVEN*), you will need to adjust the viewpoint’s parallax and convergence values. See *WTviewpoint_setparallax* and *WTviewpoint_setconvergence*.

This is an example of calling *WTuniverse_new* to create a display appropriate for monoscopic, flat-screen viewing directly from the monitor:

```
WTuniverse_new(WTDISPLAY_DEFAULT, WTWINDOW_DEFAULT);
```

while the following is an example of calling *WTuniverse_new* to create a display with a stereo window which has no border:

```
WTuniverse_new(WTDISPLAY_DEFAULT, WTWINDOW_STEREO |  
               WTWINDOW_NOBORDER);
```

WTuniverse_delete

```
void WTuniverse_delete(  
    void);
```

This function frees all of the objects in the universe, including those that have been removed from the simulation with the remove function appropriate for that object type, such as *WTnode_remove*. *WTuniverse_delete* also cleans up and closes the graphics hardware or WTK display. This should be the last WTK call in your main program.

Simulation Management

The simulation loop is the heart of a WTK application. Every aspect of the simulation takes place in the universe. The simulation loop is entered by calling *WTuniverse_go* and is exited by calling *WTuniverse_stop*. Alternatively, you can use the function *WTuniverse_go1* to go through the simulation loop exactly once and then exit the loop automatically. Figure 2-1 shows the default order of events in the simulation loop. The order can be changed by using the function *WTuniverse_seteventorder*.

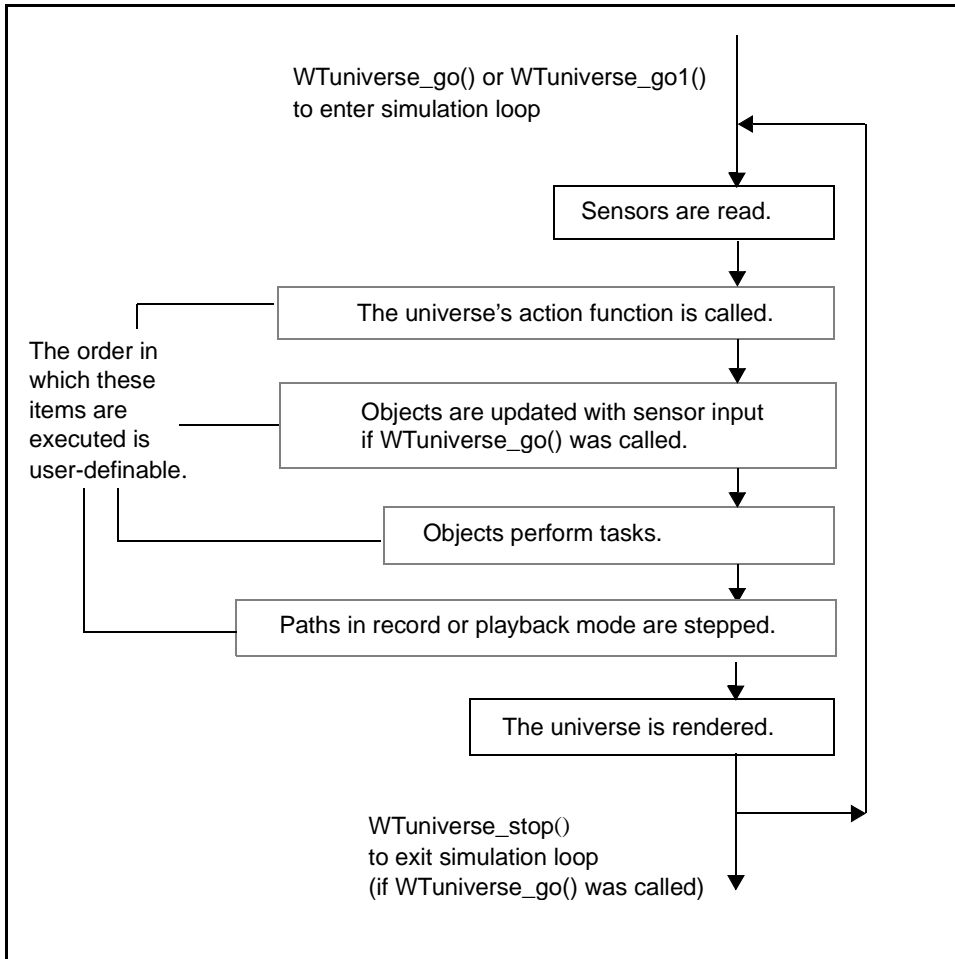


Figure 2-1: The default simulation loop

WTuniverse_ready

```
void WTuniverse_ready(  
    void);
```

This function prepares your application for entry into the main simulation loop. Call this function before starting the simulation for the first time (i.e., before the first call to either *WTuniverse_go* or *WTuniverse_go1*), but after all graphical entities have been created.

Subsequently, you also need to call *WTuniverse_ready* before re-entering the simulation loop if new graphical entities have been created or existing ones removed or deleted.

WTuniverse_go

```
void WTuniverse_go(  
    void);
```

This function starts the main simulation loop, and can only be called once. Control does not return to the statement following the call to *WTuniverse_go* until the *WTuniverse_stop* function is called. However, your application can gain control through a universe action function or through a task function. The universe has a user-specifiable action function (set by calling *WTuniverse_setactions* described on page 2-12), which is called before the rendering occurs for each frame. Individual objects can also have task functions, which are called for the object once per frame (see *WTtask_new* on page 11-2). The principle is similar to the “callback” or “event” functions typically provided by a window management system.

Note: You cannot call *WTuniverse_go* from the universe’s action function or from a task function.

Before calling *WTuniverse_go* for the first time you must call *WTuniverse_ready*. You should also call *WTuniverse_ready* before subsequent calls to *WTuniverse_go* if new graphical entities have been created since the last call to *WTuniverse_ready*. See the above description of *WTuniverse_ready*.

Note: If using WTK’s UI functionality to create a user interface, call *WTui_go* instead of *WTuniverse_go*.

WTuniverse_go1

```
void WTuniverse_go1(  
    void);
```

This function starts the main simulation loop for one loop only. This is useful for creating a splash screen, for example, which is displayed while a program is loading. It is not necessary to call *WTuniverse_stop* to exit the simulation loop when *WTuniverse_go1* is called.

An example is shown here:

```
main()
{
    WTuniverse_new();
    ....      /* code to load graphical entities */
    WTuniverse_ready();
    WTuniverse_go1();      /* draw 1 frame */
    ....      /* code to initialize simulation */
    WTuniverse_go();
    WTuniverse_delete();
}
```

Before calling *WTuniverse_go1* for the first time, *WTuniverse_ready* must be called. You should call *WTuniverse_ready* before subsequent calls to *WTuniverse_go1* if new graphical entities have been created since the last call to *WTuniverse_ready*. See the previous description of *WTuniverse_ready*.

Note that *WTuniverse_go1* is not reentrant; it must not be called from the universe's action function or from an object task function.

WTuniverse_stop

```
void WTuniverse_stop(
    void);
```

Call this function to exit the main simulation loop, which was entered by calling *WTuniverse_go*. When *WTuniverse_stop* is called, the simulation continues to the bottom of the loop and then exits. It does not exit mid-way through the loop.

Typically *WTuniverse_stop* is called from the universe's action function. The following is an example of such an action function, where *mouse* is a pointer to a sensor object created earlier in the application:

```

/* Exit the simulation loop if the left mouse button has been pressed. */
void actions()
{
    if ( WTsensor_getmiscdata(mouse) &
         WTMOUSE_LEFTBUTTON ) {
        WTuniverse_stop();
    }
}

```

WTuniverse_seteventorder

```

FLAG WTuniverse_seteventorder(
    short nevents,
    short *events);

```

This function allows you to change the order of activity in the simulation loop. When a WTK application is running, the simulation loop (as illustrated in figure 2-1 on page 2-6) is repeatedly executed. The function *WTuniverse_seteventorder* allows you to change the order of activity in the simulation loop from the default order shown.

There are four items in the simulation loop that you can rearrange. They are specified in the function *WTuniverse_seteventorder* using the following constants, (listed here in the default order in which the corresponding events occur in the simulation):

<i>WTEVENT_ACTIONS</i>	The user-defined universe action function is called.
<i>WTEVENT_OBJECTSENSOR</i>	Graphical objects and viewpoints are updated by the sensors attached to them.
<i>EVENT_TASKS</i>	Object task functions are called.
<i>WTEVENT_PATHS</i>	Paths in record or playback mode are stepped.

To change the order of events, define an array of shorts containing the constants in the desired order and pass it to *WTuniverse_seteventorder* as the *events* argument. The *nevents* argument should always be 4.

For example, you may use this function in an application where input from a sensor is used to move the viewpoint, while keeping the viewpoint within a certain room. To accomplish this, you would want to have the viewpoint moved with input from the sensor before it is tested for inclusion in the room. *WTEVENT_OBJECTSENSOR* is the constant corresponding to the viewpoint update with sensor input, and *WTEVENT_ACTIONS* is the constant corresponding to the test of the viewpoint location relative to the “room” geometry (assuming that a universe action function has been written to perform this test).

The following code fragment shows how to set this event order by calling *WTuniverse_seteventorder* so that the universe action function is called last:

```
short myevents[4];

/* set the order so the action function is last */
myevents[0] = WTEVENT_OBJECTSENSOR;
myevents[1] = WTEVENT_TASKS;
myevents[3] = WTEVENT_PATHS;
myevents[2] = WTEVENT_ACTIONS;
WTuniverse_seteventorder(4, myevents);
```

Each of the four constants must occur exactly once in the array passed in to *WTuniverse_seteventorder*. If a valid array is passed in, then the event order is set and the function returns TRUE. If an invalid array is passed in (for example, if one of the tokens occurs twice), then the event order is unaffected and the function returns FALSE.

WTuniverse_geteventorder

```
short *WTuniverse_geteventorder(
    void);
```

This function returns the order of events currently set to occur in the simulation loop. See the function *WTuniverse_seteventorder* for a description of the constants and their default order. The return value of this function is an array of shorts where:

array[0] : is the number of events in this array (4 in the current release), and
array[1] ... array[N] : are the N event tokens, where N is given in array[0]

Note: Do not modify the returned array, or the results may be undefined!

The following illustrates how to use this function:

```
short *events, event, i;
events = WTuniverse_geteventorder();
for (i = 0; i < events[0]; i++) {
    event = events[i+1];
    WTmessage("Event %d is ", i);
    if ( event==WTEVENT_OBJECTSENSOR )
        WTmessage("OBJECTSENSOR\n");
    if ( event==WTEVENT_TASKS )
        WTmessage("TASKS\n");
    if ( event==WTEVENT_ACTIONS )
        WTmessage("ACTIONS\n");
    if ( event==WTEVENT_PATHS )
        WTmessage("PATHS\n");
}
```

The Universe Action Function

You use the universe action function to define and control the activity in your simulations. Using the action function, you can specify actions involving any WTK objects, graphical or otherwise. The action function is a user-defined function that is called by the simulation manager each time through the simulation loop. Figure 2-1 on page 2-6 shows the order in which the action function is called with respect to the other events in the simulation loop. This order can be changed with the function *WTuniverse_seteventorder*.

Some examples of actions that might be specified in the universe action function are:

- Program termination by having a button press trigger a call to *WTuniverse_stop*.
- Simulation activities such as terrain-following, object manipulation, intersection testing, or others.
- Changes to rendering parameters such as lighting conditions or background color.
- Event handling for a user interface, for example, calling *WTwindow_pickpoly* to enable the user to interactively select a polygon, and specifying what is to be done with the selected polygon; processing keyboard input using the *WTkeyboard* functions.

Actions pertaining to a specific graphical object can be specified in the object's task function using *WTtask_new*.

WTuniverse_setactions

```
void WTuniverse_setactions(  
    void (*actionfn) (void));
```

This function sets the universe action function. An example of a simple but useful action function is the following, which tests whether **BUTTON1** of a Spaceball has been pressed, and calls *WTuniverse_stop* if so. This example assumes that a Spaceball sensor object has been previously constructed in the application.

```
WTsensor *spaceball;  
  
void myactions(void)  
{  
    /* stop by pressing the 1 button on the Spaceball. */  
    if ( WTsensor_getmiscdata(spaceball)  
        & WTSPACEBALL_BUTTON1 )  
        WTuniverse_stop();  
}
```

The universe action function is set for the example above by calling:

```
WTuniverse_setactions(myactions);
```

The Universe's Objects

The following functions provide access to the universe's lists of objects (for example, *WTuniverse_getsensors* returns a pointer to the sensor list). To iterate through this list, use the corresponding iterator function (such as *WTsensor_next*) to return the next object on the list.

WTuniverse_getsensors

```
WTsensor *WTuniverse_getsensors(  
    void);
```

This function returns a pointer to a list of all sensors currently in the universe. This list includes all sensors that have been constructed using the function *WTsensor_new* or one of the sensor macros, such as *WTspaceball_new*, but does not include any sensors deleted with *WTsensor_delete*. Use the function *WTsensor_next* to iterate through this list.

WTsensor_next

See *WTsensor_next* on page 13-10 for a description.

WTuniverse_getpaths

```
WTpath *WTuniverse_getpaths(  
    void);
```

This function returns a pointer to a list of all paths in the universe. You can use *WTpath_next* to iterate through the universe's list of paths.

WTpath_next

See *WTpath_next* on page 14-10 for a description.

WTuniverse_getwindows

```
WTwindow *WTuniverse_getwindows(  
    void);
```

This function returns a pointer to a list of all windows currently in the universe. You can then iterate through the list of existing windows using *WTwindow_next*. When new windows are created, they are added onto the end of the universe's list of windows, so the first window returned by *WTuniverse_getwindows* is the first window that was created (i.e., the window opened by the *WTuniverse_new* call, unless that window has been deleted with *WTwindow_delete*).

Consult your Hardware Guide for information about support for the WTK window class on your system.

WTwindow_next

See *WTwindow_next* on page 17-8 for a description.

WTuniverse_getcurrwindow

```
WTwindow *WTuniverse_getcurrwindow(  
    void);
```

This function returns a pointer to the window that has input focus. Normally, this is the WTK window in which the mouse cursor lies, or NULL if the cursor is not in a WTK window.

However, if the mouse is moved out of a window while a mouse button is held down, the window will still retain input focus. See also *WTwindow_getidx* on page 17-29.

WTuniverse_getcurrwinidx

```
WTuiwinidtype WTuniverse_getcurrwinidx(  
    void);
```

This function returns the system-specific window ID of the window that has input focus. The return value's type is host-system specific; on UNIX platforms the return type is Widget, while on the Windows platform, the return type is HWND.

WTuniverse_getcurrscridx

```
int WTuniverse_getcurrscridx(  
    void);
```

This function returns the number of the screen that has input focus.

WTuniverse_getviewpoints

```
WTviewpoint *WTuniverse_getviewpoints(  
    void);
```

This function returns a pointer to a list of all viewpoints in the universe. When the universe is created with *WTuniverse_new*, a viewpoint is automatically created and is by default the first viewpoint returned by *WTuniverse_getviewpoints*. You can then iterate through the list of existing viewpoints using *WTviewpoint_next*.

If VRML files have viewpoint information, new viewpoints are created and added to the beginning of the list of viewpoints. Thus, once you load a VRML file, a call to *WTuniverse_getviewpoints* will not return the same viewpoint as it would have before the VRML file was loaded. To associate a sensor with a viewpoint, you would usually call *WTuniverse_getviewpoints* to get a pointer to the viewpoint associated with the window. If a VRML file with viewpoints was just loaded, *WTuniverse_getviewpoints* does not return the viewpoint associated with the window. That is why you have to get a pointer to the window's viewpoint before you load a file.

WTviewpoint_next

See *WTviewpoint_next* on page 16-5 for a description.

WTuniverse_setviewpoint

```
void WTuniverse_setviewpoint(  
    WTviewpoint *viewpoint);
```

Some WTK functions make use of the concept of a “current” viewpoint. This function allows you to designate a particular viewpoint as the current viewpoint.

Note: *WTK won't delete the viewpoint if it is the universe's current viewpoint. You can, however, delete any other viewpoint.*

In the following example, a new viewpoint is created with the same position as the viewpoint initially constructed for the universe, but pointing in the opposite direction. It is then set to be the current viewpoint.

```
WTviewpoint *newview; /*new viewpoint for rotated view */

/*make a new viewpoint by copying the universe's initial viewpoint */
newview = WTviewpoint_copy(WTuniverse_getviewpoints());

/*rotate the viewpoint to point in opposite direction */
WTviewpoint_rotate(newview, Y, PI, WTFRAME_VPOINT);

/*finally, switch to this new viewpoint */
WTuniverse_setviewpoint(newview);
```

WTuniverse_getinitialview

```
void WTuniverse_getinitialview(
    WTPq *position);
```

When *WTnode_load* reads in a model from a DXF or NFF file, it reads in and saves the viewpoint information contained in the file. *WTuniverse_getinitialview* extracts the saved viewpoint position and orientation from the most recently loaded model and places it in *pq*. The initial viewpoint location is useful for resetting the viewpoint to a specific start-up location, like when the end-user has moved around enough to become lost with respect to the model.

Here's how to set the viewpoint to the stored location, so that you can return to where you started:

```
/* get the stored viewpoint from the most recently loaded universe */

WTPq initialview;
WTuniverse_getinitialview(&initialview);

/* move the current viewpoint to the location read in from the model */
WTviewpoint_moveto(WTuniverse_getviewpoints(), &initialview);
```

See also *WTnode_load* on and page 4-46 and *WTnode_save* on page 4-48.

WTuniverse_getrootnodes

```
WTnode *WTuniverse_getrootnodes(  
    void);
```

This function returns a pointer to the first root node in the universe's list of root nodes. You can use *WTrootnode_next* to iterate through the universe's list of root nodes.

WTrootnode_next

See *WTrootnode_next* on page 4-77 for a description.

WTuniverse_getmotionlinks

```
WTmotionlink *WTuniverse_getmotionlinks(  
    void);
```

This function returns a pointer to the first motion link in the universe's list of motion links. You can use *WTmotionlink_next* to iterate through the universe's list of motion links.

WTmotionlink_next

See *WTmotionlink_next* on page 15-8 for a description.

WTuniverse_deletelink

```
void WTuniverse_deletelink(  
    void *source,  
    void *target);
```

This function deletes any motion link connecting the indicated source and target objects from the universe's list of motion links. All memory used by the motion link is released.

Global Rendering Parameters

The functions in this section pertain to WTK rendering parameters and display options. Please consult your Hardware Guide for system-specific information on these subjects.

Rendering Options

The rendering style applied to a geometry is a combination of the universe's rendering style and the geometry's rendering style. For example, if you set the universe's rendering style to smooth shaded and textured, but set one geometry's rendering style to wireframe, then all of the objects in the universe will be rendered smooth shaded and textured except for the one specified geometry, which is rendered in wireframe mode. For more information on rendering styles for geometries, see page 6-33.

WTuniverse_setrendering

```
void WTuniverse_setrendering(  
    FLAG style);
```

This function specifies the universe's rendering style. The *style* argument is a bitmask that allows you to set several different rendering flags simultaneously. The default rendering style for the universe is lighting enabled, smooth shaded and texturing enabled (i.e., the *style* flag is set to `WTRENDER_LIGHTING | WTRENDER_SMOOTH | WTRENDER_TEXTURED`).

Note: `WTuniverse_setrendering` has no effect on a prebuilt geometry, since you cannot change the rendering style of prebuilt geometry.

The following are valid styles:

`WTRENDER_ANTIALIAS`

Enables anti-aliasing. Note that solid fill polygon anti-aliasing is only available on SGI RE systems that have at least two raster managers. Wireframe anti-aliasing is available when using the OpenGL version of WTK (i.e., `WTRENDER_ANTIALIAS | WTRENDER_WIREFRAME`).

<i>WTRENDER_BEST</i>	Enables lighting, smooth shading, textures, perspective texturing, and anti-aliasing (i.e., <i>WTRENDER_LIGHTING WTRENDER_SMOOTH WTRENDER_TEXTURED WTRENDER_PERSPECTIVE WTRENDER_ANTIALIAS</i>).
<i>WTRENDER_GOURAUD</i>	Enables gouraud shading and lighting (this is an outdated style from WTK 2.1, see note below).
<i>WTRENDER_LIGHTING</i>	Turns on lighting.
<i>WTRENDER_NOSHADE</i>	Disables shading, lighting, and texturing.
<i>WTRENDER_PERSPECTIVE</i>	Enables perspective correct texturing.
<i>WTRENDER_SMOOTH</i>	Enables smooth shading (gouraud shading).
<i>WTRENDER_TEXTURED</i>	Enables texturing.
<i>WTRENDER_WIREFRAME</i>	Enables wireframe mode. Note that when WTK renders in wireframe mode, all texturing and lighting is ignored, and the lines are drawn in a solid color based on the diffuse material components of the vertices.

For example, if the style parameter is set to *WTRENDER_WIREFRAME*, all of the geometries in the universe will be rendered as wireframe entities, instead of solid entities (with *WTRENDER_WIREFRAME*, the only additional rendering option available is *WTRENDER_ANTIALIAS*, which causes the wireframe image to be anti-aliased).

Note: *WTRENDER_GOURAUD* is an outdated WTK 2.1 style that has been replaced with *WTRENDER_LIGHTING* and *WTRENDER_SMOOTH*.

In many cases you will want to simply change one of the rendering flags while leaving the current set active. For example, if you want to turn texturing off while leaving the rest of the currently active rendering flags on, you would do something like this:

```

FLAG style;
/* get the current rendering flags */
style = WTuniverse_getrendering();
/* now turn off the texturing flag, leaving the rest alone */
style = style & ~(WTRENDER_TEXTURED);

```

```
/* now pass the modified flag set back to WTK */  
WTuniverse_setrendering(style);
```

WTuniverse_getrendering

```
FLAG WTuniverse_getrendering(  
    void);
```

This function returns the current value of the universe rendering style. See the *WTuniverse_setrendering* function above, for possible return values.

Other Global Functions

WTscreen_setyblank

```
void WTscreen_setyblank(  
    int distance);
```

This function allows you to adjust the vertical blanking interval between the left and right eye images, which are stacked vertically on the display. This interval is measured in pixels, and appears as a solid bar between the upper and lower images. Use *WTscreen_setyblank* for certain hardware platforms when a field-sequential viewing device is being used and you are using the display option *WTDISPLAY_CRYSTALEYES*.

If you are using one of these devices and experience a rolling vertical sync problem or problem with the vertical alignment of the left and right eye images, adjust this value until the problem disappears. Examine your Hardware Guide for platform-specific information about this feature.

It is often useful to be able to interactively adjust the vertical blanking interval one pixel at a time, until the correct value is found. Increasing the vertical blanking interval value by one is accomplished by calling:

```
WTscreen_setyblank(WTscreen_getyblank() + 1);
```

For more information about creating a display appropriate for CrystalEyes or the BOOM2C, see your WTK Hardware Guide. See the *Sensors* chapter for information about the BOOM (page 13-55) and CrystalEyesVR (page 13-108) as serial port devices.

WTscreen_getyblank

```
int WTscreen_getyblank(  
    void);
```

This function returns the current value of the screen blanking interval used for field-sequential devices as described under *WTscreen_setyblank*. Check your Hardware Guide to see if your platform supports this feature.

WTuniverse_setbboxrgb

```
void WTuniverse_setbboxrgb(  
    float r,  
    float g,  
    float b);
```

This function sets the color of all active bounding boxes in the universe. The default color is white.

WTuniverse_setbgrgb

```
void WTuniverse_setbgrgb(  
    unsigned char r,  
    unsigned char g,  
    unsigned char b);
```

This function sets the background color of the universe (0 to 255 are valid rgb values). The default color is blue (0, 0, 255).

WTuniverse_getbgrgb

```
void WTuniverse_getbgrgb(  
    unsigned char *r,  
    unsigned char *g,  
    unsigned char *b);
```

This function returns the current background color of the universe.

WTuniverse_setsubfaceoffset

```
void WTuniverse_setsubfaceoffset(  
    float val);
```

This function sets the distance by which a subface is offset from its parent polygon. The default offset value is 0.65, which seems to work well for many models. For more information, see *Subfaces in MultiGen/ModelGen* on page 6-7.

Note: The subface offset set through this function is only applicable to models loaded from MultiGen .FLT files.

WTuniverse_getsubfaceoffset

```
float WTuniverse_getsubfaceoffset(  
    void);
```

This function returns the value of the subface offset.

Performance and Timer Functions

For optimizing performance, it is often useful for applications to know how fast the simulation is running. Although the speed could easily be checked outside of WTK by making calls to system timer functions, the need for such functions is common enough that they are provided as part of the WTK library.

When *WTuniverse_new* is called, the universe clock starts and WTK provides access to the following simulation statistics: simulation time, frame count, and average frame rate.

WTuniverse_time

```
float WTuniverse_time(  
    void);
```

This function returns the number of seconds the simulation has been running since the last time *WTuniverse_new* or *WTuniverse_resetime* was called.

WTuniverse_resetime

```
void WTuniverse_resetime(  
    void);
```

This function resets the universe time as if the simulation had just started.

WTuniverse_framecount

```
int WTuniverse_framecount(  
    void);
```

This function returns the number of frames drawn since the last time *WTuniverse_new* or *WTuniverse_resetframecount* was called.

WTuniverse_resetframecount

```
void WTuniverse_resetframecount(  
    void);
```

This function resets the universe frame count as if the simulation had just started.

WTuniverse_framerate

```
float WTuniverse_framerate(  
    void);
```

This function returns the number of frames per second at which the simulation is currently running. The number returned is actually a running average of the frame rate of the preceding 30 frames, in an attempt to stabilize the reading to at least one decimal digit. Therefore, you should wait at least 30 frames prior to accessing this function to obtain a meaningful result.

WTuniverse_avgframerate

```
float WTuniverse_avgframerate(  
    int samples);
```

This function returns the number of frames per second, averaged over a user-specified number of updates. Currently the maximum number of samples is 30. Passing in a sampling value of less than zero or greater than 30 will return -1. Passing in 30 is the same as calling *WTuniverse_framerate*.

Universe Options

WTuniverse_setoption

```
void WTuniverse_setoption(  
    int option,  
    int value);
```

This function sets certain global parameters. It can be called at any time after *WTuniverse_new* has been called. However, for the option to have effect, *WTuniverse_setoption* must be called before calling the WTK function that the option will affect.

The following options, allowed values, and default values (shown in parentheses) are currently supported:

WTOPTION_3DSCHGTEXEXT

This option pertains only to the reading of 3D Studio files when a file references a texture whose name ends in “.gif”, “.tif”, or “.cel”. These formats are unsupported in WTK, so if *WTOPTION_3DSCHGTEXEXT* is set to TRUE, then the texture extension is automatically changed to “.rgb” for UNIX platforms or “.tga” for Windows platforms. Note that “.jpg” extensions are no longer automatically changed. TRUE/FALSE (FALSE)

<i>WTOPTION_MGENREADVCOLOR</i>	This option reads vertex colors when loading MultiGen/ModelGen .flt files. (Sometimes vertex colors are computed and saved out with .flt files which are not needed when using the file with another program.) TRUE/FALSE (FALSE)
<i>WTOPTION_NFFWRITE12</i>	This option writes polygon colors out in 12-bit format rather than 24-bit when NFF files are written out. (Note that vertex colors are always written out in 24-bits.) TRUE/FALSE (FALSE)
<i>WTOPTION_NFFWRITEUV</i>	This option writes texture uv information out (as part of the vertex description) when NFF files are written out. (See <i>Appendix E, WTK Neutral File Format</i> , for information about how uv values are stored in the NFF file.) TRUE/FALSE (FALSE)
<i>WTOPTION_NFFWRITEV21</i>	This option writes model files out in version 2.1 NFF format. It is only used with <i>WObject_save</i> , since there was no <i>WTgeometry</i> entity in WTK 2.1. When this option is set to FALSE, the new version 3.0 NFF is used when writing and separate .mat files are written which contain material information. TRUE/FALSE (FALSE)
<i>WTOPTION_OLD3DS</i>	This option loads 3D Studio files using the object and texture orientation used in Version 2.0. TRUE/FALSE (FALSE)
<i>WTOPTION_OLDTEXTROT</i>	This option pertains to non-SGI systems only. In Version 2.1 Beta and earlier releases of WTK on non-SGI platforms, <i>Wtpoly_rotatetexture</i> (and the <i>rot</i> parameter in the NFF file) rotated the texture opposite to the direction indicated under <i>Wtpoly_rotatetexture</i> . This was corrected in Version 2.1. To obtain the previous (incorrect) behavior, set this option to TRUE. TRUE/FALSE (FALSE)

<i>WTOPTION_OLDWFRONT</i>	This option loads in Wavefront files using the object orientation used in Version 2.0. TRUE/FALSE (FALSE)
<i>WTOPTION_VERTWARN</i>	This option produces a warning when vertices not referenced by a geometry object are found and discarded. TRUE/FALSE (FALSE)
<i>WTOPTION_USEWTPUMP</i>	This option pertains to windows applications. If it is FALSE, WTK stops processing events, and it is up to the application to process events. TRUE/FALSE (TRUE)
<i>WTOPTION_XFORMSCALE</i>	This option causes WTK to use the scaling factors (if any) contained in transform and movable nodes. Normally, WTK ignores scaling factors in the transformations contained in transform and movable nodes due to some severe side effects. If this option is enabled so that scaling factors are incorporated into WTK's computations, it is likely that intersection tests and math functions pertaining to matrices will operate incorrectly.
<i>WTOPTION_NOPOSTQUIT</i>	This option must be set if you want to incorporate WTK in a Netscape or ActiveX plug-in. This option prevents WTK from automatically shutting down when WTK's last rendering window is closed. Since a Netscape or ActiveX plug-in may require a WTK window at certain times and not others, you must set this option so that a kill signal is not sent to your plug-in when WTK is not active at a particular time.
<i>WTOPTION_NOAUTOALPHA</i>	Causes the alpha value of texture elements (texels) of textures which do not contain alpha values to be set to 255 (completely opaque). If this option is not set, texels whose R, G, and B values are equal to 0, i.e. those texels which are colored black, will have their

alpha value set to 0 (completely transparent) while pixels which are not completely black will have their alpha value set to 255. By setting this option, you can prevent WTK from assigning black texels an alpha value of 0. See *Wtpoly_settexture*.

WTOPTION_NEWMGENREAD

This option allows you to select the method used to read in MultiGen .flt files. If this option is set to TRUE, WTK will use MultiGen's Read/Write API to read in the .flt file. By using the MultiGen Read/Write API, WTK can read in even the newest versions of .flt files (including v15.x). If this option is set to FALSE, WTK will use an older reader which is only capable of reading in .flt files up to v14.2. For backward compatibility with WTK R8, set this option to FALSE. Valid values: TRUE/FALSE (FALSE).

In the following example, the writing out of texture uv information to NFF and binary NFF files is enabled:

```
WTuniverse_setoption(WTOPTION_NFFWRITEUV, TRUE);
```

WTuniverse_getoption

```
int WTuniverse_getoption(  
    int option);
```

This function returns the value of the specified option. The *option* parameter can be any of the following:

```
WTOPTION_3DSCHGTEXEXT  
WTOPTION_MGENREADVCOLOR  
WTOPTION_NFFWRITE12  
WTOPTION_NFFWRITEUV  
WTOPTION_NFFWRITEV21  
WTOPTION_OLD3DS
```

WTOPTION_OLDTEXTROT
WTOPTION_OLDWFRONT
WTOPTION_VERTWARN
WTOPTION_USEWTPUMP
WTOPTION_XFORMSCALE
WTOPTION_NOPOSTQUIT
WTOPTION_NOAUTOALPHA
WTOPTION_NEWMGENREAD

If the *option* parameter is invalid, this function will return -1.

Resource Files

WTK provides the ability to set certain parameters from a file when your application starts up. For example, you can specify background color, viewing angle, window size and window position this way. On UNIX platforms, you do this with X Resources. For other platforms, consult your Hardware Guide.

To use this capability, follow these steps:

1. Add the desired parameters to the appropriate X Resource file.
2. Register those resources with the X Resources Database using `xrdb`. For example, if using the `.Xdefaults` file, when adding new resource values to the file, use:

```
xrdb -merge ~myuserid/.Xdefaults
```

To find out what values are currently in your X Resource Database, you can type:

```
xrdb -query
```

3. Call `WTinit_defaults` before calling `WTuniverse_new`.

Each of these steps is examined more closely, in the following sections.

The Resource Hierarchy

This is the order in which WTK processes resource files:

1. `/usr/lib/X11/app-defaults/Wtk`
(except for SUN which uses `/usr/openwin/lib/app-defaults/Wtk`)
2. `$HOME/.Xdefaults`
3. File specified by `XENVIRONMENT` environment variable
4. `$HOME/app-defaults/Wtk`
5. `$HOME/app-defaults/<app-name>`

where `app-name` is the name of the application executable, with its first letter capitalized. For example, if your application is named `Kitchen`, then the file processed is named `Kitchen`. If the `-name` option is specified on the command line, the next parameter following this option is used instead of the application name.

6. Finally, command line arguments used when you run your application will override X Resource values obtained from any of the above files. Use of command line arguments is described below under *Specifying Parameters on the Command Line* on page 2-31.

When resource specifications are made in more than one of the files above, the last file processed takes precedence over previously processed files.

The resource database “class” chosen for WTK resources is `Wtk` while the “name” is `wtk`.

Choosing an Appropriate Resource File

It is recommended that you create files in the application defaults directory that will allow different resource definitions for different applications. If you use the `.Xdefaults` file, then it is recommended that you use the `Wtk` class instead of the `wtk` name.

WTK Parameters Specified in a Resource File

The following table describes the WTK parameters specified in a resource file, and indicates which parameters take boolean values (TRUE or FALSE):

Parameter	Boolean Values (i.e., TRUE/FALSE)	Description
<i>bgcolor</i>		Universe background color (default: 0x0000ff)
<i>ambient</i>		Ambient light intensity (default: 0.4)
<i>ambientrgb</i>		Ambient light color (default: 0xffffff)
<i>geometry</i>		Window size and placement (default: system-dependent)
<i>fov</i>		Total horizontal view angle, in degrees (default: 80.0)
<i>hither</i>		Hither clipping value (default: 1.0)
<i>yon</i>		Yon clipping value (default: system-dependent)
<i>border</i>	X	Whether initial windows have a border (default: TRUE)
<i>coplanar</i>	X	Whether coplanarity testing is on when models are loaded (default: TRUE)
<i>write12</i>	X	Write out NFF files using 12-bit polygon color (default: FALSE)
<i>writeuv</i>	X	Write out NFF using texture uv coordinate values, rather than texture rotation, translation, scale, and mirror values (default: FALSE)
<i>old3ds</i>	X	3D Studio objects and textures load in as they did in Version 2.0 (upside down) (default: FALSE)
<i>oldwfront</i>	X	Wavefront objects load in as they did in Version 2.0 (default: FALSE)

HOW TO SPECIFY THESE PARAMETERS

The following examples show how to specify these parameters in an X Resource file:

Wtk.fov: 40 /* specifies a default field of view of 40 degrees */
Wtk.coplanar: True /* enables coplanarity testing */
Wtk.border: False /* does not display window border */
Wtk.writeuv: True /* Enables writing of uv values in nff files */
Wtk.write12: True /* Enables write of 12-bit color value in nff */
Wtk.geometry: 640x480+0-0 /* places a 640 by 480 window in bottom-left
 corner of screen */
Wtk.bgcolor: 0xff0000 /* specifies a red background color */
Wtk.ambient: 0.5 /* specifies an ambient intensity of 0.5 */
Wtk.hither: 2.0 /* specifies a default hither value of 2.0 */
Wtk.screen: 1 /* specifies default screen to be screen # 1 */
Wtk.ambientrgb: 0x0000ff /* specifies an ambient color of blue */

Note: Do not place the comments in the above examples into the resource file. To put comments into a resource file, begin a line with an exclamation mark “!”—the line is then considered to be a comment.

SPECIFYING PARAMETERS ON THE COMMAND LINE

In addition to the values specified in the resource files, you can also use the command line to specify the display and the resource file (using the *name* option). For example:

```
wtk -fov 60 -border -hither 2.0 -name xyz
```

This runs the WTK application with an fov of 60 degrees, no border, and a hither value of 2.0, using the resource file *\$HOME/app-defaults/xyz* instead of *\$HOME/app-defaults/Wtk*.

Note: When specifying parameters on the command line, the full resource name must be used. WTK does not support abbreviations.

Telling WTK to Use Resource Information

To have WTK use the resource information, simply call *WTinit_defaults* before calling *WTuniverse_new*. Make this call before the call to *scan_args* in the demos since it removes the X resource arguments.

To use the WTK's support for X Resources, you must call *WTinit_defaults* before calling *WTuniverse_new*.

For example:

```
int main(int argc, char **argv)
{
    /* initialize WTK application defaults. NOTE the use of "&" before argc */
    WTinit_defaults(&argc, argv);

    /* Call scan args fn for this demo */
    scan_args(argc, argv);

    /* Now call WTuniverse_new */
    WTuniverse_new(WT....., WT.....);

    /* ..... */
}

/* scan_args fn, for example, as provided in many WTK demo programs. */
void scan_args(int argc, char **argv)
{
    /* ..... */
}
```

WTinit_defaults

```
FLAG WTinit_defaults(
    int *argc,
    char **argv);
```

This function creates an X Resource database that overrides *WTuniverse* structure values (UNIX only). Given an X display and the command line arguments, this function creates

an X Resource Database, then looks through it for program option values. Usually, all option values modify information in the *WTuniverse* structure. Thus, X Resources processed here override *WTuniverse* structure values set via function calls in the main application before calling this function. Precedence for default option values (lowest to highest) is as follows:

1. Function calls in the main application before calls to this function.
2. X Resources set from resource file.
3. Command line arguments.

WTinit_setmodels

```
void WTinit_setmodels(  
    const char *paths);
```

This function sets the path to the models directories, so that WTK functions that read in geometry, light and/or sound files will search for the file in the specified directory path. This function is an embeddable alternative to using the `WTMODELS` environment variable. Refer to your system-specific Hardware Guide for more information about environment variables.

WTinit_setimages

```
void WTinit_setimages(  
    const char *paths);
```

This function sets the path to the images directories, so that WTK functions that read in image, texture, and/or bitmap files will search for the file in the specified directory path. This function is an embeddable alternative to using the `WTIMAGES` environment variable. Refer to your system-specific Hardware Guide for more information about environment variables.

Modes of Stereoscopic Viewing

Depending on the graphics hardware installed in your computer, you may find one or more of the following methods useful to generate a stereo image. You may find it helpful to have at hand the vendor's manual that describes your hardware.

To display a stereoscopic effect, the software must render two images – one as seen from the left eye, and the second as seen from the right eye. There are essentially three different ways in which these two images can be displayed:

- Render the full images of both eyes into one single window.
- Divide the display into two along a horizontal axis and render the left eye image in the top part of the display and the right eye image in the bottom part of the display.
- Interleave the left and right eye images as alternate scan lines on a display.

These three stereoscopic methods are described below in more detail.

Field Sequential Mode

This is also known as quad-buffering, since it requires the graphics hardware to have quad-buffers – left, right, front and back buffers. This means it has sufficient memory and performance capabilities to render two full views (the left and right eye images) and then swap the images at 120Hz to generate a field sequential view at 60Hz. Both the images are thus drawn onto a single display, which is why this mode is called 'stereo in a window'. The monitor should be capable of supporting a 120Hz update frequency. However, since this mode uses twice as much frame buffer memory, you may have to lower your screen resolution.

You can turn on this option by passing in `WTDISPLAY_DEFAULT` and `WTWINDOW_STEREO` as arguments to the `WTuniverse_new` call. WTK will determine whether your graphics hardware is indeed capable of supporting this mode. If it is supported, a single window will be created into which both eye views will be drawn. If not, WTK will default to the *Over/Under Mode* as described in the next section.

This mode requires the graphics hardware to have an emitter signal to synchronize the swapping with the LCD displays. This mode will work with most LCD shutter systems as

long as the emitter signal is compatible. Your graphics card should have the capability to plug in an emitter box that sends the sync signal.

The advantages of using this mode are:

- you obtain stereo in a window
- there is no loss of vertical resolution
- if you are using any GUI, there is no distortion

The stereoscopic hardware known to support this mode are Stereographics, NuVision and most LCD shutter systems. The graphics hardware Intergraph Z13/Z25 as well as most SGI and SUN systems support this mode.

Over/Under Mode

This is another form of the field sequential mode and hence there is some confusion about the terms used. If the graphics board is incapable of supporting quad-buffers (four buffers), the display is divided into two parts along the horizontal axis, to have two borderless viewports within one borderless window. The left eye image is drawn into the top half and the bottom eye image is drawn into the bottom half.

This mode also requires a monitor that supports a 120Hz update frequency. Moreover, if the graphics board is incapable of generating a 120Hz vertical sync signal, a special adapter box is used to double the vertical frequency to 120Hz (in this case the graphics board is set to 60Hz vertical sync). This adapter box is called a vertical sync doubling box. The video signal is passed through this box before it is fed to the monitor. The top and bottom images are merged into one by the adapter box.

You can force this option to be activated by using the `WTWINDOW_STEREOVSPLIT` as the window configuration flag in the call to *WTuniverse_new*. When you use this flag, you are forcing a vertical split in the display to generate the over/under images, even if your system is capable of supporting stereo in a window.

This mode works with any graphics system. The Diamond 4000 is a special case because Evans & Sutherland has provided the ability to generate the 120Hz vertical signal from their graphics board. You don't need the sync doubling box in this case.

The stereoscopic hardware known to support this mode are Stereographics, NuVision and most LCD shutter systems.

Interlaced Mode

The interlaced mode interleaves the left and right eye images as alternate scan lines in a single window. All the even scan lines belong to the left eye image and all the odd scan lines belong to the right eye image (or vice-versa). There are two distinct approaches to the interlaced mode – stencil interlaced and hardware interlaced. WTK supports both these approaches.

STENCIL INTERLACED

This is a method for supporting interlaced displays through stencils. However, your graphics board must support stencils for this to work. You must inform WTK that you want to use the hardware stencils via the display option `WTDISPLAY_NEEDSTENCIL`.

You may use the window options `WTWINDOW_INTERLACEEVENODD` (or `WTWINDOW_INTERLACEODDEVEN`) to inform WTK to draw the left and right eye images as even and odd scan lines respectively (or vice-versa).

The advantages to this mode are

- you obtain stereo in a window
- it supports many different LCD glasses
- it works with a 60Hz monitor (and graphics board).

However, not all graphics boards support stencils. There is also a performance impact because of the use of stencils and the vertical resolution is halved because of the alternating scan lines.

The stereoscopic hardware known to support this mode are VREX, Virtual i-O i-glasses!, and many cheap LCD solutions.

The graphics hardware known to support this mode are the Diamond 4000, the Intergraph Z25, 3D Labs' GLiNT TX/MX designs, most SGI systems and some SUN systems.

HARDWARE INTERLACED

This mode of interlaced display is controlled entirely by the hardware. Intergraph supports two methods of hardware-interlacing their displays. Their modes are called "Interlaced" and "Hardware Interlaced". You can select these from the driver settings. Support for these modes have varied with versions of Intergraph drivers. The latest versions support these modes.

The "Interlaced" mode forces the hardware to generate an interlaced display at 120Hz. The main reason for doing this is to support the 120Hz display devices.

The stereoscopic hardware known to support Intergraph's interlaced mode are Stereographics, and 120Hz LCD shutters. The Intergraph Z13/Z25 boards support this mode.

The "Hardware Interlaced" mode lets you set the vertical sync frequency to something other than 120Hz. This is necessary in order to work with 60Hz interlaced stereo devices like VREX and Virtual i-O i-glasses!.

The stereoscopic hardware known to support Intergraph's hardware interlaced mode are VREX, Virtual i-O i-glasses!, and 60Hz interlaced devices. The Intergraph GLZ1T/Z13/Z25 boards support this mode.

Object/Property/Event Architecture

Overview

WorldToolKit (WTK) Release 8 has been enhanced through the addition of an Object/Property/Event (OPE) architecture. This new architecture provides you with the following capabilities:

- Treat most WTK object types as generic (or *base*) objects, which can all be stored, manipulated, and retrieved in a uniform manner using certain `WTbase_*` functions. (See page 3-2 for a list of WTK object types supported by the new architecture.)
- Create your own properties for objects, in which to easily store user-defined data. This provides a convenient alternative to the `setdata` and `getdata` functions for each object type.
- Trigger reactions to property changes for both user-defined and pre-defined properties (see page 3-3 for a list of the WTK pre-defined properties.). A property change is known as an *event*, and the optional reaction that is triggered in response to an event is controlled by the property's *event handler(s)*.
- Share properties, allowing you to create multi-user simulations to be used with Sense8's World2World product. If you have not purchased World2World, contact Sense8 to learn more about this client/server networking solution.

The OPE architecture can simplify many of the programming tasks encountered by WTK programmers and represents an alternative programming paradigm to the one described in the WorldToolKit Reference Manual. If you are developing multi-user simulations that connect to Sense8's World2World servers, use the OPE programming paradigm described here. Otherwise, you can use either programming paradigm.

For new WTK applications, we recommend that you make use of the OPE architecture programming paradigm, for the following reasons.

- It is easier to associate user-defined data with objects.
- The event-based architecture corresponds more closely with other modern event-based programming paradigms.
- Should you decide to extend your simulation to be used with World2World as a multi-user simulation, you will save development time if the application has already been written using the OPE paradigm.

Supported Types and Supplied Properties

The OPE architecture supports the following WTK object types:

- WTnode
- WTviewpoint
- WTwindow
- WTsensord
- WTpath
- WTbase

Note: WTbase is a new object type that you can use to create generic, empty objects distinguished only by the properties that you add to them. Use the WTbase object type when you want to create properties for unsupported object types (such as WTgeometry or WTpoly), or when you want to create an object that does not suit the characteristics of any of the WTK supplied object types. For more information on the WTbase object type, see page 3-7.

The tables below list the pre-defined properties supplied by WTK for each of the supported object types.

WTnode Properties

WTnode Properties	Data Type
WTANCHOR_LOCATION	WTSTRING
WTFOG_COLOR	WTQ (R,G,B,A)
WTFOG_LINEARSTART	WTFLOAT
WTFOG_MODE	WTINT
WTFOG_RANGE	WTFLOAT
WTINLINE_LOCATION	WTSTRING
WTLIGHT_AMBIENT	WTP3
WTLIGHT_ANGLE	WTFLOAT
WTLIGHT_ATTENUATION	WTP3
WTLIGHT_DIFFUSE	WTP3
WTLIGHT_DIRECTION	WTP3
WTLIGHT_EXPONENT	WTFLOAT
WTLIGHT_INTENSITY	WTFLOAT
WTLIGHT_POSITION	WTP3
WTLIGHT_SPECULAR	WTP3
WTLOD_CENTER	WTP3
WTLOD_RANGE	WTSTRING (form of "range;range;range")
WTMOVNODE_ATTACHMENTS	WTSTRING (form of "name;name;name")
WTNODE_BOUNDINGBOX	WTINT
WTNODE_CHILDREN	WTSTRING (form of "name;name;name")
WTNODE_ENABLED	WTINT
WTNODE_ROTATION	WTQ
WTNODE_TRANSLATION	WTP3

WTnode Properties	Data Type
--------------------------	------------------

WTSEP_CULLMODE	WTINT
WTSWITCH_WHICHCHILD	WTINT

WTviewpoint Properties

WTviewpoint Properties	Data Type
-------------------------------	------------------

WTVIEWPOINT_ASPECT	WTFLOAT
WTVIEWPOINT_CONVDISTANCE	WTFLOAT
WTVIEWPOINT_CONVERGENCE	WTINT
WTVIEWPOINT_ORIENTATION	WTQ
WTVIEWPOINT_PARALLAX	WTFLOAT
WTVIEWPOINT_POSITION	WTP3

WTwindow Properties

WTwindow Properties	Data Type
----------------------------	------------------

WTWINDOW_BGRGB	WTP3
WTWINDOW_BOTTOMRIGHT	WTP2
WTWINDOW_ENABLED	WTINT
WTWINDOW_EYE	WTINT
WTWINDOW_HITHER	WTFLOAT
WTWINDOW_KEY	WTINT
WTWINDOW_LBUTTON	WTINT (1=down, 0=up)
WTWINDOW_LBUTTONDBLCLK	WTINT

WTwindow Properties	Data Type
WTWINDOW_RBUTTON	WTINT (1=down, 0=up)
WTWINDOW_RBUTTONDBLCLK	WTINT
WTWINDOW_MBUTTON	WTINT (1=down, 0=up)
WTWINDOW_MBUTTONDBLCLK	WTINT
WTWINDOW_POSITION	WTP2
WTWINDOW_PROJECTION	WTINT
WTWINDOW_ROOTNODE	WTSTRING
WTWINDOW_SIZE	WTP2
WTWINDOW_TOPLEFT	WTP2
WTWINDOW_VIEWANGLE	WTFLOAT
WTWINDOW_VIEWPOINT	WTSTRING
WTWINDOW_VIEWPOINT2	WTSTRING
WTWINDOW_YON	WTFLOAT

WTsensor Properties

WTsensor Properties	Data Type
WTSENSOR_ANGULARRATE	WTFLOAT
WTSENSOR_LASTROTATION	WTQ
WTSENSOR_LASTTRANSLATION	WTP3
WTSENSOR_MISCDATA	WTINT
WTSENSOR_RAWDATA	WTPOINTER
WTSENSOR_ROTATION	WTQ
WTSENSOR_ROTATIONALOFFSET	WTQ

WTsensor Properties	Data Type
WTSSENSOR_SENSITIVITY	WTFLOAT
WTSSENSOR_TRANSLATION	WTP3
WTSSENSOR_UPDATEFN	WTPOINTER

WTpath Properties

WTpath Properties	Data Type
WTPATH_CONSTRAINTS	WTINT
WTPATH_DIRECTION	WTINT
WTPATH_MARKER	WTPOINTER)
WTPATH_MODE	WTINT
WTPATH_PLAYING	WTINT
WTPATH_PLAYSPEED	WTINT
WTPATH_ROTATION	WTQ
WTPATH_RECORDING	WTINT
WTPATH_RECORDLINK	WTPOINTER
WTPATH_SAMPLES	WTINT
WTPATH_TRANSLATION	WTP3
WTPATH_VISIBILITY	WTINT

WTbase Objects and Functions

The addition of the WTbase object type with Release 8 allows you to create generic, empty objects, distinguished only by the properties that you add to them. Just like the other object types supported by the OPE architecture (see page 3-2), you can add properties to WTbase objects, add event handlers to those properties to react to their value changes, and share those properties across the network when using World2World servers. WTK object types that are not supported by the OPE paradigm cannot contain properties and, thus, do not generate events or allow for the sharing of data over a network. To extend the OPE paradigm, create WTbase objects and user-defined properties (see page 3-14) to represent the desired attributes of the unsupported objects.

Suppose you have a texture applied to a geometry, and each time the texture on the geometry changes, you want to intensify one of your lights. Since the WTgeometry object type is not supported by the OPE architecture, you would create a WTbase object, add a Texture property to that object (see *WTproperty_new* on page 3-15), and add an event handler (see *WTproperty_addhandler* on page 3-25) to the Texture property. In the event handler, you would call *WTgeometry_settexture* to modify the actual geometry, followed by a call to *WTlightnode_setintensity*. Remember to always modify the geometry's texture via the WTbase object in order to trigger the property change event. By using this technique, you can share the Texture property with other clients when using World2World servers.

You can arrange WTbase objects in a hierarchy so that user data can be organized in a coherent fashion.

WTbase Functions for WTbase Objects

This section describes the WTbase functions that operate on WTbase objects only. See page 3-10 for information on the WTbase functions that operate on WTbase objects as well as other object types supported by the OPE architecture.

WTuniverse_getbases

```
WTbase* WTuniverse_getbases(  
    void);
```

This function returns a pointer to the first WTbase object in the universe's list of WTbase objects.

WTbase_next

```
WTbase* WTbase_next(  
    WTbase *object);
```

This function returns the next WTbase object in the universe's list of WTbase objects. Use *WTuniverse_getbases* to obtain a pointer to the first WTbase object in the universe's list of WTbase objects.

WTbase_new

```
WTbase* WTbase_new(  
    WTbase *parent);
```

This function creates a new WTbase object as a child of *parent* and returns a pointer to it. If *parent* is NULL, the WTbase object will be created as an orphan, i.e. it will not have any parent(s) unless *WTbase_addparent* is used to add a parent to the newly created WTbase object. This new WTbase object is added to the universe's list of WTbase objects.

WTbase_addparent

```
void WTbase_addparent(  
    WTbase *object  
    WTbase *parent);
```

This function adds the specified *parent* WTbase object as a new parent of *object*.

WTbase_removeparent

```
void WTbase_removeparent(  
    WTbase *object  
    WTbase *parent);
```

This function removes the specified *parent* WTbase object as a parent of *object* so that the *object* WTbase object is no longer a child of *parent*.

WTbase_numparents

```
int WTbase_numparents(  
    WTbase *object);
```

This function returns the number of parents of the specified WTbase object.

WTbase_getparent

```
WTbase* WTbase_getparent(  
    WTbase *object  
    int parentnum);
```

This function returns a pointer to the *parentnum*'th parent of a WTbase object. *parentnum* can range from 0 to (*WTbase_numparents* - 1).

WTbase_numchildren

```
int WTbase_numchildren(  
    WTbase *object);
```

This function returns the number of children of the specified WTbase object.

WTbase_getchild

```
WTbase* WTbase_getchild(  
    WTbase *object  
    int childnum);
```

This function returns a pointer to the *childnum*'th child of the specified WTbase object. *childnum* can range from 0 to (*WTbase_numchildren* - 1).

WTbase_ischild

```
FLAG WTbase_ischild(  
    WTbase *parent  
    WTbase *child);
```

This function returns TRUE if *child* is a first generation child of *parent*.

WTbase_findchild

```
WTbase* WTbase_findchild(  
    WTbase *object  
    const char *name);
```

This function returns a pointer to the WTbase object which is a first generation child of the specified WTbase object and whose name matches the *name* parameter.

WTbase Functions for the Supported WTK Object Types

In addition to the WTbase_* functions described in the previous section, there are a number of additional WTbase_* functions that can be used with WTbase objects as well as with the other WTK object types supported by the OPE architecture (see page 3-2). This provides WTK programmers a uniform way to store, manipulate, and retrieve user and WTK data.

By using these WTbase functions, WTK applications can now generically access any of the supported WTK objects and their properties. So, for example, if you wish to set the name of a WTnode object, you can use either the *WTnode_setname* function or the *WTbase_setname* function on the WTnode object.

WTbase_gettype

```
int WTbase_gettype(  
    void *object);
```

This function returns the type of the object passed in (WTBASE, WTNODE, WTWINDOW, WTVIEWPOINT, WTSENSOR, or WTPATH).

WTbase_delete

```
void WTbase_delete(  
    void *object);
```

This function deletes an object.

WTbase_print

```
void WTbase_print(  
    void *object);
```

This function prints information about an object.

WTbase_setdata

```
void WTbase_setdata(  
    void *object  
    void *data);
```

This function sets the user-defined data field in an object.

WTbase_getdata

```
void* WTbase_getdata(  
    void *object);
```

This function returns the user-defined data field from an object.

WTbase_setname

```
void WTbase_setname(  
    void *object  
    const char *name);
```

This function sets the name of an object.

WTbase_getname

```
char* WTbase_getname(  
    void *object);
```

This function returns the name of an object.

WTbase_numproperties

```
int WTbase_numproperties(  
    void *object);
```

This function returns the number of properties associated with an object.

WTbase_getproperty

```
char* WTbase_getproperty(  
    void *object  
    int propnum);
```

This function returns the *propnum*'th property of the specified object. *propnum* can range from 0 to (*WTbase_numproperties* - 1). To get the value of the property, use *WTproperty_get* (see page 3-20).

WTbase_nfindproperty

```
char* Wtbase_nfindproperty(  
    void *object  
    const char *propname  
    int ntocmp);
```

This function returns the full property name of the first property of an object whose first *ntocmp* characters of the property name match the first *ntocmp* characters of *propname*.

WTbase_deleteproperties

```
FLAG Wtbase_deleteproperties(  
    void *object);
```

This function deletes all user-defined properties from the specified object.

WTbase_find

```
void* Wtbase_find(  
    int objtype  
    const char *name);
```

This function finds an object of the specified *objtype* by name. *objtype* can be WTBASE, WTNODE, WTWINDOW, WTVIEWPOINT, WTSENSOR, or WTPATH.

WTbase_nfind

```
void* Wtbase_nfind(  
    int objtype  
    const char *name  
    int ntocmp);
```

This function finds an object of the specified *objtype* whose name's first *ntocmp* characters matches the first *ntocmp* characters of *name*. *objtype* can be WTBASE, WTNODE, WTWINDOW, WTVIEWPOINT, WTSENSOR, or WTPATH.

Properties

Objects that are supported by the OPE architecture (see page 3-2) are distinguished from one another by their properties. Properties describe characteristics of an object. For example, *WTviewpoint* objects (as shown in the table on page 3-2) have the following pre-defined properties:

- WTVIEWPOINT_ASPECT
- WTVIEWPOINT_CONVDISTANCE
- WTVIEWPOINT_CONVERGENCE
- WTVIEWPOINT_ORIENTATION
- WTVIEWPOINT_PARALLAX
- WTVIEWPOINT_POSITION

One of the advantages of the OPE architecture is that additional user-defined properties can be added to an object of any of the supported types. This allows WTK to treat user data in a similar fashion to pre-defined properties. Consequently, changes to user data (events) can now trigger reactions to those changes and can be shared on the network. Note that WTK still allows user data to be associated with WTK objects through their ‘data’ field via calls to functions such as *WTviewpoint_setdata* and *WTviewpoint_getdata*. However, you cannot trigger event reactions or share data across the network for data associated with WTK objects in this manner.

Properties are specified by their property name. Each property’s name (within a given object) must be unique. That is, no two properties of an object can have identical property names. Each property is of a specific data type. The table below lists the property data types available in WTK.

WTdatatype	Actual data	Range	WTProperty_getasString
WTINT	int	-2,147,483,648 to 2,147,483,647	"10"
WTUINT	unsigned int	0 to 4,294,967,295	"100003223"
WTFLOAT	float	3.4E +/- 38 (7 digits)	"10.25"
WTDDOUBLE	double	1.7E +/- 308 (15 digits)	"1034.2342342343424"

WTdatatype	Actual data	Range	WTProperty_getasstring
WTP2	float [2]	3.4E +/- 38 (7 digits)	"1.0,2.0"
WTP3	float [3]	3.4E +/- 38 (7 digits)	"1.0,2.0,3.0"
WTQ	float [4]	3.4E +/- 38 (7 digits)	"0.0,0.0,0.0,1.0"
WTSTRING	char*	–	"this is a string value"
WTPOINTER	void*	–	"0x00000000"

Following are the functions that allow you to create, access, and delete properties. Note that some of the functions listed below take values of type ‘void *’ as a parameter and its usage is dependent upon the property’s data type. See the examples shown for *WTProperty_set* on page 3-17 and *WTProperty_get* on page 3-20 for clarification.

WTProperty_new

```
FLAG WTProperty_new(
    void *object
    const char *propname
    WTdatatype dtype);
```

This function creates a new user-defined property whose name is *propname* and whose data type is *dtype* for the specified object.

WTProperty_delete

```
FLAG WTProperty_delete(
    void *object
    const char *propname);
```

This function deletes the user-defined property whose name is *propname* from a specified object.

WTproperty_exists

```
FLAG WTproperty_exists(  
    void *object  
    const char *propname);
```

This function returns TRUE if the property whose name is *propname* exists on a specified object.

WTproperty_setdata

```
void WTproperty_setdata(  
    void *object  
    const char *propname  
    void *data);
```

This function sets the user-defined data field for a property.

WTproperty_getdata

```
void* WTproperty_getdata(  
    void *object  
    const char *propname);
```

This function returns the user-defined data field for a property.

WTproperty_getdatatype

```
WTdatatype WTproperty_getdatatype(  
    void *object  
    const char *propname);
```

This function returns the datatype of the specified object's *propname* property.

WTproperty_getsizeofdata

```
unsigned int WTproperty_getsizeofdata(
    void *object
    const char *propname);
```

This function returns the number of bytes used by the specified object's *propname* property value.

WTproperty_set

```
FLAG WTproperty_set(
    void *object
    const char *propname
    void *value);
```

This function sets the specified object's *propname* property's value to *value*.

Usage of WTproperty_set:

WTINT	int v = 10;	WTproperty_set(o, p, (void*)&v);
WTUINT	unsigned int v = 23423452345;	WTproperty_set(o, p, (void*)&v);
WTFLOAT	float v = 2314.2134f;	WTproperty_set(o, p, (void*)&v);
WTDOUBLE	double v = 234234.234234234;	WTproperty_set(o, p, (void*)&v);
WTP2	WTp2 v = {0.0f, 1.0f};	WTproperty_set(o, p, (void*)v);
WTP3	WTp3 v = {0.0f, 1.0f, 0.0f};	WTproperty_set(o, p, (void*)v);
WTQ	WTq v = {0.0f, 0.0f, 0.0f, 1.0f};	WTproperty_set(o, p, (void*)v);
WTSTRING	char v[] = "Test message";	WTproperty_set(o, p, (void*)v);
WTPOINTER	void *v = WTuniverse_getrootnodes();	WTproperty_set(o, p, v);

Alternatively, you could use the following type-specific `WTproperty_set` functions:

WTproperty_seti

```
FLAG WTproperty_seti(  
    void *object  
    const char *proprname  
    int value);
```

WTproperty_setui

```
FLAG WTproperty_setui(  
    void *object  
    const char *proprname  
    unsigned int value);
```

WTproperty_setf

```
FLAG WTproperty_setf(  
    void *object  
    const char *proprname  
    float value);
```

WTproperty_setd

```
FLAG WTproperty_setd(  
    void *object  
    const char *proprname  
    double value);
```

WTproperty_setp2

```
FLAG WTproperty_setp2(  
    void *object  
    const char *proprname  
    WTp2 value);
```


WTproperty_setp3

```
FLAG WTproperty_setp3(  
    void *object  
    const char *proprname  
    WTp3 value);
```

WTproperty_setq

```
FLAG WTproperty_setq(  
    void *object  
    const char *proprname  
    WTq value);
```

WTproperty_sets

```
FLAG WTproperty_sets(  
    void *object  
    const char *proprname  
    const char *value);
```

WTproperty_setp

```
FLAG WTproperty_setp(  
    void *object  
    const char *proprname  
    void *value);
```

WTproperty_setat

```
FLAG WTproperty_setat(  
    void *object  
    const char *proprname  
    void *value  
    double time);
```

This function sets the specified object's *proprname* property's value to *value* at a specified *time*. For more information on time values, see page 3-27. For examples of usage, see *WTproperty_set* on page 3-17.

WTproperty_get

```
FLAG WTproperty_get(  
    void *object  
    const char *proprname  
    void *value);
```

This function retrieves the specified object's *proprname* property value.

Usage of WTproperty_get:

WTINT	int v;	WTproperty_get(o, p, (void*)&v);
WTUINT	unsigned int v;	WTproperty_get(o, p, (void*)&v);
WTFLOAT	float v;	WTproperty_get(o, p, (void*)&v);
WTDOUBLE	double v;	WTproperty_get(o, p, (void*)&v);
WTP2	WTp2 v;	WTproperty_get(o, p, (void*)v);
WTP3	WTp3;	WTproperty_get(o, p, (void*)v);
WTQ	WTq v;	WTproperty_get(o, p, (void*)v);
WTSTRING	char *v;	WTproperty_get(o, p, (void*)&v);
WTPOINTER	void *v;	WTproperty_get(o, p, (void*)&v);

Note: The *char** result of a *WTproperty_get* or *WTproperty_gets* on a *WTSTRING* property is a pointer to the actual string stored in *WTK*. **DO NOT** modify this string directly with calls to *WTfree*, *WTrealloc*, *strcat*, *strcpy*, etc. If you need to modify the string value, make a local copy of the string before modifying it.

For example:

```
{  
    char *value;  
    char *newvalue;  
    value = WTproperty_gets(obj, "myprop");  
    newvalue = WTmalloc(strlen(value)+strlen("addtostring")+1);  
    strcpy(newvalue, value);
```

```
    strcat(newvalue, "addtostring");
    WTproperty_sets(obj, "myprop", newvalue);
    WTFree(newvalue);
}
```

As an alternative to the *WTproperty_get* function, you could use the following type-specific functions:

WTproperty_geti

```
int WTproperty_geti(
    void *object
    const char *propname);
```

WTproperty_getui

```
unsigned int WTproperty_getui(
    void *object
    const char *propname);
```

WTproperty_getf

```
float WTproperty_getf(
    void *object
    const char *propname);
```

WTproperty_getd

```
double WTproperty_getd(
    void *object
    const char *propname);
```

WTproperty_getp2

```
FLAG WTproperty_getp2(
    void *object
    const char *propname
    WTP2 value);
```

WTproperty_getp3

```
FLAG WTproperty_getp3(  
    void *object  
    const char *propname  
    WTp3 value);
```

WTproperty_getq

```
FLAG WTproperty_getq(  
    void *object  
    const char *propname  
    WTq value);
```

WTproperty_gets

```
char* WTproperty_gets(  
    void *object  
    const char *propname);
```

Note: See page 3-20 for special information on the usage of *WTSTRING* properties.

WTproperty_getp

```
void* WTproperty_getp(  
    void *object);
```

WTproperty_getasstring

```
char* WTproperty_getasstring(  
    void *object  
    const char *propname);
```

This function returns the specified object's *propname* property value as a string. (Strings returned are only good until another call to *WTproperty_getasstring* is executed).

WTvalue_tostring

```
char* WTvalue_tostring(  
    WTdatatype dtype  
    void *value);
```

This function returns the data *value* of type *dtype* as a string. (Strings returned are only good until another call to *WTvalue_tostring* is executed).

Events

Events occur when the value of a property changes. The value of a property can change due to a property being set with a call to *WTproperty_set* or via typical WTK calls such as *WTnode_settranslation*, *WTviewpoint_setposition*, etc., or through internal processes like motion link updates.

These value changes can be acted on by adding any number of *event handlers* (callback functions) to the property. When a property value changes, an event is internally generated which will trigger the execution of that property's event handlers in the main WTK simulation loop, right before the universe's actions function is called. If WTK's simulation loop is not active (that is, if your application has not called *WTuniverse_go* or *WTuniverse_go1*), a call to *WTuniverse_processevents* will execute the handlers.

The event handler is given a pointer to the object, the property that generated the event, the new value of the property, the source from which the property change event was generated, and the time the event was generated. The *object* pointer points to an object of one of the object types supported by the OPE architecture (see page 3-2). To find out what type of object it is, use *WTbase_gettype* (see page 3-11). The *propname* passed to the event handler is the name of the property that generated the event (this is either a pre-defined WTK property like *WTNODE_TRANSLATION* or a user-defined property created with *WTproperty_new*).

The *value* argument contains the new value of the property. The value is passed into the function as a void* and must be treated differently depending on the datatype of the property. To find out what datatype the property is, call *WTproperty_getdatatype* (see page 3-16). The table below describes how to treat this void* for each datatype.

Property change events can be triggered from several *event sources*. WTLOCAL events are events that originate from the local computer, while WTNETWORK events originate from another computer on the network involved in a World2World simulation. WTLOCAL_TIMER and WTNETWORK_TIMER events can occur due to property changes that result from a call to *WTproperty_setat* (see page 3-19).

The *time* argument, which is passed to a connection callback for networked simulations, is the time at which the property changed. This may not be the current time, since events are queued and executed at different times.

An event handler callback function takes the form:

```
void WTevenhandler(
    void *object,           ← object which generates the event
    const char *proprname, ← property whose value has changed
    void *value,           ← new value of the property
    double time,           ← time of the change
    WTevensource src);    ← source of the change (WTLOCAL, WTNETWORK, etc.)
```

Treat the value parameter which is of type 'void *' as follows:

'void *value'	Cast to	Usage
WTINT	int *value	printf("Value: %d\n", *((int*)value));
WTUINT	unsigned int *value	printf("Value: %u\n", *((unsigned int*)value));
WTFLOAT	float *value	printf("Value: %f\n", *((float*)value));
WTDDOUBLE	double *value	printf("Value: %f\n", *((double*)value));
WTP2	float *value	printf("Value: X=%f\n", ((float*)value)[0]);
WTP3	float *value	printf("Value: Y=%f\n", ((float*)value)[1]);
WTQ	float *value	printf("Value: W=%f\n", ((float*)value)[3]);
WTSTRING	char*	printf("Value: %s\n", (char*)value);
WTPOINTER	void*	Wprintf("Value: %x\n", value);

Note that if your application does not call *WTuniverse_go* or *WTuniverse_go1* then it must call *WTuniverse_processevents* in order for WTK to process these events and execute the callback handlers.

Following are the functions that pertain to events and event handlers.

WTproperty_addhandler

```
FLAG WTproperty_addhandler(  
    void *object  
    const char *propname  
    WTevenhandler eh);
```

This function adds an event handler callback to the specified object's *propname* property.

WTproperty_removehandler

```
FLAG WTproperty_removehandler(  
    void *object  
    const char *propname  
    WTevenhandler eh);
```

This function removes an event handler callback from the specified object's *propname* property.

WTproperty_numhandlers

```
int WTproperty_numhandlers(  
    void *object  
    const char *propname);
```

This function returns the number of handlers assigned to the specified object's *propname* property.

WTproperty_gethandler

```
WTevenhandler WTproperty_gethandler(  
    void *object  
    const char *propname  
    int handlenum);
```

This function returns the *handlenum*'th handler for the specified object's *propname* property.

WTproperty_removeallhandlers

```
void WTproperty_removeallhandlers(  
    void *object  
    const char *propname);
```

This function removes all handlers for the specified object's *propname* property.

WTbase_removeallhandlers

```
FLAG WTbase_removeallhandlers(  
    void *object);
```

This function removes all event handlers from all the properties of the specified object.

WTuniverse_processevents

```
void WTuniverse_processevents(  
    void);
```

This function processes all events in the universe. You must call it if your application's simulation loop is not active (that is, if your application has not called *WTuniverse_go* or *WTuniverse_go1*) in order to ensure that all of the events are processed.

Time

Time values are measured in seconds since January 1st, 1970 and include fractions of a second. These time values are used by event handlers and timed functions like *WTproperty_setat* (see page 3-19).

WTime_update

```
void WTime_update(void);
```

If not in a *WTuniverse_go* loop, call this function to update the time value returned from *WTime_getcurrent*.

WTime_getcurrent

```
double WTime_getcurrent(void);
```

This function returns the current Greenwich mean time (GMT) in seconds from 01-01-70.

WTime_getcurrentlocal

```
double WTime_getcurrentlocal(void);
```

This function returns the current (local timezone) time in seconds from 01-01-70.

WTime_getcurrentsec

```
int WTime_getcurrentsec(void);
```

This function returns the whole number of seconds from 01-01-70 Greenwich mean time (GMT).

WTime_getcurrentseclocal

```
int WTime_getcurrentseclocal(void);
```

This function returns the whole number of seconds from 01-01-70 in the local timezone.

WTtime_getcurrentmsec

```
unsigned short WTtime_getcurrentmsec(void);
```

This function returns the number of milliseconds beyond the current second in Greenwich mean time (GMT).

WTtime_getcurrentmseclocal

```
unsigned short WTtime_getcurrentmseclocal(void);
```

This function returns the number of milliseconds beyond the current second in local time.

WTtime_getdouble

```
double WTtime_getdouble(  
    int sec  
    unsigned short msec);
```

This function returns a 'double' version of a seconds and milliseconds time value.

WTtime_getsec

```
int WTtime_getsec(  
    double time);
```

This function returns the seconds part of a 'double' time value.

WTtime_getmsec

```
unsigned short WTtime_getmsec(  
    double time);
```

This function returns the milliseconds part of a 'double' time value.

Scene Graphs

This chapter describes the fundamental building blocks of your WTK application, how they are created, their properties, and how to assign behaviors. The main sections of this chapter are as follows:

- *Introduction* – introduces the hierarchical scene graph, and explains why WTK uses the scene graph structure. (see page 4-2)
- *Scene Graph Concepts in Detail* – offers a more detailed discussion on the scene graph and its concepts. (see page 4-5)
- *Building a Scene Graph* – provides a general discussion of how to build a hierarchical scene graph and explains several important scene building concepts. (see page 4-29)
- *WTK Scene Graph Functions* – gives descriptions of the scene graph functions. (see page 4-39)
- *Additional Topics Related to the Scene Graph* – includes a section of additional topics related to the scene graph, including node paths, intersection testing, picking polygons, and attaching sensors. (see page 4-79)

Introduction

This section introduces the hierarchical scene graph, and explains why WTK uses the scene graph structure.

The Scene

A *scene* is a collection of geometries and lights, along with the positional information that places these elements at particular locations. In WTK, the only other element that is considered to be part of the scene directly is fog. So, in very simple terms, a scene is built from the following four content elements: geometries, lights, positional information, and fog.

Elements Of A Scene

GEOMETRIES

Geometries include static geometry files you load into WTK (using WTK's file import functions) and dynamic geometries you create within WTK at the polygon and vertex levels.

LIGHTS

Lights include the lights that may be part of a file you load into WTK and the lights you dynamically create in WTK. You can use lights in WTK to illuminate some or all of the geometries in a scene.

POSITIONAL INFORMATION

Positional information includes any positional information that is associated with geometries and lights read in from a file, and dynamic positional information created and managed within WTK. It describes where particular elements (geometries and lights) should be placed in the scene, either in relation to another object, or in relation to the scene as a whole.

Fog

Fog is a special effect that simulates environmental conditions such as smoke, haze, mist, and of course fog. A geometry that is further away from the viewer becomes obscured by fog. You can use fog to affect some or all of the geometries in a scene.

The Viewpoint

In order to render a collection of objects in 3D space to the screen, WTK needs to know where and how in the scene the viewer is oriented. *WTviewpoint* is the WTK object that contains this information. *WTviewpoint* defines the position of the viewer in the scene, the direction in which the viewer is looking, and the viewer's twist about the directional axis. For example, if you look at a scene with your head tilted to one side, you will see an image that is the same as the one you see with no tilt, except that it has been slightly rotated in the direction opposite to your head's tilt.

The Scene Graph

WTK maintains your scene in a hierarchical structure known as a *scene graph*. You can think of the scene graph as an upside-down tree, where the root is on the top and the branches and leaves are on the bottom.

The scene graph is the structure that holds all of the current elements of the scene, such as geometries, lights, fog, and positional information. As shown in figure 4-1, the scene graph is an ordered collection of nodes, in the form of a directed acyclic (defined on page 4-29) graph, which holds hierarchical scene data.

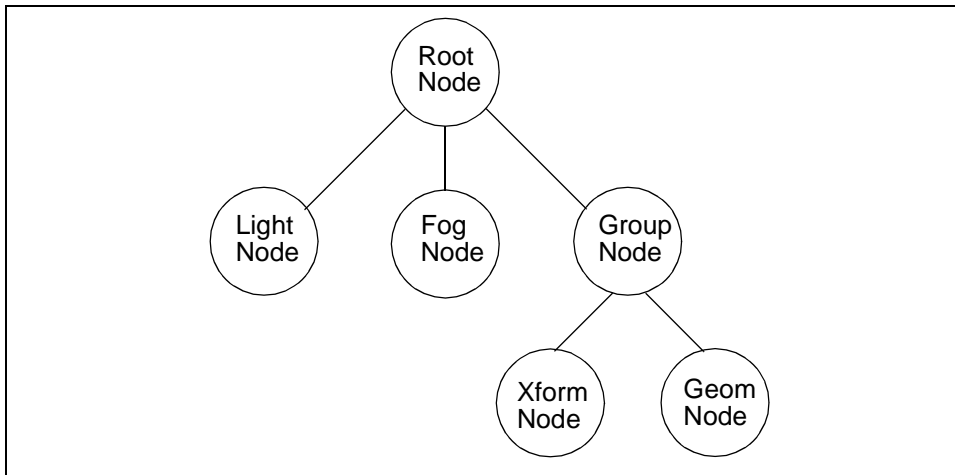


Figure 4-1: Simple scene graph

The basic element of the scene graph is the node, which either holds geometry, light, fog, and position data, or is a structural element that you use to maintain the hierarchy of the graph.

Why WTK Uses the Scene Graph Structure

The scene graph provides a very powerful scene structure for real-time 3D simulation. Specifically, it provides the hierarchical framework for easily grouping objects together spatially. This is essential for maintaining performance in scenes that contain many individual objects. Because you can group objects together in a positional hierarchy, you are able to use the scene graph to easily construct and maintain efficient simulations which contain individual moving parts.

The scene graph enhances performance of the WTK's rendering stage (drawing the scene) because it facilitates spatial culling of the scene. In other words, WTK calculates which parts of the scene (or scene graph) are visible from the current viewpoint, and quickly rejects non-visible geometry before drawing begins.

In addition to culling, the scene graph enhances the performance of both picking into the scene (using a mouse, for example) and general intersection testing (e.g., collision detection). WTK's scene graph also easily supports advanced procedural elements such as

level-of-detail (LOD) switching and hierarchical file formats such as VRML and MultiGen.

To create a 3D scene, you build a scene graph that describes this scene. One of the primary functions of the WTK API (Application Programmer's Interface) is to provide you with the tools and methods to build scene graphs. This includes functions to create the core elements (nodes), and functions to assemble, disassemble, rearrange and query the relationships between these elements.

Scene Graph Concepts in Detail

This section offers a detailed discussion of the scene graph and its concepts.

The Node

The *node* is the fundamental element of the scene graph; it is the basic building block that you use to construct scene graphs. A node is simply an element of *content*, or a *grouping/procedural* element you use to maintain scene hierarchy.

CONTENT NODES

Content nodes are easy to understand. They are containers for the four basic elements of a scene: geometry, light, position, and fog. Geometry nodes contain geometry information, light nodes contain light information, and fog nodes contain fog information. Positional information is contained in nodes called *transform nodes*.

Objects (geometry or lights) in 3D scenes can have both a position (X,Y,Z in cartesian space), and an orientation about this position (pitch, yaw, roll). For example, you can stand at some particular place in a room (position), and turn your body to face any direction (orientation). Transform nodes store both a position and an orientation internally in a 4x4 matrix.

GROUPING NODES

Grouping (organizational) nodes contain no content directly, however they are the essential structuring nodes used in building a scene graph. To understand the role that grouping nodes play, it is important to understand the structure of the scene graph. (See *The Scene Graph Hierarchy* below.) Briefly, organizational nodes, such as the group node, the separator node and the transform separator node, let you group together and encapsulate a set of nodes that share common states, such as position or lighting effects.

Procedural nodes are like organizational nodes but they contain additional information that they use to activate one of their child nodes while deactivating the others. The level-of-detail (LOD) node and the switch node are procedural nodes.

The Scene Graph Hierarchy

Nodes are ordered in a directed hierarchical fashion. In other words, nodes are attached together from top to bottom, in a tree-like structure. A node that has nodes attached to it from the bottom is a “parent” to those nodes. Those nodes attached immediately underneath another node are the “children” of that node. If two nodes share the same parent, then they are considered to be “siblings.” Figure 4-2 illustrates the parent, child, sibling structure.

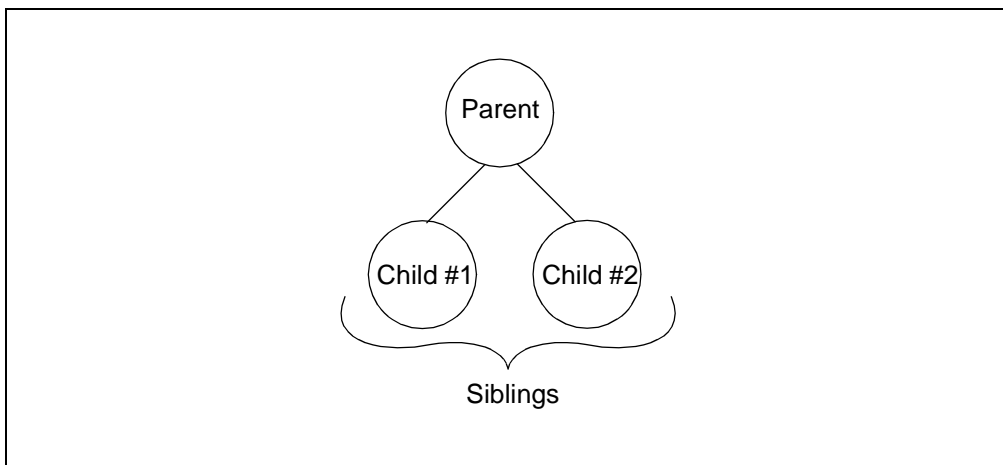


Figure 4-2: Parent, child, and sibling relationships

All scene graphs in WTK require a starting point. This starting point is called a *root node*. Because the scene graph is hierarchical, or structured in a top to bottom manner, the root node represents the top point on the scene graph. Each scene graph has a single root node, and this node cannot be shared with other scene graphs. WTK allows multiple scene graphs, and these are uniquely identified by their individual root nodes.

SCENE GRAPH TERMINOLOGY

Figure 4-3 illustrates a number of terms that are used throughout this guide to discuss the way WTK implements scene graphs.

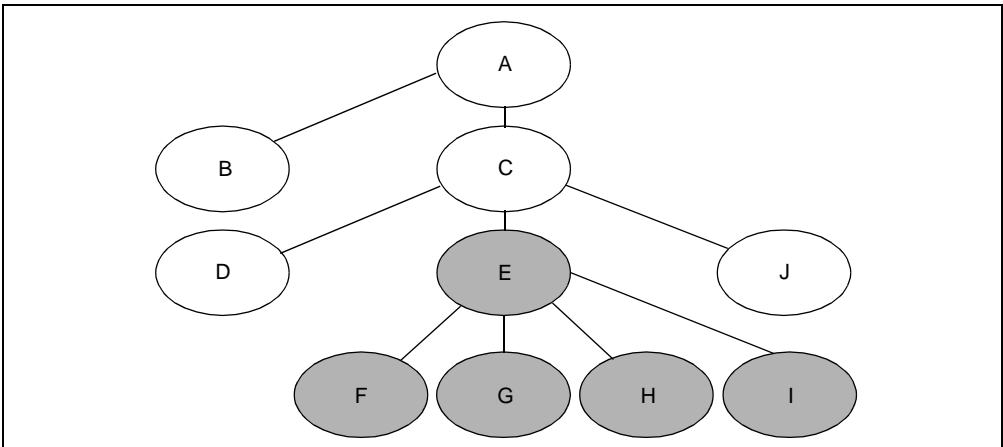


Figure 4-3: A schematic diagram of a scene graph

Ancestor	Since node A (in figure 4-3) has a sub-tree that includes node E, it is an <i>ancestor</i> of node E. Note that node J is not an ancestor of node I.
Child node	A node's direct descendant. In figure 4-3, nodes B and C are both children of node A. J is not a <i>child</i> of A.
Descendant	Any node that is contained in the sub-tree of another node is considered to be a <i>descendant</i> of that node. In figure 4-3, node F is one of the descendants of node A.
Leaf node	A node without children. Nodes B, D, F, G, H, I, and J are all leaf nodes.

Parent node	A node's direct ancestor. Node A is a <i>parent</i> of node C, but not of node E. It is possible for a node to have several parents.
Predecessor	Since nodes B and C are processed before node J, they are its <i>predecessors</i> . A node's predecessors can affect the rendering of that node, even though they may not be ancestors.
Root node	Each scene graph has only one <i>root node</i> . The root node in figure 4-3 is node A.
Scene graph tree	All of the nodes in a scene graph, arranged in a hierarchical order. The nodes in figure 4-3 are all in one <i>scene graph tree</i> .
Sub-tree	A node and all of its descendants. A <i>sub-tree</i> in figure 4-3 is shaded.
Sibling	Children of the same parent node are <i>siblings</i> . Nodes F, G, H, and I are siblings.
Traversal order	The order in which nodes in a scene graph are processed while the simulation is running. The nodes in the scene graph in figure 4-3 have been labeled so that their alphabetical order indicates the proper <i>traversal order</i> . For more information, see <i>Traversing the Scene Graph Tree</i> on page 4-9.

Viewing your Scene Graph

It may help you visualize the scene graph(s) in your simulation by seeing a printout of the scene graph hierarchy (or any part of it). Use the `WTnode_print` function (see page 4-76) to print the scene graph hierarchy, starting at the specified node. If you specify the root node, the whole scene graph hierarchy is printed; if you specify any other node, only the specified node and its sub-tree hierarchy are printed.

How WTK Draws the Scene Graph

As previously discussed, the scene graph is the hierarchical structure that contains all of the elements of a scene, and their relationships to each other. Every WTK window has both a scene graph (referenced via its root node) and a viewpoint associated with it, providing WTK with everything necessary to draw a scene (the scene graph), as viewed from a certain position and orientation (the viewpoint), to some place on the screen (the window).

TRAVERSING THE SCENE GRAPH TREE

The root node is the entry point into the scene graph, and is the point where WTK starts to draw the scene. Once at the root node, WTK begins “walking” (traversing) the scene graph tree. This “walking” process is always the same. WTK traverses the tree, visiting each node of the tree in a top to bottom, left to right order. In other words, when WTK encounters a node with more than one child, it walks down the first child’s branch, completely traversing this portion of the tree before returning back up and processing the second child’s branch. Figure 4-4 illustrates this “walking” the scene graph tree process.

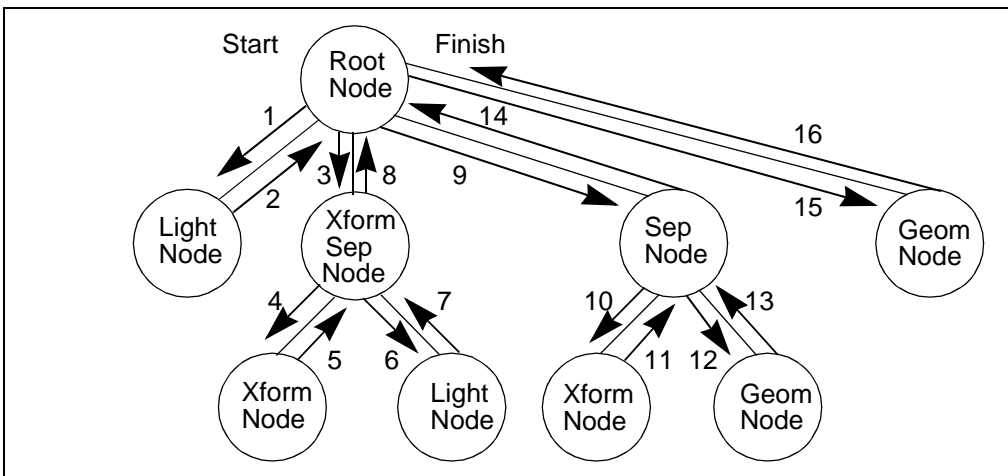


Figure 4-4: Walking the scene graph

It is during this process of traversing the scene graph tree that WTK draws the scene. As WTK encounters nodes in the scene graph tree, it evaluates and processes them depending on their type. Very simply, when WTK encounters a geometry node, it draws it (at the current position and orientation, with the current lighting and current fog); when it

encounters a light node, it adds this light to the currently active set of lights; when it encounters a transform node, it modifies the current orientation and position information; and when it encounters a fog node, it sets the current fog.

WTK traverses the entire scene graph tree once per frame. Recall from the Universe chapter, WTK runs in a simulation loop that includes six different stages: reading the sensors, calling the universe action function, updating objects, performing tasks, stepping paths, and rendering the universe. It is in this last stage that WTK traverses the scene graph (or multiple scene graphs) and draws the scene to a window (or multiple scenes to multiple windows).

ENCOUNTERING CONTENT NODES

WTK performs the actual act of drawing when it encounters a geometry node in a scene graph. Since WTK traverses the entire scene graph tree once per frame, it will encounter all the active geometry nodes in that tree for that frame, and thus draw all of the active geometries (objects) in the scene. The other three content nodes (light, transform, and fog) do not directly result in drawing by WTK, but they do affect *how* WTK draws the geometry. All three node types contribute to, and/or modify, the current drawing “state” which WTK maintains as it traverses the scene graph tree.

Every geometry in a 3D scene has a specific position, orientation, color, and brightness that is affected by any active lights shining on the geometry. Also, the geometry is obscured by any surrounding fog. When WTK traverses the scene graph tree, it maintains a current transformation state, a current lighting state, and a current fog state. When it encounters a geometry node while traversing the tree, WTK draws this geometry at the position and orientation defined by the current transformation state, lit by the lights of the current lighting state, and obscured by the fog of the current fog state.

It is important to know that the current transformation, lighting, or fog state at any time during scene graph traversal is totally dependent on which transform, light and fog nodes WTK has encountered up to that point. When WTK encounters a transform node, it updates the transformation state. Transformation update is a concatenation, or combination of the newly encountered transform node with the current transformation state. If the current transformation state contains a rotation and the newly encountered transform node contains a translation, the resulting transformation state will contain a translation and a rotation.

When WTK encounters a light node, it adds the light contained in the light node to the list of active lights maintained in the current light state. When WTK processes a geometry node, all of the active lights in the current light state cumulatively affect how the geometry

is rendered. It is important to note that the current transformation state comes into play when WTK encounters a light node. Though light nodes have their own position and direction, these positions and directions are *modified* by the current transformation state. When WTK encounters a directed light node, it modifies the direction of this light by the current transformation state. If the current transformation state is not set (that is, WTK has not encountered any transform nodes at this point), then the direction remains as set in the light node. If there is a current transformation state, then WTK rotates the direction of this light by the orientation component of the current transformation state *before* it adds the light to the current list of active lights. When WTK encounters a point light, the same process occurs, but with the positional component (point lights only have position), and when WTK encounters a spot light in the scene graph, this same process occurs with both the position and direction components. When WTK encounters a fog node, it replaces the current fog state based on the values of the current fog node.

Table 4-1 summarizes the content node types, and indicates where you can find descriptions of them in this manual.

Table 4-1: Content Nodes

Node	What it does	Can have children?	Affects state?	Where described
Geometry	Displays a set of polygons, together with a WTK material.	No	No	Page 4-2
Fog	Simulates fog, smoke, murkiness.	No	Yes	Page 4-3
Light	Specifies a WTK light.	No	Yes	Page 4-2
Transform	Provides position and orientation information.	No	Yes	Page 4-24
Movable light	Specifies a movable light node. There are three types of movable light nodes.	No	Yes	See the <i>Movables</i> chapter, starting on page 5-1.
Movable geometry	Specifies a movable geometry node.	No	No	See the <i>Movables</i> chapter, starting on page 5-1.

ENCOUNTERING GROUPING NODES

The four content-specific nodes discussed so far can only exist as children of other nodes, they cannot have children themselves. The content nodes are not involved in forming the actual hierarchical structure of the scene graph tree because they are all leaf nodes of the tree. The hierarchy of the scene graph tree is provided by a general class of nodes called *grouping nodes*. The grouping node class is made up of all the WTK node types capable of having children attached to them, and in general, this includes all the other node types available in WTK beyond the four content-specific nodes.

The grouping nodes include the following:

- group node
- separator node
- transform separator node
- procedural nodes (the level-of-detail node and the switch node)
- specialized nodes (the root node, the inline node, and the anchor node)
- movable separator

The group node itself is a grouping node with no other special properties; its only function in the scene graph is to serve as a parent to one or more nodes in the tree. A group node, as shown in figure 4-5, can serve as a parent to both content nodes and other grouping nodes.

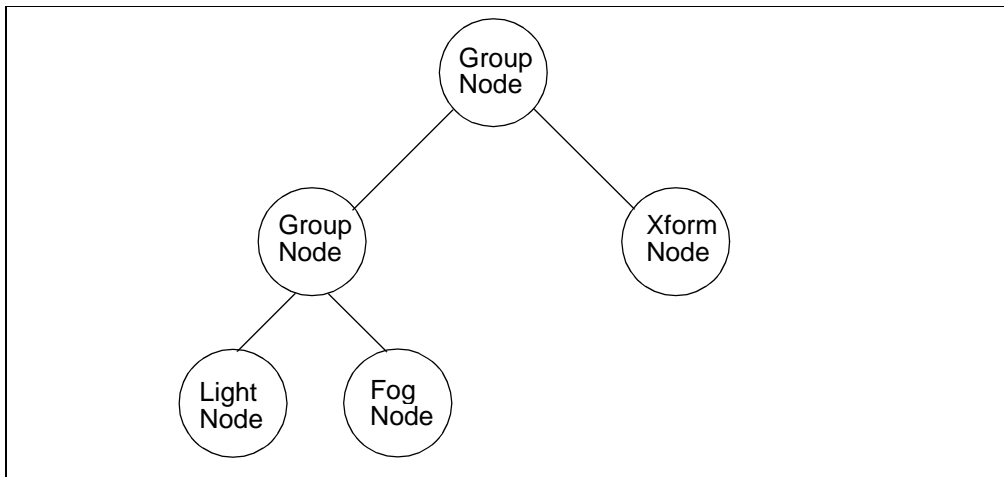


Figure 4-5: Group node

Table 4-2 summarizes the grouping node types, and indicates where you can find descriptions of them in this manual.

Table 4-2: Grouping Nodes

Node	What it does	Can have children?	Affects state?	Where described
Anchor	Contains a string property (URL) that references a data file.	Yes	No	Page 4-28
Group	Has child nodes, but no other properties.	Yes	No	Page 4-40
Inline	Contains a string property (URL) that references a data file which can be automatically read in.	Yes	No	Page 4-28
Level-of-detail (LOD)	Swaps in objects as a function of viewpoint distance.	Yes	No	Page 4-26
Root	Acts as the top node in a scene graph. Each scene graph has only one root node, which is not shared with any other scene graph. As the top node in its hierarchy, this node has no parent node.	Yes	No	Page 4-7
Separator	Prevents state information from propagating from its descendant nodes to its sibling nodes.	Yes	No	Page 4-21
Switch	Controls which of its children are traversed.	Yes	No	Page 4-25

Table 4-2: Grouping Nodes (continued)

Node	What it does	Can have children?	Affects state?	Where described
Transform separator	Prevents just the transformation state from propagating from its descendant nodes to its sibling nodes. All other states are allowed to propagate.	Yes	No	Page 4-24
Movable separator	Specifies a movable separator node.	Yes	No	See the <i>Movables</i> chapter, starting on page 5-1.

THE IMPORTANCE OF A SCENE GRAPH TREE HIERARCHY

One of the main reasons why you would want to use the scene graph hierarchy is that it is often very desirable to group several objects (such as geometries) together spatially in a scene. A good example of this is a car object that is composed of four tire geometries and a car body geometry. It is much easier to deal with this composite car object in the scene if you can group all of its parts together under one node, rather than five disparate geometry nodes. Remember, these geometries have positions and orientations too, so that gives you (potentially) another five transform nodes that you need to maintain as well.

Imagine if you wanted to move this five piece car around the scene as a whole, you would have to figure out a position and orientation to move the car body to, and then individually move each wheel to its proper location. The scene graph allows you to set up a hierarchy, under a grouping node, which creates a composite car object, which holds all the geometric information for the car, as well as the position and orientation of all the wheels relative to the car body.

Why the Ordering of Children is Important

Recall the order in which WTK traverses the scene graph tree. Starting from the root node, WTK walks down the tree to the first child node of the root node. If this child node is one of the four content nodes, then WTK processes this node. If this child node is a grouping node, then the traversal continues down to this node's first child node, and so on.

Because WTK processes the children of a grouping node in fixed order (first child branch first, second child branch second, etc.), the ordering of children *is* important. Suppose, for example, you have a simple case of a scene graph with five nodes, wherein the root node has one child, being a group node, as shown in Case #1 of figure 4-6. The group node's three children are a transform node, a light node and a geometry node. Assume the light node is *child #1*, the transform node is *child #2*, and the geometry node is *child #3* of the group node.

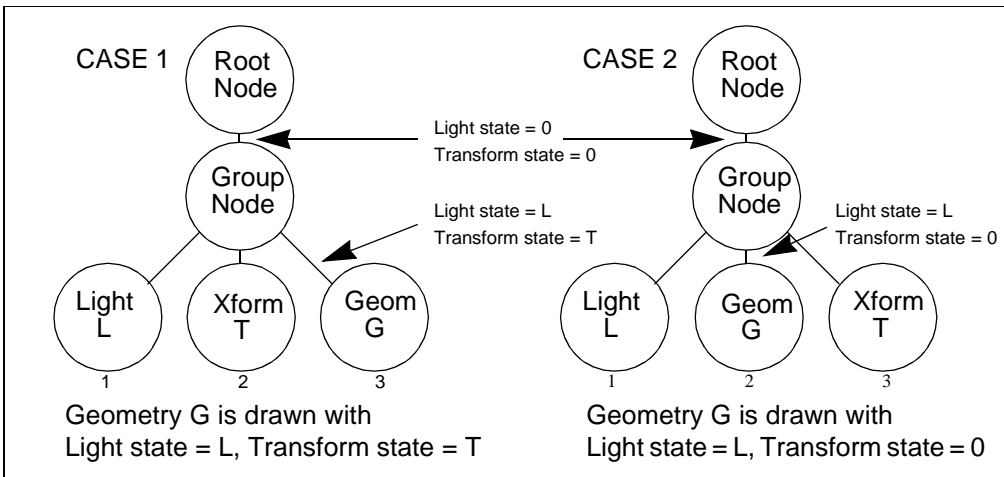


Figure 4-6: The importance of child order in the scene graph

WTK starts the tree traversal at the root node, and goes to its first (and only) child, the group node. When WTK encounters a group node, it does not do anything special, it just processes all the group node's children in order. The first child of the group node in this case is the light node. Encountering the light node, WTK adds this light to the current lighting state. After processing this node, the current lighting state contains one light, the light that was set by this node. (Recall the current lighting state defines the lights that will affect the drawing of any future geometry encountered in the tree.) Because the light node does not

(and cannot) have any children, WTK moves back up to the group node and processes its second child.

The second child is a transform node, so WTK updates its current transform (position and orientation) state. Since there is no current transformation state set, this node now defines the current transformation state. (Recall that the transformation state defines the position and orientation where any geometry encountered on the tree will be drawn. Again, since the transform node does not (and cannot) have any children, WTK now moves back up to the group node and processes the third and final child.

The third child is a geometry node, so WTK draws this geometry node, using the current transform state to position and orient the geometry in the scene, and using the lights in the current lighting state to illuminate the geometry. Note that the transform node has told WTK where to draw this geometry, so you can say the geometry was “affected” by this transform node. Note that the light node has told WTK how to light this geometry node, so you can say the geometry was “affected” by this light node.

Suppose you take exactly the same case as above, but make the geometry node the second child of the group node, the transform the third child of the group node, and the light node remains the first child (see Case #2 of figure 4-6). When WTK walks this tree, and processes the children of the group node, it encounters the light node first, adding the light to the current light state, just like before. The second child of the group node is the geometry node, so WTK draws it with the current lighting state and the current transformation state. Just like last time, the light node has been processed before the geometry node, so it affects the illumination of the geometry. The current transformation state, however, has not been set yet, so the geometry is drawn in the default position in the scene, the origin (0,0,0) with the default orientation. Because the transform node is the third child, it has not been processed yet, so it has not altered the current transformation state, and thus, has not affected the drawing of the geometry.

So you can see that the ordering of children, even in very simple cases, affects how things are drawn. *Make sure transform nodes that you want to affect a geometry are processed prior to that geometry node, and make sure light nodes you want to affect a geometry are processed before that geometry node. Remember that directed light nodes, point light nodes and spot light nodes have either an orientation component or a position component, or both, and are affected by the current transformation state. If you don't want your lights' positions and/or orientations changed, they must be at a place in the scene graph where no transform nodes can affect them (usually, this is at the top of the scene graph, as children of the root node).*

State Accumulation and State Propagation

As discussed previously, WTK has a concept of “state” (that is, the current transformation, lighting and fog state) for any particular place in the scene graph. These states affect the way any geometry nodes at that particular place in the scene graph are drawn. Because the current transformation state at some point in a scene graph is a concatenation of the transform nodes processed up to that point, and the current lighting state includes all the lights activated by processing light nodes up to that point, you can say that transform and light state “accumulate” as the scene graph tree is traversed, as shown in figure 4-7.

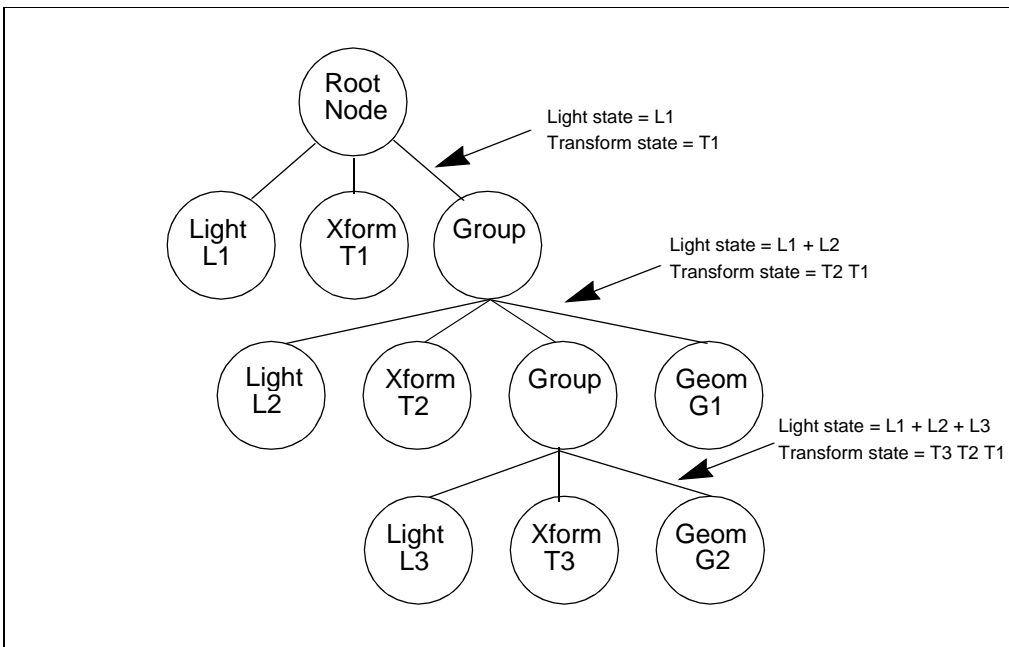


Figure 4-7: State accumulation

WTK also uses the concept of state “propagation” when traversing a scene graph tree. That is, should it encounter and process a transform node, the transformation state created by this node propagates, or moves along, as WTK continues to process the remainder of the scene graph. Geometry nodes that are drawn after encountering a transform, light or fog node are affected by these transform, light, and fog nodes, even if these nodes were processed long before, or farther up the tree than the current geometry.

It is important to be aware of the state propagation because, if you are not careful, transform nodes can affect geometries that you did not want affected. For example, suppose you wanted to build a scene with two geometrical objects, each independently placed in the scene, and both lit by a scenewide directed light source. To do this, you would build a scene graph piece by piece. The following example describes the scene graph illustrated in figure 4-8 in detail.

Example: Building a Simple Scene Graph Piece by Piece

First, consider the case of a geometry independently placed in the scene. This means you need a geometry, say a model of a dog, and a method to place this dog somewhere in the scene. The method you need to place this dog somewhere in the scene is a transformation (which is contained in a transform node). Thus, you need both a geometry node (the dog), and a transform node (the dog's position in the scene). Since this geometry node and this transform node conceptually go together (that is, you don't want this transform node to affect anything else but the dog), you would group these two nodes together as children of a group node, making the transform node the first child and the geometry node the second child of the group node. Now, the dog and its position are grouped together, and can be added to the scene. The scene is currently empty (i.e., it contains just a root node). Once the group node is attached to the root node, you have a scene containing a dog (and its position).

Suppose you want to add light to the scene. Assuming you want the light to affect the dog geometry, put it in a place where it will be processed in the scene graph prior to the dog geometry. You accomplish this by making the light node the first child of the root node. Since that is the very first node processed in the scene graph after the root node, *every* geometry in the scene will be affected by it. This means the dog object group node should be the second child of the root node. If you wanted, you could now create a second geometry/transformation pair for your second object in the scene, say a fire hydrant. You create the fire hydrant object the same way you created the dog object. Create a fire hydrant geometry node, a transform node for its position, and a group node to group the two together. Then, you add your new fire hydrant group node to the scene graph as the third child of the root node. The scene graph now contains all the content to fulfill the goal, two geometries independently placed in the scene, both lit by scenewide light.

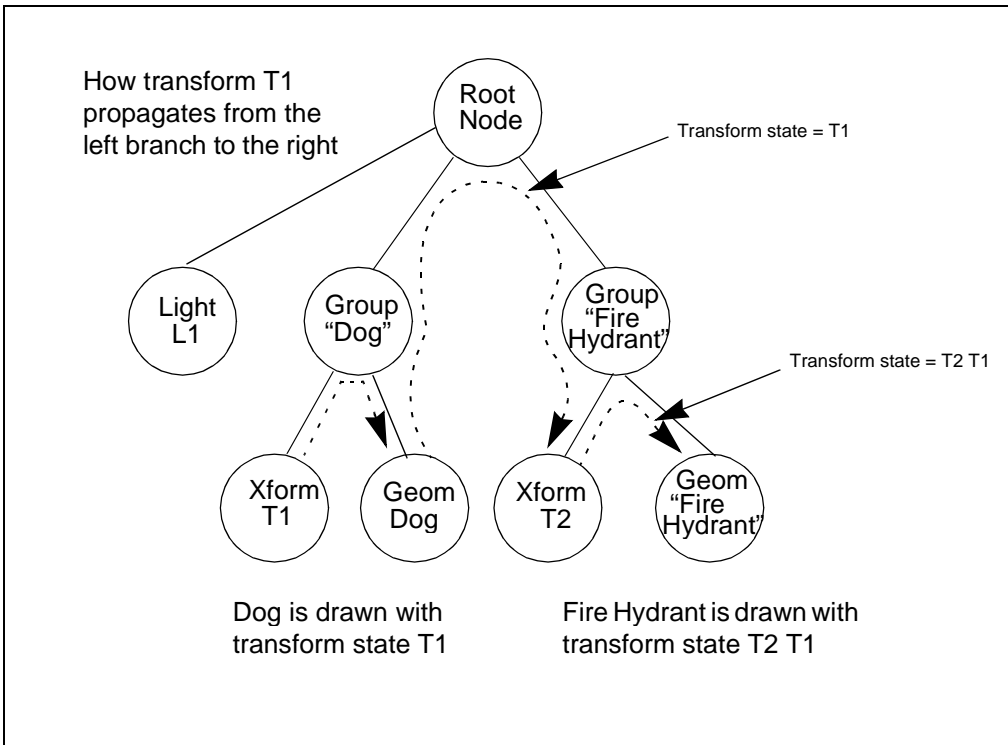


Figure 4-8: State propagation

Since you independently placed your geometries in the scene, you should expect to be able to move them in the scene independently of each other. To do this, simply modify the information contained in the transform nodes which affect the individual geometries. There are a number of WTK functions that allow you to do this, such as *WTnode_settranslation* (see page 4-59) and *WTnode_setrotation* (see page 4-60).

Assuming that you have set the dog's transform node to position it at the left of your scene and that you have set the fire hydrant's transform node to position it at the right of your scene, you can take a walk through the scene graph to make sure you have constructed it properly. The root node's first child is the light node, so now there is a single active light that will affect all succeeding geometry nodes.

The root node's second child is the dog object group node, so now walk down into the children of the group node. The first child is the dog's transform node, which you want to

use to place the dog somewhere in the scene. The current transform state is set to the information in the dog's transform node, as you move to the group node's second child, the dog's geometry node. This geometry of the dog itself is now drawn at the position defined by the dog's transform node, and lit by your one light. So far, so good. There are no more children in the dog group node, so you move back up the tree to the root node and continue to process the root node's children.

The root node's next and last child is the fire hydrant object's group node. You process the first child of this group node, the fire hydrant's transform node. It is important to remember that when WTK encounters a transform node, it takes the information in the node and concatenates it with the current transform state. In this case, the current transform state previously set by the dog's transform node has "propagated," so the resulting transformation state after processing the fire hydrant's transform node is an *accumulation* of both the dog's transform node and the fire hydrant's transform node. A discussion of how transforms are combined is given in *Using Frames of Reference (Coordinate Frames)* on page 4-32.

Assume, however, that you do not want the fire hydrant's transformation to be affected by the dog's transformation. The following section describes how you can prevent a transform node from affecting the remaining nodes in the scene graph.

When WTK processes the fire hydrant's second child, the geometry node, it draws this geometry not at the place defined by the fire hydrant's transform node as you intended, but instead at the place defined by the combination of the dog's transform *and* the fire hydrant's transform. Your scene graph containing objects with independent positions behaves as expected, because the first transform "propagates" and affects both the intended geometry, the dog, as well as succeeding geometries in the scene graph, the fire hydrant. The transform state at the point in the scene graph tree where the fire hydrant geometry exists is an "accumulation" of all the transform nodes before it.

How do you remedy this situation? Very simply. Instead of using group nodes, you can use a different type of node, the separator node. Replacing the group nodes with separator nodes solves the problem, because the separator node "encapsulates" all of the state changes underneath itself.

State Encapsulation

SEPARATOR NODE

As you have seen, transform nodes combine with the current states in the scene graph tree to set new values for the states. There are often times when you want to have a transform node affect only certain geometries in the scene graph. In order to do this, you must “encapsulate” the effects of a transform node to just one part of the scene graph tree. Continuing the example in the previous section, let’s say you wish to encapsulate the effects of the dog’s transform node so that it only affects the dog’s geometry. You need some way to prevent the transform state changes in the dog object’s branch of the scene graph tree from propagating over to the fire hydrant object’s branch. This is exactly what you use the separator node for.

The separator node is a grouping node that encapsulates all of the effects of the transform nodes, light nodes, and fog nodes beneath it, as shown in figure 4-9.

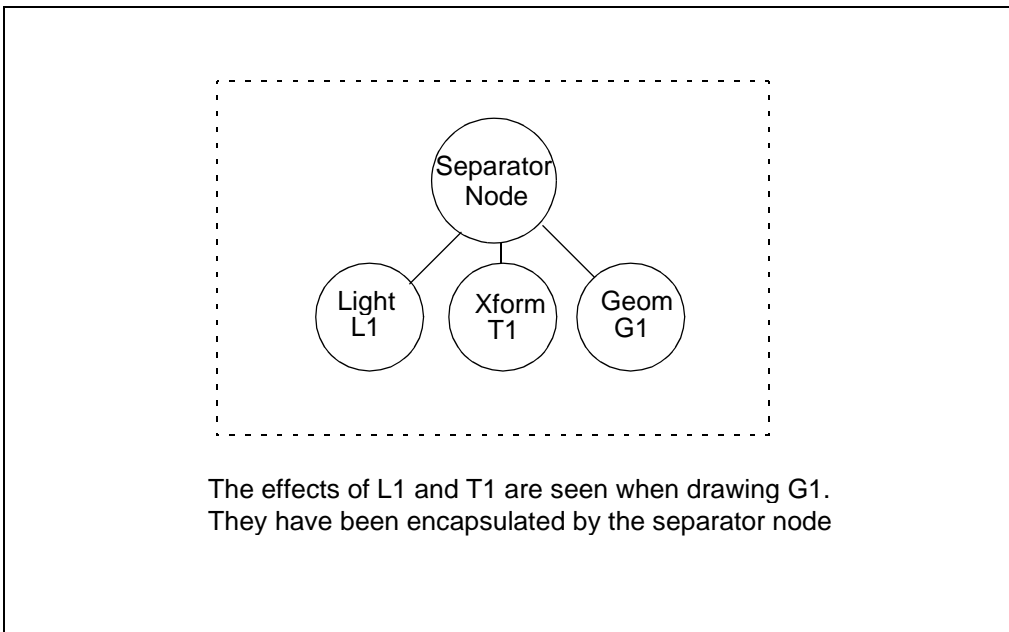


Figure 4-9: Separator nodes

Internally, the separator node makes copies of the current transformation, lighting and fog state, processes it and all of its children, and then restores those states to what they were prior to processing this portion of the scene graph. In other words, the transform, light and fog state changes which occur underneath the separator only affect the nodes underneath the separator, as shown in figure 4-10.

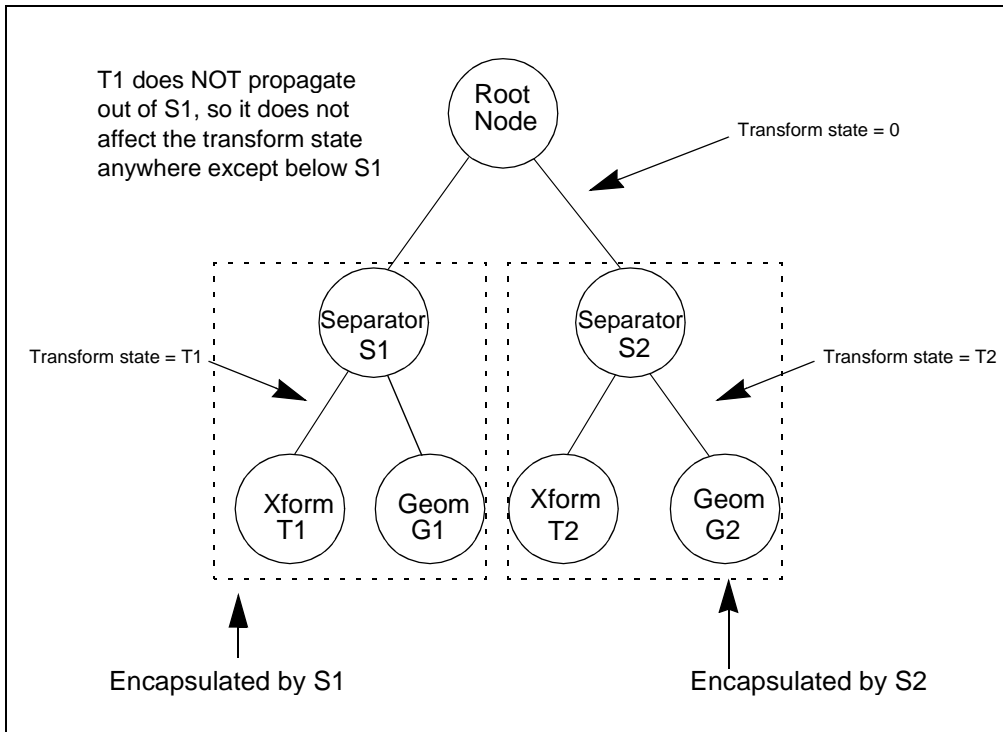


Figure 4-10: Separator diagram

As stated, you can use the separator node instead of the group node in cases where you want to prevent states from propagating across branches of the scene graph tree. You will use separators often to encapsulate transform nodes, *but when would you want to encapsulate light and fog nodes?*

If a separator node sits above a light node or fog node, it prevents this light or fog from affecting any other part of the tree beyond the nodes underneath the separator. Usually, you use a separator to constrain a light to only a certain group of geometries, having it light only those geometries and no others. Because lighting geometry is a computationally expensive

task, you are able to improve application performance by using as few lights as possible, and only having lights affect the specific geometries you need affected. Imagine a building, where each room may have several local lights, say point lights simulating lamps, or spot lights simulating track lighting. Because you only need and want lights in a particular room to light the geometry in that room, you can use a separator to group the lights and room geometry together, as shown in figure 4-11.

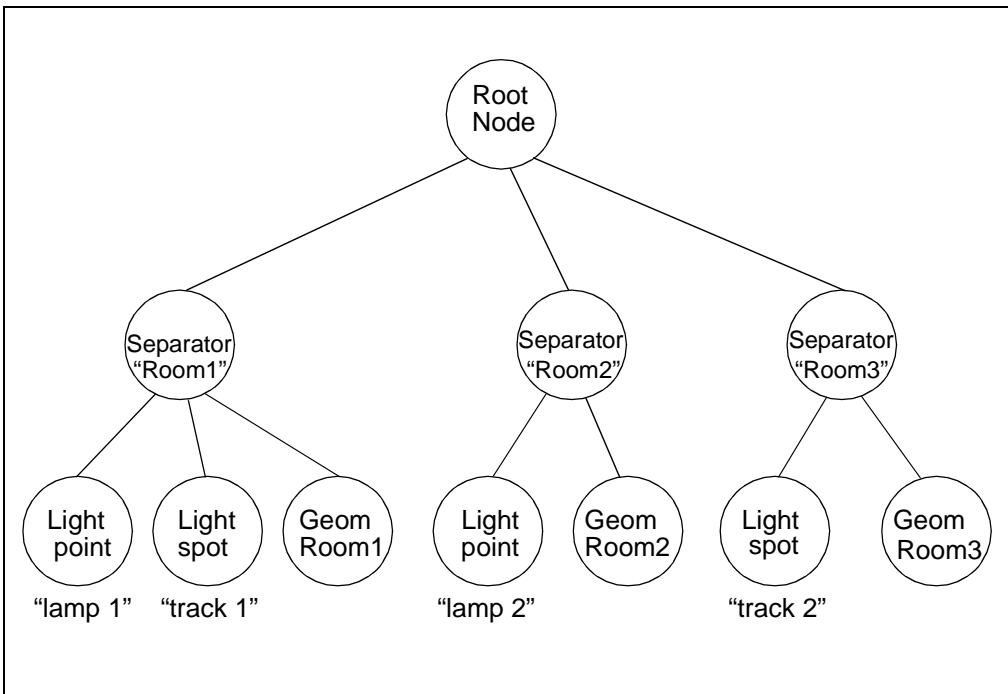


Figure 4-11: Light encapsulation

This results in both the desired effect of having lights localized to their respective rooms as well as improved performance since WTK only needs to apply a light to a specific number of polygons, rather than every polygon in the scene.

TRANSFORM SEPARATOR NODE

The transform separator node is exactly like the separator node, except it only encapsulates the transformation state, *not* the light or fog state. In other words, the effects of light nodes and fog nodes propagate out through the transform separator node, and only the transform node's effects are blocked by it. Generally, you use the transform separator node in conjunction with the pairing of a light node and transform node. For example, assume you want to have a light whose position/direction are controlled by a transform node. You want to have this light illuminate the whole scene, but you want to encapsulate the transform node, so it only affects the light's position/direction, and no other elements of the scene. A plain separator node would encapsulate the transformation, but it would also encapsulate the light, thus preventing the light from affecting any part of the scene. The solution is to use a transform separator node instead, allowing the encapsulation of the light's position/direction, but still allowing the light itself to add to the current light state, as shown in figure 4-12.

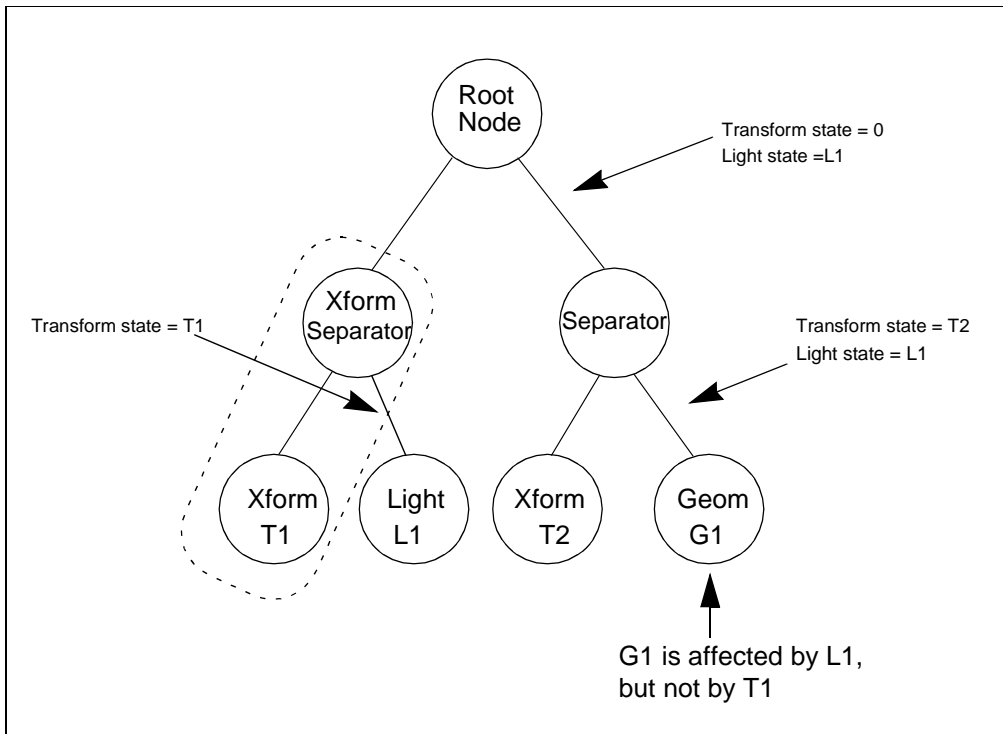


Figure 4-12: Transform (Xform) Separator

Other Node Types

This section describes the other scene graph node types that have yet to be defined.

SWITCH NODES

A switch node is a grouping node that enables only one of its child branches at a time, as shown in figure 4-13. The switch node can have multiple children, but only one of the children is active at any one time. When WTK traverses the scene graph tree and encounters a switch node, the switch node informs WTK which child branch to process. After processing this child branch, WTK traverses back out of the switch node, leaving all the other children unprocessed. You control which child is active via the call to `WTswitchnode_setwhichchild` (see page 4-57).

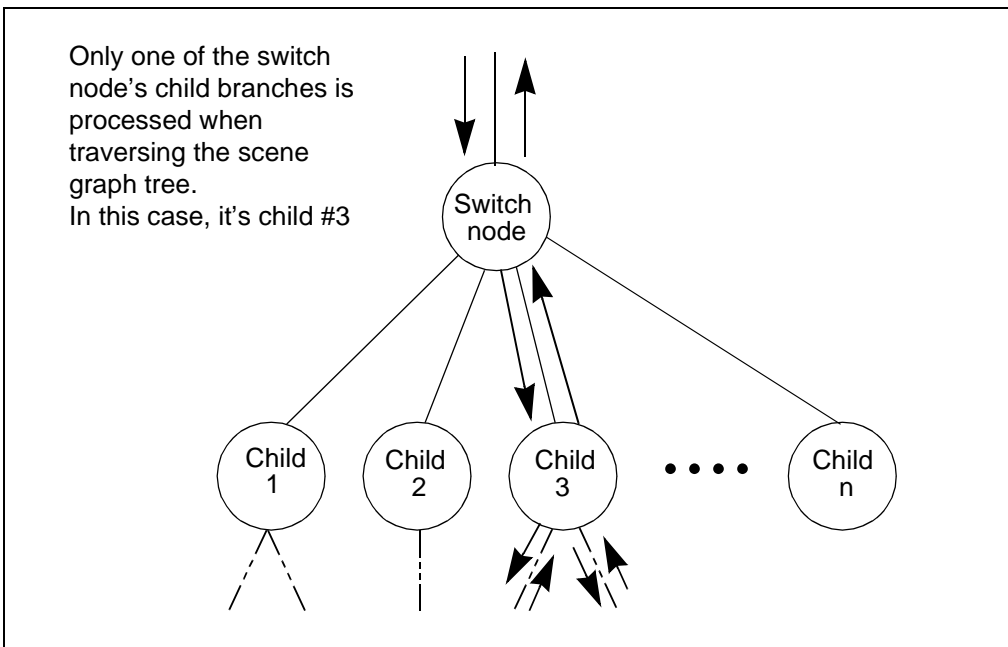


Figure 4-13: Switch node

Although you can duplicate the functionality of a switch node by using a group node and manually adding the active child and removing the previously active child, this is generally

both more work and results in poorer performance than using a switch node. Because there can be a significant amount of internal work involved in adding and removing children from the scene graph tree, it is best to use a switch node whenever it is applicable.

A switch node is useful in a number of situations, including multiple representations, geometry animations, and portalling, to name a few. Multiple representations means having several representations for a particular object, only one of which you want drawn at any one time. Take the simple example of a tank simulator which might have three different geometrical representations for an enemy tank, either undamaged, damaged or destroyed. You could use a switch node, having each geometry as its child. Changing the representation of the tank based on whether its been hit and/or destroyed is as simple as calling `WTswitchnode_setwhichchild` to choose the proper geometry.

Switch nodes are useful in any case where you might wish to do animation using a sequence of geometries rather than dynamically altering a single geometry at the vertex level. You can simulate a human walking by having multiple geometries, each being the same human geometrical model, but in a different sequential position of walking forward. Together, you can play the models back in sequence, like a flipbook animation, showing a human walking in place. You can define the sequence of geometries representing the human in the different walking positions as children of a switch node. Using a transform node to move the human forward in space, you can use the switch node to sequence through the different walking position models, simulating a human walking forward.

Portalling refers to jumping from one scene to another, or one part of the scene to another part. In a walkthrough environment, taking an elevator to another floor would be an example of portalling. You can build a scene graph that has a switch node near the top, and each floor of the building is a child branch of the switch node. When someone rides the elevator to a new floor, the proper child branch of the switch node is activated to draw that floor of the building. In cases like this, you definitely want to turn off any parts of the scene graph that you know are not visible, as this vastly increases performance. Switch nodes are a good way to disable the portions of a scene graph you know will not be visible.

LOD NODES

The level-of-detail (LOD) node is a specialized switch node that selects its currently active child automatically based on its distance from the viewpoint, as shown in figure 4-14. You use an LOD node to dynamically select between a set of different representations, each of which is a different level of detail. For example, suppose your application involves a train that passes close to the viewer and then recedes into the distance. After you create an LOD node, you would add several children to it — each of which is a less-detailed representation

of the same train. WTK allows you to specify the distance at which the LOD node could “swap in” a new representation. As the train recedes into the distance, simpler models (which require less and less computational effort to process) are progressively swapped in, freeing up memory and system resources.

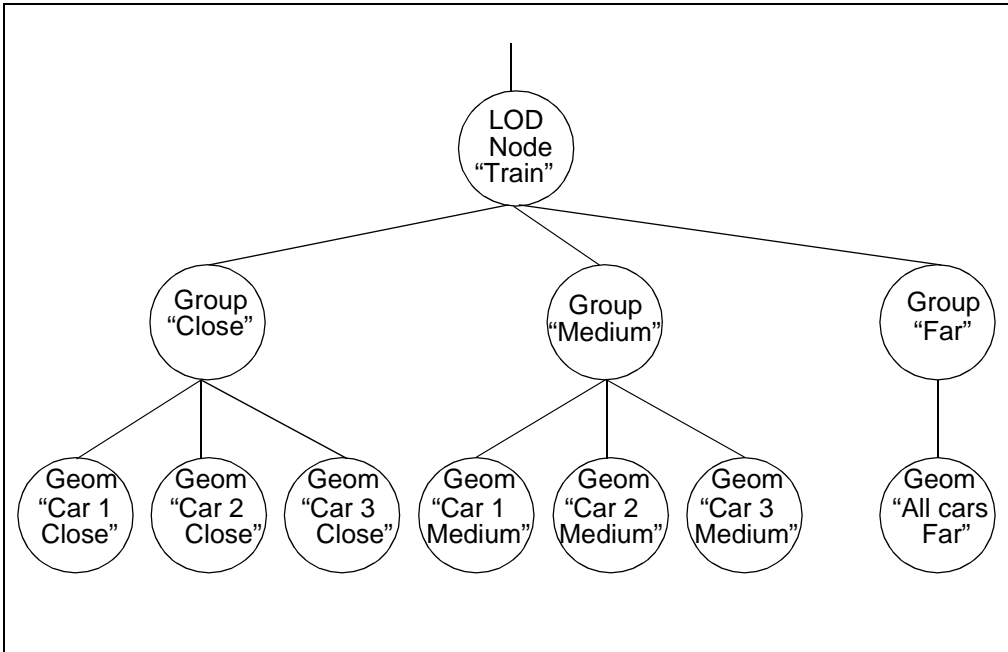


Figure 4-14: LOD nodes

Of course, your object doesn’t have to be moving — LOD nodes are useful whenever the distance varies between the viewpoint and a geometric object.

The LOD node allows you to specify the same object with varying level of detail. An LOD node’s children are arranged from highest level of detail (closest) to lowest level of detail (furthest away). In general you build geometries with a larger number of polygons (more detail) for the highest level of detail representation and you build geometries with fewer polygons for the lowest level of detail representation.

An LOD node automatically chooses among its child nodes based on the distance between the user’s viewpoint and the position you have designated as the *center* of your (LOD) representation. WTK computes the distance between the viewpoint and the position of the

center, compares that distance with the ranges that you have specified, and chooses the appropriate representation.

The range for an LOD node is an array of floats specifying the switch-out distances for the children of the LOD node. The input parameter *num* is the number of ranges passed in. There does not have to be a one-to-one correspondence between range values and number of child nodes. If there are more child nodes than range values, then the excess child nodes are never traversed. If there are more range values than child nodes, then range values that have no corresponding child nodes are not entered into the determination of which child node to traverse.

ANCHOR NODES

An anchor node is a group node which contains a string property indicating the path and filename of a VRML file that is associated with the node. However, an anchor node does not retrieve the file automatically. In order to retrieve the file, some sort of user action (e.g., a mouse click) is required to trigger the execution of a programmer defined action function which causes the file to be read.

The string property is a character string representing a URL (Uniform Resource Locator) and can be set using the functions *WTanchornode_setlocation* and *WTinlinenode_setlocation*. The default URL of an anchor node is NULL. You can set or retrieve the anchor string (URL) corresponding to the anchor node using the functions, starting page 4-63.

INLINE NODES

An inline node is a group node whose children are read from a file without user interaction. An inline node contains a string property indicating the path and filename of a VRML file that is associated with the node. You set or retrieve the inline string (URL) corresponding to the inline node using the functions starting on page 4-63.

Note that an inline node's children are only read into WTK when inline nodes are created by reading VRML (.wrl) files. If you manually create an inline node by calling *WTinlinenode_new* (see page 4-40), and then call *WTinlinenode_setlocation* (see page 4-63) to set the node's string property, WTK will not read the inline node's children into the scene graph. Therefore, you would have to manually create the nodes representing the inline node's children.

When a scene graph is written out to a VRML file using *WNode_save* (see page 4-48), WTK outputs the inline node along with the string property but does not write out the inline node's children.

Building a Scene Graph

This section offers a general discussion on how to build a scene graph, and explains several important concepts related to building scene graphs.

How to Create the Scene Graph Tree

There are several ways to build a scene graph. You can build it dynamically, as discussed in the example on page 4-18, you can load it directly from a file, or you can use a combination of both methods. The hierarchical file formats that WTK reads are VRML and MultiGen. You can use *WNode_load* (see page 4-46) on a VRML *.wrl* file or MultiGen *.flt* file to load in the entire scene graph sub-tree stored in the file and attach it to the WTK scene graph tree at the point specified by the “parent” argument field. Inline nodes that are part of a VRML file will also be loaded in and added to the scene graph tree.

When you build and modify a scene graph tree, there are several issues you need to be aware of.

As discussed on page 4-7, every scene graph requires a starting point, a node that is unique to its specific scene graph tree, the root node. The root node is the only node that cannot be shared between multiple scene graphs; its only purpose is to serve as the unique parent to all the top nodes of a particular scene graph. Once you have a root node, either using the default root node created by WTK (accessed by calling *WUniverse_getrootnodes*) or creating a new root node via the function call *WRootnode_new* (see page 4-39), you are ready to begin building your scene graph tree.

It is important to keep in mind that the scene graph tree must be acyclic, that is, no cycles are allowed in the tree structure. A cycle occurs when a node is an ancestor of itself. WTK disallows this because it becomes impossible for it to traverse a cyclic tree without going into an infinite loop, as shown in figure 4-15. In fact, WTK prevents you from adding a child node if it causes a cycle in the scene graph tree.

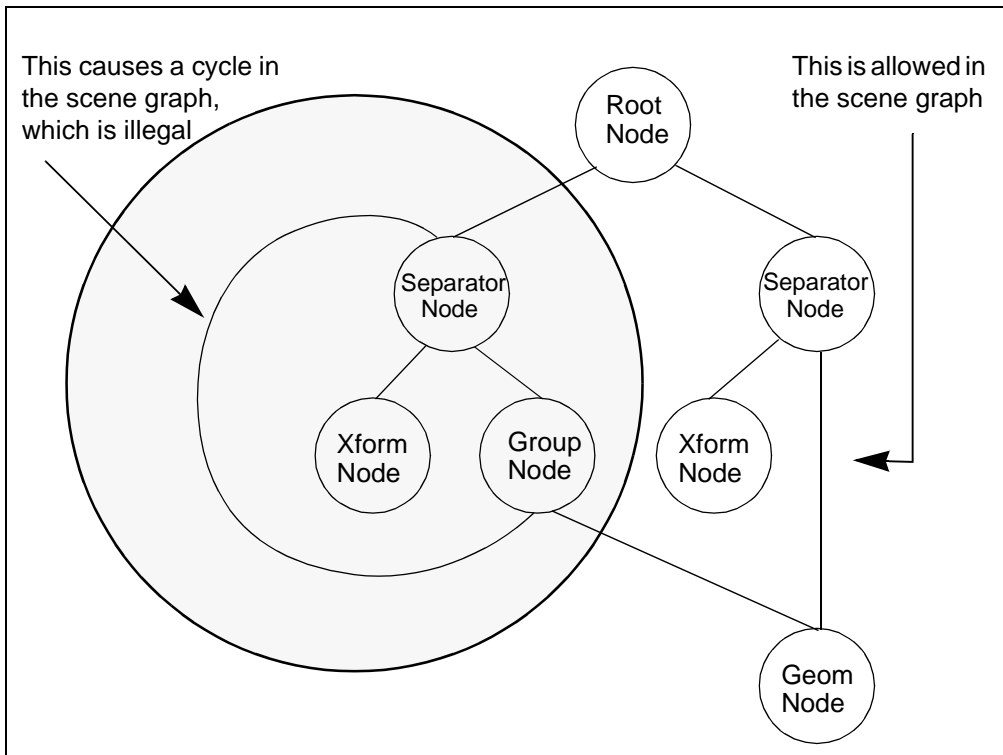


Figure 4-15: Cyclic scene graph tree

Building a Composite Object in the Scene – Composite Transformations

Accumulated transformation occurs when the current transformation state in a scene graph tree is an accumulation of all the transform nodes processed before the current traversal position in the tree. By using separator nodes or transform separator nodes, you are able to prevent accumulated state from propagating from sibling branch to sibling branch, as in the case of the dog and fire hydrant example (see page 4-19).

In general however, you *do* want the transformation state to accumulate as WTK walks down any particular branch of the scene graph tree. One of the most important reasons to

have a hierarchical structure to hold your scene data is to help define the composite positional and orientational relationships between objects in the scene.

It is often the case that you want to build a composite object in the scene. This composite object is treated as a single object in relation to the rest of the scene, however its individual parts are considered to be a collection of distinct objects in relation to the composite object. A common example of this would be a composite car, which would be built from five parts: the car body, the left front wheel, the right front wheel, the left rear wheel, and the right rear wheel, as shown in figure 4-16. When the composite car moves forward, all of its parts move forward together as a whole (i.e., as the car moves forward 20 feet, each part of the car moves forward exactly the same amount, 20 feet.) You can think of this as a frame of reference. All of the parts of the car are in a composite car frame of reference. When the composite car frame of reference moves forward, all the sub-parts of the car also move forward.

The composite car can also be thought of as having sub-frames of reference, that is, there are parts of the composite car which move individually in relation to the whole car, as well as moving in conjunction with the whole car. The wheels are an example of a sub-frames of reference for the composite car. Although they move forward when the whole car moves forward, they also rotate around the axles individually when the car rolls forward. The orientation and position of a wheel is a combination of its individual frame of reference (which rotates about the axle) and the composite car frame of reference (which moves forward, backward, etc., in the scene).

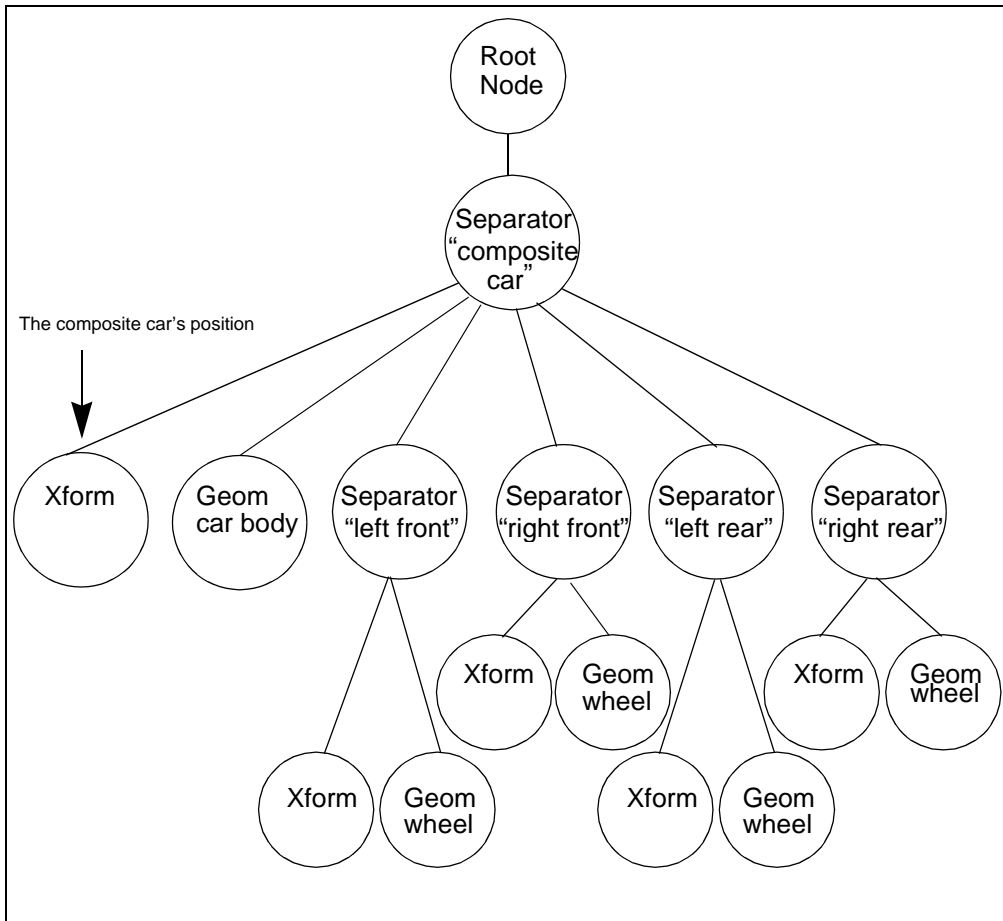


Figure 4-16: Composite car

USING FRAMES OF REFERENCE (COORDINATE FRAMES)

In building scene graphs and composite objects, it is important to understand the concept of *frames of reference* (or coordinate frames). This section discusses this concept in more detail.

Since the current transformation state at any place in the scene graph defines a unique frame of reference, as WTK traverses down the scene graph tree and encounters transform nodes,

these nodes change the current transformation state and thus change the current frame of reference in which WTK draws the ensuing geometry. A transform node essentially defines a relationship between the frame of reference (transformation state) prior to it being processed and the frame of reference after it is processed.

Just as transformations accumulate as you traverse down the scene graph tree, frames of reference accumulate as you walk *up* the tree. From any point in the scene graph, when you walk up the tree (opposite from how WTK actually traverses the scene graph), you encounter transform nodes in the reverse order as you would encounter in a normal traversal. You can view each transform node in a reverse walk as a change in coordinate systems.

This may seem complicated, however, the concept is important. The purpose of a transformation is to place and orient an object (geometry or light) in the scene *or* to place and orient an object in the scene *relative* to another object. Usually, you do not care exactly where an object is in the scene, only where it is in relationship to some other object.

Consider the previous example of the composite car made from four wheels and a body, and suppose you break down the transformations you use to place the components together and then move them as a whole. Assume you do not care exactly where the wheels are in the scene, only that they are placed in their proper positions relative to the body of the car. In other words, you don't want to specify the wheels' position relative to the whole scene's frame of reference, but only their position relative to the composite car's frame of reference. In this case, you would want to use another transformation to specify the composite car's frame of reference to the scene.

Assume you want your composite car to move around the scene, and the wheels to roll as your car drives along.

What are all the coordinate frames involved here?

From the scene's point of view, there is only one coordinate frame, the base coordinate frame that exists when no transformations have been applied. This coordinate frame is WTK's world coordinate frame, and defines the three dimensional space in which the entire scene exists. With no transformations applied, geometries are drawn in the world coordinate frame. Move down the scene graph to your first encapsulated object, the composite car. The transformation associated with the composite car defines how the composite car will be placed into its parent's coordinate system.

What is the composite car's parent coordinate system?

It is the coordinate system defined by the current transformation state before processing the composite car's transform node, the *world coordinate system*. So from the composite car's point of view, there are two coordinate systems, its *own (local) coordinate system*, which is the coordinate system all of its parts will be placed into, and its *parent's coordinate system*, where it will be placed after its transform node is processed. *So why is this useful?* You can assemble all of the car's parts together in the composite car's local coordinate system, and then move them together as a whole using the composite car's transform node.

So, you can use a transform node to move all the parts assembled in an object's local coordinate system to some place in the object's parent coordinate system. In this case, you have placed (drawn) the car body in the composite car's coordinate system.

Assume you modeled the car body in a way such that the origin (0,0,0) is at the center of the car body. Also assume that you have modeled the wheel geometries at the origin, such that the center axis point on the wheel is at (0,0,0). Obviously if you put the wheels into the composite car's coordinate frame without transforming them first, they will all end up in the same place, at the center of the car body, not where you want them. What you need to do is add a transform node for each wheel that translates it from its local coordinate system (where 0,0,0 is its center) to the proper place in its parent coordinate system (the composite car, where 0,0,0 is in the middle of the car). So from a wheel's point of view, there are three coordinate systems, its local coordinate system, its parent coordinate system, and the world coordinate system. In general, you will only be concerned with the first two coordinate systems. However, there will be cases when you are interested in an object's position relative to the scene as a whole (world coordinate system), particularly when you are interested in positional relationships between two independent objects. You now have an assembled car in its coordinate system, as shown in figure 4-17.

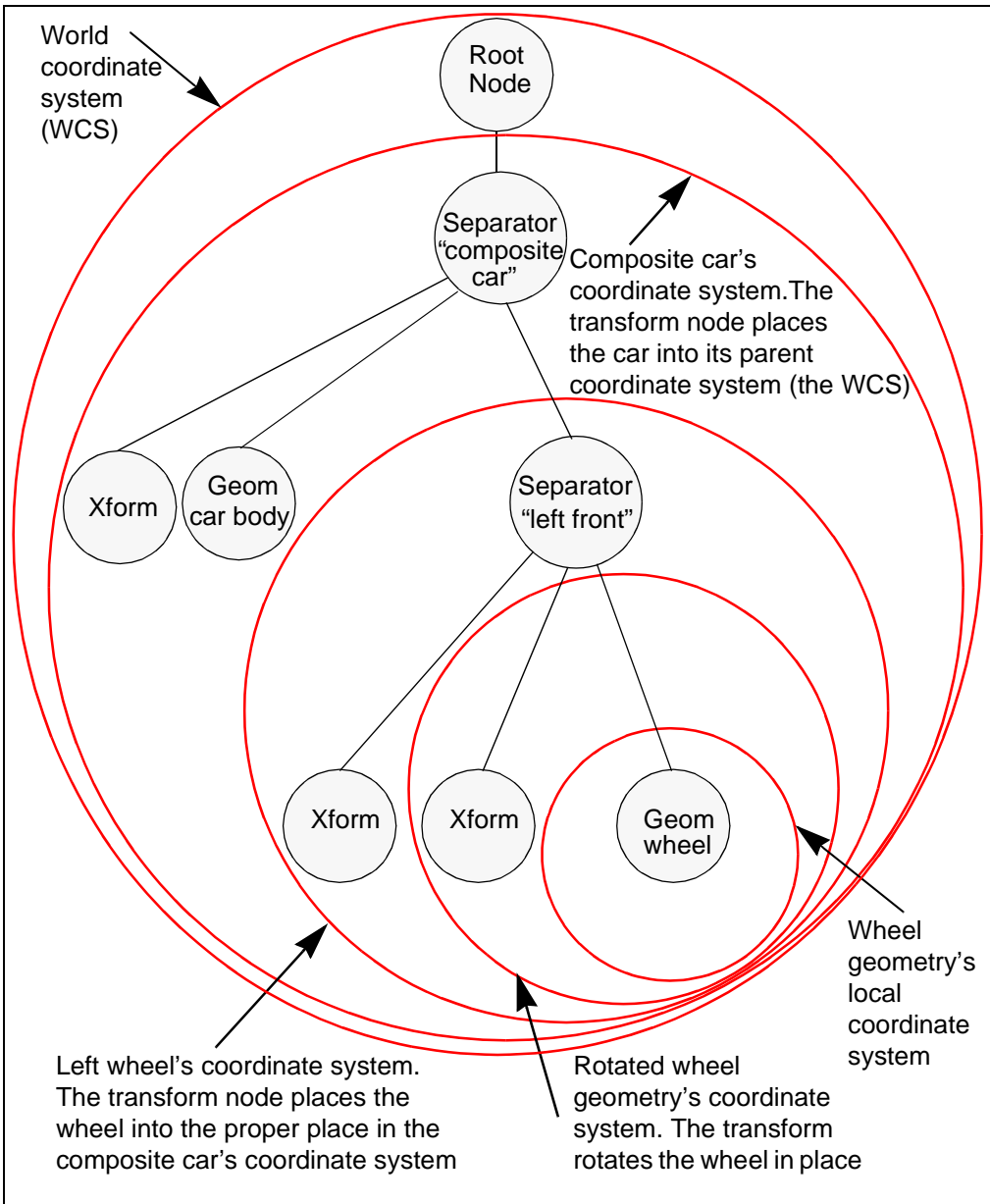


Figure 4-17: Car's frames of reference

You can move the car about the scene using its transform node. Now, you want to have the wheels roll (rotate) when you move the car along. Rolling is simply rotation about the wheel's center axis, and since you have built this car using several different coordinate systems, it will be very easy to make these wheels rotate as you move the car about. Remembering that you have modeled the wheels with their center axis point being at (0,0,0), all you have to do is rotate the wheel within its local coordinate system prior to transforming it into its parent coordinate system. Although you can combine this rotation with the transformation into the composite car coordinate system, for simplicity, just insert another transform node to hold this rotation.

Assuming you rotate each wheel slightly about its local center axis each frame, then across multiple frames, you will see the wheels rotating in place. Mixing this with the translation of the composite car along a single direction, you are able to simulate a car rolling along. As WTK traverses down the scene graph, it preconcatinates each transform node's internal 4x4 matrix with the current transform state's internal 4x4 matrix, forming a new current transform state, as shown in figure 4-18.

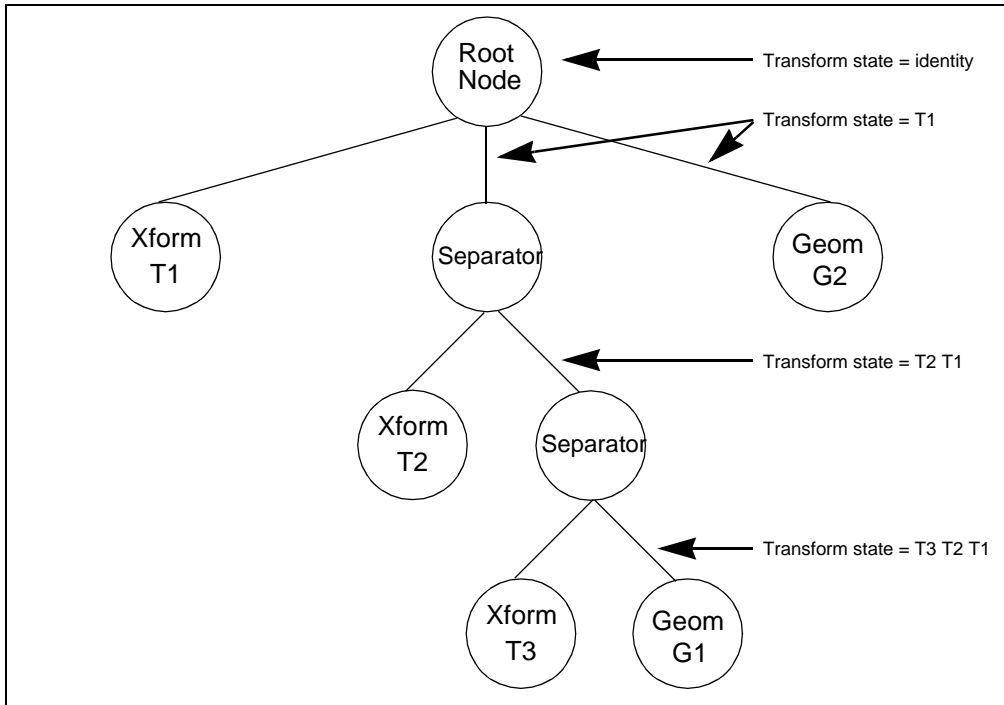


Figure 4-18: Internal matrix

ADDING A NODE TO YOUR SCENE MULTIPLE TIMES – INSTANCING

You can add a single node to a scene graph multiple times; this is known as *instancing*. Instancing is another important concept to understand when building your scene graph. When you add a node to a scene graph multiple times, WTK creates a reference to the original node and then adds that reference to the node at the point of insertion. It does not copy this node internally, thus significantly saving memory for every additional reference to a node in a scene graph(s).

There are a number of cases where instancing can significantly improve performance. For example, suppose you have a terrain with trees on it. Assuming the trees on this terrain are identical in shape and size, then this is an ideal situation to use instancing. Instead of having a separate transformation and a separate geometry for each tree, you instead have only a separate transformation for each tree and a single instance of the tree geometry, as shown in figure 4-19.

Each instance has a unique position in the scene graph. The route (in the scene graph) to that position is called a *node path*. See page 4-79 for a description of node paths.

Instancing obviously saves memory usage, particularly when you can instance a large number of geometries. Instancing also improves performance in such situations by significantly improving data cache hit ratios, thereby decreasing memory usage requirements. This can be critical when running on machines that do not have a large amount of physical RAM.

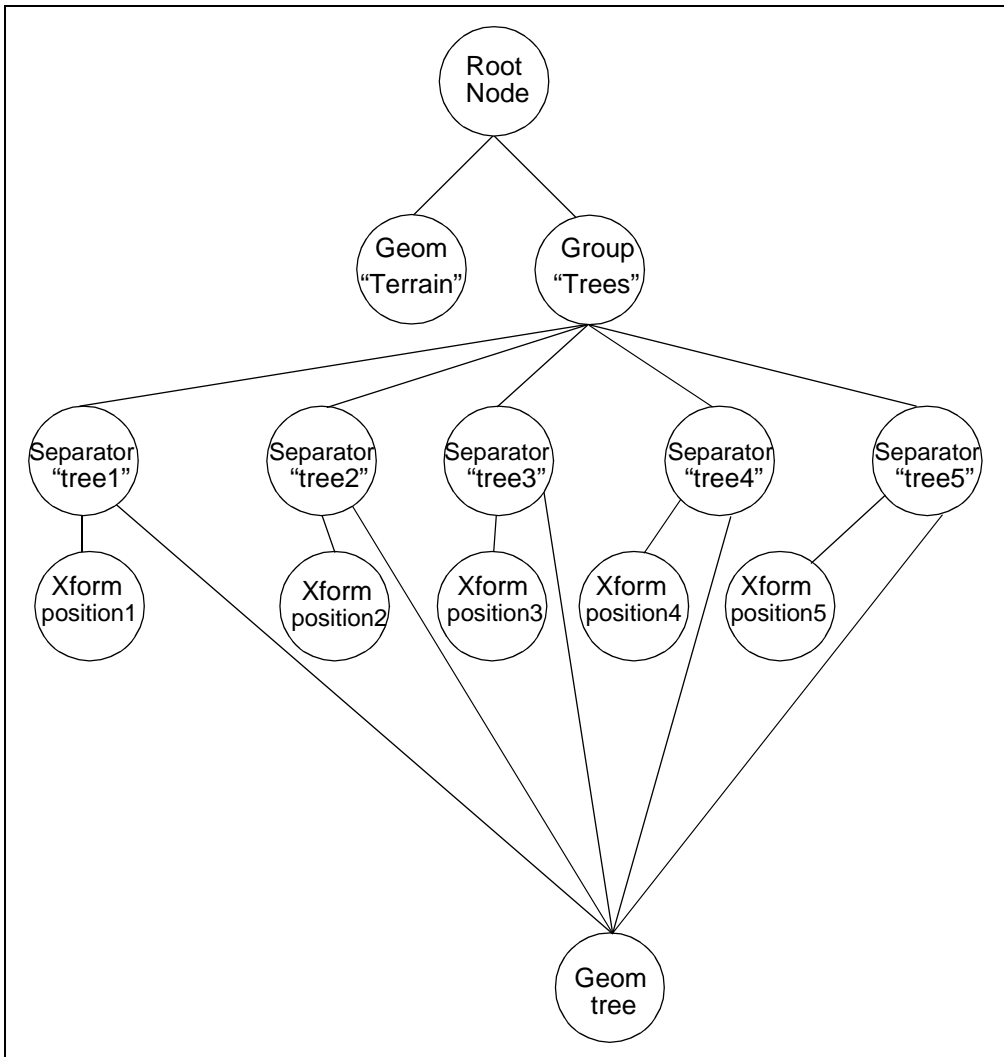


Figure 4-19: Instancing

WTK Scene Graph Functions

This section lists descriptions of all of the WTK scene graph functions.

Constructing Node Types

Each of the node types listed in table 4-1 and table 4-2 has a corresponding constructor function. For example, to create a new group node, you call *WTgroupnode_new*; to create a new transform node, you call *WTxformnode_new*. The new node is attached to the graph below the specified parent, after the last child already attached to this same parent. This section lists the WTK functions that you use to create node types.

WTrootnode_new

```
WTnode *WTrootnode_new(  
    void);
```

This function creates a new root node (and therefore a new scene graph). This root node constructor function is different from other node constructor functions in that it does not have an argument for specifying a parent node. This is because the root node, as the top node in the scene graph, has no parent node. The root node is the only node without a parent.

Each scene graph has only one root node. When you create a new root node, you are creating a new scene graph.

If your application requires only a single scene graph, then it is not necessary to call this function in your application. This is because *WTuniverse_new*, which is called at the beginning of every WTK application, automatically creates an initial root node. A pointer to this root node can be obtained by calling *WTuniverse_getrootnodes*.

For a scene graph to be rendered into a WTK window (see the *Windows* chapter, starting on page 17-1), the root node of the scene graph must be associated with the window using the function *WTwindow_setrootnode*. Note that *WTuniverse_new* automatically associates the initial root node created by *WTuniverse_new* to each of the windows created by *WTuniverse_new*.

See also *WTwindow_setrootnode* on page 17-8. Note that root nodes cannot be deleted via calls to *WTnode_delete*.

WTanchornode_new

```
WTnode *WTanchornode_new(  
    WTnode *parent);
```

This function creates an anchor node and adds it to the scene graph after the last child of the specified parent. If NULL is specified for the parent argument, then the node is created without a parent. Such nodes can be added to the scene graph by calling *WTnode_addchild* or *WTnode_insertchild*.

An anchor node is a group node which contains a string property (URL) used to retrieve a file. However, an anchor node does not retrieve the file automatically. In order to retrieve the file, some sort of user action (e.g., a mouse click) is required to trigger the user-defined action function that causes the file to be read. The default URL of an anchor node is NULL. See *WTanchornode_setlocation* on page 4-63 to set an anchor node's URL.

WTgroupnode_new

```
WTnode *WTgroupnode_new(  
    WTnode *parent);
```

This function creates a group node and adds it to the scene graph after the last child of the specified parent. If NULL is specified for the parent argument, then the node is created without a parent. Such nodes can be added to the scene graph by calling *WTnode_addchild* or *WTnode_insertchild*.

A group node is a node which can have children but has no other special properties. This would be useful if your application involved a set of geometries which needed to be treated as a single entity.

WTinlinenode_new

```
WTnode *WTinlinenode_new(  
    WTnode *parent);
```

This function creates an inline node and adds it to the scene graph after the last child of the specified parent. If NULL is specified for the parent argument, then the node is created without a parent. Such nodes can be added to the scene graph by calling *WTnode_addchild* or *WTnode_insertchild*.

An inline node is a group node which contains a string property (URL) representing the name of a file from which the inline node's children are read *without user interaction*. Note that an inline node's children are only read into WTK when inline nodes are created by reading VRML (.wrl) files. If you manually create an inline node by calling *WTinlinenode_new*, and then call *WTinlinenode_setlocation* to set the node's string property, WTK will not read in the inline node's children into the scene graph. Therefore you would have to manually create the nodes representing the inline node's children. When a scene graph is written out to a VRML file using *WTnode_save*, WTK will output the inline node along with the string property but does not write out the inline node's sub-tree. The default URL of an inline node is NULL. See *WTinlinenode_setlocation* on page 4-63 to set an inline node's URL.

WTlodnode_new

```
WTnode *WTlodnode_new(  
    WTnode *parent);
```

This function creates an LOD (Level of Detail) node and adds it to the scene graph after the last child of the specified parent. If NULL is specified for the parent argument, then the node is created without a parent. Such nodes can be added to the scene graph by calling *WTnode_addchild* or *WTnode_insertchild*.

An LOD node is used to dynamically select between different representations, each of which is a different level of detail, as a function of viewpoint distance. See *LOD Nodes* on page 4-26 for more information.

WTsepnode_new

```
WTnode *WTsepnode_new(  
    WTnode *parent);
```

This function creates a separator node and adds it to the scene graph after the last child of the specified parent. If NULL is specified for the parent argument, then the node is created without a parent. Such nodes can be added to the scene graph by calling *WTnode_addchild* or *WTnode_insertchild*.

A separator node prevents the state information from propagating from its descendent nodes to its sibling nodes.

Separator nodes also allow for a quick-reject test to be performed on the extents box of the separator node's sub-tree. When the simulation is run, if an extents box lies outside the area that is being viewed, then the sub-tree is not visible and is therefore not traversed (or rendered). Using separator nodes and their quick-reject test capability can drastically improve the performance of your simulation. For more information on the quick reject test, see *Separator Node Functions* on page 4-56.

WTswitchnode_new

```
WTnode *WTswitchnode_new(  
    WTnode *parent);
```

This function creates a switch node and adds it to the scene graph after the last child of the specified parent. If NULL is specified for the parent argument, then the node is created without a parent. Such nodes can be added to the scene graph by calling *WTnode_addchild* or *WTnode_insertchild*.

A switch node controls which of its several children is to be processed. By default, none of the children of a switch node is processed. See *WTswitchnode_setwhichchild* on page 4-57 to select which child of a switch node gets processed.

WTxformnode_new

```
WTnode *WTxformnode_new(  
    WTnode *parent);
```

This function creates a transform node and adds it to the scene graph after the last child of the specified parent. If NULL is specified for the parent argument, then the node is created without a parent. Such nodes can be added to the scene graph by calling *WTnode_addchild* or *WTnode_insertchild*.

A transform node provides position and orientation information which can affect subsequent geometry and light nodes.

By default, a transform node's matrix is set to the identity matrix. An identity matrix (for the purpose of matrix multiplication) is identical to the number 1 (one) for numeric multiplication. An identity matrix is shown below.

```
[1.0 0.0 0.0 0.0  
0.0 1.0 0.0 0.0  
0.0 0.0 1.0 0.0  
0.0 0.0 0.0 1.0]
```

Note that by default, WTK ignores scaling factors (if any) within a Transform (and Movable) node's transformation. If you want WTK to use the scaling factors of transformations within transform and movable nodes, you can do so by setting the *WTOPTION_XFORMSCALE* option in *WTuniverse_setoption*. However, by doing so, it is likely that intersection tests and math functions pertaining to matrices will operate incorrectly.

WTxformsepline_new

```
WTnode *WTxformsepline_new(  
    WTnode *parent);
```

This function creates a transform separator node and adds it to the scene graph after the last child of the specified parent. If NULL is specified for the parent argument, then the node is created without a parent. Such nodes can be added to the scene graph by calling *WTnode_addchild* or *WTnode_insertchild*.

A transform separator node prevents just the transformation state information from propagating from its descendent nodes to its sibling nodes. All other state is allowed to propagate.

Constructing Light Nodes

This section lists the functions you use to create light nodes.

WTlightnode_newdirected

See *WTlightnode_newdirected* on page 12-6.

WTlightnode_newspot

See *WTlightnode_newspot* on page 12-8.

WTlightnode_newpoint

See *WTlightnode_newpoint* on page 12-7.

WTlightnode_newambient

See *WTlightnode_newambient* on page 12-5.

Constructing Geometry Nodes

This section gives descriptions of the functions you use to create geometry nodes.

WTgeometrynode_new

```
WTnode *WTgeometrynode_new(  
    WTnode *parent,  
    WTgeometry *geom);
```

This function creates a geometry node with the specified geometry and adds it to the scene graph after the last child of the specified parent. If `NULL` is specified for the parent argument, then the node is created without a parent. You can add such nodes to the scene graph by calling *WTnode_addchild* or *WTnode_insertchild*.

Note that you can only create one geometry node for a particular geometry, i.e., WTK does not allow multiple geometry nodes to be created from the same geometry. Of course, you can instance (see page 4-37) the geometry node multiple times in the scene graph and you can also create movable instances of a geometry node. For more information refer to the *Geometries* chapter and/or the *Movable Nodes* chapter.

WTnode_getgeometry

```
WTgeometry *WTnode_getgeometry(  
    WTnode *node);
```

This function returns a pointer to the *WTgeometry* referenced by the specified geometry node.

Constructing Movable Nodes

This topic is discussed in the *Movable Nodes* chapter (starting on page 5-1).

Constructing Fog Nodes

WTfognode_new

```
WTnode *WTfognode_new(  
    WTnode *parent);
```

This function creates a fog node and adds it to the scene graph after the last child of the specified parent. If NULL is specified for the parent argument, then the node is created without a parent. These nodes can be added to the scene graph by calling *WTnode_addchild* or *WTnode_insertchild*. Also see *Fog Node Functions* on page 4-64.

Loading a File into a Scene Graph

In addition to providing low-level functions to create individual nodes and manually assembling a scene graph, WTK also provides high-level automatic methods to load in hierarchical data from a file directly into WTK's scene graph structure.

WTnode_load

```
WTnode *WTnode_load(  
    WTnode *parent,  
    char *filename,  
    float scale);
```

This function creates one or more nodes from data read in from a file, and adds these nodes to the scene graph after the last child of the specified parent. The data read in from the file may contain geometry data or data which corresponds to any of the supported node types.

If the specified data file is organized in a hierarchical fashion, then this function creates a node that corresponds to each data construct in the file, and adds the top-most node of the hierarchy to the WTK scene graph after the last child of the specified parent and returns the top-most node created. If the data is in a “flat” (non-hierarchical) file, each node that is created to correspond to each data construct is added to the scene graph after the last child of the specified parent. In this case, the function returns the first node that was created.

The scale parameter is used to scale the coordinates of geometries contained in the specified file. If you do not wish to scale the file’s geometries, pass in 1.0 as the scale value.

A geometry node is created for each geometry contained in the file; the node name assigned to each geometry node is taken from the name of the corresponding geometry in the file.

Note: The argument filename is a string that specifies the name of the file from which the data is read. This file could be on your local system (in which case you specify the path to it), or it could be a URL. If you are using a URL to read in data, the file name should contain the full http address (e.g., <http://www.sense8.com/models/oplan.wrl>).

WTK supports http URLs to VRML files only. The WTnode_load function does not support any other file type by way of a URL. Make sure your system has an http server if you intend on using URLs in the filename argument.

WTgeometrynode_load

```
WTnode *WTgeometrynode_load(  
    WTnode *parent,  
    char *filename,  
    float scale);
```


This function creates a single geometry node from data read in from a file, and adds the newly created node to the scene graph after the last child of the specified parent.

The data read in from the file must contain only geometry data; the only file formats which can be processed by this function are the following:

- 3DS (3D Studio)
- BFF (SENSE8)
- DXF (AutoCAD)
- GEO (VideoScape)
- NFF (SENSE8)
- OBJ (Wavefront)
- SLP (ProEngineer “RENDER”)

These formats may contain one or more geometric objects which are incorporated into a single geometry node. The scale parameter is used to scale the coordinates of geometries contained in the specified file. If you do not wish to scale the file’s geometries, pass in 1.0 as the scale value. You cannot use *WTgeometrynode_load* to read file formats such as FLT (MultiGen) or WRL (VRML) because those file formats are hierarchically organized and contain non-geometric information. Use *WTnode_load* to read FLT and WRL files.

WTurl_download

```
char *WTurl_download(  
    char *url,  
    char *localfile);
```

This function copies a file from an http server to a file on the local machine. The *url* argument takes the form “http://...”, or “file://...”. The *localfile* argument is the full path and file name of the file to be copied onto the local machine. If this function is successful in copying the file to the local machine, the return value of this function will be the full path and file name of the copied file. If this function fails to successfully copy the file, NULL is returned.

Saving a Scene Graph

WTnode_save

```
FLAG WTnode_save(  
    WTnode *node,  
    char *filename,  
    WTviewpoint *view,  
    int filetype,  
    int options);
```

This function saves the specified node to a file. If you save it to the VRML format (.wrl files) the node and its sub-tree are saved. The filename and filetype are specified by the parameters *filename* and *filetype*. Valid filetypes are:

- *WTFILETYPE_NFF*
- *WTFILETYPE_BFF*
- *WTFILETYPE_DXF*
- *WTFILETYPE_WRL*

The *options* parameter must be set to 0 (zero) for all filetypes. You can also specify a viewpoint (with the view parameter) to be saved with the node.

Note: Only geometry nodes can be saved to *NFF*, *BFF*, and *DXF* files. To save a sub-tree, use the *WRL* filetype.

This function returns *FALSE* if the filetype is not *WTFILETYPE_WRL* and the node is not a geometry node.

Node Property Functions

Certain node properties are generic — they can pertain to all node types. These properties include the name of the node, the node type, and any tasks assigned to nodes. Other node properties are specific to the type of node being considered. For example, Level of Detail switching information is stored only in LOD nodes, while position and orientation information is stored only in transform nodes. This section gives descriptions for the node property functions.

WTnode_setname

```
FLAG WTnode_setname(  
    WTnode *node,  
    char *name);
```

This function sets the name of the specified node. All nodes have a name; by default, a node's name is "" (i.e., a NULL string). More than one node can have the same name.

WTnode_getname

```
char *WTnode_getname(  
    WTnode *node);
```

This function returns the name of the specified node.

WTuniverse_findnodebyname

```
WTnode *WTuniverse_findnodebyname(  
    char *name,  
    int num);
```

This function finds the numbered occurrence of a specified node. If no nodes have the specified name, or if there are fewer nodes with the specified name than the number passed in as *num*, then NULL is returned. If more than one node has the same name, and *num* is 0, then a pointer is returned to the most recently created node with that name. See *How Do I Get A Pointer To A Node Using Its Name?* on page A-20 for an example of when to use this function.

WTnode_enable

```
FLAG WTnode_enable(  
    WTnode *node,  
    FLAG flag);
```

This function enables or disables the specified node during rendering or traversal of the scene graph. Valid node types are geometry, separator, transform separator, light, fog, and ambient.

The default value of the enable flag for all nodes is enabled (TRUE). If a node's enable flag has been set to FALSE, the node is disabled, meaning that it will be ignored during a rendering or picking traversal. The node enable flag does not affect intersection testing nor the values returned by any of the following functions: *WTnode_getradius*, *WTnode_getmidpoint*, or *WTnode_getextents*. Active tasks associated with a disabled node are still active.

WTnode_isenabled

```
FLAG WTnode_isenabled(  
    WTnode *node);
```

This function indicates whether a specified node is enabled (or disabled) for rendering and picking traversals.

WTnode_ismovable

```
FLAG WTnode_ismovable(  
    WTnode *node);
```

This function returns TRUE if the specified node is a movable node, otherwise it returns FALSE. See the *Movable Nodes* chapter (starting on page 5-1) for more information about movable nodes.

WTnode_gettype

```
int WTnode_gettype(  
    WTnode *node);
```

This function returns the type of the specified node. The node types supported are described in table 4-1 on page 4-11 and table 4-2 on page 4-13. The functions used to construct these nodes are described under *Constructing Node Types* on page 4-39.

Possible return values are the following defined constants: *WTNODE_ANCHOR*, *WTNODE_FOG*, *WTNODE_GEOMETRY*, *WTNODE_GROUP*, *WTNODE_INLINE*, *WTNODE_ILLEGAL*, *WTNODE_LOD*, *WTNODE_LIGHT*, *WTNODE_MGEOMETRY*, *WTNODE_MLOD*, *WTNODE_MSEP*, *WTNODE_MSWT*, *WTNODE_ROOT*, *WTNODE_SEP*, *WTNODE_SWT*, *WTNODE_WTK*, *WTNODE_XFORM*, *WTNODE_XFORMSEP*, *WTNODE_GLNODE* and *WTNODE_WTOBJECT*.

If the node is illegal, it returns *WTNODE_ILLEGAL*.

WTnode_setdata

```
void WTnode_setdata(  
    WTnode *node,  
    void *data);
```

This function sets the user-defined data field in a node. Private application data can be stored in any structure. To store a pointer to a structure within a node, pass in a pointer to the structure as the *data* argument, cast to a *void**.

WTnode_getdata

```
void *WTnode_getdata(  
    WTnode *node);
```

This function retrieves private data stored within a node. You should cast the value returned by this function to the data type of the data stored with the node using *WTnode_setdata*.

WTnode_canaddchild

```
FLAG WTnode_canaddchild(  
    WTnode *parent,  
    WTnode *child);
```

This function tests to see if the specified node can be added to the scene graph as the child of the specified parent node without creating a cycle in the scene graph. It returns *TRUE* if the specified child node can be added. If a cycle would be created or if the parent node is not one of the group nodes (meaning that it cannot have children) then this function returns *FALSE*.

Geometrical Property Functions

WorldToolKit functions provide access to three useful parameters that describe the space occupied by the geometries in a scene graph. Figure 4-20 illustrates these parameters: the extents box, the midpoint, and its radius.

The extents box is the smallest box that fits around the geometries. The extents box of a node in a scene graph is relative to its position in the coordinate system (the X, Y, and Z axes), which are defined by the transformations accumulated by traversing the scene graph up until that node. A node's extents box encloses the geometries *beginning at that node* and including its sub-tree (the sub-tree is its children and grandchildren, etc.)

The midpoint is the midpoint of the extents box. The radius is the distance from the midpoint of the extents box to one of its corners.

Since there is only one root node for each scene graph, the extents box of the root node encloses all of the geometry in the scene graph.

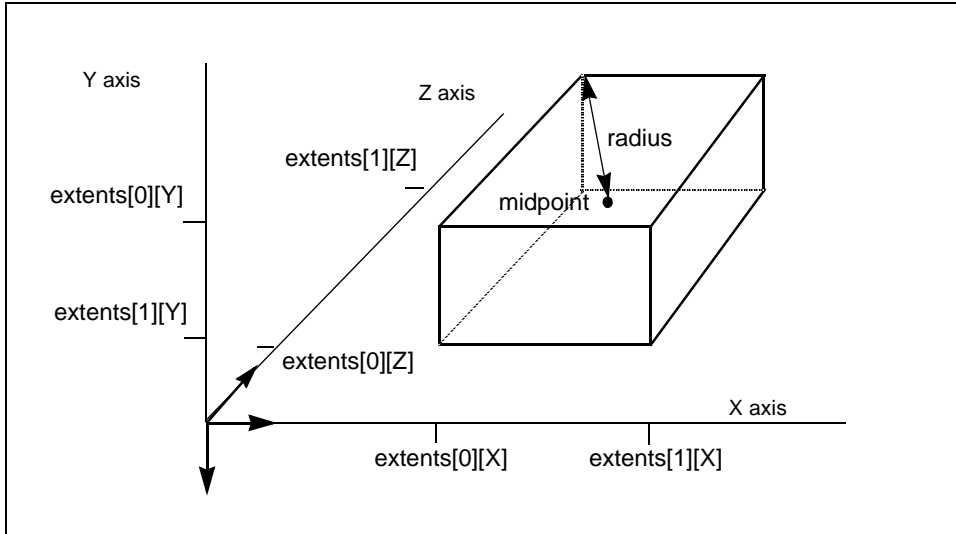


Figure 4-20: Node geometrical parameters: extents, radius and midpoint

Note: Although these parameters can be retrieved with WTK calls, they cannot be directly set. Their values are determined by the locations and extents of the geometries in the simulation.

In the remainder of this section, all of the functions described return FALSE if the node cannot have children (or if the node is not a geometry node).

WTnode_getextents

```
FLAG WTnode_getextents(  
    WTnode *node,  
    WTp3 extents);
```

This function obtains the extents of a specified node (including the node's sub-tree). The coordinates of the vector returned via the *extents* parameter represents the X, Y, and Z distance between the midpoint of the node's extents box and any corner of the extents box.

One use for this function is to restrict viewpoint motion. Your node action function might look to see whether the viewpoint is within the node's extents, and if not, call *WTviewpoint_moveto* to ensure that the viewpoint stays within the spatial extents of the geometries contained in the node's sub-tree.

To obtain the minimum and maximum world coordinate values of all graphical entities contained in the node and its sub-tree (i.e., the node's extents box, as illustrated in figure 4-20), you can use the following code segment:

```
WTnode *node;  
WTp3 midpoint;  
WTp3 extents;  
float extentsbox[2][3];  
  
WTnode_getmidpoint(node, midpoint);  
WTnode_getextents(node, extents);  
  
extentsbox[0][X] = midpoint[0] - (extents[0]);  
extentsbox[1][X] = midpoint[0] + (extents[0]);  
extentsbox[0][Y] = midpoint[1] - (extents[1]);  
extentsbox[1][Y] = midpoint[1] + (extents[1]);  
extentsbox[0][Z] = midpoint[2] - (extents[2]);  
extentsbox[1][Z] = midpoint[2] + (extents[2]);
```

WTnode_getradius

```
float WTnode_getradius(  
    WTnode *node);
```

This function obtains the distance from the midpoint of the specified node's extents box to a corner of the box. This is the same as the length of the extents vector. See *WTnode_getextents* page 4-53.

The node's radius is the distance from the midpoint of the node's "extents box" to one of its corners (see figure 4-20 on page 4-52).

It is often useful to scale distances in an application (for example, the velocities of moving objects or the parallax of a viewpoint) according to the dimensions of a node and its sub-tree. The node's radius is convenient for this purpose.

```
WTsensor *sensor;  
WTnode *node;  
  
/* scale sensor sensitivity with the size of the geometries  
   contained in the node's sub-tree */  
WTsensor_setsensitivity(sensor, 0.01 * WTnode_getradius(node));
```

WTnode_getmidpoint

```
FLAG WTnode_getmidpoint(  
    WTnode *node,  
    WTp3 p);
```

This function obtains the midpoint of the specified node's extents box.

The node's midpoint is the midpoint of the node's extents box (see figure 4-20 on page 4-52). *WTnode_getmidpoint* places this three-dimensional point in *p*.

LOD Node Functions

This section gives descriptions for LOD node functions.

WTlodnode_setrange

```
FLAG WTlodnode_setrange(  
    WTnode *node,  
    float *range,  
    int num);
```

This function sets an array of floats that specify the switch-out distances for the children of the specified LOD node. The array of floats must be in increasing order, as each float represents a distance where WTK switches to a lower level of detail. An LOD node's first child node represents the highest level of detail, while subsequent children of an LOD node represent decreasing levels of details. When an LOD node is processed, the distance between the viewpoint and the LOD center is computed. If it is less than or equal to the first range value, then WTK selects the first level of detail (i.e., the LOD node's first child node). If the computed distance is greater than the first range value, but less than or equal to the second range value, then WTK selects the second level of detail (i.e., the LOD node's second child node). By default, an LOD node has no range values.

WTlodnode_getrange

```
FLAG WTlodnode_getrange(  
    WTnode *node,  
    float *range,  
    int num);
```

This function returns an array of floats specifying the switch-out distances for the children of the specified LOD node.

The parameter *num* indicates the size of the arrays of floats, and must be at least as large as the number of range entries in the LOD node. Use `WTlodnode_numranges` to obtain the number of range values contained in an LOD node.

WTlodnode_numranges

```
int WTlodnode_numranges(  
    WTnode *node);
```

This function returns the number of range entries in the LOD node.

WTlodnode_setcenter

```
FLAG WTlodnode_setcenter(  
    WTnode *node,  
    WTp3 center);
```

This function sets the center used by an LOD node to compute distance from the viewpoint. This distance is then used to determine which child node (of the LOD node) to traverse.

The default center of an LOD node is (0.0, 0.0, 0.0) in world coordinates.

WTlodnode_getcenter

```
FLAG WTlodnode_getcenter(  
    WTnode *node,  
    WTp3 center);
```

This function returns the center position of an LOD node.

Separator Node Functions

Separator nodes, in addition to preventing state information from propagating from its descendent nodes to its sibling nodes, allow for a quick-reject test to be performed on the extents box of a separator node's sub-tree. When the simulation is run, if a separator node's extents box lies outside the area that is being viewed, then the sub-tree is not visible and is therefore not traversed (or rendered). Using separator nodes and their quick-reject test capability can drastically improve the performance of your simulation. Use the *WTsepnode_setcullmode* function to perform a quick-reject test.

This section gives descriptions for separator node functions.

WTsepnode_setcullmode

```
FLAG WTsepnode_setcullmode(  
    WTnode *node,  
    int mode);
```

This function sets the specified separator node's culling mode. Valid modes are *WTNODE_CULLON* and *WTNODE_CULLOFF*. The default is on.

WTNODE_CULLON means that a quick-reject test will be performed on the extents box of the separator node's sub-tree. If the extents box lies outside the viewing area, then the sub-tree is not visible and is therefore not traversed. If the cull mode is set to *WTNODE_CULLOFF*, the quick-reject test is not applied, and the node's sub-tree will be traversed.

WTsepnode_getcullmode

```
int WTsepnode_getcullmode(  
    WTnode *node);
```

This function returns the specified separator node's culling mode.

Switch Node Functions

This section gives descriptions for switch node functions.

WTswitchnode_setwhichchild

```
FLAG WTswitchnode_setwhichchild(  
    WTnode *node,  
    int which);
```

This function allows you to specify which child of a switch node is processed (the default is none). Valid values for *which* are: 0, 1, 2, etc. You can also use *WTnode_numchildren-1*, *WTSWITCH_ALL*, and *WTSWITCH_NONE*.

By default, the value is *WTSWITCH_NONE*, which means that none of the children of the switch node will be processed.

WTswitchnode_getwhichchild

```
int WTswitchnode_getwhichchild(  
    WTnode *node);
```

This function returns the index of the current child being processed. Note that *WTSWITCH_ALL* and *WTSWITCH_NONE* are defined as negative numbers so that actual child numbers will not conflict with these two settings.

Transform Node Functions

This section gives descriptions for transform node functions. Note that by default, WTK ignores scaling factors (if any) within a Transform (and Movable) node's transformation. If you want WTK to use the scaling factors of transformations within transform and movable nodes, you can do so by setting the *WTOPTION_XFORMSCALE* option in *WTuniverse_setoption*. However, by doing so, it is likely that intersection tests and math functions pertaining to matrices will operate incorrectly.

WTnode_settransform

```
FLAG WTnode_settransform(  
    WTnode *node,  
    WTm4 m);
```

This function replaces the transformation matrix of the specified node.

WTnode_gettransform

```
FLAG WTnode_gettransform(  
    WTnode *node,  
    WTm4 m);
```

This function returns the transformation matrix of the specified node.

WTnode_settranslation

```
FLAG WTnode_settranslation(  
    WTnode *node,  
    WTp3 p);
```

This function replaces the translation component of the specified node's transformation matrix.

WTnode_gettranslation

```
FLAG WTnode_gettranslation(  
    WTnode *node,  
    WTp3 p);
```

This function returns the translation component of the specified node's transformation matrix.

WTnode_translate

```
FLAG WTnode_translate(  
    WTnode *node,  
    WTp3 pos,  
    int frame);
```

This function creates an incremental translation to the existing transform either in the local frame or in the parent frame – as opposed to *WTnode_settranslation*, which replaces the translation value.

Valid frames are *WTFRAME_LOCAL* and *WTFRAME_PARENT*.

WTnode_setrotation

```
FLAG WTnode_setrotation(  
    WTnode *node,  
    WTm3 m);
```

This function replaces the rotational component of the specified node's transformation matrix.

WTnode_getrotation

```
FLAG WTnode_getrotation(  
    WTnode *node,  
    WTm3 m);
```

This function returns the rotational component of the specified node's transformation matrix.

WTnode_setorientation

```
FLAG WTnode_setorientation(  
    WTnode *node,  
    WTq q);
```

This function replaces the rotational component of the specified node's transformation matrix using a quaternion as an output parameter, unlike *WTnode_setrotation*, which uses a 3x3 matrix as an output parameter.

WTnode_getorientation

```
FLAG WTnode_getorientation(  
    WTnode *node,  
    WTq q);
```

This function returns the rotational component of the specified node's transformation matrix using a quaternion as an input parameter, unlike *WTnode_getrotation*, which uses a 3x3 matrix as an input parameter.

WTnode_rotation

```
FLAG WTnode_rotation(  
    WTnode *node,  
    float y,  
    float x,  
    float z,  
    int frame);
```

This function creates an incremental rotation to the existing transform either in the local frame or in the parent frame — as opposed to *WTnode_setrotation*, which replaces the rotation value. The incremental transformation matrix is the product of the 4x4 matrices formed by the rotational angles (in radians) specified in the y, x, and z parameter (in that order). Note that the order of rotation is also reflected in the order of the parameter list of this function.

Valid frames are *WTFRAME_LOCAL* and *WTFRAME_PARENT*.

WTnode_rotateq

```
FLAG WTnode_rotateq(  
    WTnode *node,  
    WTq q,  
    int frame);
```

This function creates an incremental rotation to the existing transform either in the local frame or in the parent frame by the amount specified by the quaternion. Valid frames are *WTFRAME_LOCAL* and *WTFRAME_PARENT*.

WTnode_rotatem3

```
FLAG WTnode_rotatem3(  
    WTnode *node,  
    WTm3 m3,  
    int frame);
```

This function creates an incremental rotation to the existing transform either in the local frame or in the parent frame by the amount specified by the 3x3 matrix. Valid frames are *WTFRAME_LOCAL* and *WTFRAME_PARENT*.

WTnode_rotatem4

```
FLAG WTnode_rotatem4(  
    WTnode *node,  
    WTm4 m4,  
    int frame);
```

This function creates an incremental rotation to the existing transform either in the local frame or in the parent frame by the amount specified by the 4x4 matrix. Valid frames are *WTFRAME_LOCAL* and *WTFRAME_PARENT*.

WTnode_axisrotation

```
FLAG WTnode_axisrotation(  
    WTnode *node,  
    int axis,  
    float angle,  
    int frame)
```

This function creates an incremental rotation to the existing transform either in the local frame or in the parent frame around the specified axis. Valid frames are *WTFRAME_LOCAL* and *WTFRAME_PARENT*. The axis can be X, Y, or Z. The angle is specified in radians.

URL for Anchor and Inline Nodes

WTvrml_seturl

```
void WTvrml_seturl(  
    char *basepath);
```

This function sets the URL base path, so that relative pathnames can be specified using *WTanchornode_setlocation* and *WTinlinenode_setlocation*.

Since it may be desirable to use relative pathnames in anchor and inline nodes, this function allows you to set the base path (location), so that files with relative pathnames can be located. Note that if *WTvrml_seturl* is used to set the base path, it is still acceptable to use full pathnames in the *WTanchornode_setlocation* and *WTinlinenode_setlocation* functions.

Anchor Node Functions

This section gives descriptions for anchor node functions.

WTanchornode_setlocation

```
FLAG WTanchornode_setlocation(  
    WTnode *node,  
    char *url);
```

This function replaces the anchor string (URL) reference of the specified anchor node with the new character string (the *char* string given in *url*).

WTanchornode_getlocation

```
char *WTanchornode_getlocation(  
    WTnode *node);
```

This function returns the anchor string (URL) reference of the specified anchor node.

Inline Node Functions

This section gives descriptions for inline node functions.

WTinlinenode_setlocation

```
FLAG WTinlinenode_setlocation(  
    WTnode *node,  
    char *url);
```

This function replaces the inline string (URL) reference of the specified inline node with the new character string (the *char* string given in *url*).

WTinlinenode_getlocation

```
char *WTinlinenode_getlocation(  
    WTnode *node);
```

This function returns the inline string (URL) reference of the specified inline node.

Fog Node Functions

You can control fog effects by setting the following attributes of a fog node:

<i>fogcolor</i>	The color to which objects in the scene are blended to. Default fog color is black (0.0, 0.0, 0.0). For best results, the color of the fog should match the background color of the simulation. A light grey fog color works well.
<i>range</i>	The distance upon which all objects will blend (completely) into the fogcolor. Default is 0.0 which means that the range will be set to the window yon plane distance.
<i>mode</i>	The fog blending ramp (linear, exponential, exponential-squared). The default is <i>WTFOG_LINEAR</i> .
<i>linearstart</i>	The distance at which objects are affected by the fog color. (Only applicable if the fog mode is linear.) The default is 0.0.

*Note: If you have two or more fog nodes in the same state (that is, not “state-separated” using a separator), only the most recently traversed one will be used. The fog effect is **not** cumulative.*

This section gives descriptions for fog node functions.

WTFognode_setcolor

```
FLAG WTFognode_setcolor(  
    WTnode *node,  
    float red,  
    float green,  
    float blue,  
    float alpha);
```

This function sets the fog color of a fog node. The default fog color is black. Objects in the scene will be blended to this color as a function of the distance between it and the viewpoint.

WTFognode_getcolor

```
FLAG WTFognode_getcolor(  
    WTnode *node,  
    float *red,  
    float *green,  
    float *blue,  
    float *alpha);
```

This function retrieves the fog color of a fog node.

WTFognode_setrange

```
FLAG WTFognode_setrange(  
    WTnode *node,  
    float range);
```

This function specifies the distance at which all objects are completely blended into the fog. For example, if you wish to model a scene where visibility is limited to 500 feet due to heavy fog, you would call this function with a range value of 500.0f. The specified node must be a fog node. If a range value of 0.0 is specified, then the range will be set to the window you plane distance. The default range value is 0.0 so it is recommended that you use *WTFognode_setrange* to set the range to an appropriate value.

WTFognode_getrange

```
float WTFognode_getrange(  
    WTnode *node);
```

This function returns the range (distance) of the specified fog node.

WTFognode_setmode

```
FLAG WTFognode_setmode(  
    WTnode *node,  
    int mode);
```

This function sets the mode of the specified fog node. Valid values for the *mode* argument are: *WTFOG_LINEAR*, *WTFOG_EXP*, *WTFOG_EXPSQUARED*, and *WTFOG_NONE*. The mode of a fog node specifies the fog blending ramp to be used. The default mode is *WTFOG_LINEAR*.

WTFognode_getmode

```
int WTFognode_getmode(  
    WTnode *node);
```

This function returns the mode of the specified fog node.

WTFognode_setlinearstart

```
FLAG WTFognode_setlinearstart(  
    WTnode *node,  
    float start);
```

This function specifies the starting distance where the fog color will affect the appearance of objects. The default value is 0.0. This start distance is only applicable if the fog node's mode is *WTFOG_LINEAR*.

WTFognode_getlinearstart

```
float WTFognode_getlinearstart(  
    WTnode *node);
```

This function returns the linear start distance of the specified fog node.

Open GL Callback Node Functions

OpenGL callback nodes (WTglnode) are nodes intended for use by advanced users. This node type allows a developer the flexibility to make custom Open GL calls during the scene graph's traversal. This functionality supplants that offered via the use of the WorldToolKit 3D drawing function WTwindow_setdrawfn (this was previously the only way to accomplish custom GL calls without fear of interfering with WorldToolKit's normal operation.)

The glnode functions are all prefixed with WTglnode_ and exist only to allow the insertion/creation of a callback function and to get/set the culling/boundingbox of the node. Normal node manipulation can be accomplished via the standard WTnode_ functions.

How to use glnodes

OpenGL callback nodes in WorldToolKit are used in a similar fashion as are geometry nodes; however, instead of containing a pointer to a WTgeometry, your glnode will contain a pointer to your own custom function that makes OpenGL calls. This requires not simply calling WTglnode_new, but also having defined a separate function that contains the OpenGL calls you wish to make. As WorldToolKit traverses the scene graph to draw all of the objects contained therein, it will call your custom function at the appropriate time. The benefit of doing your custom OpenGL code in this manner is that you can accumulate the transformation state of the scene graph and apply it to your custom function if you so wish (you may also choose to ignore this by simply loading the identity matrix in your callback function.)

Shown below is a simple example of how to implement an OpenGL callback node in WorldToolKit:

(Note: Sample files may be found in the directory \wtkinstall\demo\glnode)

```
#ifdef WIN32
#include <windows.h>
#endif
#include <gl\gl.h>
#include "wt.h"

static void actionfn( void );
void MyGLCallback( void );

void main( int argc, char *argv[] )
{
    WTnode *rootnode;
    WTnode *node;
    WTuniverse_new( WTDISPLAY_DEFAULT, WTWINDOW_DEFAULT );
    rootnode = WTuniverse_getrootnodes();
    WTlightnode_load( rootnode, "lights" );
    /* Create OpenGL callback node */
    node = WTglnode_new( rootnode, MyGLCallback, 0 );
    WTnode_setname( node, "MyOpenGLNode" );
    WTuniverse_setactions( (void *) actionfn );
    WTkeyboard_open();
    WTuniverse_ready();
    WTuniverse_go();
    WTuniverse_delete();
}

static void actionfn()
{
    short key;
    key = WTkeyboard_getlastkey();
    if ( 'q' == key )
        WTuniverse_stop();
}
```

```
void MyGLCallback( void )
{
    /*
       It is critical that we save the current GL attributes,
       You only need to save attributes/states that you will
       be changing; but, if you use glaux or glu functions,
       attributes and/or states may be changed that you are
       not aware of so if you are unsure, simply use the
       GL_ALL_ATTRIB_BITS flag to save everything ; however,
       this can an unnecessary computational expense if you only
       actually need to save very few states.
    */
    /* Save state(s) */
    glPushMatrix();
    glPushAttrib( GL_ALL_ATTRIB_BITS );
    /* WTK uses these states, so you may wish to disable them to have a clean state */
    glDisable(GL_CULL_FACE);
    glDisable(GL_LIGHTING);
    glDisable(GL_TEXTURE_2D);
    glDisable(GL_ALPHA_TEST);
    glDisable(GL_COLOR_MATERIAL);
    glDisable(GL_BLEND);
    glDisable(GL_LINE_SMOOTH);

    /* We now have a clean state, your OpenGL code goes here */

    /* Done with our custom code, restore WTK state(s) */
    glPopAttrib();
    glPopMatrix();
    glFlush();
}
```

It is very important that you manage the state of OpenGL properly when using this node type. Indications of improper state/attribute management can be diverse and subtle: i.e. textures are not lit properly, line widths are incorrect, anti-aliasing is incorrect, lighting is wrong, colors and/or material are incorrect, et cetera. The simplest way to accomplish this management properly is to use the call `glPushAttrib(GL_ALL_ATTRIB_BITS)` because this simply pushes the entire current state on the GL attributes stack. This is far better for

performance reasons than calling `glQuery` or `Set/Get` repeatedly as (1) you are incurring excessive function call overhead and more significantly (2) you may be taking network delay penalties if you are remotely rendering because the state information must be pushed and pulled across the network each time by the client. `glPushAttrib` avoids this by saving the current attributes on the server's attribute stack, therefore restricting delays to the actual client's calls of `glPushAttrib` and `glPopAttrib`. Note: The use of the method `WTtexture_cache` can cause unusual problems with texturing in your OpenGL callback function. We highly recommend caching all textures prior to calling `WTuniverse_go` if possible. This problem rarely surfaces; however, it has been noted to occur when incorporating 3rd party OpenGL calls such as those found in the DiGuy API from Boston Dynamics Inc.

`WTglnodes` are not allowed to make any `WorldToolKit` calls since your function is being processed in the middle of what `WorldToolKit` considers a 'known state'. It is highly recommended that you limit your callback function to OpenGL usage. Examples of how to implement OpenGL callback nodes can be found in the `wtkinstall\demo\glnode` directory. An example on how to implement 3rd party OpenGL based products such as DiGuy has been included in the `wtkinstall\demo\DiGuy` directory.

WTglnode_new

```
WTnode *WTglnode_new(  
    WTnode *parent,  
    void *GLCallbackFunction,  
    int flags);
```

This function is used to create a new OpenGL callback node. `GLCallbackFunction` should be a void pointer to a function that takes an argument list of type `void`. `flags` determines the behavioral state of the OpenGL callback node in the scene graph. There are currently 3 flags you can pass:

<code>WTGLNODE_ENABLED</code>	Allows <code>WorldToolKit</code> to process this node during scene graph traversal.
<code>WTGLNODE_BBOXENABLED</code>	Allows <code>WorldToolKit</code> to process a user-defined bounding box for this node.
<code>WTGLNODE_DEFAULTS</code>	Bitwise OR of the previous two flags.

WTglnode_replacecallback

```
FLAG WTglnode_replacecallback(  
    WTnode *node,  
    void *GLCallbackFunction);
```

This function simply replaces the currently assigned callback function with a new one.

WTglnode_setcullingbox

```
FLAG WTglnode_setcullingbox(  
    WTnode *node,  
    WTp3 midpoint,  
    WTp3 extents);
```

In order for a culling and bounding box to be processed by WorldToolKit you must supply the midpoint and dimensions of the 'box' you wish to use for this purpose. If the node was created with the flag `WTGLNODE_BBOXENABLED` or `WTGLNODE_DEFAULTS`, you should use this method immediately following creation of the OpenGL callback node. If the flag(s) were not set prior to this call, WorldToolKit will not use the node's boundingbox for culling purposes.

WTglnode_setflags

```
FLAG WTglnode_setflags(  
    WTnode *node,  
    int flags);
```

This function replaces the currently set flags with flags. It does not perform any bitwise operations on the current flag settings.

WTglnode_getflags

```
int WTglnode_getflags(  
    WTnode *node);
```

This function returns the status of a particular node's flags.

Possible uses of the OpenGL callback node

The OpenGL callback node has many uses. Most notably it allows a developer the freedom to introduce custom OpenGL calls into their WTK simulations without the expensive, laborious and 'hacky' method of tracking and maintaining state in a window's 3D drawing function.

For example, if a user developed a flight simulator for helicopters he/she may want to include smoke trails and shadows for their missiles as well as a special effect during the launch. As a helicopter launched a missile, the application would add 3 glnodes just after or before the geometry in the scene graph (so that all of these effects occur in the same local frame as the missile moves.) The node for the shadows could construct a basic cube that was roughly the same dimensions as the missile. The function would then determine where the 'sun' is in relation to the missile's position and the surface and then change the viewing transform to 'collapse' the cube's geometry onto a plane, and use that information to construct the shadow which would then need to be placed on the surface. The node for the launch affect could construct a simple alpha blended and textured cone whose textures could be modified via the 2D image functions in OpenGL and then applied. The node for the missile trails could construct of cylinders described by curves whose control points were described by the missile's frame by frame 3D position. Each segment of the cylinder's geometry could be filled with a volumetric fog at an intensity determined over time.

Now, none of these examples are simple; however, they are now more easily accomplished by the new OpenGL callback node's functionality, and they are possible.

Bounding Boxes

A bounding box represents the maximum spatial extent of an object, in its current position and orientation. You can use bounding boxes for collision detection. This section gives descriptions of bounding box functions.

You can also make bounding boxes visible in your simulation; this is a way of highlighting some of the scene's geometric entities. Any geometry node or any node which can have children (see table 4-1 on page 4-11 and table 4-2 on page 4-13) can have a bounding box. No other nodes can have bounding boxes.

When a node's bounding box is enabled, a white wireframe box is drawn at the node extents when the scene graph is rendered. These extents are the same as those obtained by *WTnode_getextents*, i.e., the extents of the node and its entire sub-tree. If you want the wire-frame box to be drawn in a color other than white, use *WTuniverse_setbboxrgb*.

WTnode_boundingbox

```
FLAG WTnode_boundingbox(  
    WTnode *node,  
    FLAG onoff);
```

This function enables a bounding box for the specified node. Use TRUE to enable the bounding box (which makes it visible) and FALSE to disable it. By default, the bounding box of a node is disabled.

Note that a bounding box can be enabled only for geometry nodes or nodes that can have children. If any other type of node is passed in to this function, then FALSE is returned.

WTnode_hasboundingbox

```
FLAG WTnode_hasboundingbox(  
    WTnode *node);
```

This function allows you to see if the specified node's bounding box is enabled. If a node's bounding box has been enabled by calling *WTnode_boundingbox* with a value of TRUE, then *WTnode_hasboundingbox* returns TRUE. If *WTnode_boundingbox* has not been called for the particular node, or if it has been called with a value of FALSE, then FALSE is returned.

Scene Graph Assembly

Note that you can add a node to the same parent node more than once. For this reason, you must refer to child nodes by number (0,1,2, etc..) rather than by pointer. This section gives descriptions for functions you use in assembling the scene graph.

WTnode_addchild

```
FLAG WTnode_addchild(  
    WTnode *parentnode,  
    WTnode *child);
```

This function adds the specified child to the scene graph after the last child of the specified parent. Note that the node may already be in the scene graph when this function is called. This function does not replace any existing child nodes, it merely adds another child node to the parent node.

WTnode_insertchild

```
FLAG WTnode_insertchild(  
    WTnode *parentnode,  
    WTnode *child,  
    int childnum);
```

This function adds the child node as the numbered node of the specified parent. Note that the node may already be in the scene graph when this function is called.

WTnode_removechild

```
FLAG WTnode_removechild(  
    WTnode *parentnode,  
    int childnum);
```

This function removes the numbered child and its sub-tree from the specified parent node, possibly leaving the node with no parents (an “orphan”). If any task has been associated with the node by a call to *WTtask_new* and the node is removed from the scene graph (i.e., the node no longer has any parents), the task is no longer performed.

WTnode_remove

```
FLAG WTnode_remove(  
    WTnode *node);
```

This function removes the specified node from all of its parent nodes, disconnecting it from the scene graph. If the specified node is a *WTgeometry* node, then it is no longer rendered because it is not encountered in any scene graph traversal during rendering. If the specified node is one of the container-type nodes, it is still possible for the children of this node to be rendered, if they have other parents that are still connected to the scene graph.

If any task has been associated with this node by a call to *WTtask_new* and the node is removed from the scene graph, the task is no longer performed.

WTnode_deletechild

```
FLAG WTnode_deletechild(  
    WTnode *parent,  
    int childnum);
```

This function detaches the numbered occurrence of the specified node from its parent node. All nodes in the sub-tree beginning with the this node are deleted (if they have no other children in the scene graph).

Tasks associated with deleted nodes will no longer be performed.

WTnode_delete

```
FLAG WTnode_delete(  
    WTnode *node);
```

This function detaches all occurrences of the specified node from all of their parent nodes. All nodes in the sub-tree beginning with this node will have the specified child nodes deleted (if they have no other parent in the scene graph).

Tasks associated with deleted nodes are no longer performed. Note that root nodes cannot be deleted; if a root node is specified as the parameter, this function does nothing and returns `FALSE`.

WTnode_vacuum

```
void WTnode_vacuum(  
    void);
```

This function deletes all non-root nodes from all the scene graphs in the universe.

Utility Functions

This section gives a description for the function used to obtain a formatted printout of a hierarchical scene graph.

WTnode_print

```
void WTnode_print(  
    WTnode *node);
```

This function generates a formatted printout of a scene graph, starting at the specified node. If you specify the root node, the whole scene graph is printed; if you specify any other node, only the specified node and its sub-tree are printed. Information printed is depth (with depth 0 assigned to the node passed in to the function), node type, and node name. Traversal of the tree is depth first. Each node of the printout is on a separate line, and each line is indented according to depth. For example, to print the first scene graph in the universe, use:

```
WTnode_print( WTuniverse_getrootnodes() );
```

Internally *WTnode_print* is implemented using *WTmessage* so that you can redirect the text output of *WTnode_print* to a file if you wish. This is especially useful for non-console Windows applications. See *WTmessage_sendto* for more information on how text output can be redirected to a file or elsewhere.

Scene Graph Structure Inquiry

This section gives descriptions for functions you use to query the scene graph.

WTuniverse_getrootnodes

See *WTuniverse_getrootnodes* on page 2-17 for a description.

WTronode_next

```
WTnode *WTronode_next(  
    WTnode *rootnode);
```

This function returns the next root node in the universe's list of root nodes. A pointer to the first root node is obtained with a call to *WTuniverse_getrootnodes*. You can then iterate through the list of existing root nodes using *WTronode_next*.

WTnode_numchildren

```
int WTnode_numchildren(  
    WTnode *node);
```

This function returns the number of children of the specified node.

WTnode_getchild

```
WTnode *WTnode_getchild(  
    WTnode *parentnode,  
    int childnum);
```

Returns the numbered child of the specified parent node. Valid values for *childnum* are 0, 1, 2, etc. up to the value returned by *WTnode_numchildren* minus 1.

WTnode_numparents

```
int WTnode_numparents(  
    WTnode *node);
```

This function returns the number of parents of the specified node. If the return value is 0 (i.e., the node has no parents), the node is inactive in the simulation.

WTnode_getparent

```
WTnode *WTnode_getparent(  
    WTnode *node,  
    int num);
```

This function returns the numbered parent of the specified node. Valid values for *num* are 0, 1, 2, etc. up through the value returned by *WTnode_numparents* minus 1.

WTnode_numpolys

```
int WTnode_numpolys(  
    WTnode *node);
```

This function returns the number of polygons contained in the specified node's sub-tree.

Scene Graph Traversal

Occasionally, you may need to find and perhaps modify certain types of nodes within a scene graph. In order to perform such a task, you are required to traverse the scene graph and then process each node as it is encountered. The following code segment is provided as a template so that you are able to easily write a scene graph traversal function which caters to your specific needs.

The following example prints out the name of all light nodes within the specified scene graph:

```
void traverse_node(WTnode *node)
{
    int nChildren, nAttachments;
    /* Put your node manipulation code here */
    if (WTNODE_LIGHT == WTnode_gettype(node)) {
        WTmessage("Light Name is %s\n", WTnode_getname(node));
    }
    nChildren = WTnode_numchildren(node);
    if (nChildren > 0) {
        for(i=0; i<nChildren; i++) {
            traverse_node(WTnode_getchild(node,i));
        }
    }
    nAttachments = WTmovnode_numattachments(node);
    if(nAttachments > 0) {
        for(i=0; i<nAttachments; i++) {
            traverse_node(WTmovnode_getattachment(node,i));
        }
    }
}
```

Additional Topics Related to the Scene Graph

This section contains some additional topics related to the scene graph, such as node paths, intersection testing, picking polygons, and sensor attachment.

Node Paths

One of the advantages of a scene graph is the ability to *instance* a node (see page 4-37). An instance is a reference to the original node. Instancing means that you can have only one object loaded into memory, but you can make as many references to it as you need. The

ability to have multiple instances of a node requires that WTK have a mechanism to uniquely identify a specific instance of a node. The mechanism that WTK uses to uniquely identify a node or node instance is called a *node path*.

A node path is actually a mathematical entity that allows you to distinguish between multiple instances of a node. A specific instance can be uniquely defined by the “node path” through the scene graph from the root node to the node instance, and hence the term node path is used.

For example, say you have one car model that is instanced several places in the simulation, which gives you several cars on the road at the same time, all of which look the same. In this simulation, you would use a node path to refer to a specific instance of the car model.

There are two things you can do with node paths:

- Perform intersection tests between a specific node path and other nodes in the scene graph — this allows intersection testing between an instance of an object and another object in the universe.
- Pick graphical entities rendered into WTK windows. The WTK picking functions generate the node path of the picked geometry node.

Note that you must create each node path that your simulation needs and you must delete it when you no longer need it (to free up the memory that it uses). If you change your scene graph after creating a node path, the node path may no longer be valid.

LOCATING NODES IN THE SCENE GRAPH

If you create a geometry node and attach it to the scene graph’s root node, the geometry is drawn at the universe origin. If you then create a transform node and attach it to the scene graph’s root, then attach the same geometry to the root node after the transform node, the geometry is drawn a second time, wherever the transform dictates. The location of that *instance* of the geometry (remember, there is only one actual geometry) depends on the path (node path) you take through the scene graph tree to reach it.

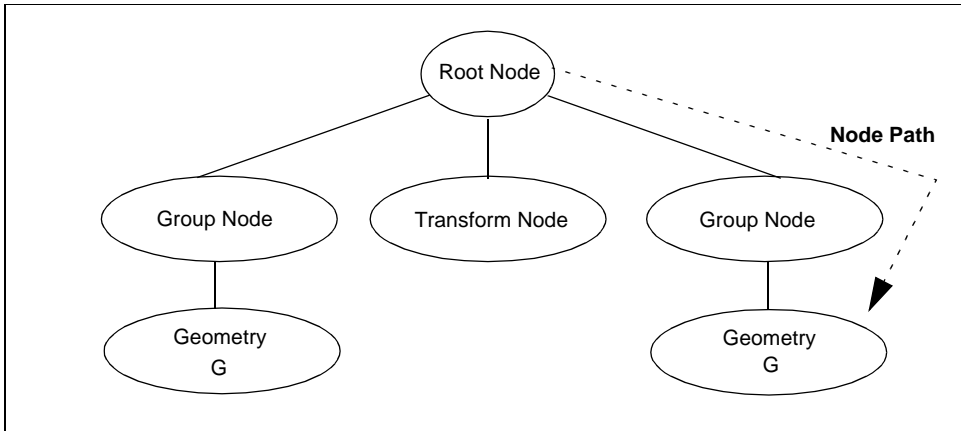


Figure 4-21: Node path to an instance of a geometry

For example, if you want a node path to the second instance of the geometry (as shown in figure 4-21), you can create it by calling `WTnodepath_new` and specifying the following three parameters: the geometry node, the ancestor node (the scene graph's root node), and the instance number 1 (since 0 refers to the first instance, and you are interested in the second instance). This newly created node path can then be used to uniquely specify the second instance of the geometry.

WTnodepath_new

```

WTnodepath *WTnodepath_new(
    WTnode *node,
    WTnode *ancestor,
    int which);
  
```

This function creates a new node path. A node path must be fully specified by giving three arguments to this function:

- the node that the instance references
- the *ancestor* of that node (in figure 4-21, the ancestor is the root node)
- and the occurrence number *which*. The *which* parameter is the number of that instance of the node

The *which* parameter must be an integer between 0 and the total number of ways (minus 1) of traversing the scene graph to go from the ancestor to the specified node. Otherwise, NULL is returned.

WTnodepath_delete

```
FLAG WTnodepath_delete(  
    WTnodepath *nodepath);
```

This function deletes a node path.

WTnodepath_numnodes

```
int WTnodepath_numnodes(  
    WTnodepath *nodepath);
```

This function obtains the number of nodes in the specified node path. This is the number of nodes in the path from (but not including) the ancestor node to the bottom-most node of the node path. In the example in figure 4-21 on page 4-81, the number of nodes is two.

WTnodepath_getnode

```
WTnode *WTnodepath_getnode(  
    WTnodepath *nodepath,  
    int num);
```

This function obtains the specified node in the specified node path. Think of the node path as a specific path through the scene graph. Pass in 0 for *num* to obtain a pointer to the first node in the node path. The first node in the node path is **not** the ancestor node that was specified when the node path was created. The first node in the node path is actually a child node of that ancestor node. In figure 4-21 on page 4-81, the first node in the node path is the (second) group node.

Pass in:

`WTnodepath_numnodes(nodepath) - 1`

for *num* to obtain a pointer to the bottom-most node in the node path. The bottom-most node in figure 4-21 on page 4-81 is the instance of the geometry.

Pass in a value for *num* between these two extremes to obtain a pointer to the nodes of the node path between the child of the ancestor and the bottom-most nodes.

WTnodepath_gettraversal

```
int WTnodepath_gettraversal(  
    WTnodepath *nodepath,  
    int *numarray,  
    int maxsize);
```

This function obtains a description of the specified node path in terms of the scene graph traversal order. This function will return a number for each node in the node path, where the number indicates which child number the node represents.

Your application must declare the integer array called *numarray*, and pass it in to this function. The size of *numarray* must be at least as big as the value returned by *WTnodepath_numnodes* for this node path. You must also specify the size of the array passed in.

The meaning of the values returned in *numarray* are as follows.

<i>numarray</i> [0]	the number of the first child of the ancestor node along this node path.
<i>numarray</i> [1]	the number of the grandchild (the child of the first child) of the ancestor node along the node path. And so on, up to:
<i>numarray</i> [<i>WTnodepath_numnodes</i> ()-1]	which is the number of the bottom-most node along the node path.

Note: The numbers returned by this function are relative to each parent node. The numbers tell you which child of each parent node is along the specified node path.

WTnodepath_getextents

```
FLAG WTnodepath_getextents(  
    WTnodepath *nodepath,  
    float ext[2][3]);
```

This function places the center and extents of the node path in the specified floating point array (given by *ext*). The extents box of a node path is the smallest rectangular box that encloses all the geometries of the node path and which is aligned with the world coordinate axes. Use this function to test for collisions anywhere on the node path.

OBTAINING AN ACCUMULATED TRANSFORMATION

The functions in this section enable you to obtain the accumulated transformation (both position and orientation) of the node path. This takes into account all transformations accumulated by traversing down and then to the right (“depth first”) between the ancestor and bottom-most node of the node path.

If the node path passed in to any of the functions in this section have become invalid since the node path was created, then FALSE is returned. The node path can become invalid if parts of the scene graph associated with the node path have been modified after the node path was created.

WTnodepath_gettransform

```
FLAG WTnodepath_gettransform(  
    WTnodepath *nodepath,  
    WTm4 m4);
```

This function returns the transformation matrix that would be applied to the leaf node of the node path.

WTnodepath_gettranslation

```
FLAG WTnodepath_gettranslation(  
    WTnodepath *nodepath,  
    WTp3 p);
```

This function returns the translational component of the transformation matrix that would be applied to the leaf node of the node path.

WTnodepath_getorientation

```
FLAG WTnodepath_getorientation(  
    WTnodepath *nodepath,  
    WTq q);
```

This function returns the rotational component of the transformation matrix that would be applied to the leaf node of the node path. The rotational component is returned in quaternion form.

Intersection Testing

Node paths can be used to test for intersections between specific instances of geometries. Because a node may be referenced more than once in a scene graph, it is not enough to simply ask whether two nodes in your scene graph intersect. You must specify the specific node paths you are interested in. For example, suppose your application is a simulation of a sailboat race, with several sailboats navigating a course defined by several buoys. To find out whether a sailboat has collided with a buoy, you must specify exactly which sailboat and which buoy. To do so, node paths are used.

The intersection functions provided in this section are meaningful only if the two node paths passed in as arguments have the same ancestor node. This is the case, for example, if both node paths are created by calling *WTnodepath_new* with the same second argument. By having a common ancestor node, there is a common frame of reference in which the proximity of the node paths can be determined. Note that this common ancestor node can be the root node. Therefore it is always possible to test for intersections of node paths which are in the same scene graph. However it is not possible to test for intersections of node paths which are in completely disjoint scene graphs.

Wtpoly_intersectpolygon

```
FLAG Wtpoly_intersectpolygon(  
    Wtpoly *poly1,  
    WTnodepath *nodepath1,  
    Wtpoly *poly2,  
    WTnodepath *nodepath2);
```

This function tests whether two polygons intersect and returns TRUE if they intersect and FALSE if they do not intersect. Since polygons are contained within geometry nodes and nodes may be referenced more than once in a scene graph, it is not enough to simply specify the two polygons. In addition to the two polygons, you must specify the node path of the specific polygon instance for each polygon.

Wtpoly_intersectnode

```
FLAG Wtpoly_intersectnode(  
    Wtpoly *poly1,  
    WTnodepath *nodepath1,  
    WTnodepath *nodepath2);
```

This function tests whether a polygon instance (*nodepath1*) intersects any polygons in the scene graph's sub-tree whose start node is the bottom-most node of the node path (*nodepath2*). It returns TRUE if there is an intersection and FALSE otherwise. Since the polygon may be referenced more than once in the scene graph, you must specify the node path of the specific polygon instance (*nodepath1*).

Wtpoly_intersectbbox

```
FLAG Wtpoly_intersectbbox(  
    Wtpoly *poly1,  
    WTnodepath *nodepath1,  
    WTnodepath *nodepath2);
```

This function tests whether a polygon instance (*nodepath1*) intersects any part of the bounding box of the scene graph sub-tree whose start node is the bottom-most node of the node path (*nodepath2*) and returns TRUE if there is an intersection and FALSE otherwise. Since the polygon may be referenced more than once in the scene graph, you must specify the node path of the specific polygon instance (*nodepath1*).

WTnodepath_intersectpoly

```
FLAG WTnodepath_intersectpoly(  
    WTnodepath *nodepoly1,  
    WTnodepath *nodepath2);
```


This function tests for the intersection of any polygons in two node paths (nodepath1 and nodepath2) and their sub-trees. It returns TRUE if there is an intersection, FALSE otherwise.

WTnodepath_intersectbbox

```
FLAG WTnodepath_intersectbbox(  
    WTnodepath *n1,  
    WTnodepath *n2);
```

This function tests for the intersection of two node paths, n1 and n2, based on their bounding boxes. Remember that these bounding boxes are the bounding boxes of the entire sub-tree of the scene graph beginning at the bottom-most node of the node path.

This function returns TRUE if an intersection is found and FALSE otherwise. If n1 and n2 weren't constructed with a common ancestor, then FALSE is returned. If the node paths are equivalent, i.e., represent the same exact path through the scene graph, or if one of the node paths represents a subset of the path through the scene graph represented by the other, then FALSE is returned.

WTnodepath_intersectnode

```
WTnodepath *WTnodepath_intersectnode(  
    WTnodepath *nodepath,  
    WTnode *node,  
    int which);
```

This function performs a bounding box intersection test between the specified node path and the numbered occurrence of the specified node (and its sub-tree). If they do not intersect, it returns NULL. If they do intersect, then this function traverses down the specified node's sub-tree in search of the node whose bounding box has the smallest extents and yet still intersects the bounding box of the specified node path. Then a node path to that node is created and a pointer to it is returned.

For example, suppose your simulation contains multiple conveyor belts, each with several links, onto which a box is dropped. You want to know which link of a specific belt the box intersects as it lands. To do this, pass in the node path corresponding to the specific box as the first argument, pass in a pointer to the group or separator node representing the belt, and as the last argument pass in the number to specify the specific belt to test.

WTnode_rayintersect

```
WTPoly *WTnode_rayintersect(  
    WTnode *node,  
    WTP3 dir,  
    WTP3 origin,  
    float *distance,  
    WTnodepath **nodepath);
```

This function obtains the frontmost intersected polygon along a specified ray contained in any geometry node in the specified nodes sub-tree. The ray is defined by the *dir* and *origin* arguments (specifying the direction and the origin respectively) in the same coordinate system as the specified node. This function only tests visible (i.e., front-facing) polygons that are beyond the hither clipping plane. Back-facing polygons and polygons between the viewpoint and the hither clipping plane are not tested for intersection.

If the *distance* argument is non-NULL, then this memory location is set to the distance along the ray from the origin to the intersection point.

If you supply a non-NULL *nodepath* argument, then a node path is created which defines the path to the geometry containing the polygon that was intersected. This node path begins at the specified node. You are responsible for deleting the node path created by this function. Call *WTnodepath_delete* to do so, once you are through using the node path.

See *What Is Terrain Following?* on page A-31.

WTPoly_rayintersect

```
FLAG WTPoly_rayintersect(  
    WTPoly *poly,  
    WTnodepath *npath,  
    WTP3 direction,  
    WTP3 origin,  
    float *dist);
```

This function tests whether a ray specified by an origin and a direction vector intersects a given polygon, *poly*. The origin and the direction should be specified in world coordinates.

If the ray intersects the polygon, `TRUE` is returned. In this case, the distance along the ray from its origin to the intersection point is returned in `dist`. If the ray does not intersect the polygon, `FALSE` is returned and `dist` is not updated.

This function takes a pointer to a node path as one of its arguments. You should create a node path from the root node to the geometry node that contains the polygon `poly`. Note that it is possible that your scene graph has multiple instances of the geometry node that references `poly`. The node path indicates the instance of the geometry node with which to perform the intersection test. If the node path does not start at the root node or does not end at the geometry node that contains `poly`, `FALSE` is returned.

`WTPoly_rayintersect` takes into account the accumulated transform along the node path from the root node to the geometry node. That is why you need to specify the ray origin and direction in world coordinates.

This function is similar to `WTnode_rayintersect`, though in certain cases, it is more efficient than the latter. For example, if you need to determine whether a ray intersects any polygon of a particular geometry, you could loop through the geometry's polygons calling `WTPoly_rayintersect` for each one.

Now consider implementing this using `WTnode_rayintersect`. You would simply test whether the polygon returned by `WTnode_rayintersect` belongs to the relevant geometry. This is less efficient, however, because `WTnode_rayintersect` tests for an intersection with every polygon in every geometry that is below the specified node, before it returns the closest polygon. Remember that since `WTnode_rayintersect` takes the origin and ray in the node's local coordinates, the node has to be sufficiently high up in the scene graph such that all relevant transform nodes are considered. This might prove to be expensive if you have your scene graph organized such that `WTnode_rayintersect` is forced to check for intersections with irrelevant geometries. There are circumstances of course, where `WTnode_rayintersect` is the better suited function.

See *What Is Terrain Following?* on page A-31.

WTviewpoint_intersectpoly

```
FLAG WTviewpoint_intersectpoly(  
    WTviewpoint *vpoint,  
    WTPoly *poly,  
    WTnodepath *npath,  
    float distance);
```

This function tests whether the viewpoint *vpoint* intersected the polygon *poly* as a result of the viewpoint's motion in the current frame.

npath should be a node path from the root node to the geometry node that contains the polygon *poly*. It is possible that your scene graph has multiple instances of the geometry node that contains the polygon *poly*. The node path *npath* indicates exactly which instance you want the intersection test to be performed with. If the node path specified does not start at the root node and end at the geometry node, FALSE is returned.

Use the argument *distance* to specify how close you want the viewpoint to get to the polygon before it is detected as an intersection. The value usually specified for *distance* is 0.0. In some cases, however, you might want to detect whether the viewpoint is as close as the hither clipping distance to the polygon, even if it has not intersected the polygon. You should then pass in the hither clipping value as *distance*. (See *WTwindow_gethithervalue* on page 17-18). Negative values for *distance* are invalid and will result in this function returning FALSE.

WTviewpoint_intersectpoly returns TRUE if the motion of the viewpoint during the current frame resulted in an intersection with the polygon, or if the distance between the new position of the viewpoint and the polygon is less than the *distance* parameter passed into this function.

Since you would usually want to test to see whether the viewpoint has intersected the polygon each and every frame, you should call this function from within the universe actions function. Also, the *universe event order* is critical to the functioning of *WTviewpoint_intersectpoly*. If the viewpoint is being controlled by a sensor, sensor updates have to be done before this function is called. If the sensor updates have not been done, *WTviewpoint_intersectpoly* will find no difference between the position of the viewpoint in the last frame and that in the current frame. That is why you have to call *WTuniverse_seteventorder* (see page 2-9) such that the sensors are updated before the actions function is called. Then, with a call to *WTviewpoint_intersectpoly* in your actions function, you can determine whether the sensor updates in the current frame resulted in the viewpoint intersecting a given polygon.

One application for *WTviewpoint_intersectpoly* is portals. Your code can call this function every frame to check whether the viewpoint intersected a portal polygon. If *WTviewpoint_intersectpoly* returns TRUE, you should have code that appropriately loads in a new world or switches to a different root node.

Refer to the *portal.c* demonstration (located in the *demos* directory on your WTK distribution) for a detailed example of how to use this function. Also see *How Do I Handle Portals In This Release?* on page A-22.

Picking Polygons

The functions described in this section enable you to pick the top-most rendered polygon in the specified window. These functions provide you with not just the intersected polygon, but also with the coordinate of the point at which the polygon is intersected, as well as the *WTnodepath* indicating to which node occurrence in the scene graph the intersected polygon belongs.

WTscreen_pickpoly

```
WTPoly *WTscreen_pickpoly(  
    int screennum,  
    WTP2 pt,  
    WTnodepath **nodepath,  
    WTP3 p);
```

This function obtains a pointer to the frontmost polygon rendered at the specified 2D screen point on the specified screen. Screen coordinates are specified as 2D floating point values, with (0.0, 0.0) representing the top-left corner of the screen, and the bottom-right corner of the screen represented by (screen width, screen height). If there is no *WTwindow* at the specified screen coordinate of the specified screen, or if there is no polygon at that coordinate, then NULL is returned.

The *WTP3* obtained is the 3D point in world coordinates at which the selected polygon was intersected.

This function also fills in the value of the *WTnodepath* pointer, indicating the node path to which the selected polygon belongs. If the polygon selected is in a *WTgeometry* node which is referenced more than once in the scene graph, it may be useful to know for which occurrence of the *WTgeometry* node the polygon was selected. You are allowed to pass in NULL for the *nodepath* argument. If you do pass in NULL, then the function does not provide the *WTnodepath* pointer information to you and does not create a *WTnodepath* for you.

If you do pass in a non-NULL value for *nodepath*, then a node path is created. You are responsible for deleting this *WTnodepath*, when you no longer need it. To do so, call *WTnodepath_delete*.

WTwindow_pickpoly

See *WTwindow_pickpoly* on page 17-20 for a description.

Sensor Attachment

Sensors can be attached to transform nodes or to node paths, as long as the bottom-most node of the node path is a transform node. Motion links, which are described in the *Motion Links* chapter, are a more powerful and general-purpose mechanism for attaching sensors to various objects than using transform nodes.

WTnode_addsensor

```
WTmotionlink *WTnode_addsensor(  
    WTnode *node,  
    WTsensor *sensor);
```

This function attaches a sensor to a transform node. You can only pass a transform node into this function.

Transform nodes have a property — a list of attached sensors — that automatically updates position and orientation stored in the node, in the local frame.

Some sensors, like the FASTRAK, ISOTRAK, InsideTRAK, and Flock of Birds, return absolute records. To get the expected results with these sensors, you have to set their reference frame to their parent node's reference frame using the *WTmotionlink_new* function.

WTnode_removesensor

```
void WTnode_removesensor(  
    WTnode *node,  
    WTsensor *sensor);
```

This function detaches the specified sensor from the specified node.

WTnodepath_addsensor

```
WTmotionlink *WTnodepath_addsensor(  
    WTnodepath *nodepath,  
    WTsensor *sensor,  
    int frame);
```

This function allows you to attach a sensor to a node path (if the bottom-most node of the node path is a transform node).

The sensor input is applied relative to the top-most node of the node path (this is the ancestor node argument to *WTnodepath_new*).

WTnodepath_removesensor

```
void WTnodepath_removesensor(  
    WTnodepath *nodepath,  
    WTsensor *sensor);
```

This function detaches the specified sensor from the specified node paths's leaf node.

Movable Nodes

Introduction

Movable nodes are self-contained entities that save you time and effort when constructing a scene graph. Because movable nodes contain position and orientation information, movable nodes make it easier to position the object corresponding to the movable node. In this manual, movable nodes are also referred to as “movables.”

What Makes Up a Movable Node?

As shown in figure 5-1, the three basic components of a movable node are a separator, a transform, and a content.

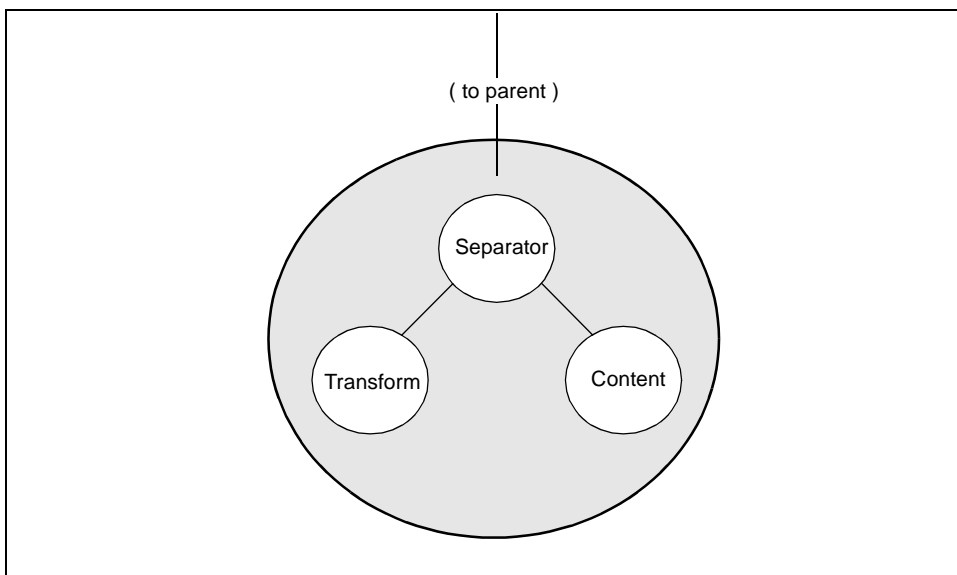


Figure 5-1: The basic structure of a movable node

Table 5-1 describes each component in a movable and what it accomplishes.

Table 5-1: The three basic components in a movable node

Node	What it controls	Remarks
Separator	Keeps the transformation within this movable from affecting sibling nodes.	Movable light nodes have a transform separator instead of a separator.
Transform	How the content is positioned.	Each movable node has a transformation component which allows you to control the position and orientation of a movable node. See <i>Movable Node Position and Orientation</i> on page 5-7.
Content	What the movable displays or accomplishes.	<p>The five types of content components are the following:</p> <p>Geometry (a series of vertex positions and surface definitions).</p> <p>Light (a defined source of illumination).</p> <p>Separator (prevents state information from propagating from its descendant nodes to its sibling nodes).</p> <p>Switch (a group that allows the user to control which of its children is in the simulation at any given time).</p> <p>Level of Detail (LOD) (a switch that chooses the active child automatically, based on the range to the viewpoint). Use LOD to improve rendering speed by displaying simpler objects at a distance and switching to more complex objects as you approach them in the simulation.</p>

The last three types of content components — Separator, Switch, and LOD — are “group” types. Group movable nodes can have children under them in the scene graph. See *Group Movable Node Creation* on page 5-4.

Movable Node Creation Functions

This section lists the functions you use to create movable nodes.

Geometry and Light Movable Node Creation

These movable nodes (like geometry and light nodes) cannot have children.

WTmovgeometrynode_new

```
WTnode *WTmovgeometrynode_new(  
    WTnode *parent,  
    WTgeometry *geom );
```

This function creates a movable geometry node from the existing geometry and adds it to the scene graph after the last child of the specified parent. If the parent is `NULL`, the movable geometry node is created without a parent. If there is an error, `NULL` is returned. Any of the *WTnode* functions that are applicable to geometry nodes are also applicable to movable geometry nodes.

WTmovlightnode_newpoint

```
WTnode *WTmovlightnode_newpoint(  
    WTnode *parent);
```

This function creates a movable point light node and adds it to the scene graph after the last child of the specified parent. If the parent is `NULL`, the movable light node is created without a parent. If there is an error, `NULL` is returned. You can use the regular *WTlightnode* functions to set and retrieve a movable point light's attributes. See the *Lights* chapter (starting on page 12-1).

WTmovlightnode_newdirected

```
WTnode *WTmovlightnode_newdirected(  
    WTnode *parent);
```

This function creates a movable directed light node and adds it to the scene graph after the last child of the specified parent. If the parent is NULL, the movable light node is created without a parent. If there is an error, NULL is returned. You can use the regular *WTlightnode* functions to set and retrieve a movable directed light's attributes. See the *Lights* chapter (starting on page 12-1).

WTmovlightnode_newspot

```
WTnode *WTmovlightnode_newspot(  
    WTnode *parent);
```

This function creates a movable spot light node and adds it to the scene graph after the last child of the specified parent. If the parent is NULL, the movable light node is created without a parent. If there is an error, NULL is returned. You can use the regular *WTlightnode* functions to set and retrieve a movable spot light's attributes. See the *Lights* chapter (starting on page 12-1).

Group Movable Node Creation

Group movable nodes can have children.

WTmovsepnode_new

```
WTnode *WTmovsepnode_new(  
    WTnode *parent );
```

This function creates a movable separator node and adds it to the scene graph after the last child of the specified parent. If the parent is NULL, the movable separator node is created without a parent. If there is an error, NULL is returned. A separator prevents state information from propagating from its descendant nodes to its sibling nodes.

You can use the regular *WTsepnode* functions with movable separator nodes. See *Separator Node Functions* on page 4-56.

WTmovswitchnode_new

```
WTnode *WTmovswitchnode_new(  
    WTnode *parent );
```

This function creates a movable switch node and adds it to the scene graph after the last child of the specified parent. If the parent is NULL, the movable switch node is created without a parent. If there is an error, NULL is returned. A switch is a group that allows the user to control which of its children should be in the simulation at any given time.

You can use the regular *WTswitchnode* functions with movable switch nodes. See *Switch Node Functions* on page 4-57.

WTmovlodnode_new

```
WTnode *WTmovlodnode_new(  
    WTnode *parent );
```

This function creates a movable Level of Detail (LOD) node and adds it to the scene graph after the last child of the specified parent. An LOD is a switch that chooses the active child automatically, based on the range to the viewpoint. The children of an LOD are typically the same object with differing numbers of polygons. High-detail models are selected when the viewer is close for better realism. Low-detail models are selected when the viewer is farther away, which increases the frame rate by rendering fewer polygons. If the parent is NULL, the movable LOD node is created without a parent. If there is an error, NULL is returned.

You can use the regular *WTlodnode* functions with movable LOD nodes. See *LOD Node Functions* on page 4-55.

WTmovnode_load

```
WTnode *WTmovnode_load(  
    WTnode *parent,  
    char *filename,  
    float scale);
```

This function creates a movable node (or node hierarchy) from data read in from a file, and adds the movable node to the scene graph after the last child of the specified parent. The

data read in from the file may contain geometry data or data which corresponds to any of the supported node types. If the file contains a single geometry then the name assigned to the movable node will be the name of the geometry. If the file contains multiple geometries then the movable node's name will be set to NULL.

This function can read data from light files (see the *Lights* chapter, starting on page 12-1) to create a movable (point, directed, or spot) light. If the light file contains multiple lights, then a single movable containing all of the lights is created. So, if you need to create individual movables for each light in a light file, you should break the file down into "single-light" files.

See *What Is The Difference Between WTmovnode_load and WTnode_load?* on page A-4.

Note: The argument *filename* is a string that specifies the name of the file from which the data is read. This file could be on your local system (in which case you specify the path to it), or it could be a URL. If you are using a URL to read in data, the file name should contain the full http address (e.g., <http://www.sense8.com/models/oplan.wrl>).

WTK supports http URLs to VRML files only. The WTmovnode_load function does not support any other file type by way of a URL. Make sure your system has an http server if you intend on using URLs in the filename argument.

Movable Nodes Compared to 'Regular' Nodes

Aside from the fact that the transformation functions such as *WTnode_setrotation* can be used with the class of movable nodes, each type of movable node is identical to the corresponding regular (non-movable) node.

Each of the movable nodes created by the functions in the left column below are identical to the corresponding regular (non-movable) nodes created by the functions in the right column. Their functionality is identical and the functions that are applicable to the regular (non-movable) nodes are also applicable to the movable version.

<i>WTmovgeometrynode_new</i>	<i>WTgeometrynode_new</i>
<i>WTmovlightnode_newpoint</i>	<i>WTlightnode_newpoint</i>
<i>WTmovlightnode_newdirected</i>	<i>WTlightnode_newdirected</i>
<i>WTmovlightnode_newspot</i>	<i>WTlightnode_newspot</i>
<i>WTmovsepline_new</i>	<i>WTsepline_new</i>

*WTmovswitchnode_new**WTswitchnode_new**WTmovlodnode_new**WTLodnode_new*

Movable Node Position and Orientation

As illustrated in figure 5-1 on page 5-1, each movable node has a built-in transformation component that allows you to control its position and orientation. Thus, you do not have to create a transformation node for the movable node.

To set the position and/or orientation of a movable node, you can use any of the position and orientation *WTnode* functions that are applicable to transform nodes, such as *WTnode_settransform*, *WTnode_setrotation*, etc. (see *Transform Node Functions* on page 4-58). It is important to remember that a movable node's position and orientation may also be affected by transformation nodes or movable nodes that are its ancestors in the scene graph.

When positioning an object, WTK functions use the geometry's *origin* when moving a geometry to the specified position. For example, when *WTnode_settranslation* is called, the geometry is translated so that its origin is placed at the 3D world coordinate passed in to that function.

The following example shows how you can create and position a movable geometry node at the world coordinates 100.0, 0.0, 0.0 (given that the corresponding *WTgeometry* has already been created):

```
WTgeometry *geo;
WTnode *root;
WTnode *movgeo;
WTp3 position;
root = WTuniverse_getrootnodes();
movgeo = WTmovgeometrynode_new(root, geo);
position[0] = 100.0;
position[1] = 0.0;
position[2] = 0.0;
WTnode_settranslation(movgeo, position);
```

WTmovnode_axisrotation

```
FLAG WTmovnode_axisrotation(  
    WTnode *movnode,  
    int axis,  
    float angle);
```

This function rotates a movable node in its local frame (i.e., it rotates around its own axis). The specified movable is rotated by the number of radians in the *angle* parameter about the specified (X, Y, or Z) axis. Note that the rotation angle specified here has an incremental effect, i.e., it is combined with the existing transformation component of the movable node; it does not “replace” the transformation component.

If the specified node *movnode* is NULL, or if it’s not a movable node, then this function returns FALSE.

WTmovnode_alignaxis

```
FLAG WTmovnode_alignaxis(  
    WTnode *movnode,  
    int axis,  
    WTp3 dir);
```

This function rotates the movable node about its midpoint in such a way that the specified axis of the movable aligns with (i.e., points in the same direction) as the direction vector *dir*. This function “replaces” the WTK V2.1 function *WTobject_alignaxis*.

The argument *movnode* should point to a movable node. *axis* should be one of the defined constants X, Y or Z, and it identifies the axis of the movable node that needs to be aligned with the direction vector *dir*. This function is not available for regular transform nodes or geometry nodes.

The following example aligns a graphical object (flashlight) with a light. It is assumed that “flashlight” is a movable created with *WTmovnode_load*, and “lightnode” is a directional light node.


```
{  
    WTp3 dir;  
    WTlightnode_getdirection( lightnode, dir );  
    /* X axis assumed to point along flashlight length */  
    WTmovnode_alignaxis( flashlight, X, dir );  
}
```

WTmovnode_alignaxis returns TRUE if it succeeds in aligning the movable as required. It returns FALSE if *movnode* is not a movable, or if *axis* is not one of the constants X, Y, or Z, or if *dir* is a zero vector (a vector whose magnitude is 0).

Movable Node Hierarchies

A movable hierarchy is a group of nodes that move together as a whole but whose parts can move independently. For example, consider the hierarchically assembled robot arm illustrated in figure 5-2.

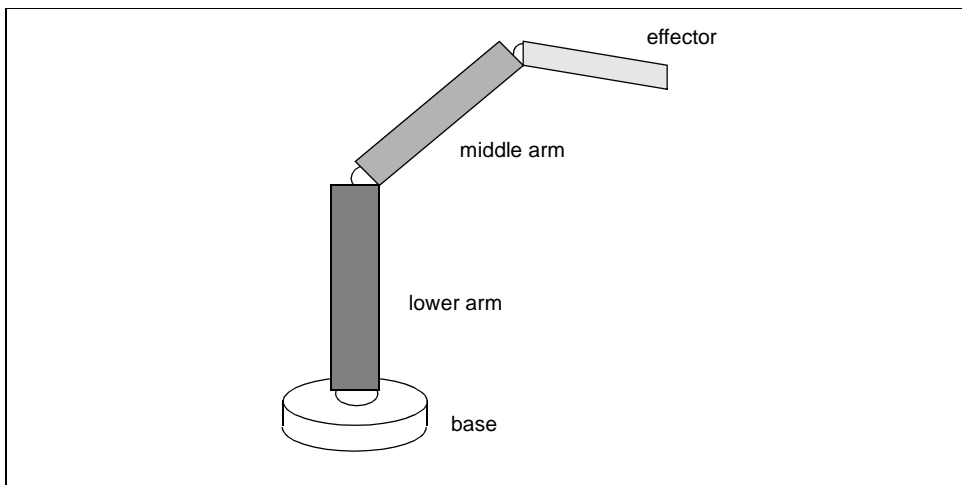


Figure 5-2: Hierarchically assembled robot arm

Each part of the robot arm — the base, the lower segment, the middle segment, and the effector — must be created as a separate node, using, for example, the function *WTmovnode_load*. The pointers to these four movable nodes are called *base*, *lower*, *middle*, and *effector*. To assemble the robot arm as shown in figure 5-2, you would make the following calls to *WTmovnode_attach* (see page 5-11):

```
WTmovnode_attach(base, lower, 0);
WTmovnode_attach(lower, middle, 0);
WTmovnode_attach(middle, effector, 0);
```

These calls result in a geometry hierarchy in which *base* is the root, and moving down through the hierarchy is *lower*, then *middle*, then *effector*. (Don't be confused by the fact that "down" in the hierarchy corresponds to "up" in figure 5-2.)

When a geometry in the hierarchy moves, it moves all of the geometries that are below it, as if the geometries were rigidly attached. Geometries that are above the moved geometries are not affected by the geometry's motion. For example, when the lower arm moves, this causes the middle arm and effector to move with it, while the base is unaffected. When the effector moves, none of the other geometries are affected because the effector is at the bottom of the hierarchy. Since sub-geometries move automatically with their parent geometries, if you wish to move an entire geometry hierarchy, you need only move the topmost geometry in the hierarchy. In the robot arm example, to move the entire arm you would simply move the base.

Keyboard input, mouse button presses, or other sensor device input could be used to control the robot arm. For example, to rotate the effector using the left mouse button, you could use the following effector task function (assigned with *WTtask_new*). Note that the following code fragment assumes the existence of a global WTK sensor object pointer called *mouse*.

```
void effector_task(WTnode *mnode)
{
    float w; /* amount of rotation (radians) */

    /* return if the left mouse button isn't pressed. */
    if ( ! (WTSensor_getmiscdata(mouse) & WTMOUSE_LEFTBUTTON) )
        return;

    /* Rotate the movable.
    Rotation about the effector's Y or Z axis will cause the
    arm to pitch or yaw, rather than to twist about its length. */
```

```
w = WTsensor_getangularrate(mouse);
WTnode_rotate(mnode, Y, w, WTFRAME_LOCAL);
}
```

WTmovnode_attach

```
FLAG WTmovnode_attach(
    WTnode *parent,
    WTnode *child,
    int attachmentnum);
```

This function attaches the child node to the parent node as the *attachmentnum*'th attachment. The parent node must be a movable node (a node created by *WTmovgeometrynode_new*, *WTmovlodnode_new*, etc.), while the child node can be a movable or a regular (non-movable) node.

The parameter *attachmentnum* must be an integer between the range of 0 (zero) and the total number of attachments. (You can find the total number of attachments by using the *WTmovnode_numattachments* (see page 5-13) function. For example, if the movable node specified by the parent has two attachments, then calling *WTmovnode_attach* with an attachmentnum value of 2, will attach the child node as the third attachment of the parent node. You can also set *attachmentnum* to the constant `WTNODE_APPEND`, which attaches the child node to the end of all the attachments.

Note that if the specified child node is already contained in the scene graph or has been previously attached to another movable node, then this function will not detach existing connections but instead will create an additional connection. Attaching a node to a movable node is similar to adding a child node to a parent; the child node may already be placed somewhere in the scene graph so *WTmovnode_attach*, like *WTnode_addchild*, will not alter the child node's existing connections within the scene graph.

Note: This function creates a new instance of the child node in the specified location in the hierarchy. It does **not** move an existing node from one location to another.

Use this function when the child you are attaching is different from the other children of the parent node. For example, if you have a movable Level of Detail (LOD) node with several children (representing the different levels of detail of a single object), and you want to add a node that is unrelated to LOD (i.e., it does not represent one of the level of detail objects of the LOD), you would attach it with this function rather than adding it with *WTnode_addchild*.

If the specified parent node is not a movable node, or if parent is NULL, or if child is NULL, or if *attachmentnum* is outside the valid range, then this function returns FALSE, otherwise it returns TRUE.

WTmovnode_detach

```
FLAG WTmovnode_detach(  
    WTnode *parent,  
    int attachmentnum);
```

This function detaches the child node, whose attachment number is specified by *attachmentnum*, from the parent node, possibly leaving the detached node with no parents (i.e., the node becomes an orphan). The parent node must be a movable node (a node created by *WTmovgeometrynode_new*, *WTmovlodnode_new*, etc.).

Attachmentnum must be an integer between the range of 0 (zero) and the total number of attachments minus one (the number returned by the function *WTmovnode_numattachments* (see page 5-13) minus one).

If the specified parent node is not a movable node, or if *attachmentnum* is outside the valid range, then this function returns FALSE, otherwise it returns TRUE.

WTmovnode_deleteattachment

```
FLAG WTmovnode_deleteattachment(  
    WTnode *parent,  
    int attachmentnum);
```

This function detaches the child node, whose attachment number is specified by *attachmentnum*, from the parent node, and deletes all nodes of the sub-tree beginning with that attachment node (if they have no parents in other branches of any scene graph).

The parent node must be a movable node (a node created by *WTmovgeometrynode_new*, *WTmovlodnode_new*, etc.). *Attachmentnum* must be an integer between the range of 0 and the total number of attachments minus one (the number returned by the function *WTmovnode_numattachments* (see below) minus one).

If the specified parent node is not a movable node, or if *attachmentnum* is outside the valid range, then this function returns FALSE, otherwise it returns TRUE.

WTmovnode_numattachments

```
int WTmovnode_numattachments(  
    WTnode *node);
```

This function returns the movable node's number of attachments. For example, if the specified parent node has a single attachment, then *WTmovnode_numattachments* returns 1 (one). If the specified parent node is not a movable node, then the parent node cannot have any attachments and this function returns 0 (zero).

WTmovnode_getattachment

```
WTnode *WTmovnode_getattachment(  
    WTnode *node,  
    int attachmentnum);
```

This function returns a pointer to the node whose attachment number is specified by *attachmentnum*. *Attachmentnum* must be an integer between the range of 0 and the total number of attachments minus one (the number returned by the function *WTmovnode_numattachments* (see above) minus one).

If the specified parent node is not a movable node or if *attachmentnum* is outside the valid range then this function returns NULL.

Movable Node Instancing

WTmovnode_instance

```
WTnode *WTmovnode_instance(  
    WTnode *parent,  
    WTnode *movable);
```

This function creates a separate instance of a movable node. When multiple instances of a movable node are desired, this function allows you to create them as efficiently as possible because all of the information stored in the movable node (except transformation information) is shared by every instance of that particular movable node.

Use this function when you want multiple instances of a particular movable node in a scene graph. Since every movable node has a built-in transformation component, it would not be possible to have instances of a movable node where each instance has its own position and orientation, unless a function such as *WTmovnode_instance* were available to create separate instances of the movable node.

The specified source node must be a movable node. If the specified source node is a regular (non-movable) node, then this function does nothing and returns NULL. If *parent* is not NULL, then the newly created movable node is added to the scene graph after the last child of the specified parent, and a pointer to the newly created movable node is returned.

See *How Do I Display Multiple Instances Of An Object?* on page A-5.

Note: When you create an instance of a movable node, the instanced node does not inherit the name from the movable node, instead, the instanced node's name is NULL. So if you want the instanced node to have the same name as the movable node, you need to explicitly set its name using *WTnode_setname* (see page 4-49).

Introduction

Geometries are the three-dimensional (3D) objects that form the building blocks for any real-time simulation; they are what you see on your screen. Examples of these graphical objects might include balls, platforms, vehicles, houses, landscapes, but are by no means limited to any particular shape or form. Because geometries are a composition of 3D points (vertices) and the surfaces (polygons) formed from these points, along with material properties (such as color) and texture, you have great freedom in creating just about any shape or form you can imagine as a geometry.

As WTK traverses the scene graph and encounters geometry nodes, it renders these three-dimensional graphical objects to the screen. Geometry nodes contain a pointer to the actual geometry (or *WTgeometry* structure). Geometry nodes are the content in the scene graph that you see drawn to the screen. Light and fog nodes affect the way geometries are drawn (the way they look), but are not visible themselves. Likewise, procedural nodes (such as switches, LOD's) have no visual representation; rather, they choose an appropriate geometry node to draw at the given time.

You can use the following resources to create WTK geometries:

- A CAD or modeling program such as AutoCAD or the World Up Modeler, with the 3D geometry written out in one of the formats supported by WTK.
- WTK's neutral file format (NFF) import facility.
- Importing a geometry defined in one of the file formats supported by WTK (see page 6-2).
- WTK's functions for dynamically constructing predefined geometry types such as cylinders, blocks, and cones.
- WTK's polygon and vertex constructor functions for dynamically constructing custom graphical objects.
- WTK's functions for creating 3D text objects.

- Copying an existing geometry with *WTgeometry_copy* (see page 6-26), with the option of modifying this copy by using WTK's polygon and vertex editing functions.

This chapter discusses some important factors you should consider when constructing geometries for your simulations. It also includes WTK functions for creating, importing, modifying, and optimizing geometries.

Modeling Considerations

The way in which you model geometrical entities affects the appearance as well as the run-time performance of a simulation. This section describes considerations that are important when you are modeling geometries for use in a WTK application. It discusses the file formats that WTK supports, and some techniques for constructing a virtual world consisting of multiple geometrical entities using a CAD or other 3D modeling program.

File Formats Supported by WTK

WTK supports the following 3D geometry and attribute file formats. Geometrical entities are constructed when you call *WTgeometrynode_load*, which is described on page 4-46 in the *Scene Graphs* chapter.

1. **Autodesk DXF format.** Many 3D modeling programs generate this common format. WTK can also output files in DXF format (see *WTgeometry_save* on page 6-26).
2. **Wavefront OBJ format.** The Wavefront modeling tool generates this format. WTK imports the 3D polygonal geometry and curved surfaces that have been polygonalized. Vertex normals and texture vertices are supported for Gouraud shading and texture draping. WTK reads map files and material files, but the only supported properties are diffuse color (Kd) and diffuse texture (map_Kd). An OBJ file describes a single geometry.
3. **Autodesk 3D Studio mesh format.** WTK reads polygonal information from a 3DS file including color and texture information. WTK uses the “ambient” color material value as the color for each polygon, and supports 3DS texture uv values

to allow correct reproduction of the 3D Studio texture application methods. Smoothing groups are supported for Gouraud shading. A 3DS file can contain multiple geometries. See *Notes on the Autodesk 3DStudio Mesh reader* on page 6-5 for more information on 3DS files.

4. **Pro/Engineer RENDER SLP format.** WTK reads the facets in an SLP file as colored polygons with vertex normals for smooth shading. A SLP file contains only one geometry.
5. **MultiGen/ModelGen Flight format.** WTK supports textures, subfaces, external references, transforms, LODs, instances and replicas. A FLT file can contain multiple objects. See *Notes on the MultiGen OpenFlight File Reader* on page 6-5 for more information on MultiGen/ModelGen files.
6. **VideoScape GEO format.** This is a simple 16-color format in which all polygons are back face rejected. A GEO file describes a single geometry.
7. **WorldToolKit Neutral File Format (NFF) and Binary NFF format** (see Appendix F). The NFF format is an efficient and readable representation of 3D geometry. It is also useful as an intermediary format between WTK and formats not otherwise supported. NFF files can be written directly by WTK functions (see *Wtnode_save* on page 4-48). An NFF or binary NFF file can contain multiple geometries.
8. **Virtual Reality Modeling Language (VRML) format.** WTK can read and write VRML 1.0 (.wrl) files. If you are using a URL to read in data, you should specify an http link in your call to the *Wtnode_load* function (see page 4-46). Note that WTK supports http URLs to VRML files only.

WTK supports the VRML files output by CATIA version 4.1.7.

You can load in many other file formats into WTK using third-party geometry conversion programs capable of writing formats that WTK can read. A program such as KANDU software's CADMOVER reads and writes most popular 3D file formats.

WTK VRML 1.0 Limitations

WTK supports most of the VRML 1.0 specification. The VRML 1.0 limitations of WTK include:

- No support for `AsciiText`, `FontStyle`, `IndexedLineSet`, and `PointSet` nodes.
- The crease angle field within `ShapeHints` nodes is ignored.
- By default, WTK ignores scaling factors (if any) within a `Transform` node's transformation. If you want WTK to use the scaling factors of transformations within transform nodes, you can do so by setting the `WTOPTION_XFORMSCALE` option in `WTuniverse_setoption`. However, by doing so, it is likely that intersection tests and math functions pertaining to matrices will operate incorrectly.
- WTK can read and process geometric primitives (such as cone, cube, cylinder, and sphere), but they are internally decomposed into polygons (i.e., they are not internally retained as cone, cube, cylinder and sphere primitives).
- WTK uses its own convention to apply textures to faces without texture coordinates (see *How WTK Applies a Texture to a Polygon* on page 10-5).
- WTK's support for instancing (USE/DEF scheme) does not include all node types. The `Coordinate3`, `Material`, and `Normal` node types cannot be instanced unless they are in the same scope (i.e., there is no separator that differentiates the state of one instance from that of the other).

Exporting a File in the VRML Format

If you are planning to export your scene graph in the VRML format, you will need to ensure that all of your textures are stored as JPEG files. This is because web browsers do not support *.rgb or *.tga files. They require JPEG or GIF image files (GIF images are currently unsupported by WTK.).

When WTK exports a scene graph in the VRML format, the color of textured polygons will be white if texture blending is off. Textured polygons retain their color (i.e., the color is saved in the output file) only if the blending attribute is on. WTK works this way in order to conform with the VRML specifications. (See `WTPoly_settexturestyle` on page 10-23 for information about texture blending).

Notes on the Autodesk 3DStudio Mesh reader

WorldToolKit supports the Autodesk 3DStudio format for Releases 3 and 4. WorldToolKit does not currently support the 3DStudioMAX file format; however, 3DStudioMAX supplies an exporting tool that allows you to save your files in the *.3ds file format that WorldToolKit can utilize. It is important to note that in the original release of 3DStudioMAX there were numerous bugs in the *.3ds exporter that made these files unreadable. A patch for this shortcoming is available on the Kinetex website (www.ktx.com) and has been implemented in the later releases of 3DStudioMAX.

WTK supports the 3DStudio R3/R4 specification except for the following: Points, Lines, Splines, Curves, Face mapping of textures, Box Mapping of textures, and Masks. Multiple geometries in a *.3ds file will be treated by WTK as a single geometry when loaded. If you need to maintain hierarchy in respect to your geometries, you should export your file from 3DStudio/3DStudioMAX as VRML1.0 or VRML2.0 because WTK retains the hierarchical information from these file formats.

Notes on the MultiGen OpenFlight File Reader

In an effort to read newer versions of OpenFlight (.flt) files with greater fidelity, WTK R9 introduces a new OpenFlight reader based upon MultiGen's OpenFlight Read/Write API. The new reader supports MultiGen files greater than V14.2. To gain access to the new reader, *WTuniverse_setoption* now has a new option: `WTOPTION_NEWMGENREAD`. This option must be set to `TRUE` in order to utilize the new reader. See *WTuniverse_setoption* on page 2-24 for a description. The following are notes pertaining to the use of the new reader.

- At the time of this writing, due to limitations in MultiGen's OpenFlight Read/Write API library on UNIX platforms, this reader currently only works with the Microsoft Windows NT/98/95 operating systems.
- WTK supports MultiGen OpenFlight files from (but not including) 14.2 through 15.5. By replacing the reader DLLs distributed by MultiGen, the reader can be updated to take advantage of new versions as they become available.
- The new reader reads the following MultiGen nodes: material palette, texture palette, object, group, group with animation 1, light source records (infinite, point, spot), level of detail, subfaces, switch 2, and external reference. Note 1: A MultiGen animation record is translated to a WTK switch node. Each frame of the animation sequence is a child object of the switch. The first frame is the default

active child. The WTK user must explicitly activate successive frames to produce an animation effect. Note 2: Translated MultiGen switch nodes do not maintain a list of masks. The default active node under the resulting WTK switch node will be the first child of the switch node.

- The reader does not support the following MultiGen nodes: header, eye point, light point, binary space partition, curve, DOF, sound, text, road, and path.
- Using the new reader requires the distribution of API libraries along with the WTK executable. See the “Installation and Hardware Guide” for details.
- Primary colors are applied to polygons only if there is no material applied to the polygon. Secondary colors are unsupported.
- Material properties are always blended with textures.
- If a texture specified in the MultiGen file is missing, a texture representing a red 'X' on a white field will be applied in its place. The user can change this texture by replacing the existing "notex.tga" image in the WTK images directory with one of their own creation.
- A separate material table is created for the MultiGen file and each external reference.
- The name of the table is the name of the file with an "MT" appended to the front and missing the ".flt" suffix. For example, the externally referenced file: "test.flt" will have a corresponding material table called: "MTtest".
- Material table indices in WTK will be one greater than the same entry in the MultiGen material palette. This is to allow the addition of a default material for those polygons without a material or color.

The following are some notes on the old MultiGen OpenFlight file reader. This is the default reader unless you've set the `WTOPTION_NEWMGENREAD` option to `TRUE` using `WTuniverse_setoption`:

- The old reader supports MultiGen OpenFlight files V14.2.
- The old reader reads the following MultiGen nodes: group, object, polygon, subface polygon, LOD, instance, and transformation.
- The old reader does not support the following MultiGen nodes: switch, animation sequences, paths, roads, sounds, and other specialty nodes.
- The old reader supports any hardware platform that WTK runs on.
- The old reader supports the scene graph hierarchy information and color, material and texture information.

Subfaces in MultiGen/ModelGen

Another issue that commonly arises in the OpenFlight file format is that ModelGen and MultiGen permit “subfaces,” polygons that generally are oriented in the same plane as another polygon, but that are intended to appear as if they are on top of the other polygon. When polygons with subfaces are translated literally into the WTK viewing format, Z-buffer roundoff becomes pronounced, resulting in flickering between the coplanar faces as the object is rendered.

Therefore, when WTK encounters subfaces in an OpenFlight file, it translates them by a constant amount in the direction of the parent polygon’s normal vector. When there are multiple levels of subfaces, WTK multiplies the translation magnitude by the number of levels of subfacing. (For example, the subface of a subface is translated twice as far as its parent is.) This moves the polygons so that they no longer lie in the plane of their parent polygon.

Depending on the details of the application, different models may require a different magnitude translation. The following WTK functions are available for accessing and changing this value:

WTuniverse_setsubfaceoffset

See *WTuniverse_setsubfaceoffset* on page 2-22 for a description.

WTuniverse_getsubfaceoffset

See *WTuniverse_getsubfaceoffset* on page 2-22 for a description.

Constructing a World with Multiple Objects

Using a CAD program, you can create a graphical environment for your WTK application in which the various graphical entities have the desired spatial relationships. One technique for accomplishing this is to initially build all of the geometries into one CAD file, positioning the various entities as desired, and then to save out each graphical entity into a separate file. Alternatively, you can save them out as separate objects in a single multi-object file. *File Formats Supported by WTK* on page 6-2 indicates which file formats support multiple objects.

For example, suppose you want to create an office model that consists of office walls, a desk, a chair, and a book on the desk, and that only the chair and the book are movable (dynamic) objects. You might use the following approach:

1. Construct the model containing all of these components and save the file.
2. To create the file that contains the stationary universe geometry (which will be passed to *WTnode_load*), start from the original file, erase the book and the chair, and save the resulting model, which contains just the walls and the desk, to a file.
3. Similarly, to create the file for the chair (which will be passed in to *WTnode_save*), load in the original file, erase the walls, desk, and book, and save the result to a separate file.
4. Similarly, you can create the file from which the book object will be constructed.

If you are using AutoCAD, another approach is to create each graphical object that you wish to load in separately to WTK on a separate layer. Once you construct your model, you can successively, for each object, turn off all layers except the one that the object is on and save the model to a file.

Vertex Normals and Gouraud Shading

A significant improvement can be made in the shading of continuous surfaces if lighting is calculated at each vertex, instead of at the center of each polygon. This is called Gouraud shading, and results in smooth surfaces when used correctly.

A few points about this type of shading:

- It is intended for curved, continuous surfaces, not structures like boxes.
- It requires you to define a normal vector at each vertex.
- It incurs a (usually small) speed penalty since it requires more computation.

WTK automatically uses Gouraud shading to render universes and shaded objects for which vertex normals are present.

You can generate vertex normals in a variety of ways:

- Create them with a modeling program. WTK reads in vertex normals from Wavefront .obj files, 3D Studio .3ds files (using shading groups), MultiGen/ModelGen .flt files, Pro/Engineer RENDER .slp files, and VRML .wrl files.
- Enter them yourself in an NFF file (this is difficult).
- Use the NFF automatic-normal-generation feature to make them for you (see *Appendix F*).
- Call a geometry constructor such as *WTgeometry_newsphere* (see page 6-16) or *WTgeometry_newcylinder* (see page 6-15) with the *gouraud* argument TRUE.
- Create your own geometries in your application code and set vertex normals with *WTgeometry_setvertexnormal* (see page 6-45).
- Create your geometries and use the function *WTgeometry_computevertexnormal* (see page 6-46). This is the simplest way of generating vertex normals.

The vertex normals generated by WTK, when either the auto-normal feature of the NFF file is used or in the geometry constructor functions, have a unit magnitude equal to 1 (one). However, if you specify your own vertex normals (either by using *WTgeometry_setvertexnormal* or by specifying them in an NFF file), the normal you specify is not required to have unit magnitude.

When using vertex normals, you should keep the following in mind:

- WTK does not normalize vertex normals for you.
- The magnitude of vertex normals should be no greater than 1 (one). Typically, vertex normals have unit magnitude. However, for special applications it can be useful to vary the lighting effect by varying the magnitudes of the normal vectors.

Vertex Colors and Radiosity

As with Gouraud shading (described in the preceding section), you can use vertex colors to increase the visual realism of your virtual scene.

For example, vertex color support enables you to render models that have been radiosity preprocessed. A radiosity-preprocessed model stores lighting information such as shadows and reflections as vertex colors—this lighting doesn't then have to be computed at run-time. The result is complex lighting with real-time performance.

A radiosity preprocessor is a program that takes a model and a light source specification as input and generates a new model with lighting information (such as for shadows or reflections) built into it. This involves meshing the original model to contain more detailed color information. This color information is stored at the vertices of the mesh, and WTK (or the hardware that WTK is running on) interpolates between these vertex color values to produce a smooth effect.

The price of better rendering quality is greater polygon complexity, as illustrated by two models in the *models* directory of the WTK distribution. The *oplan.nff* office model has only 157 polygons. By contrast, *oplanrad.nff* — generated from *oplan.nff* by the National Computer Board of Singapore using their radiosity preprocessing program — has 1372 polygons.

A number of radiosity-preprocessed models are provided with this release of WTK. Please see the README file in the *models* subdirectory on your WTK distribution.

In addition to storing lighting information, vertex colors can also represent other values such as the temperature or pressure throughout an object. As with radiosity, you would need to have a program that computes the appropriate vertex colors, and then pass them to WTK.

You can set vertex colors for geometries in the following ways:

- In an NFF file (see *Appendix F*).
- Using a radiosity preprocessing program. ATMA's program called Real Light is a radiosity preprocessor that reads and writes NFF files.
- With the function *WTgeometry_setvertexmatid*, described in *Vertex-level Geometry Editing* on page 6-42.

Back Face Rejection

Using back face rejection is another important technique you should consider when modeling. By eliminating the rendering of polygons that face away from the viewer, you can significantly increase frame rates.

In WTK, the front face of a polygon is the side of the polygon for which the vertices are ordered counter-clockwise. It is also the side from which the polygon normal points. The order of vertices in a polygon is the order in which they are returned using *WTPoly_getvertex* (see page 7-8). For geometries constructed from an NFF file, a

polygon's vertex order is the order specified in the line of the file where the polygon is defined.

WTK's NFF format is very flexible for specifying the back face and front face of polygons, and whether the back faces should be rejected. To switch the back face and front face of a polygon in this format, simply reverse the vertex order in the line of the file where you define the polygon. In addition, you can use the keyword "*both*" in the polygon definition if both sides of the polygon are to be visible, or omitted if the polygon's back face is to be rejected. (See *Appendix F* for a complete specification of the NFF file format.)

Most geometrical entities in the AutoCAD DXF standard are 2 1/2D entities—planar curves with extrusions. When these curves are "closed," it is possible for WTK to unambiguously interpret them in 3D as solids, and know which polygons are seen from their "inside" and which from their "outside." In such cases, in the interest of rendering efficiency, the inside surfaces—the back faces—are rejected at an early stage of the rendering pipeline. The result is that when you go inside these closed solid objects, they disappear, because you are looking at the back faces.

If you wish to have the inside of AutoCAD-modeled geometry appear, you have two choices. You can construct the models so back faces are not rejected. To guarantee this, you should construct a geometry of individual 3D polygons, and extrude open polylines, or polyface meshes, which are not closed in both directions. Alternatively, you can use the *WTPoly_setbothsides* function to change the back face rejection status of polygons.

When using 3D studio, set the two-sided property of the material to TRUE if you want both sides of the polygons with this material to be rendered.

Because of the convenience of the NFF format, it can be advantageous to convert models in other formats (such as DXF) to NFF for greater control over back face rejection. For example, to reverse the front face and back face of an AutoCAD polyline entity would require defining the polyline in the reverse order. It may be more convenient to convert the file to NFF and reverse the order there.

For geometries constructed with WTK's geometry-constructor functions such as *WTgeometry_newsphere* (see page 6-16), polygon back faces are on the inside of the object. Their visibility is specified with the argument *bothsides* passed to these functions. Similarly, WTK's functions for constructing individual polygons take an argument specifying the visibility of back faces.

The function *WTPoly_setbothsides* (see page 7-4) enables you to specify whether both sides of a polygon are visible. The polygons that are passed to this function can be obtained

interactively, using *WTwindow_pickpoly* (see page 17-20) or, programmatically with the functions *WTgeometry_getpolys* (see page 6-32), *WTgeometry_id2poly* (see page 6-33), or *WTgeometry_beginpoly* (see page 6-23).

Overlapping Polygons

When building models it is best to avoid the use of coplanar polygons or surfaces — that is, surfaces that overlap and lie in the same plane. An example of coplanar polygons is a building facade with a door in it. If this model is loaded into WTK, WTK would not know which surface is to appear in front, which can produce unexpected results. On Z-buffered systems, Z-buffer roundoff results in image flashing between coplanar surfaces where they overlap. On non-Z-buffered systems, it is possible for the order in which the coplanar polygons are drawn to change as the simulation runs, for example, if you were to interactively texture or change the color of these surfaces.

To avoid this problem, you should construct your model either:

1. So that the surfaces are not in the same plane, or
2. So that they do not overlap.

In the first approach, you would construct the wall and door surfaces so that the door is in a plane in front of the plane of the wall. On Z-buffered systems, how far the planes must be separated to avoid flashing depends on the resolution of the Z-buffer and on the locations of the window's hither and yon clipping planes (see *WTwindow_sethithervalue* on page 17-18 and *WTwindow_setyonvalue* on page 17-19). On non Z-buffered systems, the planes must be separated by at least the value of *WTFUZZ* (a defined constant equal to 0.004). If you are using MultiGen or ModelGen as your modeler, see also the discussion of subface offsets under *Notes on the MultiGen OpenFlight File Reader* on page 6-5.

In the second approach (constructing the model so that the surfaces do not overlap) you would create a hole in the wall and fit the door rectangle into the hole. This approach has the advantage that the surfaces would appear exactly coplanar when viewed from an angle. A disadvantage of this approach is that creating the hole in the wall generates extra polygons, either in the modeling program, or when the surface is loaded into WTK and rendered there.

Because models are stored with finite precision, coplanarity problems may sometimes also arise when reading in a DXF or NFF file that was previously written out with

WTgeometry_save. This problem is particularly severe when scaling models up by factors of 10 or more. The solution is to load the model into a CAD program, scale it up there, and then load it into WTK.

Roundoff and Scaling

On a digital computer, floating-point quantities are usually represented imprecisely. In traditional hardware Z-buffered rendering, this finite resolution frequently results in undesirable “flashing,” as distant surfaces that are parallel and close to one another alternately obstruct one another.

For performance and memory efficiency reasons, WTK stores all coordinates as single-precision, floating-point values. Since roundoff can occur in a number of places in the rendering pipeline, it is important that the geometry that is read in be scaled appropriately to avoid mistaking vertices that are close to one another for identical vertices.

For this purpose, a scale factor is supplied as one of the parameters to the *WTnode_load* (see page 4-46) function. This factor should be specified so that distinct vertices, once scaled, are separated by a distance of at least the defined constant *WTFUZZ* (equal to 0.004), a value used in many WTK comparisons.

When passing in a scale factor to *WTnode_load*, the factor should be no smaller than necessary. If you load in tiny models, or if you scale down models by supplying an extremely small scale factor, vertices in the model may not cleanly connect and polygons in the model may disappear. This is because WTK merges vertices in a polygon that are separated by less than the floating-point fuzz value in all dimensions. When vertices are merged, a small polygon can end up with fewer than three vertices and is discarded. These discarded polygons do not reappear even if *WTgeometry_scale* is later called for the object. An example of a model that might exhibit these problems is one that is very small (less than one unit) and contains many polygons. To fix this, simply load the model with a larger scale value. This should make rendering problems disappear.

It is also important not to use a scale factor that is *larger* than necessary. As the scale factor increases, the number of distances within the model that are larger than the fuzz value also increases. When scaled up, polygons that abut also tend to overlap. On non Z-buffered systems, this results in an increase in the number of polygons that are used to draw the scene, so that as the scale factor increases, the frame rate decreases.

Scaling your model up or down too much is not desirable. As a rule of thumb, you should not scale by more than 2 orders of magnitude, i.e., by more than 100.0 or less than 0.01. If you need to scale by more than this, it is best to perform the scaling in your modeling program prior to saving out the geometry file. Then load the file into WTK. In general, model your objects so that you can use a scale factor of 1.0 for *WTnode_load*.

In addition, WTK expects the vertices in each polygon to lie approximately in the same plane. If they do not, then calculations involving polygon normals will be inaccurate to whatever degree the surface is non-planar. If this is the case, WTK automatically subdivides each non-planar polygon into two or more planar polygons. WTK also expects all polygons to be convex. If a non-convex polygon is detected, WTK automatically subdivides it into a number of convex polygons.

Creating Predefined Geometries

WTK provides functions for dynamically creating a variety of basic geometry types: cylinders, blocks (boxes), cones, spheres, hemispheres, rectangles, truncated cones, extrusions, and 3D text objects. These functions provide an alternative to constructing graphical objects by loading them in from a geometry file, as is done with *WTnode_load* or by creating custom geometries using *WTgeometry_begin*.

Each of the functions in this section takes an argument, *bothsides*, which specifies whether both sides of each polygon in the geometry are to be visible. If *FALSE*, then back-facing polygons are rejected (not rendered). Polygon back faces for these geometrical entities are their inside surfaces. See *Back Face Rejection* on page 6-10 for more information on this subject.

All the geometries constructed with the functions in this section are colored white by default.

WTgeometry_newcylinder

```
WTgeometry * WTgeometry_newcylinder(  
    float height,  
    float radius,  
    int tess,  
    FLAG bothsides,  
    FLAG gouraud);
```

This function creates and returns a pointer to a new cylinder geometry centered at the world coordinate frame origin and oriented vertically. The cylinder's height and radius are given by the *height* and *radius* arguments. The *tess* (tessellation) argument gives the number of polygons to use in approximating the cylinder. For example, a *tess* value of 4 creates a rectangular block-shaped cylinder, *tess=3* creates a triangular prism-shaped cylinder, and *tess=20* creates a cylinder with 20-sides along its curved surface.

If the *gouraud* flag is TRUE, then on systems that support Gouraud shading, outward-pointing vertex normals parallel to the cylinder base are defined.

The parameter *bothsides* indicates whether polygon back faces are visible. This function returns a pointer to the constructed geometry if successful. If unsuccessful, NULL is returned.

The default name of a new geometry created with this function is "cylinder."

WTgeometry_newblock

```
WTgeometry * WTgeometry_newblock(  
    float lx,  
    float ly,  
    float lz,  
    FLAG bothsides);
```

This function creates and returns a pointer to a new block (box) geometry. The block is created with X, Y, and Z dimensions given by the *lx*, *ly*, and *lz* arguments, and is centered at the world origin.

The parameter *bothsides* indicates whether polygon back faces are visible. This function returns a pointer to the constructed geometry if successful. If unsuccessful, NULL is returned.

For example, to construct a white cube that is 10.0 units on a side and visible only from the exterior, you would call:

```
WTgeometry *cube;
cube = WTgeometry_newblock(10.0, 10.0, 10.0, FALSE);
```

The default name of a new geometry created with this function is “block.”

WTgeometry_newcone

```
WTgeometry * WTgeometry_newcone(
    float height,
    float radius,
    int tess,
    FLAG bothsides);
```

This function creates and returns a pointer to a new cone-shaped geometry. The cone is centered at the world coordinate frame origin and oriented vertically. The cone’s height and radius are given by the *height* and *radius* arguments. The *tess* argument gives the number of polygons to use in approximating the cone. For example, a *tess* value of 4 creates a 4-sided pyramid, *tess=3* creates a tetrahedron, and *tess=20* creates a cone with 20-sides along its curved surface.

The parameter *bothsides* indicates whether polygon back faces are visible. This function returns a pointer to the constructed geometry if successful. If unsuccessful, NULL is returned.

The default name of a new geometry created with this function is “cone.”

WTgeometry_newsphere

```
WTgeometry * WTgeometry_newsphere(
    float radius,
    int nlat,
    int nlong,
    FLAG bothsides,
    FLAG gouraud);
```

This function creates and returns a pointer to a new sphere geometry. The sphere is centered at the world coordinate frame origin. The sphere's radius is given by the *radius* parameter. The *nlat* and *nlong* parameters give the number of latitude and longitude subdivisions to use in approximating the sphere. For example, *nlat=2* and *nlong=4* creates an octahedron, and *nlat=8* with *nlong=16* creates a sphere with 128 polygons.

The parameter *bothsides* indicates whether polygon back faces are visible. This function returns a pointer to the constructed geometry if successful. If unsuccessful, NULL is returned.

If the *gouraud* flag is TRUE, then on systems that support Gouraud shading, outward-pointing vertex normals are defined.

The default name of a new geometry created with this function is "sphere."

WTgeometry_newhemisphere

```
WTgeometry * WTgeometry_newhemisphere(  
    float radius,  
    int nlat,  
    int nlong,  
    FLAG bothsides,  
    FLAG gouraud);
```

This function creates and returns a pointer to a new hemisphere. It is exactly like *WTgeometry_newsphere* except that only the top half of the sphere is created.

The default name of a new geometry created with this function is "hemisphere."

WTgeometry_newrectangle

```
WTgeometry * WTgeometry_newrectangle(  
    float height,  
    float width,  
    FLAG bothsides);
```

This function constructs a new geometry composed of a single rectangle (*height* is the Y dimension, *width* is the X dimension). The rectangle is created in an upright position with its center at the world coordinate frame origin.

When viewed from $Z = -\infty$ its vertices run counterclockwise, so that the rectangle normal points in the $-Z$ direction. In other words, a viewpoint facing in the $+Z$ direction would see the front face of the rectangle.

The parameter *bothsides* indicates whether polygon back faces are visible. This function returns a pointer to the constructed geometry if successful. If unsuccessful, NULL is returned.

NULL is also returned if there is insufficient memory, or if the *height* or *width* parameter is less than or equal to zero.

The default name of a new geometry created with this function is “rectangle.”

WTgeometry_newtrunccone

```
WTgeometry *WTgeometry_newtrunccone(  
    float height,  
    float toprad,  
    float baserad,  
    int tess,  
    FLAG bothsides,  
    FLAG gouraud);
```

This function constructs a new geometry consisting of a single truncated cone. The truncated cone is in an upright position and centered at the world coordinate frame origin.

The cone's height is given by the *height* parameter, its top radius by *toprad* and bottom radius by *baserad*. The *baserad* can be larger or smaller than *toprad*, i.e., the cone can point up or down. The tessellation argument *tess* gives the number of polygons to use to approximate the cone. The parameter *bothsides* indicates whether polygon back faces are visible. If the *gouraud* flag is TRUE, outward-pointing normals for the side polygons are defined on systems that support Gouraud shading. The normals are perpendicular to the edges along the sides of the geometry.

This function returns a pointer to the constructed geometry if successful. If unsuccessful, NULL is returned. NULL is also returned if the height, top radius or base radius is smaller than the WTK tolerance factor (WTFUZZ) or if the tessellation value is smaller than three.

The default name of a new geometry created with this function is “trunc_cone.”

WTgeometry_newextrusion

```
WTgeometry *WTgeometry_newextrusion(  
    WTp2 points[],  
    int numpts,  
    float height,  
    FLAG bothsides,  
    FLAG gouraud);
```

This function makes a new geometry by extruding a given contour the distance specified (in the parameter *height*). The parameter *bothsides* indicates whether polygon back faces are visible. This function returns a pointer to the constructed geometry if successful. If unsuccessful, NULL is returned.

If the *gouraud* flag is TRUE, outward-pointing normals for the side polygons are defined on systems that support Gouraud shading. The normals are perpendicular to the edges along the sides of the geometry.

The extruded geometry will be created in an upright position and centered at the world coordinate frame origin. The contour is extruded a distance of (*height/2.0*) above and below the x-z plane.

The number of points must be between 3 and 256. Note that the input data array is of type *WTp2*, not *WTp3*. The point array is assumed to be a non-self-intersecting (does not intersect with itself) contour in the X-Z plane. The contour can include concavities.

The input data array is scanned and points closer than WTFUZZ to another point are discarded. For this reason and others, the contents of the data array may be changed by the function. Therefore, a copy should be retained elsewhere if the data is needed after the function call.

If successful, the function returns a pointer to the new geometry, otherwise NULL is returned. NULL is also returned if *height* is smaller than the WTK tolerance factor (WTFUZZ) or if *numpts* is less than 3.

The default name of a new geometry created with this function is “extrusion.”

WTgeometry_newtext3d

```
WTgeometry *WTgeometry_newtext3d(
    WTfont3d *font,
    char *string);
```

This function takes a character string (such as “Hello”) and creates a WTK geometry from that string, using the specified 3D font. The geometry is created with the first character’s base point placed at the world coordinate frame origin (0, 0, 0). The characters are placed sequentially along the +X axis using the current font spacing. This is shown below in figure 6-1.

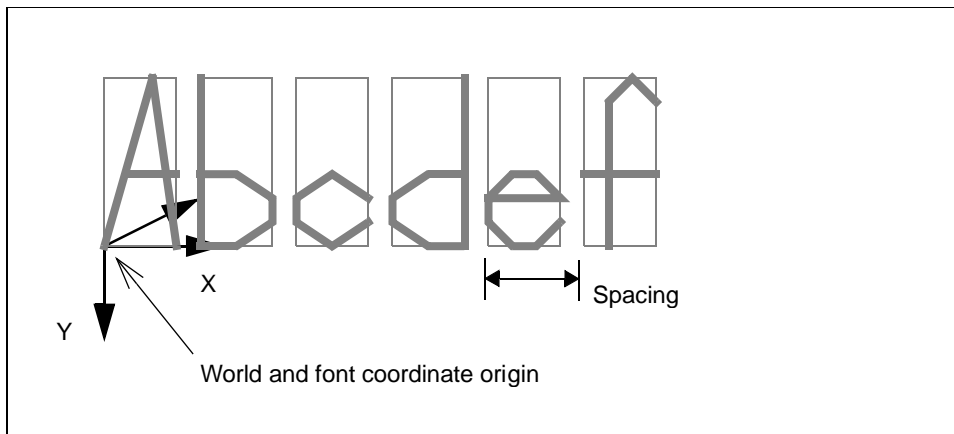


Figure 6-1: 3D text geometry creation

Once the geometry is constructed, you must use the function *WTgeometrynode_new* to place the text in a geometry node in your scene graph.

You may also wish to use *WTgeometry_scale* or *WTgeometry_stretch* to scale the size or stretch the height, width and/or depth of the text geometry. The color and/or texture of characters in a text geometry are specified in the NFF font file. Once the text geometry is constructed, you can change the texture and color attributes using *Wtpoly_settexture* or the material functions such as *WTgeometry_getmtable* and *WTmtable_setvalue*.

Geometries constructed with *WTgeometry_newtext3d* are assigned the name “text.” This is what is returned if you call *WTgeometry_getname*, to query the name of the geometry. If the character string passed to *WTgeometry_newtext3d* has characters that are not represented in the font, blank spaces are created in the text geometry in place of those characters. If none of the characters in the character string are represented in the font, or if

the character string passed in is made up entirely of spaces, then *WTgeometry_newtext3d* returns NULL. The function *WTfont3d_charexists* can be used to determine whether a particular character is in a WTK 3D font (a *WTfont3d*).

This example uses WTK 3D text functions to add a virtual “Hello world” to a scene graph.

```
WTnode *root
WTfont3d *font;
WTgeometry *geo;
WTnode *child;
font = WTfont3d_load("alphabet.nff");
if ( !font )
    WTwarning("Couldn't load 3D font.\n");
else {
    /* create the text geometry from a character string */
    geo = WTgeometry_newtext3d(font, "Hello world");
    child=WTgeometrynode_new(root,geo);
}
```

Creating Custom Geometries

In addition to loading geometries from files and creating them with the high-level object constructor functions, you can also define a geometry polygon-by-polygon with lower-level calls.

To create a geometry with WTK’s geometry constructor functions, follow these steps:

1. Initialize the geometry by calling *WTgeometry_begin*.
2. Add vertices to the geometry using *WTgeometry_newvertex*.
3. Add polygons to the geometry.
4. For each polygon, call *WTgeometry_beginpoly*.
5. Add vertices (that have already been added to the geometry with *WTgeometry_newvertex*) to the polygons using *WTPoly_addvertex* or *WTPoly_addvertexptr* (see page 7-10).

6. Call *Wtpoly_close* for each polygon.
7. When you are done adding polygons, call *WTgeometry_close*.

Note that only one geometry can be constructed at a time. You must complete the definition of one geometry before beginning the definition of a new one. The default color assigned to polygons contained in custom geometries is white and each back facing polygon is rejected by default.

WTgeometry_begin

```
WTgeometry * WTgeometry_begin(  
    void);
```

This function begins the construction of a new geometry and obtains a pointer to a new, empty geometry. As described above, the geometry is then defined by adding vertices and polygons to it and then finishing the geometry with a call to *WTgeometry_close*. The completed geometry is not part of the simulation until the geometry is added to a scene graph with a call to *WTgeometrynode_new*.

Note that only one geometry can be constructed at a time. Before calling this function to begin the definition of a new geometry, you must call *WTgeometry_close* for any geometry currently under construction.

An example of using this function is provided under *WTgeometry_close*.

WTgeometry_newvertex

```
WTvertex *WTgeometry_newvertex(  
    WTgeometry *geom,  
    Wtp3 p);
```

This function adds a vertex with coordinates (X, Y, Z) in the *Wtp3* structure to a geometry. The return value is TRUE if successful, otherwise FALSE is returned. For example, if a NULL geometry pointer is passed in, then FALSE is returned.

An example of using this function is provided under *WTgeometry_close*.

WTgeometry_beginpoly

```
WTPoly *WTgeometry_beginpoly(  
    WTgeometry *geom;
```

This function returns a new, empty polygon assigned to the specified geometry (the geometry pointed to by the *geom* argument). The polygon should be subsequently defined with *WTPoly_addvertexptr* (or *WTPoly_addvertex*) and finally completed with *WTPoly_close*. See the example provided under *WTgeometry_close* (page 6-23) for the usage of this function.

By default, the color of the polygon is white with its back face rejected. The function *WTPoly_setrgb* can be used to change the polygon's color and *WTPoly_setbothsides* can be used to change its back face rejection status. See *Back Face Rejection* on page 6-10 for more information about back face rejection.

NULL is returned if this function is unsuccessful; for example, if a NULL geometry pointer is passed in.

WTgeometry_close

```
FLAG WTgeometry_close(  
    WTgeometry *geom);
```

This function finishes the definition of a geometry (no more vertices or polygons can be added), and internally prepares the geometry for use in the simulation. WTK will automatically subdivide each non-planar polygon as well as each non-convex polygon into two or more convex/planar polygons. The geometry is not, however, part of a scene graph (and therefore is not rendered) until you call *WTgeometrynode_new* on page 4-44.

If you later decide to edit the geometry, use the function *WTgeometry_beginedit* (see page 6-42).

The default name of a new geometry created with this function is “untitled.”

This function returns FALSE if *geom* is not a valid pointer, or if the geometry consists of less than three vertices. Note that vertices which are too close together (a distance of less than WTFUZZ units apart), will be merged into one vertex. *WTgeometry_close* or *WTgeometry_closesmooth* should be used in conjunction with *WTgeometry_begin*. *WTgeometry_begin* creates a new 'open' geometry, which can then be constructed by adding

vertices and polygons. Use *WTgeometry_close* or *WTgeometry_closesmooth* to end the construction of an open geometry. If *WTgeometry_close* is called on a geometry that is not currently open, *FALSE* is returned.

The following example illustrates the use of the geometry constructor functions to create a geometry consisting of a single rectangular polygon.

```
/* Construct a geometry consisting of a single rectangle in the
X-Y plane using the WTK geometry constructor functions. */
WTgeometry *make_rect(float lx, float ly, FLAG bothsides)
{
    WTgeometry *geom;
    WTPoly *poly;
    WTP3 p;
    int j, k;

    /* initialize a new, empty geometry*/
    geom = WTgeometry_begin();

    /* add vertices to the geometry */
    p[Z] = 0.0;
    for ( j = 0 ; j < 2 ; j++ )
        for ( k = 0 ; k < 2 ; k++ ) {
            p[X] = j ? -lx/2 : lx/2;
            p[Y] = k ? -ly/2 : ly/2;
            WTgeometry_newvertex(geom, p);
        }

    /* add an empty polygon to the geometry */
    poly = WTgeometry_beginpoly (geom);

    /* add the vertices to the polygon */
    for ( k = 0 ; k < 4 ; k++ ) WTPoly_addvertex(poly, k);

    /* mark the end of the polygon's construction */
    WTPoly_close(poly);
}
```

```
/* make both sides of the polygon visible
(polygons are by default back face rejected) */
if ( bothsides) WTpoly_setbothsides(poly, TRUE);

/* finish the geometry definition */
WTgeometry_close(geom);
return geom;
}
```

WTgeometry_closesmooth

```
FLAG WTgeometry_closesmooth(
    WTgeometry *geom);
```

This function is identical to *WTgeometry_close* except that it also computes the vertex normals for each of the geometry's vertices. Like *WTgeometry_close* this function finishes the definition of a geometry (no more vertices or polygons can be added), and internally prepares the geometry for use in the simulation. WTK will automatically subdivide each non-planar polygon as well as each non-convex polygon into two or more convex/planar polygons. The geometry is not, however, part of a scene graph (and therefore is not rendered) until you call *WTgeometrynode_new* on page 4-44.

If you later decide to edit the geometry, use the function *WTgeometry_beginedit* (see page 6-42).

The default name of a new geometry created with this function is “untitled.”

This function returns `FALSE` if `geom` is not a valid pointer, or if the geometry consists of less than three vertices. Note that vertices which are too close together (a distance of less than `WTFUZZ` units apart), will be merged into one vertex. *WTgeometry_closesmooth* or *WTgeometry_close* should be used in conjunction with *WTgeometry_begin*. *WTgeometry_begin* creates a new 'open' geometry, which can then be constructed by adding vertices and polygons. Use *WTgeometry_closesmooth* or *WTgeometry_close* to end the construction of an open geometry. If *WTgeometry_closesmooth* is called on a geometry that is not currently open, `FALSE` is returned.

Other Geometry Functions

See the description of *WTnode_load* on page 4-46 and *WTgeometrynode_load* on page 4-46 in the *Scene Graphs* chapter for information about loading a new geometry.

WTgeometry_copy

```
WTgeometry *WTgeometry_copy(  
    WTgeometry *geom);
```

This function creates and returns a new geometry by copying an existing geometry. The name of the geometry is also copied.

WTgeometry_delete

```
int WTgeometry_delete(  
    WTgeometry *geom);
```

This function deletes a geometry and frees its memory. If there is a geometry node that uses this geometry, the geometry node is also deleted.

WTgeometry_save

```
FLAG *WTgeometry_save(  
    WTgeometry *geom,  
    char *filename,  
    WTPq *pq,  
    int filetype);
```

This function saves a geometry to file using the specified filename. If you pass in a non-NULL *pq* argument, then this position and orientation are also saved to the file.

If the geometry you are saving is textured and *WTFILETYPE_DXF* is specified, it is important to keep in mind that any texture rotations, scale factors, translations, mirroring operations, and texture uv values are *not* saved out. When the object is read back in, textures must be reapplied with *Wtpoly_settexture*.

If *WTFILETYPE_NFF* or *WTFILETYPE_BFF* is specified, then information about these texturing operations *is* saved out. Refer to the NFF format specification described in *Appendix F*.

Refer to the *Textures* chapter (starting on page 10-1) for geometry-texturing functions. Note that this function saves only individual geometries. If you want to save a group of geometries, you have to first merge them together into a single one using *WTgeometry_merge* described below.

WTgeometry_merge

```
WTPoly *WTgeometry_merge(  
    WTgeometry *geometry1,  
    WTgeometry *geometry2);
```

This function merges *geometry2* into *geometry1* and returns a pointer to the merged geometry's first polygon. In the process of merging the geometries, a new material table, composed of *geometry1* and *geometry2*'s materials, is created and assigned as *geometry1*'s new material table. Please note that this function does not take into account the geometries' relative positions in the scene when merging the geometries together. To properly align the geometries, you must position and orient them using geometry transform functions such as *WTgeometry_scale*, *WTgeometry_translate*, and/or *WTgeometry_transform*.

The following is an example of how to merge two geometries maintaining the relative positions of each (i.e., as they appear in a virtual world).

```
{  
    WTgeometry *g1, *g1_copy, *g2, *g2_copy;  
    WTnodepath *np1, *np2;  
    WTM4 m4;  
    WTPq pq;  
  
    g1 = WTnode_getgeometry( node1 );  
    g2 = WTnode_getgeometry( node2 );  
    /* copy the geometries so that we don't alter the ones being rendered  
    on the scene */  
    WTgeometry_copy( g1, g1_copy);  
    WTgeometry_copy( g2, g2_copy);
```

```
/* get the nodepath to each node and the transform matrix along that
   nodepath. The geometry must be transformed by this matrix */
np1 = WTnodepath_new( node1, rootnode, 0);
WTnodepath_gettransform( np1, m4);
WTm4_2pq( m4, &pq);
WTgeometry_transform( g1_copy, &pq);
np2 = WTnodepath_new( node2, rootnode, 0);
WTnodepath_gettransform( np2, m4);
WTm4_2pq( m4, &pq);
WTgeometry_transform( g2_copy, &pq);

WTgeometry_merge( g1_copy, g2_copy );
/* we now have the merged geometry in g1_copy */
/* Note that to merge other geometries into g1_copy now,
   we do not need to consider the transform along the nodepath
   to g1 any more. The geometry g1_copy has the world positions
   of g1 and g2 */
}
```

Geometry Properties

WTgeometry_getmidpoint

```
void WTgeometry_getmidpoint(
    WTgeometry *geom,
    WTp3 p);
```

This function obtains the midpoint of a geometry's extents box in local coordinates. The midpoint is returned in *p*.

WTgeometry_getradius

```
float WTgeometry_getradius(  
    WTgeometry *geom);
```

This function returns the radius of the specified geometry. A geometry's radius is defined as the distance from the midpoint of the geometry to a corner of the geometry's bounding box.

WTgeometry_getextents

```
void WTgeometry_getextents(  
    WTgeometry *geom,  
    WTp3 extents);
```

This function retrieves the extents of the geometry's axis-aligned bounding box and stores them in *extents*. The *extents* argument is a *WTp3* containing the X, Y, and Z dimensions of the geometry's bounding box.

WTgeometry_setname

```
void WTgeometry_setname(  
    WTgeometry *geom,  
    char *name);
```

This function assigns a name to a geometry. The default name of a pre-defined geometry is described in the corresponding *WTgeometry_new** function description. The default name for a custom geometry is "untitled", and geometries read in from file are assigned the name of the corresponding geometry. Geometries read in from a file using *WTmovnode_load* assign the geometry's name to the movable node's name, while the name of the underlying *WTgeometry* is NULL. Note that this is done only if the file contains a single geometry. If the file has multiple geometries, the movable that is created is not given a name, and the individual geometries maintain their names.

WTgeometry_getname

```
char *WTgeometry_getname(  
    WTgeometry *geom);
```

This function returns the name of the specified geometry. A geometry may have obtained its name by a call to *WTgeometry_setname* or from a name read in from the file (if the geometry was loaded from a file).

If no name has been assigned to a geometry, then the default name of the geometry is “untitled” unless the geometry is one of the predefined types, in which case the name corresponds to the type, i.e., *WTgeometry_newcone* has the default name “cone.”

Materials used with Geometries

As described in the *Materials* chapter (starting on page 8-1), you can define a geometry’s material in its own material table. Each geometry references a single material table, from which the geometry’s material properties are obtained.

WTgeometry_setmtable

```
void WTgeometry_setmtable(  
    WTgeometry *geometry,  
    WTmtable *mtable);
```

This function causes the indicated geometry to reference the specified material table. It overrides any previous setting; a geometry may only refer to one material table at any one time. By default, every geometry is associated with its own material table, i.e., a material table is created for every newly constructed geometry. You can use *WTgeometry_getmtable* to retrieve the material table associated with any geometry.

Use the material table functions described in the *Materials* chapter (starting on page 8-1) to add new entries to the material table, or to alter existing ones.

Note: If a material table is not referenced by any geometry in the scene graph, it is automatically deleted. For example, suppose you load a material table using *WTmtable_load* (see page 8-11) and associate it with a geometry using

WTgeometry_setmtable. Now, if you delete this geometry, the material table also gets deleted (if it is not referenced by any other geometry). Hence, you could not associate another geometry to this material table later using *WTgeometry_setmtable*.

See also the functions *WTPoly_setmatid* on page 7-3 and *WTgeometry_setvertexmatid* on page 6-48.

WTgeometry_getmtable

```
WTmtable *WTgeometry_getmtable(  
    WTgeometry *geom);
```

This function returns a pointer to the material table referenced by the geometry.

WTgeometry_setmatid

```
FLAG WTgeometry_setmatid(  
    WTgeometry *geom,  
    int id);
```

This function changes the material table index of all the polygons in the specified geometry to the index value specified by the *id* argument. The indices remain static (do not change) if the material table is changed or deleted. A modulus operation will occur at render time if *id* is greater than the number of materials in the material table referenced by the geometry.

A negative material index is not allowed, and therefore this function returns FALSE if you pass in a negative value for *id*.

WTgeometry_setrgb

```
void WTgeometry_setrgb(  
    WTgeometry *geom,  
    unsigned char red,  
    unsigned char green,  
    unsigned char blue);
```

This function specifies the 24-bit color value of a geometry (red, green, and blue). The valid range for each of red, green, and blue is from 0 to 255. Note that even though WTK allows the user to specify color using red, green, and blue color components, WTK's internal

representation makes use of the geometry's material table by either finding a material table entry whose color (red, green, and blue) matches the color specified by the user, or by creating a new entry in the material table.

In the following example, a geometry's color is set to yellow.

```
WTgeometry *geom;  
WTgeometry_setrgb(geom,255,255,0);
```

Geometry Polygons and Vertices

The first two functions in this section can be used to obtain *WTPoly* pointers which can then be passed to other WTK functions. Other WTK functions that return polygon pointers are *WTwindow_pickpoly*, *WTgeometry_beginpoly*, *WTPoly_next* and *WTgeometry_id2poly*.

WTgeometry_getpolys

```
WTPoly *WTgeometry_getpolys(  
    WTgeometry *geom);
```

This function returns a pointer to the first polygon contained in the specified geometry. Use the function *WTPoly_next* to iterate through the list of polygons contained in the geometry.

WTgeometry_numpolys

```
int WTgeometry_numpolys(  
    WTgeometry *geom);
```

This function returns the total number of polygons in the specified geometry.

WTgeometry_getvertices

```
WTvertex *WTgeometry_getvertices(  
    WTgeometry *geom);
```

This function returns a pointer to the first vertex in the geometry. Use the function *WTvertex_next* to iterate through the list of vertices in the geometry.

WTvertex_next

```
WTvertex *WTvertex_next (  
    WTvertex *vertex);
```

This function returns the next vertex in a geometry's list of vertices.

WTgeometry_id2poly

```
WPoly * WTgeometry_id2poly(  
    WTgeometry *geom,  
    short id);
```

This function returns a pointer to the specified polygon in the specified geometry, where *id* is the polygon's *id* number. See *WPoly_setid* on page 7-6. A polygon can also be given an *id* by editing the line in an NFF file that defines the polygon. See Appendix F, *WTk Neutral File Format*.

WTgeometry_setrenderingstyle

```
FLAG WTgeometry_setrenderingstyle(  
    WTgeometry *geom,  
    int modes,  
    int style);
```

This function sets the rendering style for a geometry. The *modes* argument is a bitmask of the rendering flags to change and the *style* argument is the value to which the rendering flags are set. Valid values for the *style* argument are *TRUE*, *FALSE*, and *WTRENDER_DEFAULT*. These are explained further in the discussion that follows.

The *modes* argument can be a combination of the following bits:

<code>WTRENDER_ANTIALIAS</code>	enables anti-aliasing
<code>WTRENDER_BEST</code>	enables all of these modes
<code>WTRENDER_GOURAUD</code>	enables Gouraud shading and lighting (this is an outdated WTK 2.1 mode, see note below)
<code>WTRENDER_LIGHTING</code>	turns on lighting
<code>WTRENDER_PERSPECTIVE</code>	enables perspective texture
<code>WTRENDER_SMOOTH</code>	enables smooth shading (Gouraud shading)
<code>WTRENDER_TEXTURED</code>	enables texturing

Mode can also be set to `WTRENDER_WIREFRAME` or `WTRENDER_NOSHADER`. The actual rendering used to render the geometry will be a combination of the universe's rendering style (see `WTuniverse_setrendering` on page 2-18) and the geometry's rendering style. You can combine different modes by using the bitwise OR operator (`|`), as shown in the examples in this section.

The return value is `TRUE` or `FALSE`, depending on whether the function succeeds. For example, `WTgeometry_setrenderingstyle` returns `FALSE` if called on a prebuilt geometry, since you cannot change the rendering style of prebuilt geometry.

Note: `WTRENDER_GOURAUD` is an outdated WTK 2.1 style that has been replaced with `WTRENDER_LIGHTING` and `WTRENDER_SMOOTH`.

`WTgeometry_setrenderingstyle` lets you modify some or all of the rendering modes on a per geometry basis. By default, a geometry takes its rendering style from the universe (i.e., `WTuniverse_setrendering`). The difference between the syntax of the two functions is that `WTuniverse_setrendering` takes an absolute group of bitfields while `WTgeometry_setrenderingstyle` takes a list of bitfields to change and the value to which to change them).

The value of *style* can be `TRUE` (turn on), `FALSE` (turn off), or `WTRENDER_DEFAULT` (sets the mode to its default).

For example:

```
WTgeometry_setrenderingstyle(mygeom,WTRENDER_PERSPECTIVE|  
                             WTRENDER_TEXTURED,TRUE);
```


tells WTK to turn on perspective correction and texturing for this geometry. The other rendering modes (i.e., `WTRENDER_LIGHTING`, `WTRENDER_SMOOTH`, etc.) will continue to be taken from the universe rendering mode.

Once you've modified a geometry rendering mode (setting it to `TRUE` or `FALSE`), you may want to tell WTK to revert to using universe rendering mode again. This can be done by passing the `WTRENDER_DEFAULT` value into the *style* argument.

For example:

```
/* this forces wtk to turn off lighting for this geometry despite the universe rendering
mode for lighting */
WTgeometry_setrenderingstyle(mygeom, WTRENDER_LIGHTING, FALSE);
/* this tells wtk to go back to using the universe rendering mode for
lighting for this geometry */
WTgeometry_setrenderingstyle(mygeom, WTRENDER_LIGHTING,
                             WTRENDER_DEFAULT);
```

To revert all the rendering modes of a geometry back to the universe rendering modes, you can use `WTRENDER_ALLMODES` in the *modes* argument. You must use it with `WTRENDER_DEFAULT`.

For example:

```
WTgeometry_setrenderingstyle(mygeom, WTRENDER_ALLMODES,
                             WTRENDER_DEFAULT);
```

This renders the geometry using the universe rendering modes.

WTgeometry_getrenderingstyle

```
int WTgeometry_getrenderingstyle(
    WTgeometry *geom);
```

This function returns the current rendering style of the specified geometry. See *WTgeometry_setrenderingstyle*, above. This style is the composite effect of both the universe's rendering style and the geometry's rendering style.

By default, a geometry's rendering style is the same as the universe's rendering style. With *WTgeometry_setrenderingstyle* you can "customize" the geometry's rendering style by

overriding some or all of the universe style with different values. The composite between the universe style and the customizing you have done with *WTgeometry_setrenderingstyle* is what *WTgeometry_getrenderingstyle* returns. For example, if you set the universe rendering:

```
WTuniverse_setrendering(WTRENDER_SMOOTH|WTRENDER_LIGHTING|
                        WTRENDER_TEXTURED);
```

then *WTgeometry_getrenderingstyle(mygeom)* will return:

```
WTRENDER_SMOOTH|WTRENDER_LIGHTING|WTRENDER_TEXTURED
```

If you then turn off the geometry's lighting with this command:

```
WTgeometry_setrenderingstyle(mygeom, WTRENDER_LIGHTING, FALSE);
```

WTgeometry_getrenderingstyle(mygeom) will now return:

```
WTRENDER_SMOOTH|WTRENDER_TEXTURED
```

Note that the universe style is still `SMOOTH|LIGHTING|TEXTURED`. If you then turn on perspective texturing and anti-aliasing for the geometry with this command:

```
WTgeometry_setrenderingstyle(mygeom,
                              WTRENDER_PERSPECTIVE|
                              WTRENDER_ANTIALIAS, TRUE);
```

then *WTgeometry_getrenderingstyle(mygeom)* will return:

```
WTRENDER_SMOOTH|WTRENDER_TEXTURED|WTRENDER_PERSPECTIVE|
WTRENDER_ANTIALIAS
```

If you then set the lighting and perspective correction for the geometry to the universe default values with this command:

```
WTgeometry_setrenderingstyle(mygeom,
                              WTRENDER_LIGHTING|
                              WTRENDER_PERSPECTIVE,
                              WTRENDER_DEFAULT);
```

then `WTgeometry_getrenderingstyle(mygeom)` will return:

```
WTRENDER_SMOOTH|WTRENDER_LIGHTING|WTRENDER_TEXTURED|
WTRENDER_ANTIALIAS
```

Although you previously turned lighting off and perspective correction on for this geometry, you are now telling WTK to use the default (the universe values) for these two rendering style components. Thus, the geometry's rendering style adopts the universe style for these two components again, (i.e., lighting is on, perspective correction is off).

Geometry Modification

The following WTK functions are available for modifying geometries. For information on modifying geometries by editing vertices, see *Vertex-level Geometry Editing* on page 6-42.

WTgeometry_stretch

```
void WTgeometry_stretch(
    WTgeometry *geom,
    WTP3 factors,
    WTP3 center);
```

This function stretches a geometry in its local coordinate frame by applying a different scale factor in each of the three coordinate dimensions. The *factors* argument contains the three scale factors (for X, Y and Z) by which the geometry is to be stretched. The *center* argument is the world coordinate point about which the object is stretched.

This function can be compared to the function `WTgeometry_scale` (shown below), which scales the geometry uniformly by applying the same scale factor in each dimension. Keep in mind that stretching an object changes its shape, while scaling does not (it simply makes the geometry larger or smaller).

This example shows how to stretch a geometry by a factor of two along its X axis (about its midpoint):

```
WTgeometry *geometry;
WTP3 p, factors;
```

```
/* set factors for stretch */
factors[X] = 2.0;
factors[Y] = factors[Z] = 1.0;

/* stretch geometry along its X axis by factor of 2 about its midpoint */
WTgeometry_getmidpoint(geometry, p);
WTgeometry_stretch(geometry, factors, p);
```

WTgeometry_scale

```
void WTgeometry_scale(
    WTgeometry *geom,
    float factor,
    WTp3 center);
```

This function scales a geometry by a specified factor about a specified point in its local coordinate frame. If the scale factor equals 1.0, then this function has no effect.

In the following example, a geometry is scaled by a factor of two about its midpoint.

```
WTgeometry *geometry;
WTp3 p;

/* scale geometry by factor of 2 about its midpoint. */
WTgeometry_getmidpoint(geometry, p);
WTgeometry_scale(geometry, 2.0, p);
```

WTgeometry_translate

```
void WTgeometry_translate(
    WTgeometry *geom,
    WTp3 offset);
```

This function translates a geometry in its local coordinate frame by adding the specified offset to each of the geometry's vertices.

Note: Remember that `WTgeometry_translate` alters the vertex positions in the geometry. The geometry does not retain the original positions.

WTgeometry_transform

```
void WTgeometry_transform(  
    WTgeometry *geom,  
    WTPq *pq);
```

This function transforms a geometry in its local coordinate frame by the position and orientation specified by the *pq* argument. The offset contained in the *p* field of the *WTPq* structure is added to each of the geometry's vertices. Also, each vertex is rotated by an amount specified by the *q* field of the *WTPq* structure.

Note: Remember that WTgeometry_transform alters the vertex positions in the geometry. The geometry does not retain the original positions.

Geometry Optimization

The *WTgeometry_prebuild* function optimizes a specific geometry for rendering speed. This optimization (or prebuilding) takes place before rendering and converts the polygons in a geometry into triangle strips, which can then be rendered more efficiently. In order to make use of *WTgeometry_prebuild*, it's important to understand what this function can and cannot do.

The following must be true of two adjacent polygons in order to take advantage of this optimization function:

- The polygons must share an edge.
- If the polygons are textured, they must have the same texture, and the uv coordinates at their common vertices must be the same.
- If texturing is off and vertices have material properties, material properties at their common vertices must be the same. If vertices do not have material properties, the polygons' material IDs must match.
- Vertices must have a vertex normal. Vertices without a vertex normal will not be converted into triangle strips. If you first turn off smooth-shaded rendering using the *WTuniverse_setrendering* function, then even vertices without a vertex normal will be optimized by being converted into triangle strips. Remember however that those vertices that do not have vertex normals will be rendered as flat-shaded even if you turn smooth-shading back on.

Also note that there is an upper limit to the number of polygons that an individual geometry can have for optimization to be effective. This depends on your hardware platform, but in general, geometries composed of more than 32000 vertices will make excessive demands on memory. You are better off organizing your geometries into logical, localized units (such as pieces of furniture in a room), rather than creating massive geometries (such as an entire roomful of furniture), which are difficult to optimize.

Once you have optimized a particular geometry, you can't edit it. Specifically, you can't scale or stretch the geometry, change the colors or textures of its surfaces, or call *WTPoly_delete* or any of the functions described in *Vertex-level Geometry Editing* on page 6-42. If you want to edit an optimized object, you must undo the optimization with *WTgeometry_deleteprebuild* before you can call any of the editing functions. Once you have edited the object, you can optimize it again with *WTgeometry_prebuild*.

Note: You can, however, move and rotate the geometry.

WTgeometry_prebuild

```
FLAG WTgeometry_prebuild(  
    WTgeometry *geom);
```

This function optimizes geometry data structures so they render faster.

Once it is optimized, the geometry can be moved and rotated, but it cannot be edited. To edit it, undo the optimization with *WTgeometry_deleteprebuild*, call any of the editing functions, and then optimize it again. Refer to the detailed description on page 6-39. Also see *How Do I Measure Performance On My Machine?* on page A-38.

WTgeometry_deleteprebuild

```
FLAG WTgeometry_deleteprebuild(  
    WTgeometry *geom);
```

This function deletes the optimized data structures created when *WTgeometry_prebuild* was called. Use this function to remove the optimization, so that the geometry can be edited.

Creating Reflection Mapped Optimized Geometries

`WTgeometry_prebuildreflectmap` allows you to optimize the geometry (like `WTgeometry_prebuild`) and to also simulate highly reflective surfaces such as polished metal or mirror finished surfaces. By applying a spherically mapped image to the surface of a geometry, the UV's of which change in relation to the viewer's position, an effect very similar in appearance to true environmental reflection is achieved. Note, however, that this is not a true reflection of the 3D environment in which the geometry exists. Reflections of other objects will not appear in the reflection map. In fact, since you provide the image for the map, it can represent an environment that is completely different from the scene that the geometry exists in. Material properties are always blended with the reflection map. Note that the reflection map is a texture map. Consequently, you may not apply a texture map and a reflection map to the same geometry.

There are several ways to build an image that will provide an acceptable environment map:

- Using a 3D-rendering application such as 3D Studio Max or POVray, you can render the map image. The scene can be modeled and arranged as necessary to represent the reflected environment. A sphere that is small relative to the environment should then be placed in the center of the scene. The sphere should have a highly reflective, ray-traceable material applied to it. The viewpoint should then be set up to simulate a camera with an infinite or very great focal length centered on the sphere. A close up image, where the sphere fills the frame, will provide you with a good reflection map for use with this function.
- You can use a configuration similar to the setup above to create a scannable photograph for use as a reflection map. You need to take a photo of a large silvered sphere using a camera with a lens that has an infinite focal length. Simply take a photograph of the sphere from as far away as possible.
- Another way to create a usable photograph is to use an extremely wide-angle (or fisheye) lens to photograph the scene.

WTgeometry_prebuildreflectmap

```
FLAG WTgeometry_prebuildreflectmap(  
    WTgeometry *geom,  
    char* texmap);
```

This function optimizes geometry data structures so they render faster (similar to *WTgeometry_prebuild*) and it also applies a reflection map to the specified geometry using the image specified by *texmap* as the reflection map.

Vertex-level Geometry Editing

The functions in this section let you edit geometry at the vertex level. To access the vertices in a geometry, use *WTgeometry_getvertices* on page 6-33 and *WTvertex_next* on page 6-33. Additional information about vertex normals and colors is provided in *Modeling Considerations* on page 6-2.

You must call *WTgeometry_beginedit* before you can edit a geometry with any of the following functions:

- *WTgeometry_setvertexposition*
- *WTgeometry_setvertexrgb*
- *WTgeometry_setvertexnormal* (if your geometry is already Gouraud-shaded, calls to *WTgeometry_setvertexnormal* can be made at any time and do not need to be sandwiched between *WTgeometry_beginedit* and *WTgeometry_endedit* calls. See *WTgeometry_setvertexnormal* on page 6-46 for more information.)
- *WTgeometry_setvertexmatid* (if your geometry is already vertex-colored, then calls to *WTgeometry_setvertexmatid* can be made at any time, and do not need to be sandwiched between *WTgeometry_beginedit* and *WTgeometry_endedit* calls. See *WTgeometry_setvertexmatid* on page 6-48 for more information.)

When you have finished editing, you must call *WTgeometry_endedit*. This ensures that WTK properly updates the internal state of the geometry and all of the polygons contained in the geometry.

WTgeometry_beginedit

```
FLAG WTgeometry_beginedit(  
    WTgeometry *geom);
```

This function lets WTK know that you are going to edit a geometry. You must call this function before you can edit a geometry with any of the following functions:

- *WTgeometry_setvertexposition*
- *WTgeometry_setvertexnormal*
- *WTgeometry_setvertexmatid*
- *WTgeometry_setvertexrgb*

You must also call *WTgeometry_endedit* immediately after you have finished editing so that WTK can properly update the internal state of the geometry.

In the following example, the first vertex in a geometry is moved 100.0 units along the X axis:

```
WTgeometry *geom;
WTvertex *vertex;
WTp3 pos;

WTgeometry_beginedit(geom);
vertex= WTgeometry_getvertices(geom);
WTgeometry_getvertexposition(geom, vertex, pos);
pos[X] += 100.0; /* Move the vertex to the right in world coordinates */
WTgeometry_setvertexposition(geom, vertex, pos);
WTgeometry_endedit(geom);
```

WTgeometry_endedit

```
FLAG WTgeometry_endedit(
    WTgeometry *geom);
```

You must call this function when you have finished editing the specified geometry. Calling this function allows WTK to properly update the internal state of the geometry and all of the polygons contained in the geometry. The internal state modified by WTK can include the polygon normals, geometric extents, etc. If geometry editing has caused one or more polygons to become non-convex or non-planar, WTK automatically detects and splits such polygons because they may render in a confusing or unintended manner. It is therefore possible for a geometry to contain more polygons than originally expected.

WTgeometry_recomputestats

```
FLAG WTgeometry_recomputestats(  
    WTgeometry *geom  
    FLAG clearverts);
```

This function recomputes the specified geometry's statistics based on the locations of the geometry's vertices. The statistics computed are the geometry's extents, midpoint, radius, and bounding box. If the *clearverts* flag is TRUE, this function will also remove unused vertices (i.e., vertices that aren't referenced by any of the geometry's polygons) from the geometry. This function should be called whenever one or more polygons have been deleted from a geometry via calls to *Wtpoly_delete*.

WTgeometry_setvertexposition

```
FLAG WTgeometry_setvertexposition(  
    WTgeometry *geom,  
    WTvertex *vertex,  
    WTp3 pos);
```

This function sets a vertex position specified in world coordinates. Before calling this function, you must call *WTgeometry_beginedit* to put the geometry into geometry editing mode. Once the geometry is in edit mode, you can call this function (and the other editing functions) multiple times.

WTgeometry_getvertexposition

```
void WTgeometry_getvertexposition(  
    WTgeometry *geom,  
    WTvertex *vertex,  
    WTp3 pos);
```

This function obtains the specified vertex's position in the local coordinate system of the geometry and places it in the *pos* argument.

WTgeometry_setvertexnormal

```
FLAG WTgeometry_setvertexnormal(  
    WTgeometry *geom,  
    WTvertex *vertex,  
    WTP3 normal);
```

This function sets a vertex normal for a geometry. The vertex normal is used for Gouraud shading of polygons when all of the vertices in the polygon have normals associated with them.

Gouraud shading of polygons is enabled in three ways:

1. Either the polygon has vertex normals specified for each of its vertices at the time the polygon is first constructed (either with a WTK file reader, or with direct calls to the WTK geometry constructor functions), or
2. Calls to *WTgeometry_setvertexnormal* are sandwiched between *WTgeometry_beginedit* and *WTgeometry_endedit*, so that by the time *WTgeometry_endedit* is called, all of the vertices for the polygon have had normals set for them, or
3. You call *WTgeometry_computevertexnormal* and WTK automatically calculates normals for all vertices in the geometry.

Therefore, to enable Gouraud shading of polygons that weren't previously Gouraud shaded, you must call *WTgeometry_beginedit* to put the geometry into geometry editing mode before calling *WTgeometry_setvertexnormal*.

Once the geometry is in edit mode, you can call this function (and the other editing functions) multiple times. When you are done editing, *WTgeometry_endedit* must be called.

Performance Tip

If you are simply changing the value of a vertex normal for a vertex that already has a normal, and no change to the polygon's Gouraud-shading status is required, then you can call *WTgeometry_setvertexnormal* without calling *WTgeometry_beginedit* and *WTgeometry_endedit*. This will give you increased performance. You can determine

whether a vertex already has a normal by checking whether *WTgeometry_getvertexnormal* (see below) returns TRUE.

For additional information about Gouraud shading, see *Modeling Considerations* on page 6-2. Also see *WTgeometry_computevertexnormal* on page 6-46.

WTgeometry_getvertexnormal

```
FLAG WTgeometry_getvertexnormal(  
    WTgeometry *geom,  
    WTvertex *vertex,  
    WTp3 normal);
```

This function tests the specified vertex to see if a normal has been set for it. If a normal has been set for the specified vertex, then its value, in the geometry's local coordinate frame, is copied into the argument *normal*, and TRUE is returned. If a normal has not been set for this vertex, then the argument *normal* is zeroed and FALSE is returned.

A normal can be set for a vertex either by using the function call *WTgeometry_setvertexnormal*, or *WTgeometry_computevertexnormal*, or through one of the file formats supported by WTK which support vertex normals (such as NFF, 3D Studio, Wavefront, and MultiGen/ModelGen). The NFF format also supports automatic normal generation, as described in *Automatic Normal Generation* on page F-8.

WTgeometry_computevertexnormal

```
FLAG WTgeometry_computevertexnormal(  
    WTgeometry *geom,  
    WTvertex *v);
```

This function automatically computes the normal of the vertex “v” in geometry “geom.” The vertex's normal is computed as the average of the surrounding polygon normals. This function returns TRUE if the vertex normal could be computed, and FALSE if it could not.

Note: You must call *WTgeometry_beginedit* prior to using this function and *WTgeometry_endedit* afterwards.

The following example computes all of the vertex normals in the geometry:

```
WTvertex *norm_vertex;
WTgeometry_beginedit(geom);
norm_vertex = WTgeometry_getvertices(geom);
while (norm_vertex)
{
    WTgeometry_computevertexnormal(
        geom, norm_vertex);
    norm_vertex = WTvertex_next(norm_vertex);
}
WTgeometry_endedit(geom);
```

WTgeometry_setvertexrgb

```
FLAG WTgeometry_setvertexrgb(
    WTgeometry *geom,
    WTvertex *vertex,
    unsigned character red,
    unsigned character green,
    unsigned character blue);
```

This function sets the vertex's color to the specified red, green, and blue color components. Note that even though WTK allows the user to specify color using red, green, and blue color components, WTK's internal representation makes use of the geometry's material table by either finding a material table entry whose color (red, green, and blue) matches the color specified by the user, or by creating a new entry in the material table.

You must call *WTgeometry_beginedit* before you call *WTgeometry_setvertexrgb*, and you must call *WTgeometry_endedit* after calling *WTgeometry_setvertexrgb*.

WTgeometry_getvertexrgb

```
FLAG WTgeometry_getvertexrgb(
    WTgeometry *geom,
    WTvertex *vertex,
    unsigned character *red,
    unsigned character *green,
    unsigned character *blue);
```

This function retrieves the specified vertex's color. The individual red, green, and blue components of the 24-bit color are stored in *red*, *green*, and *blue* respectively.

WTgeometry_setvertexmatid

```
FLAG WTgeometry_setvertexmatid(  
    WTgeometry *geom,  
    WTvertex *vertex  
    int id);
```

This function changes the material table index of the specified vertex of the geometry. The indices remain static if the material table is changed or deleted. Note that a modulus operation occurs at render time if *id* is greater than the number of materials in the material table referenced by this geometry.

A negative material index is not allowed, and therefore this function returns `FALSE` if you pass in a negative value for *id*.

In order for vertex colors to take effect, the first time a given polygon is to be vertex colored, call *WTgeometry_beginedit*, then change the vertex material ids, and then call *WTgeometry_endedit*. At the time of the *endedit* call, those polygons whose member vertices all have color/id information specified are rendered with the vertex colors the next time they are rendered. If this has happened once during the simulation, or if the geometry file was loaded with vertex color/id specified, the polygon is rendered using the vertex colors for the remainder of the running time of the application. Subsequent changes of vertex color/id do not require you to make another set of calls to *WTgeometry_beginedit* and *WTgeometry_endedit*. These calls were only necessary in order to trigger the change in the rendering style (to vertex-coloring) of the polygon.

WTgeometry_getvertexmatid

```
int WTgeometry_getvertexmatid(  
    WTgeometry *geom,  
    WTvertex *vertex);
```

This function returns the material table index of the specified vertex of the geometry.

It returns -1 if the *WTgeometry_setvertexmatid* function was never called, or if the vertex does not have a color associated with it.

Polygons

Introduction

Polygons provide the three-dimensional shapes of the objects in your scene. A polygon is a planar surface defined by a set of three or more vertices. For example, a triangle is a polygon with 3 vertices, and a rectangle is a polygon with 4 vertices. Geometries in WTK are made up of polygons (polygonal surfaces) that you can color, shade, and texture.

You can create polygons in several different ways. WTK automatically creates polygons when you construct geometries with functions such as *WTgeometry_newcone* and *WTgeometry_newtext3d*. You can also construct polygons vertex by vertex, with functions described in this chapter.

WTK provides polygon functions that let you do the following:

- Set, get, and change polygon attributes, such as color, normals, and the material table index. (Polygon texturing is described in Chapter 10, *Textures*.)
- Define and assign polygon ID numbers.
- Access geometrical properties and vertices of a polygon.
- Iterate through a list of polygons.
- Dynamically construct geometries from vertices and polygons.
- Delete polygons.
- Test for intersections of polygons with other polygons.

Most of the functions in this chapter take a pointer to a WTK polygon structure. You can obtain polygon pointers in a variety of ways. You can get them interactively, using *WTwindow_pickpoly*; through polygon ID values using *WTuniverse_id2poly* or *WTgeometry_id2poly*; using the polygon access functions *WTgeometry_getpolys*, and *WTpoly_next*, or with the dynamic constructor function *WTgeometry_beginpoly*.

Polygon Attributes

Wtpoly_setrgb

```
void Wtpoly_setrgb(  
    Wtpoly *poly,  
    unsigned char r,  
    unsigned char g,  
    unsigned char b);
```

This function specifies the 24-bit color value of a polygon. The arguments *r*, *g*, and *b* are the red, green, and blue color components, each with a valid range of 0 to 255. The default color of a polygon is white (255, 255, 255).

In the following example, a polygon's color is set to bright purple:

```
Wtpoly *poly;  
Wtpoly_setrgb (poly, 255, 0, 255);
```

Another example of usage is provided under *Wtpoly_getrgb*.

Wtpoly_getrgb

```
void Wtpoly_getrgb(  
    Wtpoly *poly,  
    unsigned char *r,  
    unsigned char *g,  
    unsigned char *b);
```

This function retrieves the values of the *r*, *g*, and *b* (red, green, and blue) components of the polygon's color.

In the following example, the blue component of a polygon's color is increased:

```
WTpoly *poly;
unsigned char r, g, b;

/* get current polygon color components */
WTpoly_getrgb (poly, &r, &g, &b);
/* increase blue component by 1 if it is not already maximum value */
if (b<255 ) {
    b++;
    WTpoly_setrgb (poly, r, g, b);
}
```

WTpoly_setmatid

```
FLAG WTpoly_setmatid(
    WTpoly *poly,
    int id);
```

This function changes the polygon's material table index. You use this index to access the material properties contained in the material table associated with the geometry which contains the specified polygon. The polygon's material index remains static even if you change or delete the material table associated with the geometry containing the polygon. Note that a modulus operation will occur at render time if *id* is greater than the number of materials in the material table referenced by this polygon's geometry.

A negative material index is not allowed, therefore this function will return `FALSE` if the user passes in a negative value for *id*.

WTpoly_getmatid

```
int WTpoly_getmatid(
    WTpoly *poly);
```

This function retrieves the specified polygon's index into the material table.

WTPoly_setbothsides

```
void WTPoly_setbothsides(  
    WTPoly *poly,  
    FLAG flag);
```

This function specifies whether both sides of a polygon are visible.

If the *flag* argument is TRUE, then both sides of the polygon are visible. If the *flag* argument is FALSE, then polygon back faces are not drawn. By default, polygon back faces are rejected, i.e., they are not drawn.

In WTK, a polygon face whose vertices appear in counter-clockwise order on the screen is considered to be the front face, while a polygon face whose vertices appear in clockwise order is the back face. The front face is also the side of the polygon from which the polygon normal points. When the viewpoint is on the other side of the polygon (that is, its back face, and the flag argument in *WTPoly_setbothsides* is FALSE), then the polygon is not drawn, and hence invisible to the user.

You can also set polygon back face rejection in NFF files using the keyword “both” (see *Appendix F*). For more information about back face rejection, see *Back Face Rejection* on page 6-10 of the *Geometries* chapter.

WTPoly_getbothsides

```
FLAG WTPoly_getbothsides(  
    WTPoly *poly);
```

This function returns the polygon’s back face rejection status. If TRUE is returned, then the polygon can be viewed from both sides. If FALSE is returned, then only the front face of the polygon is visible and its back face is not drawn.

WTPoly_getnormal

```
void WTPoly_getnormal(  
    WTPoly *poly,  
    WTP3 normal);
```

This function retrieves the normal vector of the specified polygon and places it in the *normal* argument. The polygon normal is a unit vector (a vector with length equal to 1.0) perpendicular to the plane of the polygon pointing from the polygon's front face. Note that WTK computes the polygon normal; you cannot directly set it. The polygon normal together with the polygon's center of gravity define the plane of the polygon. An example of how this is used is provided under *WTPoly_getcg*. See also *WTnormal_2slope* on page 25-33.

WTPoly_getcg

```
void WTPoly_getcg(  
    WTPoly *poly,  
    WTP3 cg);
```

This function retrieves the center of gravity for the specified polygon and places it in the *cg* argument. The center of gravity of a polygon is defined as the average position of the polygon's vertices. (See the example on page 7-9 following *WTPoly_numvertices*.)

The following example shows the use of the functions *WTPoly_getnormal* and *WTPoly_getcg* to determine the distance of a 3D point from the plane of a polygon, and to determine which side of the polygon the point is on. The example uses the fact that the polygon normal points from its front face.

```
WTP3 pos;                /* position set elsewhere in application */  
WTP3 cg, normal, vector;  
WTPoly *poly;  
float distance;  
  
/* compute the vector from the polygon to pos. */  
WTPoly_getcg (poly, cg);  
WTP3_subtract (pos, cg, vector);  
  
/* If the vector from the polygon to pos points in the same direction  
as the polygon's normal, pos is on the side of the polygon's front face.  
The value of the dot product is the distance from pos to the plane  
of the polygon. */  
WTPoly_getnormal (poly, normal);  
distance = WTP3_dot(vector, normal);  
WTmessage("pos is distance %f from plane of polygon ", ABS(distance));
```

```
if (distance > 0.0 )
    WTmessage("on front-facing side\n");
else
    WTmessage("on back-facing side\n");
```

Polygon ID's

You can assign individual identifying numbers (IDs) to polygons. Polygon IDs are read from and written to NFF files, and are set with the function *Wtpoly_setid*. Polygon IDs provide a handy way of obtaining pointers to polygons, which can then be passed in to other functions.

You can use the function *WTgeometry_id2poly* to obtain a pointer to a polygon with a specified ID value. This function returns the first polygon in the geometry with that ID. To obtain a pointer to a specific polygon, the polygon must have an ID that is unique for the geometry to which it belongs.

For example, consider an application that requires access to the normal vector of a polygon in a rotating object. Suppose further that it isn't convenient within this application to interactively select the polygon, and thus a pointer to the polygon must be accessed entirely under application control. Here's how to do this:

1. Denote the polygon with a unique ID in the NFF file.
2. Call *WTgeometry_id2poly* to obtain a pointer to the polygon.
3. Call *Wtpoly_getnormal* for the polygon, as required by the application.

Wtpoly_setid

```
void Wtpoly_setid(
    Wtpoly *poly,
    short id);
```

This function sets the value of a polygon's *id*. The default polygon *id* value is 0 (zero). This function is useful if you have a pointer to a polygon that you would like to reference again by *id* rather than pointer. For example, your application might assign certain meaning to *id* values or group polygons by *id*, which is not as readily done using pointer values. Setting a

polygon *id* is also useful for identifying a polygon in an NFF file, if the geometry to which the polygon belongs is written out.

For a very large database, you may want to distinguish between a larger number of polygons than is possible with a single short *id* value. To get around this issue, you can create multiple geometries, using distinct polygon IDs within individual geometries, and then differentiate the polygons in the database using the unique combination of geometry name and polygon ID (see *WTgeometry_setname* on page 6-29). Additionally, to optimize performance when you have a large database, it is advantageous to have multiple geometries in the simulation rather than a single monolithic geometry.

Wtpoly_getid

```
short Wtpoly_getid (  
    Wtpoly *poly);
```

This function returns the value of the specified polygon's ID. By default, a polygon's ID value is 0. Polygon IDs are set either with *Wtpoly_setid* or in an NFF file.

Geometry that Contains a Polygon

Wtpoly_getgeometry

```
WTgeometry *Wtpoly_getgeometry(  
    Wtpoly *poly)
```

This function returns the geometry that contains the specified polygon.

Polygon Access

WTpoly_next

```
WTpoly *WTpoly_next(  
    WTpoly *poly);
```

This function returns the next polygon in the linked list of polygons associated with geometries. These lists are returned by *WTgeometry_getpolys*. If the *poly* argument is NULL, or if *poly* is the last polygon in the list, NULL is returned.

The following is an example of using *WTpoly_next*:

```
/* print out the number of vertices in each polygon of a geometry*/  
WTpoly *poly;  
for (poly = WTgeometry_getpolys(geometry); poly;  
     poly = WTpoly_next(poly)) {  
    WTmessage("poly %p has %d vertices\n", poly, WTpoly_numvertices(poly));  
}
```

Vertex Access

WTpoly_getvertex

```
WTvertex *WTpoly_getvertex(  
    WTpoly *poly,  
    short index);
```

This function returns the specified vertex pointer (indicated by the *index* argument) for the specified polygon. For instance, pass in *index=0* to obtain the polygon's first vertex pointer, *index=1* to retrieve the second, and so on. Use this function in conjunction with *WTpoly_numvertices* to find all vertices referenced by a polygon. An example is given below under *WTpoly_numvertices*.

A NULL pointer is returned if this function fails (for example, if *index* is greater than the actual number of vertices in the polygon minus one).

WTPoly_numvertices

```
short WTPoly_numvertices(  
    WTPoly *poly);
```

This function returns the number of vertices in a polygon. You can use this function with *WTPoly_getvertex* to access a polygon's vertex list. For example, if *WTPoly_numvertices* returns 3, then 0, 1, or 2 can be passed to *WTPoly_getvertex* to access the polygon's vertices.

The following example shows the use of the functions *WTPoly_numvertices* and *WTPoly_getvertex* to compute the center of gravity of a polygon. Note that this example is just for illustration, since the polygon center of gravity can also be obtained by simply calling *WTPoly_getcg*.

```
WTgeometry *geom;  
WTPoly *poly;  
WTvertex *v;  
WTP3 cg, p;  
short i, n;  
float inverse;  
  
/* initialize center-of-gravity */  
WTP3_init(cg);  
  
/* go through polygon vertices, accumulating vertex positions in cg */  
n = WTPoly_numvertices(poly);  
for ( i=0 ; i<n ; i++ ) {  
    v = WTPoly_getvertex(poly, i);  
    WTgeometry_getvertexposition(geom, v, p);  
    WTP3_add(cg, p, cg);  
}  
  
/* finish calculation of average vertex position */  
inverse = 1.0/n;  
WTP3_mults(cg, inverse);  
  
/* print out result */  
WTP3_print(cg, "Center of gravity: ");
```

Dynamic Polygon Creation

WTK enables you to construct geometries dynamically from vertices and polygons that are defined with WTK function calls. This section describes the functions used to define polygons and create new graphical objects.

WTgeometry_beginpoly

See *WTgeometry_beginpoly* on page 6-23 for a description.

WTgeometry_newvertex

See *WTgeometry_newvertex* on page 6-22 for a description.

Wtpoly_addvertex

```
FLAG Wtpoly_addvertex(  
    Wtpoly *poly,  
    int vindex);
```

This function adds a vertex to a polygon under construction by referencing a vertex in the geometry to which the polygon belongs. The return value indicates success or failure. The *vindex* argument is the index of the vertex in the geometry's list of vertices, which is determined by the order in which the vertices were added to the geometry. For example, *vindex=3* refers to the fourth vertex added to the geometry. *Wtpoly_addvertex* can only be called after *WTgeometry_beginpoly* has been called for this polygon and before *Wtpoly_close* has been called for this polygon.

Wtpoly_addvertexptr

```
FLAG Wtpoly_addvertexptr(  
    Wtpoly *p,  
    WTvertex *v);
```

This function enables you to add vertices to a polygon under construction by passing in a pointer to a vertex. In some situations, this is more convenient than specifying a vertex *index*, as is required by the function *Wtpoly_addvertex*. *Wtpoly_addvertexptr* can only be

called after *WTgeometry_beginpoly* has been called for this polygon and before *Wtpoly_close* has been called for this polygon. It returns FALSE if either the polygon or vertex pointer is NULL, otherwise it returns TRUE.

Please note that, when using *Wtpoly_addvertexptr*, it is your responsibility to ensure that the vertex belongs to the same geometry to which the polygon belongs. See the functions *WTgeometry_newvertex* on page 6-22 and *Wtpoly_getgeometry* on page 7-7.

The following code sample illustrates the use of this function:

```
/*
 * This function adds a triangular polygon to a geometry that is
 * under construction, i.e., a geometry for which WTgeometry_close has
 * not yet been called.
 */
void makepoly(WTgeometry *geom)
{
    Wtpoly *poly;
    Wtp3 pos;

    /* Get pointer to a new polygon structure */
    poly = WTgeometry_beginpoly(geom);

    /* Add vertices to geometry and polygon */
    pos[X] = 0.0; pos[Y] = 1.0; pos[Z] = 0.0;
    addnewvertex(geom, poly, pos);
    pos[X] = 1.0; pos[Y] = 0.5; pos[Z] = 0.0;
    addnewvertex(geom, poly, pos);
    pos[X] = 0.0; pos[Y] = 0.0; pos[Z] = 0.0;
    addnewvertex(geom, poly, pos);
    Wtpoly_close(poly);
}

void addnewvertex(WTgeometry *geom, Wtpoly *poly, Wtp3 pos)
{
    WTvertex *vertex;
    vertex = WTgeometry_newvertex(geom, pos);
    Wtpoly_addvertexptr(poly, vertex);
}
```

WTpoly_close

```
FLAG WTpoly_close(  
    WTpoly *poly);
```

This function finishes the definition of a polygon. Once you have called this function, you cannot add any more vertices to the polygon.

Deleting Polygons

WTpoly_delete

```
FLAG WTpoly_delete(  
    WTpoly *poly);
```

This function deletes the specified polygon. If successful, TRUE is returned.

You can delete polygons from existing geometries and also while the polygon is being dynamically created (see *Dynamic Polygon Creation* on page 7-10).

If the polygon is in a geometry that has been optimized, then the polygon cannot be deleted and FALSE is returned. See *Geometry Optimization* on page 6-39.

WTpoly_delete does not delete the polygon's vertices from the geometry when it deletes the polygon. The polygon's vertices are left in the geometry. Unused vertices can be removed from the object by calling *WTgeometry_recomputestats*, although this is not necessary.

Also, *WTpoly_delete* does not recompute the overall geometrical parameters of the geometry (such as, position, bounding box). The function *WTgeometry_recomputestats* can be used to recompute these parameters. You may want to perform several geometry editing functions before recomputing the geometry's geometrical properties, or you may want to recompute them each time.

Also see the function *WTgeometry_beginedit* on page 6-42 which is required for certain geometry editing functions. You do not have to call *WTgeometry_beginedit* in order to call *WTpoly_delete*.

While constructing a polygon, you can call the *Wtpoly_delete* function to free the memory used by the polygon before it has been completely defined. Specifically, you can call the *Wtpoly_delete* function for a polygon created with the *WTgeometry_beginpoly* function, as long as you have not called the *Wtpoly_close* function yet for that polygon. The ability to delete a partially-defined polygon is useful, for example, if you are writing a custom 3D file reader and encounter an error in parsing the file.

Polygon Intersection Testing

Keep in mind that polygon-level intersection testing, while more precise than bounding-box intersection tests, is also more computationally intensive. You should use these functions only as often as needed and only when bounding box tests are insufficient. See *Intersection Testing* on page 4-85 for more details.

Wtpoly_rayintersect

See *Wtpoly_rayintersect* on page 4-88 of the *Scene Graphs* chapter.

Introduction

A material is a combination of light and color attributes that you use to define the appearance of a geometry or collection of geometries. WTK functions let you create, edit, and save material information.

This chapter includes information on the properties of materials, color determination using material tables, material table functions, and some advanced topics on materials.

Material Properties

Geometries either emit light, reflect light, or both. This light is manifested as color. When designing a geometry (such as, a car), there are two kinds of color to consider:

- The colors used in the car itself.
- The colors of the light playing on the car

A realistic image of a geometry includes many colors, and potentially many ways, of reflecting light. You use a separate *material* to specify each of these differences in appearance. The *Lights* chapter (starting on page 12-1) describes the kinds of lights available for WTK simulations. This chapter describes how you can design different materials to reflect that light differently.

Each material has the following properties:

<i>Ambient</i>	The color reflected from the material in ambient white light without regard to light direction. Specified in red, green, and blue floats in a range from 0.0 – 1.0.
<i>Diffuse</i>	The color reflected from the material in diffuse white light, as a function of light direction. Specified in red, green, and blue floats in a range from 0.0 – 1.0.
<i>Specular</i>	The color reflected from the material in specular white light. The specular material property is what makes a geometry appear to be “shiny” with highlights appearing on its surface. Usually, the specular highlight is white, which means that it reflects the color of the specular light (which is also usually white). Specified in red, green, and blue floats in a range from 0.0 – 1.0.
<i>Shininess</i>	The narrowness of focus of the specular highlights. This has no meaning if the specular color is black (lighting of geometry rendered with material properties is an “additive” process; a black specular highlight will not darken the geometry; it simply won't contribute to a light highlight on the geometry). Its mathematical meaning is “specular exponent” in the lighting equations. The lower the shininess value, the more “spread out” the highlight; the higher the shininess value, the sharper the highlight. A high value for shininess makes an object look shiny. Specified as a floating point value in a range from 0.0 – 128.0.
<i>Emissive</i>	The color of light <i>produced</i> (not reflected) by the material even when there is no light. A geometry with this property can be seen even when there are no lights in the scene, however, the emissive light does not illuminate other geometry in the area. This material property is used less often than the others. Specified in red, green, and blue floats in a range from 0.0 – 1.0.
<i>Opacity</i>	The extent to which the color value of a pixel is combined with the color value behind it. Specified as a floating point value in a range from 0.0 – 1.0, where 0 is completely invisible and 1 is completely opaque.

There's also an *ambient-diffuse* property (a combination of the ambient and diffuse properties) that defines the color of the geometry in ambient *or* diffuse white light, specified in red, green, and blue floats in a range from 0.0 – 1.0. This is equivalent to setting the RGB color in version 2.1 (and previous versions) of WTK. You can continue to specify a geometry's color with this property, while using the new properties of WTK to improve the realism of individual geometries as needed.

You can obtain materials by reading in files from a modeler which specifies material properties in its export file format. WTK supports Wavefront's .obj, 3D Studio's .3ds, and VRML's .wrl file formats, which all have material information in them. When WTK reads in the file, it automatically renders the geometry using the modeler-specified material properties. For example, it will look shiny in WTK if it looked that way in the modeler.

Note that some modelers (including 3D Studio) have advanced material properties that WTK does not support; for example, properties specifying refractive properties of a transparent substance are not supported by WTK.

Note: Texture is not a property of WTK materials; it is a bitmap which may actually obscure some elements of a material, as described in the Textures chapter (starting on page 10-1).

Calculations Made to Determine Color

Figure 8-1 illustrates how WTK resolves your instructions into RGB values at each vertex in a geometry. After Gouraud shading (if enabled), the final value for each pixel is sent to the Z-buffer, if one is being used.

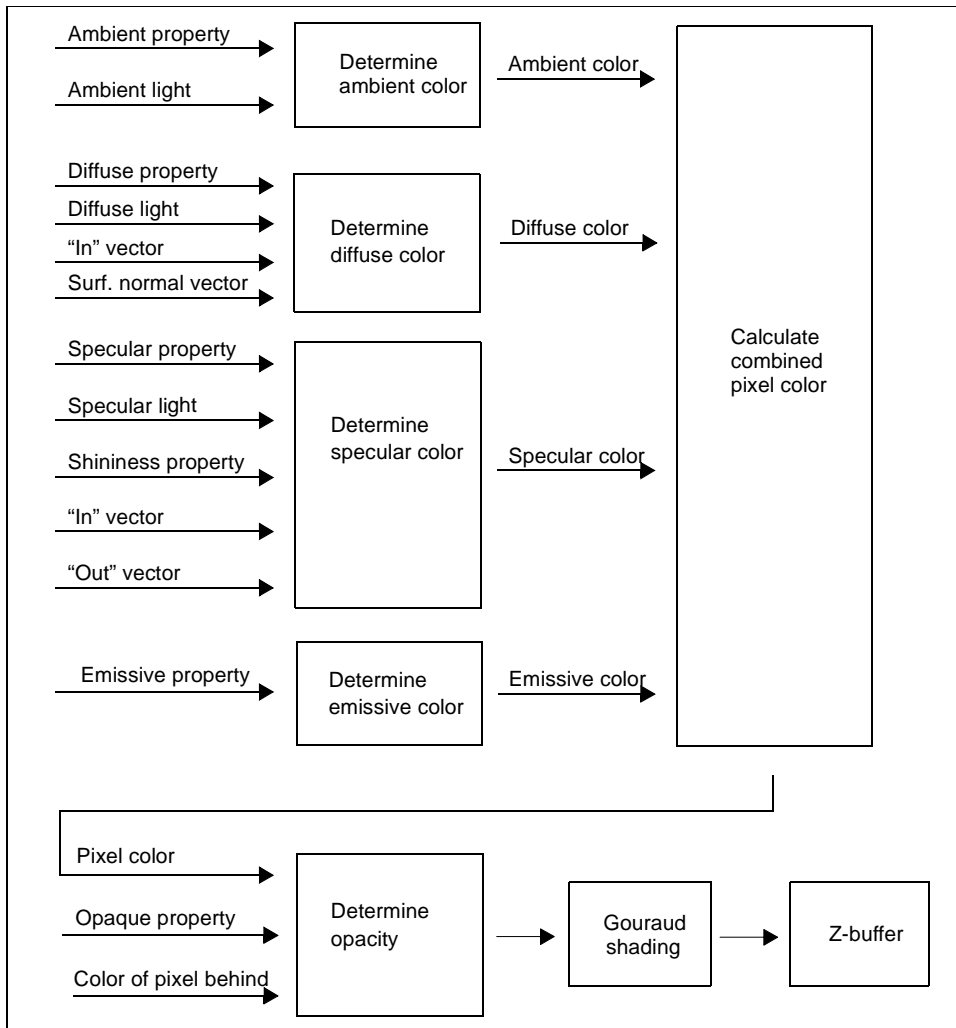


Figure 8-1: How WTK determines the color of each pixel it displays

About “In” and “Out” Vectors

The “in” vector mentioned in figure 8-1 is the vector between the light source and the vertex being lit. The “out” vector is the vector between the vertex being lit and the viewpoint. Pixels between vertices have their color and opacity values determined by Gouraud shading, if enabled.

Using Material Tables

The values for all of the materials used with a geometry are contained in its *material table*. A material table is a collection of “robust” colors. These colors are termed robust because they include more reflectance information than the “ambientdiffuse” color reflectance available in previous versions of WTK.

Material tables are indexed from 0 (zero) to the number of materials in the table. Each polygon or vertex contains an index into the material table. This means that each polygon or vertex has a number — not a color — attached to it. This number references an entry in the material table.

More than one geometry may point to the same material table, and a geometry may point to different tables depending on the effect you need. See *WTgeometry_setmtable* on page 8-18.

Once a geometry file has been loaded into a scene, you can use the functions described later in this chapter to modify the settings in this table. For example, you can use *WTmtable_setvalue* to change an existing material table entry. Since the same material may be applied to several polygons, more than one polygon in your scene could be affected when you modify a material.

To create a new material and then modify the copy, use *WTmtable_copyentry* and then *WTmtable_setvalue*.

Here's a sample WTK material file describing a material table:

```
mat
version 3.00
valid ambient diffuse specular shininess

matdef // id 0
ambient 0.345098 0.325490 0.254902
diffuse 0.376471 0.345098 0.227451
specular 0.597882 0.538353 0.225176
shininess 58.879997

matdef // id 1
ambient 0.650980 0.000000 0.000000
diffuse 0.650980 0.000000 0.000000
specular 0.890000 0.890000 0.890000
shininess 89.599998

matdef // id 2
ambient 0.200000 0.200000 0.200000
diffuse 0.600000 0.600000 0.600000
specular 0.000000 0.000000 0.000000
shininess 10.000000
```

An object does not need to have all of its material properties specified. For example, in the material file listing shown above, neither the emissive nor the opacity material properties are specified. Using fewer properties can generate a moderate improvement in performance. Properties not specified in a geometry's material table behave as though set to 0 (zero).

To alter the number of fields that are defined in a given material table, use the function *WTmtable_setproperties*, as described on page 8-9. To find out what properties are currently defined for a given material table, use *WTmtable_getproperties* as described on page 8-11.

See *How Do I Use Material Tables for Colors?* on page A-11.

Material Table Functions

WTmtable_new

```
WTmtable *WTmtable_new(  
    int definedprops,  
    int estimatedentries  
    char *name);
```

This function creates a new material table, which must have a unique name. The argument *definedprops* is a bitwise combination of the material property constants corresponding to the material properties you want to define for this material table. These are:

```
WTMAT_AMBIENT  
WTMAT_AMBIENTDIFFUSE  
WTMAT_DIFFUSE  
WTMAT_EMISSION  
WTMAT_OPACITY  
WTMAT_SHININESS  
WTMAT_SPECULAR
```

It is not permitted for *WTMAT_AMBIENTDIFFUSE* to be a defined entry of the material table at the same time as *WTMAT_AMBIENT* or *WTMAT_DIFFUSE*. If you attempt to do this, only the *WTMAT_AMBIENTDIFFUSE* field will be defined, but the *WTMAT_AMBIENT* and *WTMAT_DIFFUSE* fields will not be defined. Attempts to set the values of undefined fields of a *WTmtable* will have no effect. (See *WTmtable_setvalue* on page 8-15.)

Providing an accurate value for *estimatedentries* can be valuable for making sure material creation is quick and memory-efficient; in most cases, however, you will incur no penalty for passing in zero.

Material tables must have unique names. If you assign a name which has already been used for a material table, table creation fails and NULL is returned. However, you may pass NULL in as the *name* argument; in this case a unique material table whose name begins with *mt* and ends with a number will be created.

WTmtable_delete

```
FLAG WTmtable_delete(  
    WTmtable *mtable)
```

This function deletes the specified material table. All geometry which refers to this material table will be updated to reflect its NULL status, and will appear black. If any geometry that refers to this table has been optimized using *WTgeometry_prebuild*, these geometries will not be updated, table deletion will fail, and the function will return FALSE. Otherwise TRUE is returned.

WTmtable_merge

```
WTmtable *WTmtable_merge(  
    WTmtable *table1,  
    WTmtable *table2)
```

This function merges two material tables and returns a new material table that contains the materials from both tables. It's the equivalent of copying *table1* into a new table and then appending *table2* onto the end of it.

The materials in *table1* are indexed in the new material table just as they were in the original material table. The materials in *table2* have the number of entries in tables added to each of their indices. No action is taken to invalidate the original *table1* and *table2*; it is up to you to delete them. When the result is returned from this function, no geometry will refer to the new material table.

WTmtable_getnumentries

```
int WTmtable_getnumentries(  
    WTmtable *mtable)
```

This function returns the number of table entries in the material table specified by the *mtable* argument.

WTmtable_setproperties

```
WTmtable *WTmtable_setproperties(
    WTmtable *mtable,
    int definedprops)
```

This function adds or removes properties defined in a material table. The *definedprops* argument is a bitwise combination of the material property constants corresponding to the material properties you want to define for this material table. These are:

```
WTMAT_AMBIENT
WTMAT_AMBIENTDIFFUSE
WTMAT_DIFFUSE
WTMAT_EMISSION
WTMAT_OPACITY
WTMAT_SHININESS
WTMAT_SPECULAR
```

Note: When you use this function, the old table is invalidated and a new table is created. Any attempt to execute operations on the invalidated (freed) table can produce undefined results, including termination of your application

WTK returns a pointer to this new table you have created; geometries which used the old table are updated to refer to the new one. Table 8-1, describes how material properties are affected by the creation of a new material table.

Table 8-1: Results of creating a new material table

Condition	Result
A property in the old material table is also defined in the table that replaces it.	The value for the property is copied from the old table to the new one.
A property not defined in the old table is defined in the table that replaces it.	A value of opaque black is set for materials in the new table that use that property. The settings used to create this value are listed on page 8-14.

Table 8-1: Results of creating a new material table (continued)

Condition	Result
The old table had <i>WTMAT_AMBIENTDIFFUSE</i> defined and the new table has <i>WTMAT_AMBIENT</i> and/or <i>WTMAT_DIFFUSE</i> present.	The values for this property (for each material) are copied from the <i>WTMAT_AMBIENTDIFFUSE</i> field in the old table to the <i>WTMAT_AMBIENT</i> and/or <i>WTMATL_DIFFUSE</i> fields in the new table.
A property defined in the old table is not defined in the table that replaces it.	The property no longer has any effect on the geometry.

If a material property for a geometry is not defined in its material table, the geometry uses the default value for that property.

Example: Adding Shininess to a Multi-colored Geometry

Imagine that you load an old NFF file of a multi-colored geometry that was created with WTK V2.1 and you want to make it look shiny. When you load the file using *WTgeometrynode_load*, a material table is created for the geometry whose only defined field is *ambientdiffuse*. It will initially look just like it did under WTK V2.1.

Suppose you want to make this object uniformly shiny, and that you are starting with a material table with only *WTMAT_AMBIENTDIFFUSE* property defined.

You would use the following function call:

```
WTmtable_setproperties(table,  
    WTMAT_AMBIENTDIFFUSE | WTMAT_SPECULAR | WTMAT_SHININESS);
```

This permits each material to specify its own specular color and shininess. You then step through the material table, using *WTmtable_setvalue* to set its specular color and shininess. Note that the second argument to this function is a bitwise “OR” of the properties that you want defined for this material table.

For example:

```
WTgeometry *geom;
WTnode *root, *node;
WTmtable *tableold, *tablenew;

node = WTgeometrynode_load(root, "car.nff", 1.0);
geom = WTnode_getgeometry(node);
tableold = WTgeometry_getmtable(geom);
tablenew = WTmtable_setproperties(tableold,
    WTMAT_AMBIENTDIFFUSE | WTMAT_SPECULAR | WTMAT_SHININESS);
```

WTmtable_getproperties

```
int WTmtable_getproperties(
    WTmtable *mtable)
```

This function returns a bitwise combination of defined properties for the material table specified by the *mtable* argument. These are:

```
WTMAT_AMBIENT
WTMAT_AMBIENTDIFFUSE
WTMAT_DIFFUSE
WTMAT_EMISSION
WTMAT_OPACITY
WTMAT_SHININESS
WTMAT_SPECULAR
```

WTmtable_load

```
WTmtable *WTmtable_load(
    char *filename)
```

This function reads a material table from the specified filename. If a file with the specified filename is not found in the current directory, the extension *.mat* is added to the filename and the current directory is searched again. If the specified file is not found, NULL is returned.

Note: If a material table is not referenced by any geometry in the scene graph, it is automatically deleted. For example, suppose you load a material table using `WTmtable_load` and associate it with a geometry using `WTgeometry_setmtable` (see page 6-30). Now, if you delete this geometry, the material table also gets deleted (if it is not referenced by any other geometry). Hence, you could not associate another geometry to this material table later.

WTmtable_save

```
FLAG WTmtable_save(  
    WTmtable *mtable)
```

This function writes a material table to a file in the current directory. The name of the file will be the material table name with a `.mat` extension added. *Any existing file with the same name will be overwritten.*

If you have not set the name using `WTmtable_new` or `WTmtable_setname`, the name will be an automatically generated name; automatically generated material table names start at `mt1` (“m-t-one”) and increment numerically for each new material table created.

If you plan to save out material tables, it’s a good idea to give them unique names. Otherwise, material tables with automatically generated names written in the current session could overwrite similarly generated files from an earlier session. Since model files refer to the material tables by name, this means that when you load a model written from the earlier session, the old model would end up using the new material table and thus end up with unexpected colors.

WTmtable_setname

```
FLAG WTmtable_setname(  
    WTmtable *mtable,  
    char *name)
```

This function sets the name of the material table specified by `mtable` to `name`. Material tables must have unique names. If you assign a name which has already been used for a material table, this function returns `FALSE`.

WTmtable_getname

```
char *WTmtable_getname(  
    WTmtable *mtable)
```

This function returns the name of the material table specified by the *mtable* argument.

WTmtable_getbyname

```
WTmtable *WTmtable_getbyname(  
    char *name)
```

This function returns a pointer to the material table having the name specified by the *name* argument. It returns NULL if the material table with the given name does not exist.

WTmtable_setdata

```
void WTmtable_setdata(  
    WTmtable *mtable  
    void *data)
```

This function sets a user-defined data field in a material table. Private application data can be stored in any structure. To store a pointer to the structure within a material table, pass in a pointer to the structure, cast to a *void**, as the *data* argument.

WTmtable_getdata

```
void *WTmtable_getdata(  
    WTmtable *mtable)
```

This function retrieves user-defined data stored within a material table. You should cast the value returned by this function to the same type used to store the data with the *WTmtable_setdata* function.

Material Table Entry Functions

WTmtable_newentry

```
int WTmtable_newentry(  
    WTmtable *mtable);
```

This function creates a new entry in the material table given by the *table* argument. The new material-table entry will have all defined fields set to these default values:

Ambient	0.0, 0.0, 0.0
Diffuse	0.0, 0.0, 0.0
Emissive	0.0, 0.0, 0.0
Specular	0.0, 0.0, 0.0
Shininess	0.0
Opacity	1.0
Name	NULL

The value returned is the index into the material table which corresponds to the new material.

WTmtable_copyentry

```
int WTmtable_copyentry (  
    WTmtable *from,  
    int matid,  
    WTmtable *to)
```

This function copies an entry whose index is specified by the *matid* argument from one material table to another. This results in the creation of a new material table entry. If *from* and *to* are the same table, the material is duplicated so that there's a second copy of the material in the same table. Fields defined for the destination which weren't defined in the source table are filled in with the default values listed above for *WTmtable_newentry*. The value returned is the index into the destination material table that corresponds to the new copy of the material.

WTmtable_setvalue

```
FLAG WTmtable_setvalue(  
    WTmtable *mtable,  
    int matid,  
    float *value,  
    int propertybit);
```

This function alters the characteristics of an entry whose index is specified by the *matid* argument in the material table specified by the *mtable* argument. The *propertybit* argument is one of the following:

```
WTMAT_AMBIENT  
WTMAT_AMBIENTDIFFUSE  
WTMAT_DIFFUSE  
WTMAT_EMISSION  
WTMAT_OPACITY  
WTMAT_SHININESS  
WTMAT_SPECULAR
```

The *value* argument is an *array of three floats* when setting the ambientdiffuse, ambient, diffuse, specular, or emission properties, or an *array of one float* when setting the shininess or opacity properties. Passing an array of three floats when setting shininess or opacity is permitted, but only the *value* [0] argument is read from.

FALSE is returned if the specified *propertybit* is not defined for the given material table. Note that a single call to this command can cause changes in multiple polygons in multiple geometries, because more than one polygon may refer to the same material table entry.

WTmtable_getvalue

```
FLAG WTmtable_getvalue(  
    WTmtable *mtable,  
    int matid,  
    float *value,  
    int propertybit);
```

This function queries the characteristics of an entry whose index is specified by the *matid* argument in the material table specified by the *mtable* argument. The *propertybit* argument is one of the following:

WTMAT_AMBIENT
WTMAT_AMBIENTDIFFUSE
WTMAT_DIFFUSE
WTMAT_EMISSION
WTMAT_OPACITY
WTMAT_SHININESS
WTMAT_SPECULAR

The *value* argument must be an *array of three floats* when querying the ambientdiffuse, ambient, diffuse, specular, or emission properties, or an *array of one float* when querying the shininess or opacity properties. Passing an array of three floats when querying shininess or opacity is permitted, but only the *value[0]* argument is written into.

FALSE is returned if the specified *propertybit* is not defined for the given material table.

WTmtable_setentryname

```
FLAG WTmtable_setentryname(  
    WTmtable *mtable,  
    int matid  
    char *name)
```

This function assigns a name specified by the *name* argument to an entry whose index is specified by the *matid* argument in the material table specified by the *mtable* argument. The default name of a material table entry is NULL.

WTmtable_getentryname

```
char* WTmtable_getentryname(  
    WTmtable *mtable,  
    int matid)
```

This function returns the name of an entry whose index is specified by the *matid* argument in the material table specified by the *mtable* argument.

WTmtable_getentrybyname

```
int WTmtable_getentrybyname(  
    WTmtable *mtable,  
    char *name)
```

This function returns the index of an entry whose name is specified by the *name* argument in the material table specified by the *mtable* argument.

-1 is returned if no entry in the material table matches *name*.

Advanced Topics

How WTK Deals With Out-Of-Range Indices

When rendering, WTK assigns color to the polygon (or vertex) by using the material table index specified by the polygon (or vertex) to a material in the material table.

If the polygons and vertices in a geometry have material table indices higher than the number of materials in the material table, a modulus operation is executed on the index *at the time the polygon is rendered*. For example, if a material table has two colors in it (black and white), and the geometry consists of four polygons, with material index references 0,1,2,3, the polygons would be rendered as black, white, black, white. The material indices remain as they were: 0,1,2,3, but are rendered as if they were 0,1,0,1, because there are only two entries in the material table. If a new entry, red, is added to the material table at this point, in the next frame the geometry will be rendered as black, white, red, black. The actual indices 0,1,2,3 are rendered as if they were 0,1,2,0 because there are three entries in the material table.

A negative material index is not allowed.

Using Material Index Table Entries

Wtpoly_setmatid

See *Wtpoly_setmatid* on page 7-3 of the *Polygons* chapter.

Wtpoly_getmatid

See *Wtpoly_getmatid* on page 7-3 of the *Polygons* chapter.

WTgeometry_setmatid

See *WTgeometry_setmatid* on page 6-31 of the *Geometry* chapter.

WTgeometry_setvertexmatid

See *WTgeometry_setvertexmatid* on page 6-48 of the *Geometry* chapter.

WTgeometry_getvertexmatid

See *WTgeometry_getvertexmatid* on page 6-48 of the *Geometry* chapter.

Using Materials Tables With Geometries

As described in this chapter, you can define a geometry's material in its own material table. Each geometry references a single material table, from which the geometry's material properties are obtained.

WTgeometry_setmtable

See *WTgeometry_setmtable* on page 6-30 of the *Geometry* chapter.

WTgeometry_getmtable

See *WTgeometry_getmtable* on page 6-31 of the *Geometry* chapter.

Notes on Specific File Formats

WTK now has an expanded NFF file format which records material ID's for each polygon or vertex, instead of RGB color, as was done in WTK V2.1 and earlier (see *Changes in Reading/Writing NFF Files* on page G-24 for more information).

When an NFF 2.1 (an NFF file saved with version 2.1 of WTK) or earlier object file is loaded, a material table is created that has one material for each unique color in the object. The material table in this case has only the AMBIENTDIFFUSE field defined, so the NFF 2.1 geometry will look the same in this release as it did in WTK V2.1. The created material table has an automatically generated name. In each NFF file, there is also a reference to a material file. The new NFF file format is described in *Appendix F*.

The same is true for files read from MultiGen .flt files, ProEngineer RENDER format, AutoCad DXF, and Videoscape 3D .geo file formats.

Wavefront .obj files have material properties defined in externally referenced .mtl files; for these files all of the following material properties are read in and defined: ambient, diffuse, specular, specular exponent (shininess), and transparency (alpha). The .mtl format does not have an “emissive” material property. When a Wavefront file is loaded, any referenced .mtl file is parsed, and a new material table is created; the new material table is defined for those fields that are specified in the Wavefront .mtl file. If for example, “ambient” color is not defined for any of the materials in the .mtl file, then the new material table will be created without the “ambient” field defined. Each Wavefront material has a name as specified in the .mtl file; each *WTmaterial* defined has its name set to match the one specified in the .mtl file.

3D Studio .3ds files have material properties defined within the file. Like the Wavefront files, a material table is created and material properties are read in and defined for ambient, diffuse, specular, specular exponent (shininess), and transparency (opacity).

NFF 3.0 (the new format for WTK Release 6 and this release) files have external references to the new WTK .mat material file format. This is an easy-to-edit ASCII format.

OpenGL Compatibility

The OpenGL specification is a powerful, cross-platform definition of how lighting models are to be implemented. By providing material properties that conform to this standard, WTK preserves all of the control that users of previous versions of WTK had for

determining the coloring of geometries and polygons, but adds additional features — notably specular highlights. WTK takes full advantage of the features available with OpenGL.

Creating Three-dimensional Text in WTK

WTK allows you to create 3D text for your virtual world. These 3D text strings are simply WTK geometries, which can be used as described in the *Geometries* chapter. 3D text geometries are assembled from individual characters that can either be polygonally based or represented using bit-mapped pictures of the characters applied to polygonal surfaces. The size, shape, and style of each individual character depends on the 3D font that you are using. A 3D font is specified by an NFF file that contains one NFF object for each character in the font. You can use several different 3D fonts simultaneously to create text string geometries at any time.

Creating 3D text in WTK is a two-step process. First you load in a 3D font by calling *WTfont3d_load*. This loads a font from an NFF file (i.e., creates an NFF object for each character in the font and stores it in memory). Then the function *WTgeometry_newtext3d* is used to construct geometries from character strings. Figure 9-1 illustrates the parameters associated with a WTK 3D font. For purposes of clarity, these parameters are shown using a projection of the font onto a 2D plane. The extents of the font in 3D are illustrated in figure 9-2 on page 9-4.

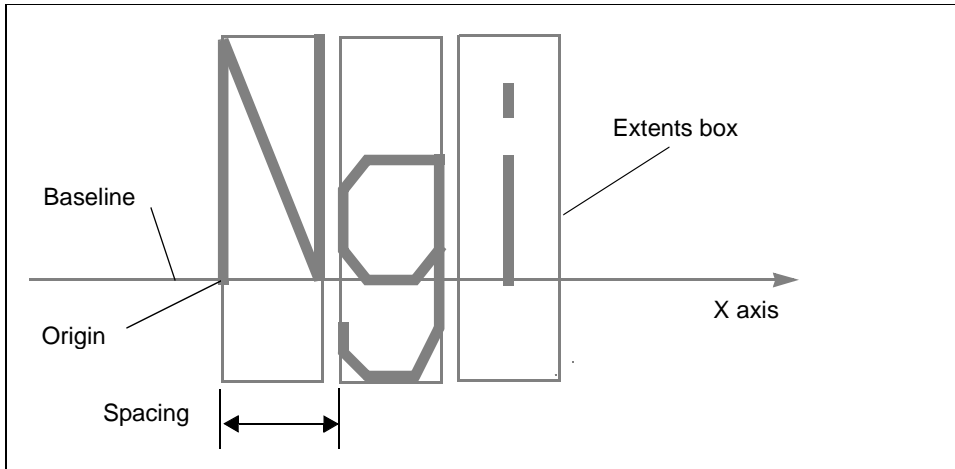


Figure 9-1: Basic font properties

WTfont3d_load

```
WTfont3d *WTfont3d_load(  
    char *filename);
```

This function loads a 3D font file into memory and returns a pointer to a *WTfont3d* structure used to refer to the font. The *filename* argument should be the name of an NFF 3D font file such as the sample font file provided with WTK. Like the functions *WTgeometrynode_load* and *WTnode_load*, the WTMODELS path is searched for the *filename*, if it is not found in the current directory.

WTK comes with at least one sample 3D font file. With this release, the sample font file is *rcfont3d.nff*, located in the modeler directory in the WTK product distribution. A description of the format of the 3D font file is provided at the end of this chapter as a reference in case you want to define your own 3D fonts.

WTfont3d_delete

```
void WTfont3d_delete(  
    WTfont3d *font);
```

This function frees the memory used by a *WTfont3d* structure. Once you have constructed all of the text strings required for your application which use this font (using *WTgeometry_newtext3d*), you can call *WTfont3d_delete* at any time.

WTgeometry_newtext3d

See *WTgeometry_newtext3d* on page 6-20 for a description.

WTfont3d_setspacing

```
void WTfont3d_setspacing(  
    WTfont3d *font,  
    float spacing);
```

This function sets the spacing for a font by setting the horizontal spacing between the base-points of the characters in a 3D text string (see figure 9-1 on page 9-2). By default, this spacing is 10% greater than the width of the widest character. Using this function affects the spacing of all subsequently created geometries.

In the following example, the font spacing is increased by 20 percent:

```
WTfont3d_setspacing(font, 1.2 * WTfont3d_getspacing(font));
```

See also *WTfont3d_getextents*, on page 9-4.

WTfont3d_getspacing

```
float WTfont3d_getspacing(  
    WTfont3d *font);
```

This function returns the current spacing value for the specified 3D font. Figure 9-1 and figure 6-1 illustrate “spacing” of a font.

Figure 9-2 illustrates the extents of a WTK 3D font.

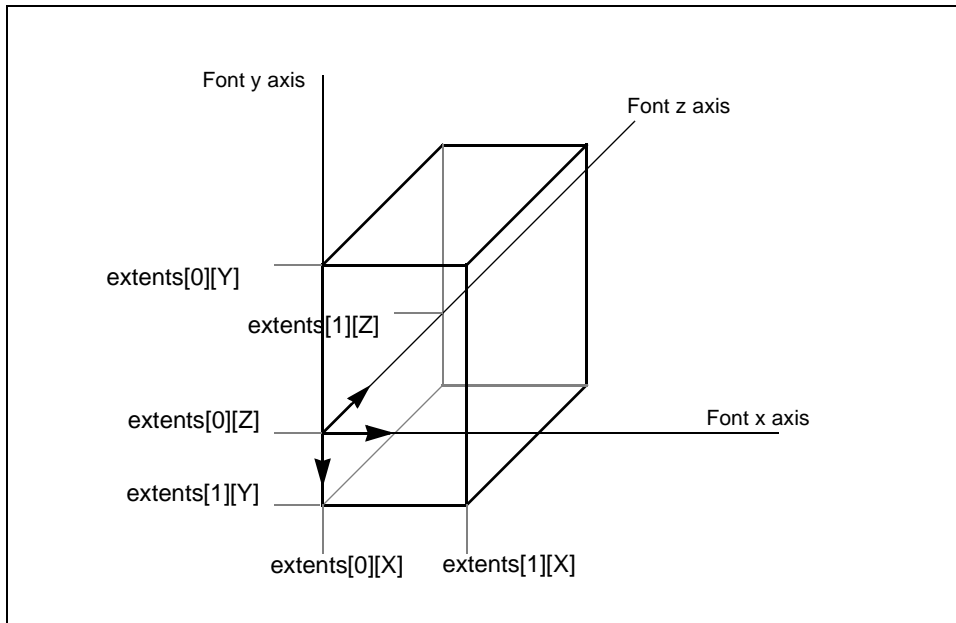


Figure 9-2: The 3D font's extents box

WTfont3d_getextents

```
void WTfont3d_getextents(
    WTfont3d *font,
    WTp3 extents[2]);
```

This function gets the 3D extents box for the specified 3D font. *WTfont3d_getextents* places the minimum and maximum spatial extents of the characters of the specified font into the *extents[0]* and *extents[1]* vectors respectively.

For example, for characters oriented to be read in the X-Y plane, the largest X coordinate value of any character in the font is placed in *extents[1][X]*, and the smallest value is placed in *extents[0][X]*. Therefore, the maximum width of any character in the font is *extents[1][X] - extents[0][X]*.

Correspondingly, for the Y value, the maximum vertical extent (height) of any character in the font is $extents[1][Y] - extents[0][Y]$ and for the Z value, the maximum depth extent of any character in the font is $extents[1][Z] - extents[0][Z]$.

The default spacing between characters in the font is ten percent greater than the maximum character width (i.e., ten percent greater than $(extents[1][X] - extents[0][X])$). You can determine the spacing by calling `WTfont3d_getspacing` after `WTfont3d_load` is called.

WTfont3d_charexists

```
FLAG WTfont3d_charexists(  
    WTfont3d *font,  
    char character);
```

This function determines whether a particular character is defined in the specified 3D font. If the specified character is in the font, TRUE is returned, otherwise FALSE is returned.

For example, to find out whether the font includes an exclamation mark, you could use:

```
WTfont3d *font;  
if ( WTfont3D_charexists(font, '!') )  
    WTmessage("Font %p contains '! \n", font);  
else  
    WTwarning("Font %p does not contain '! \n", font);
```

NFF 3D Font Files

This section describes the structure of the NFF file from which a WTK 3D font can be constructed. Also see Appendix F for a complete description of the WTK NFF format.

A 3D font file is a multi-geometry NFF file that contains one geometry for each character. The names of the character geometries are the string “char” followed immediately by the ASCII value of the character. For example, the name of the geometry representing a capital “A” would be “char65” since the ASCII code for “A” is 65. A 3D font file containing all of the capital letters would have geometries named “char65”, “char66”, “char67” and so on up to “char90” (capital “Z”). Lower-case letters are “char97” through “char122.” Table 9-1 lists ASCII character values.

The function *WTgeometry_newtext3d* constructs text geometries by assembling characters along the +X direction. Therefore, characters in your font file should read so that “left to right” corresponds to increasing X coordinate values. With this convention, depending on the way in which the characters in the font file are modeled, *WTgeometry_newtext3d* might return text geometries that are readable in the X-Y plane or in the X-Z plane (or with any angle in between). Of course, the text geometries once created can be placed at any location using the geometry move functions.

When a text string is assembled with *WTgeometry_newtext3d*, the characters are lined up based on the location of their base points. The point (0, 0, 0) is the base point for each character. You should define characters in the NFF 3D font file relative to this point.

Table 9-1: The ASCII character set

	char64 @	char96 `
char33 !	char65 A	char97 a
char34 “	char66 B	char98 b
char35 #	char67 C	char99 c
char36 \$	char68 D	char100 d
char37 %	char69 E	char101 e
char36 &	char70 F	char102 f
char39 '	char71 G	char103 g
char40 (char72 H	char104 h
char41)	char73 I	char105 i
char42 *	char74 J	char106 j
char43 +	char75 K	char107 k
char44 ,	char76 L	char108 l
char45 -	char77 M	char109 m
char46 .	char78 N	char110 n
char47 /	char79 O	char111 o
char48 0	char80 P	char112 p

Table 9-1: The ASCII character set (continued)

char49 1	char81 Q	char113 q
char50 2	char82 R	char114 r
char51 3	char83 S	char115 s
char52 4	char84 T	char116 t
char53 5	char85 U	char117 u
char54 6	char86 V	char118 v
char55 7	char87 W	char119 w
char56 8	char88 X	char120 x
char57 9	char89 Y	char121 y
char58 :	char90 Z	char122 z
char59 ;	char91 [char123 {
char60 <	char92 \	char124
char61 =	char93]	char125 }
char62 >	char94 ^	char126 ~
char63 ?	char95 _	

This chapter describes the textures that can be applied to the surfaces of graphical objects, and the functions to apply, manipulate, and animate them. The main sections of this chapter are as follows:

- *Introduction* – provides a general discussion on the use of textures in WTK and lists the texture file formats supported by WTK. (see page 10-2)
- *Applying Textures* – describes how to apply textures to geometric surfaces in WTK. (see page 10-4)
- *Changing Texture Properties* – describes how to access the shading, transparency, and blending values of a polygon's texture. (see page 10-23)
- *Filtering Textures* – describes how to specify and retrieve the filter values for a texture already applied to a polygon. (see page 10-24)
- *Manipulating Textures* – describes how to change the orientation, scale, and offset of applied texture. (see page 10-27)
- *Screen Loading* – describes how to load an image to and get an image from a WTK window. (see page 10-33)

Introduction

Surfaces of objects in the real world are not smooth and featureless — they have pattern, grain, and detail. To emulate this, you can give WTK polygons a surface texture. This texture is a bit-mapped image, which is applied to the surface of the polygon and transformed with it. For example, you can create a table top from a uniform brown-shaded polygon with an actual wood-grain image mapped onto it.

You can create textures with a bitmap image editor or derive them from video images. Basically, anything that can find its way onto a computer screen can be converted to a texture format. Figure 10-1 shows a WTK virtual world with textures applied to it.

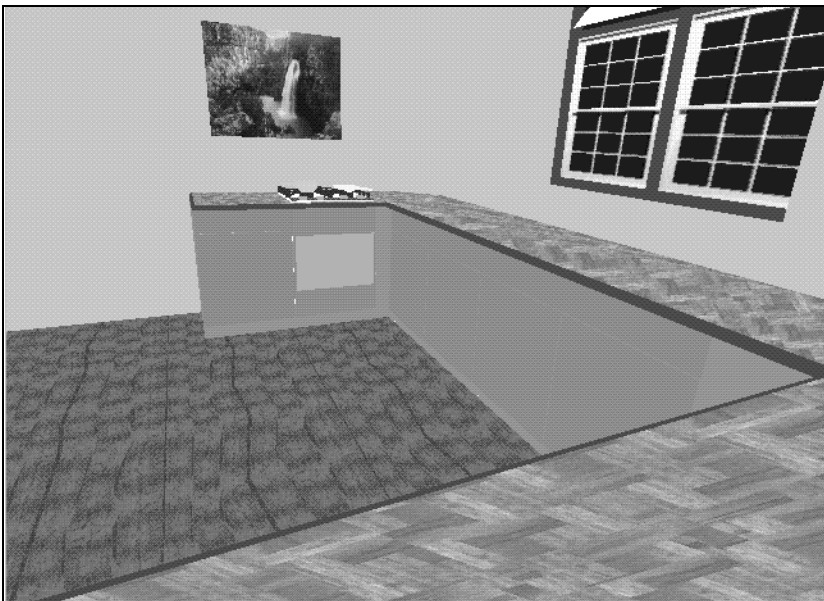


Figure 10-1: WTK virtual world with textures applied

Judicious use of textures can increase the complexity and realism of your environments, allowing you to avoid both the initial work of modeling surface details and the run-time overhead of transforming them. For example, instead of modeling as 3D details all of the windows of a distant building, you can apply a digital image of a real building to a single polygon, which then serves as an entire side of the building. Your modeling labor is conserved and rendering speed increases dramatically compared to what would have been

necessary to model all of these details in 3D. However, the frame rate of the simulation is still affected by texturing — although it is better to use textures than to model all the details, textures do slow performance compared to just rendering polygons that have neither texture nor modeling details.

Textures are automatically transformed with the polygons to which they are applied, displaying perspective shift and scaling appropriate for the viewing parameters (see *Applying Textures* on page 10-4). WTK has functions for changing the orientation, scale, and offset of applied textures (see *Manipulating Textures* on page 10-27).

You can dynamically replace a texture on a polygon, which gives the impression of animation. For example, you can use *WTtexture_replace* to sequentially load images from a video file to a polygon in the shape of a TV screen. Or you can map images from a viewpoint to a polygon to create a rear-view mirror in your simulation. See *Animating Textures* on page 10-18.

You can also make part of a texture transparent, which allows whatever is behind it in the simulation to show through. See *Changing Texture Properties* on page 10-23. Also refer to your Hardware Guide for information about system-specific texture-mapping capabilities and limitations.

Supported Texture File Formats

WorldToolKit supports the following texture file formats.

Targa	.tga extension
RGB format	.rgb and .rgba extensions
JPEG - JFIF compliant	.jpg extension

These formats are supported on all platforms. Note that 8 bit TGA format files are not supported.

Applying Textures

Use the functions in this section to apply textures to geometric surfaces. Several of the demos provided with WorldToolKit illustrate the use of these functions. See the README files in the *demos* and *images* subdirectories that were installed with WTK.

Table 10-1 below, lists the methods provided in WTK for applying textures to the surfaces of geometries.

Table 10-1: Methods for applying textures to geometry surfaces

Method	Functions used	Remarks
Automatic	<i>WTpoly_settexture</i> <i>WTgeometry_settexture</i>	The texture is applied to each polygon so that it is oriented upright on the polygon and reads from left to right when looking at the polygon's front face. <i>WTgeometry_settexture</i> calls <i>WTpoly_settexture</i> to apply a texture in this manner to each polygon in the geometry. The precise method of texture application is described under <i>How WTK Applies a Texture to a Polygon</i> on page 10-5.
Explicit uv specification	<i>WTpoly_settextureuv</i> <i>WTgeometry_settextureuv</i>	The texture is applied using the specified uv texture coordinates. The function <i>WTgeometry_settextureuv</i> calls <i>WTpoly_settextureuv</i> to apply a texture to each polygon in the geometry so that the texture appears draped or wrapped over the geometry.
Use 3D model file format	n/a	The file itself contains texturing information.

How WTK Applies a Texture to a Polygon

Here's how WTK applies a texture to a polygon:

1. The edge of the polygon with the largest upward (i.e., negative) y-axis component is found.
2. Texture is applied so that (a) the vertical edge of the texture is parallel to this polygon edge, and (b) from a viewpoint looking at the polygon's front face, the texture reads from left-to-right (the texture example in figure 10-2 below, contains text to illustrate this).
3. Given two solutions to (a) and (b), the texture is applied so it appears right-side-up rather than upside-down (with respect to a viewpoint that is right-side-up, of course).

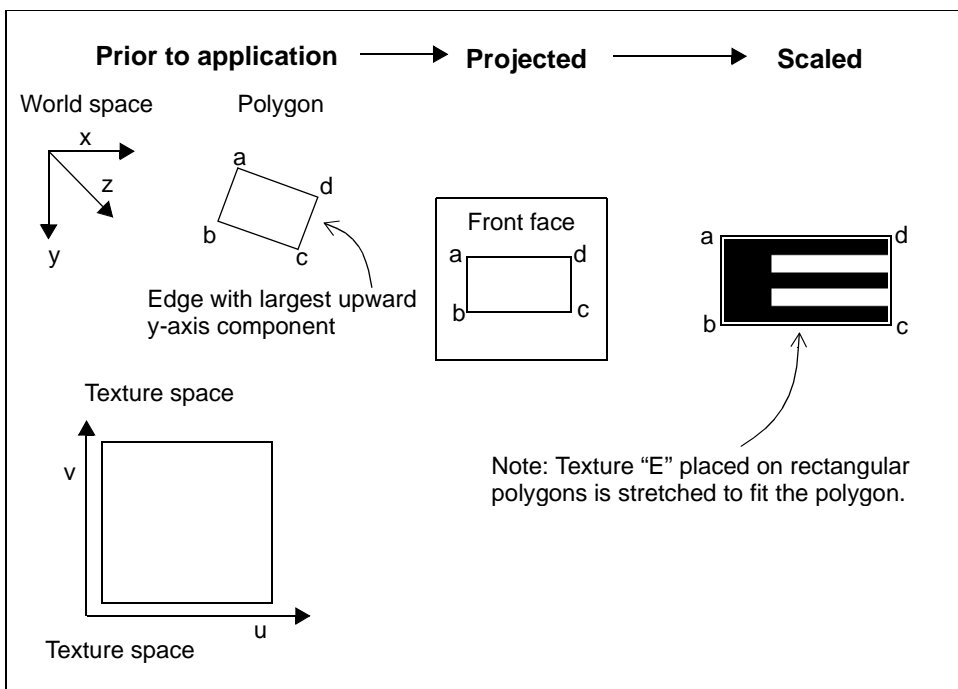


Figure 10-2: Texture application on rectangular polygons.

In the case of the rectangular polygon shown in figure 10-2, the edge with the largest upward (i.e., negative y-axis) component is c-d. (Even though edge d-a is also oriented upward and is longer than c-d, the y-axis component of d-a is less.)

Assuming the polygon vertices are stored in the polygon in the order a-b-c-d, figure 10-2 shows the front face of the polygon. The texture is applied so that if we tilt our heads (or equivalently rotate the polygon) so that the edge c-d is vertical, then the texture reads correctly from left-to-right, that is, the *E* looks like an *E* — not backwards and not upside-down.

Furthermore, on a rectangular polygon, the texture corners (in u,v space) are mapped exactly to the corners of the rectangle. In other words, the texture is stretched to fit exactly onto the rectangle. None of the texture image is cropped when applied.

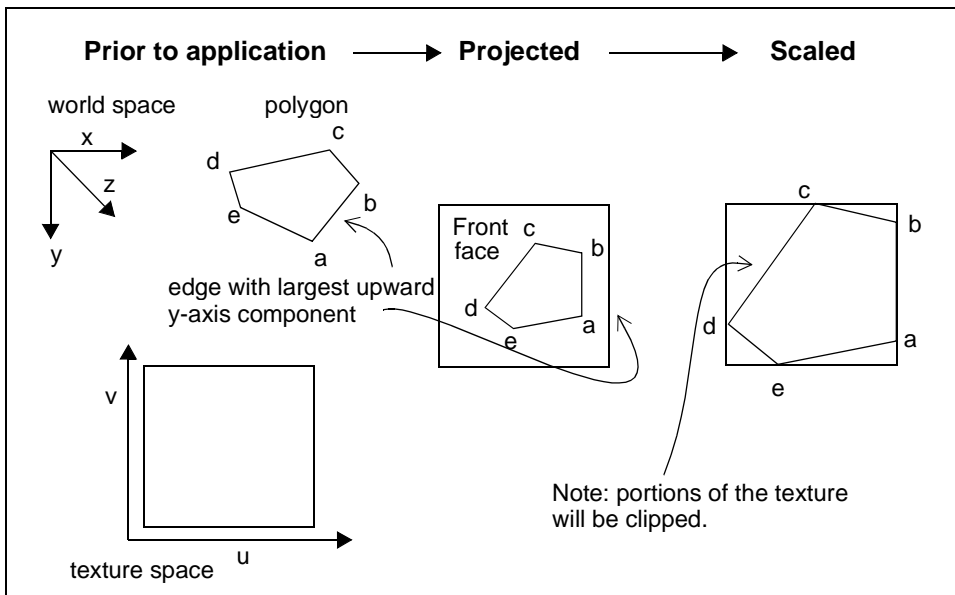


Figure 10-3: Texture application on non-rectangular polygons.

When a texture is applied to a non-rectangular polygon, the same basic technique is used as with rectangular polygons.

In figure 10-3, we start with a five-sided polygon. Since the vertices a-b-c-d-e go around counter-clockwise, we are again looking at the front face of the polygon. In this case, the

polygon is simply rotated so that the edge a-b (the edge with the largest upward y-axis component) is correctly aligned with the texture.

Then, the texture is applied so that the minimum and maximum u and v texture values map to the minimum and maximum horizontal and vertical extents of this rotated polygon. Note that portions of the texture are cropped or clipped. You can use *WTPoly_scaletexture* or *WTPoly_stretchtexture* if cropping is not desired.

The important thing to remember is that textures are aligned to polygonal surfaces based on the surface's orientation in the world reference frame and on the order in which the polygon vertices are specified. These two factors determine which polygon edge has the largest upward y-axis component. If you tried applying a texture to a moving object once per frame (as you do when using texture animation), you would see that the texture edge is sometimes aligned to different polygonal edges based on the polygon's current orientation with respect to the world. In some cases, this causes problems that must be fixed using the texture rotation feature (see *Manipulating Textures* on page 10-27).

For a polygon that is oriented horizontally, the edge to which the texture is aligned is the first polygon edge. Specifically, for a horizontal polygon, the right edge of the texture bitmap is made to lie along the first polygon edge, where moving from the first polygon vertex to the second moves you in the direction from the bottom right corner of the bitmap toward the top right corner of the bitmap. In addition, the texture may be stretched in the same way as shown in figure 10-3 for the polygon with a vertical component.

After application through this default mapping, textures may be modified using any of the texture manipulation functions described beginning on page 10-27.

Texture Size

You can also set the maximum texture size that will be loaded by your application. To do this, use the *WTMAXTEXSIZE* environment variable (see the Environment Variables Appendix for details on setting this environment variable). When you set the maximum texture size, the texture images will be shrunk, if necessary, so that the image width and height in pixels will not exceed this value. This is very valuable, as it ensures that your application does not exceed your hardware texture memory limits. The default value is 1024 (this is also the maximum), but you can, for example, set it to 512 or 256.

Texture dimensions must be a power of two (e.g. 16, 32, 64, etc.). If a texture whose width and height is not a power of two, WTK will automatically size the texture image to the

nearest power of two. For example, a 65 x 190 resolution image will be resized to 64 x 256. If the environment variable *WTKSQRTX* is enabled (set to 1), WTK will automatically shrink texture images, if necessary, so that the texture's width and height are equal. By default, *WTKSQRTX* is disabled (set to 0), and so texture images are not shrunk into square images.

Texture Naming Conventions

When using functions *WTgeometry_settexture*, *WTgeometry_settextureuv*, *Wtpoly_settexture*, and *Wtpoly_settextureuv*, one of the parameters you must supply is the filename of the texture being applied. If you don't specify the filename's extension, WTK will look for a file whose extension is recognized as a texture file (.tga, .rgb, .rgba, .jpg). By leaving off the extension, you can port your WTK application to a different hardware platform without having to change all the extension names in your source code.

Texture files specified in a WTK function call are searched for in the current directory, and along the path given by the *WTIMAGES* environment variable. If multiple files with the same filename but different extensions exist in the same directory and your file specification doesn't include the file extension, the precedence of texture file formats is dependent upon the platform you are running on. On Windows platforms, the precedence order is .tga, .rgb, .rgba, and .jpg while on UNIX platforms the precedence order is .rgb, .rgba, .tga, and .jpg.

Texture filenames are case sensitive within WTK. For example, if you were to load a 'flag.tga' texture into WTK via *WTgeometry_settexture* and then wanted to modify the texture's filtering via a call to *WTtexture_setfilter*, the filename you would need to specify in *WTtexture_setfilter* is 'flag.tga'. If you tried to refer to the texture as 'FLAG.TGA', WTK would not recognize it as the 'flag.tga' texture.

Transparent Textures

WTK textures can be transparent. A texture consists of a rectangular array of texels composed of a color component and possibly an alpha component as well. The alpha component of each texel can range from 0 to 255 and indicates the texel's degree of opacity. An alpha value of 0 means that the texel is completely transparent while an alpha value of 255 means that the texel is completely opaque. When a texture file does not contain alpha values, WTK automatically computes the alpha value for each texel using the following

schema. Texels whose color component is black, i.e. whose $R = G = B = 0$, will be assigned an alpha value of 0. Texels whose color component is non-black, i.e. either R, G, or B is non-zero, will be assigned an alpha value of 255. If you do not wish to have WTK automatically compute the alpha values for textures in this manner, you must use the `WTOPTION_NOAUTOALPHA` option of the `WTuniverse_setoption` function. If the universe's `WTOPTION_NOAUTOALPHA` option is set, then WTK will assign an alpha value of 255 (opaque) to the texels of all textures which do not contain alpha values

From the above discussion, it follows that there are two ways in which you can obtain transparencies in textures.

- Using a texture file that has an alpha component
- Using a texture file that does not have an alpha component and letting WTK automatically add the alpha values to the texels, following the process described above.

USING A TEXTURE FILE THAT HAS AN ALPHA COMPONENT

Not many texture file formats support a built in alpha component. The texture file must be in the 'rgba' format, where the 'rgb' is the color component and the 'a' is the alpha component. When the texture is applied to a polygon (using say, `Wtpoly_settexture`), you can specify whether the texture should be transparent or not via the "transparent" flag. If this flag is set to `TRUE`, the texels whose alpha values fall below a certain threshold are not drawn on the screen. This threshold is called the alpha-threshold (range 0-255) and can be controlled using the environment variable `WTKALPHATEST`. By default, this value is 0 on Windows platforms and 78 on UNIX platforms. So for example, on Windows platforms, the texels which have alpha values 0 will be transparent since they won't be drawn on the screen. An alpha value greater than 0 will not be transparent. If however, when the texture is applied, the "transparent" flag is set to `FALSE`, the alpha-threshold test will not be done and all texels will be drawn on the screen.

USING A TEXTURE FILE THAT DOES NOT HAVE AN ALPHA COMPONENT

Most common texture files do not have an in built alpha component. The texture file has an 'rgb' format, i.e., only the color component is present. When the texture file is loaded into WTK, WTK automatically inserts alpha values for each texel. If the texel's color component is black, ($r=g=b=0$), the alpha value inserted is 0. If the texel's color component is non-black, the alpha value inserted is 255. You can now use the newly added alpha component to obtain transparency. (This is called the "cookie-cutter" method). When the

texture is applied to a polygon (using say, `WTPoly_settexture`), you can specify whether the texture should be transparent or not via the "transparent" flag. If this flag is set to `TRUE`, the texels whose alpha values are 0, will be transparent. (They will not be drawn on the screen.) The texels whose alpha values are 255 will appear on the screen. This behavior is the result of the alpha-threshold test that is performed on all texels.

The "cookie-cutter" method is a simple way to obtain transparencies in textures that do not have alpha values. First, the areas of the texture that you want transparent must be colored black. Second, the texture must be tagged 'transparent' using the "transparent" flag when you use `WTPoly_settexture` (or marking the polygon with '`_t_`' when you read textured polygons from an NFF file. See Appendix F, for the NFF file format). This method sometimes leaves a "black halo" around the cut out part when texture filtering other than linear is used. This effect can be avoided by using point texture filtering (`WTFILTER_NEAREST`). (See Texture Filtering for more information). When using texture filtering methods other than point, it is possible to improve the quality of the picture by raising the value of the alpha threshold.

If you do not wish to have WTK automatically calculate the alpha values for you, you may set the universe option `WTOPTION_NOAUTOALPHA` to `TRUE`. (Use the function `WTuniverse_setoption` to do this.) If this option is set, all the alpha values (for an 'rgb' texture file) will default to 255.

TRANSPARENCY AFFECTED BY THE POLYGON'S MATERIAL PROPERTIES

In actuality, there is a third way to achieve transparencies. This method makes use of the object's material properties - in specific, the value of the "opacity" property (range 0-1). If a polygon's opacity value is less than 0.996, it is treated as one having translucency. Any value greater than 0.996 causes WTK to treat the polygon as opaque for all practical purposes. If a polygon is translucent, it's opacity value is multiplied with the texel's alpha value before that pixel is rendered. The resultant alpha affects the final color of that pixel. The color of the pixel will be a mixture of the texel's color component and the color of the background at that pixel. (This is the color that already exists in the color buffer). If the resultant alpha is closer to 1, more of the texel's color is used. If the resultant alpha is closer to 0, more of the background color is used.

This procedure can be used to enhance (or even, create) transparencies in textures. By controlling the product of the texel's alpha and the polygon's opacity, you can set areas of a texture to use more of the background color, and hence create/enhance transparency. For example, if a texel's alpha value is 10, and the polygon's opacity is 0.95, the product is

closer to 0 on a scale of 0-255. This causes that texel to be nearly transparent, since a very small fraction of the texel's color contributes to the final color.

WTpoly_settexture

```
FLAG WTpoly_settexture(  
    Wtpoly *poly,  
    char * bitmap,  
    FLAG shaded,  
    FLAG transparent);
```

This function applies a texture bitmap stored in the file to the specified polygon. The argument *bitmap* refers to the bitmap file; the argument *poly* refers to the polygon. If the polygon already had a texture applied, the new texture replaces the old texture. The FLAG arguments indicate whether the texture is to be shaded and/or transparent.

If a texture is shaded (*shaded*=TRUE), the intensity of the texture elements (texels) are affected by lighting. If colored lights are used, the color of texture elements is also affected. If the texture is not shaded (*shaded*=FALSE), the texture appears as in the source bitmap file.

The transparent parameter is used to indicate whether the texture should be treated as a transparent texture. See *Transparent Textures* on page 10-8 for more information about transparent textures.

If the specified file is not found in the current working directory, the *WTIMAGES* path is searched. See the Environment Variables Appendix for information about setting the *WTIMAGES* environment variable and see *Texture Naming Conventions* on page 10-8 about texture filename extensions.

The function *WTpoly_settexture* returns TRUE if the texture could be applied. If the texture could not be found, it returns FALSE.

See also *WTpoly_deletetexture* on page 10-23.

WTgeometry_settexture

```
FLAG WTgeometry_settexture(  
    WTgeometry *geom,  
    char * bitmap,  
    FLAG shaded,  
    FLAG transparent);
```

This function applies a texture bitmap to each polygon surface of a geometry (using the *Wtpoly_settexture* function). It returns TRUE if successful and FALSE otherwise.

If any of the geometry's polygons already have a texture applied, the old texture is replaced by the new texture. The *bitmap* argument refers to the filename of the texture bitmap. The *FLAG* arguments indicate whether the texture is to be shaded and/or transparent.

If a texture is shaded (*shaded*=TRUE), the intensity of the texture elements (texels) are affected by lighting. If colored lights are used, the color of texture elements is also affected. If the texture is not shaded (*shaded*=FALSE), the texture appears as in the source bitmap file.

The transparent parameter is used to indicate whether the texture should be treated as a transparent texture. See *Transparent Textures* on page 10-8 for more information about transparent textures.

If the specified file is not found in the current working directory, the *WTIMAGES* path is searched. See the Environment Variables Appendix for information about setting the *WTIMAGES* environment variable and see *Texture Naming Conventions* on page 10-8 about texture filename extensions.

In the following example, a shaded texture in a file *wood* is applied to every polygonal surface of a sphere geometry.

```
WTgeometry *geometry;  
geometry = WTgeometry_newsphere(5.0, 10, 10, FALSE, TRUE);  
if ( WTgeometry_settexture(geometry, "wood", TRUE, FALSE) )  
    WTmessage("Applied shaded wood texture.\n");  
else  
    WTwarning("Unable to apply shaded wood texture.\n");
```

See also *WTgeometry_deletetexture* on page 10-23.

Applying Textures with Explicit uv Values

WTK supports several methods of applying textures with explicit uv information: calling `Wtpoly_settextureuv`, `WTgeometry_settextureuv`, and through file readers that support texture uv specification. These file readers include the Neutral File Format (NFF), 3D Studio, Wavefront, and MultiGen/ModelGen.

If you wish to preserve the precise texture application information when writing a geometry out to NFF (or binary NFF) when textures have been applied in any of the above-mentioned ways, then you must first instruct WTK to write out the NFF file using uv values. To do so, before saving out the file, you must call this function:

```
WTuniverse_setoption(WTOPTION_NFFWRITEUV, TRUE);
```

or set the resource value `writeuv` to TRUE. See Appendix F for information about how uv values are stored in the NFF format.

Note: When writing out NFF files with uv values: If two polygons share a vertex, but different uv values are used for the polygons at that vertex, then a new vertex is created and appended to the geometry's vertex list. In this way, each vertex written out in the NFF file has a unique uv value. This will not occur if the geometry was textured using either of the texture draping functions `WTgeometry_settextureuv` or `WTgeometry_setuv`, because these functions ensure that shared vertices have the same texture uv coordinates.

Wtpoly_settextureuv

```
FLAG Wtpoly_settextureuv(  
    Wtpoly *poly,  
    char *bitmap,  
    float *uarray,  
    float *varray,  
    FLAG shaded,  
    FLAG transparent);
```

This function applies a texture bitmap stored in the specified file to the specified polygon and allows you to choose the way the texture is mapped onto the polygon.

Like the function `WTPoly_settexture`, this function allows you to apply a bitmap texture to a polygon, passing in a pointer to the polygon `poly`, the *name* of the bitmap file, and the values *shaded* and *transparent*. (See the function `WTPoly_settexture` on page 10-11 for more information about the parameters *bitmap*, *shaded*, and *transparent*.)

The function `WTPoly_settextureuv` enables you to specify the way in which the texture is mapped onto the polygon, by passing in to this function the arrays `uarray` and `varray`. These two arrays must be allocated by the application and *must* have at least as many elements as there are vertices in the polygon (which can be obtained using the function `WTPoly_numvertices`). The elements of `uarray` and `varray` specify, respectively, the texture *u* and *v* coordinates to use when mapping the texture to the vertices of the polygon. The polygon's vertices (and corresponding elements of `uarray` and `varray`) are taken in the order in which the vertices are stored with the polygon. The vertex order can be obtained using `WTPoly_getvertex`.

The value `u=0.0` corresponds to the left edge of the source bitmap, and `u=1.0` corresponds to the right edge. The value `v=0.0` corresponds to the bottom edge of the source bitmap, and `v=1.0` corresponds to the top edge.

In the following code fragment, the bottom half of a texture called *fish* is applied transparently to a polygon:

```
WTPoly *poly;
float u[4],v[4];
u[0] = 0.0; v[0] = 0.0;
u[1] = 1.0; v[1] = 0.0;
u[2] = 1.0; v[2] = 0.5;
u[3] = 0.0; v[3] = 0.5;
WTPoly_settextureuv(poly, "fish", u, v, FALSE, TRUE);
```

WTgeometry_settextureuv

```
FLAG WTgeometry_settextureuv(  
    WTgeometry *geom,  
    char *bitmap,  
    float (*fu)(WTp3),  
    float (*fv)(WTp3),  
    FLAG shaded,  
    FLAG transparent);
```

This function drapes or wraps a texture around a geometry. It applies the specified bitmap texture to every polygon of the geometry, using the specified functions *fu* and *fv* to determine exactly how the texture is mapped onto the geometry. The two functions *fu* and *fv* take a 3D point (a *WTp3*) as an argument and return a floating point value. These functions must be specified in your application. They define the mapping from vertex positions to texture u and v coordinates, respectively. (See the function *WTgeometry_settexture* on page 10-12 for more information about the parameters *bitmap*, *shaded*, and *transparent*.)

For example, to specify the functions *fu* and *fv*, you might use the following:

```
/* fu computes the texture "u" coordinate from a vertex position */  
float fu(WTp3 v) {  
    return 0.01 * (v[X] + v[Y] + v[Z]);  
}  
/* fu computes the texture "v" coordinate from a vertex position */  
float fv(WTp3 v) {  
    return 0.01 * (v[X] + v[Y] - v[Z]);  
}
```

Then, in your WTK application, you might call *WTgeometry_settextureuv* as shown below (where it is assumed that *g* is a *WTgeometry** declared in your application):

```
/* apply shaded texture using specified fu and fv functions */  
WTgeometry_settextureuv(g, "myimage", fu, fv, TRUE, FALSE);
```

WTgeometry_changetexture

```
FLAG WTgeometry_changetexture(  
    WTgeometry *geom,  
    char *bitmap,  
    FLAG shaded,  
    FLAG transparent);
```

This function changes all textured polygons of the specified geometry to use the new texture bitmap instead of their current texture. The new bitmap is specified by the *bitmap* argument. The shading and transparent flags are applied in a fashion similar to *WTgeometry_settexture*.

If this function fails, `FALSE` is returned and no changes are made (for example, if the texture bitmap specified by *bitmap* is not found or if you specified an invalid geometry).

WTtexture_replace

```
FLAG WTtexture_replace(  
    char *bitmap,  
    int format,  
    int width,  
    int height,  
    unsigned char *image);
```

This function dynamically replaces the image associated with a texture bitmap used for texturing polygons. This function works even if the bitmap hasn't already been loaded.

All polygons that reference the texture bitmap will display the new texture image, using the polygon's settings for shaded or transparent display. This allows special effects like playing real-time video on a polygon or performing interactive pixel-level edits to a texture. Any subsequent reference to the texture bitmap name will use the texture image defined by this function, even if a texture bitmap exists with the same filename. If you remove all polygon references to the bitmap's name, the bitmap will be deleted.

The first parameter is the name of the texture bitmap. This must be the same name used by *WTPoly_settexture* or referenced in the appropriate geometry file. It should be the filename of the texture you want to replace. It must follow the same naming restrictions as *WTPoly_settexture*.

The second parameter defines the format of the information in the array of color or Alpha values. It must be set to `WTIMAGE_RGBA`.

The *width* and *height* parameters define the size of the new texture bitmap. These do not need to be the same as the original texture bitmap, and they can change every frame. These parameters are restricted to certain values. The bitmap's size must be a power of two and can be no larger than the graphics hardware allows (for example, on the Integraph it's 512x512; on UNIX it's 1024x1024).

The image parameter points to an unsigned character array of RGB and Alpha values. This array is defined in row order with the first value being the lower-left corner of the bitmap image and the final value being the upper-right corner of the bitmap image. Alpha values can only be 0 or 255. The function returns TRUE if it successfully replaces the texture image within the specified texture and no parameter values were violated.

Note: The image array must not be modified after calling `WTtexture_replace` and before rendering the current frame. You must ensure that all array modifications occur prior to making this call.

See also `WTtexture_load` and `WTtexture_cache`, below.

WTtexture_load

```
unsigned char *WTtexture_load(  
    char *bitmap,  
    int *width,  
    int *height);
```

This function reads in a texture bitmap named `bitmap` and returns a pointer to the image file and the width and height of the texture. The return values of this function can then be used as parameters to the `WTtexture_replace` function.

WTtexture_cache

```
FLAG WTtexture_cache(  
    char *bitmap,  
    FLAG enable);
```

This function controls the caching of a texture. When the enable flag is TRUE, the specified texture bitmap (*bitmap*) is loaded (if not already loaded), then marked as cached. When the

enable flag is `FALSE` the texture is marked as not cached and is deleted if no polygons reference this texture. This function is useful for texture animations so that the texture does not have to be reloaded from disk during every pass of the animation.

WTtexture_iscached

```
FLAG WTtexture_iscached(  
    char *bitmap);
```

This function returns the caching state of a texture. If the texture bitmap referenced by the *bitmap* argument is not cached, then the function returns `FALSE`.

WTtexture_getmemory

```
int WTtexture_getmemory(  
    void);
```

This function returns the amount of texture memory used by the application. The value returned is in bytes and takes into account whether a texture is filtered (mipmapped). Once the texture is downloaded onto the hardware texture memory, WTK frees the space occupied by the texture in the system memory (RAM). However, when running on certain graphics boards, WTK maintains a copy of the textures in system memory. (Refer to your hardware guide for more information about this). This may also happen if the graphics board is not capable of storing sufficient texture information on the hardware. Note that `WTtexture_getmemory` does not take into account any additional copies that WTK might store in the system memory.

Animating Textures

There are two primary ways of animating textures on a polygon or geometry. Since performance is usually a key factor in texture animation, you want to choose the best method for your particular situation.

The first method of texture animation involves using `WTtexture_cache` to load and cache a finite number of textures, and using `WTgeometry_changetexture` to switch between them. For example:

```
WTtexture_cache(texture1,TRUE)
```

causes WTK to load this texture to memory and transfer it to hardware texture memory (if you are running WTK on a machine capable of doing hardware texturing). If you have a sequence of textures that you wish to play back on a geometry, you can load them all into WTK (and the hardware) by calling *WTtexture_cache* for each texture. To play back the textures onto a geometry you need to apply the first texture of the sequence to the geometry using *WTgeometry_settexture*.

Then, to play the remaining sequence of textures, you can use:

```
WTgeometry_changetexture(geometry,next_texture,...)
```

to optimally put the next texture onto the geometry. The advantage to using this method is purely performance. Since all of the textures are loaded into the optimal location (hardware texture memory if available) beforehand by *WTtexture_cache*, there is almost no work for *WTgeometry_changetexture* to do when changing the active texture on the geometry. This results in the fastest possible animation.

The disadvantage of this method is that it can really only be applied in cases where you have a small number of textures that are going to be played back. With hardware caching, all these textures must be loaded into texture memory, so the number of textures must be kept small enough to fit into texture memory along with all of the other textures in your scene.

This method is a good solution for special effects animations like an explosion sequence or any short repeating texture sequence. This method does not work well in situations where procedural texturing is needed or where the next texture animation is unknown as in the case of streamed video or a whiteboard application.

The second method of texture animation involves using *WTtexture_replace* to replace a currently active texture with an image that is in memory. Once a texture is loaded into WTK, it can be replaced with the image data passed into *WTtexture_replace*. All polygons that had the original texture image applied to them will now have the new image applied to them. This allows the application to animate a texture by continually calling *WTtexture_replace* on the same texture with different image data.

The advantage of this method is that it allows the application to specify the contents of a texture image from application memory, rather than a file. This allows the application to modify or create a texture image in place, or pass image data that is coming into the application in real time.

This method is a good solution for procedural texturing needs. The application can modify the image data in place and pass it off to WTK when ready. This is also a good method to

use when applying streamed video to a geometry or if scene image feedback is being used (like simulating a mirror). See the next section, *A Rear-view Mirror example using `WTtexture_replace`*.

The disadvantage of this method is that it can result in some performance degradation versus the first method. Every time `WTtexture_replace` is called (each new frame), WTK must pass this texture data to the rendering engine on which it sits.

Although this does not require a memory copy on the WTK side, it may require one in the rendering pipeline, and will require a bus transfer of texture data into texture memory if WTK is running with hardware acceleration. This performance penalty can be minimized by making the texture passed to `WTtexture_replace` as small as possible. In some cases the call to `WTtexture_replace` can be very expensive, for example, with WTK Direct, where the incoming image data must be quantized down to an 8-bit color space, something which can't be done in real-time in software.

In summary, this method is not the best solution in cases where optimal performance is needed for animating a small, fixed number of known image frames.

A REAR-VIEW MIRROR EXAMPLE USING `WTTEXTURE_REPLACE`

One common need in simulations is to create a rear-view mirror for a car or truck simulation. You can use a second viewpoint facing to the rear of your primary viewpoint to get a view of the scene behind you. Attaching this second viewpoint to a second WTK window essentially gives you a rear-view mirror, which is displayed in the second window.

You can incorporate this rear-view image back into your forward looking scene. Grab the image from the second window with `WTwindow_getimage` and texture it back into the forward looking scene using the `WTtexture_replace` function. A very simple code fragment is given below.

```
/* Initialize the "mirror" object in forward looking scene */
mirrorobj = WTgeometrynode_load(mirrorparentnode, "mirror.nff", 1.0f);

/* place a dummy texture onto the mirror object, this will be
replaced with the rear view image once we obtain it,
we'll assume there is just a single front facing poly in the geometry */
mirrorpoly = WTgeometry_getpolys(mirrorobj);
WTpoly_settexture(mirrorpoly, "mirrtex", FALSE, FALSE);
```

```
/* Initialize the window from which the rearview image will be taken */
mirrorwin = WTwindow_new(0,0, mwidth, mheight, WTWINDOW_DEFAULT);

/* allocate space for the image */
mirrorimage = (unsigned char *) malloc(mwidth*mheight*4);

/* create backward facing viewpoint from forward looking viewpoint*/
viewbackward = WTviewpoint_copy(viewforward);

/* obviously a true rear view would originate from the mirror object's
location in the scene and the rear looking viewpoint's direction would
be coming from the angle of reflection based on the forward looking viewpoint */

/* rotate rear viewpoint 180 degrees backwards from forward view */
WTviewpoint_rotate(viewbackward,Y,PI,WTFRAME_VPOINT);

/* if there are sensors attached to the forward looking viewpoint,
you should attach them to the rear looking viewpoint here */

/* draw the scene from the rear view viewport */
WTwindow_setviewpoint(mirrorwin, viewbackward);

/* grab the image of the rear view scene */
WTwindow_getimage(mirrorwin,0,0,mwidth,mheight,mirrorimage)

/* put the rear view image back into the forward looking scene */
WTtexture_replace("mirrortex",0,mwidth,mheight,mirrorimage);
```

It is also possible to have a rear-view mirror effect by using multiple viewports in a window instead of using multiple windows. See *Viewports* on page 17-30 for more information about viewports and refer to the *Rv_mirror.c* example program in the examples sub-directory of the WTK distribution for an example of how viewports can be used to achieve a rear-view mirror effect.

Assigning Textures in 3D File Formats

You can implicitly assign a texture to a polygon by applying it in the 3D model file prior to loading the file into WTK. The conventions for such annotation differ for the different file formats read by WTK.

For AutoCAD DXF files, the layer name is overloaded with texture information. Any layer name beginning with the underscore character “_” is taken to be the name of the texture file to be applied to all polygons in that layer. The next character following a leading “_” in the texture name *must* be “V”, “S”, or “T” to signify a plain vanilla, shaded, or transparent texture.

Note: “Shading” does not involve the addition of shadow effects to textures. The term merely refers to the total effect of all of the lights that illuminate a texture.

The third character in the layer name must be another “_”, and the remainder of the string is the name of the file containing the bitmap for the texture. For instance, all polygons on a layer “_T_TREE23” will have the transparent texture found in the file “TREE23” applied when the DXF file containing the layer is loaded into WTK.

For the WTK NFF format, polygons to be textured are specified by the addition of a text string with similar connotation to the AutoCAD layer name just described. Texture names indicate the file containing the bitmap to be used as a texture, and specify whether the texture is to be shaded and/or transparent.

In addition, texture placement with either keywords or uv coordinate values is supported. See Appendix F for complete information about texture specification in the NFF format.

WTK also reads texture information from file formats including 3D Studio, Wavefront, and MultiGen/ModelGen.

See *Texture Naming Conventions* on page 10-8 for usage of texture filename extensions and the Environment Variables Appendix for usage of the *WTIMAGES* environment variable.

Deleting Textures

The following functions delete a texture from a polygon or geometry, regardless of the way in which the texture was applied.

WTPoly_deletetexture

```
void WTPoly_deletetexture(  
    WTPoly *poly);
```

This function removes a texture from a polygon that has previously been textured. If the polygon is not currently textured, this function has no effect. This function also does not have any effect if the corresponding geometry has been optimized using *WTgeometry_prebuild* (see page 6-40),

WTgeometry_deletetexture

```
void WTgeometry_deletetexture(  
    WTgeometry *geom);
```

This function removes all textures from a geometry's surfaces, regardless of the way in which the textures were applied. This function also does not have any effect if the corresponding geometry has been optimized using *WTgeometry_prebuild* (see page 6-40),

Changing Texture Properties

The functions in this section access the shading, transparency, and blending values of a polygon's texture.

WTPoly_settexturestyle

```
FLAG WTPoly_settexturestyle(  
    WTPoly *poly,  
    FLAG shaded,  
    FLAG transparent,
```

```
FLAG blended);
```

This function changes the shading, transparency, and blending values of a texture that has already been applied to a polygon.

The *shaded* flag indicates whether the texture will be shaded, i.e., whether lighting should affect the texture. The *transparent* flag indicates whether black pixels in the texture should be rendered; if black pixels are not rendered, then they are effectively transparent. The *blended* flag indicates whether the polygon's material color should be blended with the texture.

If this function is called for a polygon that does not have a texture applied to it, it returns FALSE and has no effect.

See *Transparent Textures* on page 10-8 for more information about texture transparency.

Wtpoly_gettexturestyle

```
FLAG Wtpoly_gettexturestyle(  
    Wtpoly *poly,  
    FLAG *shaded,  
    FLAG *transparent,  
    FLAG *blended);
```

This function retrieves the shading, transparency, and blending settings of a texture that has been applied to a polygon.

If this function is called for a polygon that does not have a texture applied to it, it returns FALSE and has no effect.

FilteringTextures

Polygons in your simulation appear at different sizes depending on their distance from the viewpoint. Each texture, on the other hand, comes in at a specific size to take advantage of hardware capabilities. Since a large texture carelessly applied to a small polygon can produce unwanted results, WTK automatically processes each texture to match the varying size of the polygon to which it has been applied.

During this processing, called *filtering* or *mipmapping*, the texture is scaled to a size that is appropriate for the polygon's display size.

The two functions listed below let you specify and retrieve the filter values for the texture already applied to a polygon. Use the function `WTtexture_setfilter` to specify the quality of the filtering desired. (Note that higher quality requires more computation and rendering time.)

Setting the Default Texture Filter

The default texture filtering mode for all Unix versions of WTK (and on Integraph computers) is bilinear (`WTFILTER_LINEAR`). For all other Windows 32-bit systems, the default texture filtering mode is point (`WTFILTER_NEAREST`).

Although you can change the texture filtering on a per polygon basis, it is often easier to set the default texture filtering mode at the beginning of your WTK application and never change it.

Passing `NULL` into `WTtexture_setfilter()` as the first argument will set the default texture filter mode. For example:

```
WTtexture_setfilter(NULL, WTFILTER_LINEAR, WTFILTER_LINEAR)
```

sets the default magfilter and minfilter to bilinear. The minfilter and magfilter are discussed in the description of `WTtexture_setfilter` below.

WTtexture_setfilter

```
FLAG WTtexture_setfilter(  
    char *bitmap,  
    int magfilter,  
    int minfilter);
```

This function sets the magnification and minification filters of the texture bitmap, which is specified by the *bitmap* argument. If the specified bitmap is `NULL`, then this function will set the default magnification and minification filters to the values specified in the *magfilter* and *minfilter* arguments, so that all subsequently loaded texture bitmaps will take on these filter values automatically.

The texture magnification filter affects the appearance of textured polygons when the polygon occupies a portion of the screen that is larger than the texture bitmap, while the texture minification filter affects the appearance of textured polygons when the polygon occupies a portion of the screen that is smaller than the texture bitmap.

In your simulation, as a textured polygon moves closer or further away from the viewpoint, the texture filters affect the image quality of the textured polygon. In essence, the texture filters are quality/performance knobs, i.e., you can obtain the best performance if you are unconcerned about the appearance of textured polygons as the polygon moves closer or farther away from the viewpoint. At the other end of the spectrum, you can obtain the highest image quality — at the risk of incurring a significant performance penalty.

Here are the possible choices for the *magfilter* and *minfilter* arguments to this function. The choices are listed in order of increasing image quality (and decreasing performance).

Choices for the *Magfilter* Argument

WTFILTER_NEAREST
WTFILTER_LINEAR

Choices for the *Minfilter* Argument

WTFILTER_NEAREST
WTFILTER_LINEAR
WTFILTER_NEARESTMIPMAP_NEAREST
WTFILTER_LINEARMIPMAP_NEAREST
WTFILTER_NEARESTMIPMAP_LINEAR
WTFILTER_LINEARMIPMAP_LINEAR

The default value for *minfilter* and *magfilter* is *WTFILTER_LINEAR*, except on low end NT and WIN95 systems where the default *minfilter* and *magfilter* is *WTFILTER_NEAREST*. This function will cause the specified texture to be loaded if it is not already loaded.

WTtexture_getfilter

```
FLAG WTtexture_getfilter(  
    char *bitmap,  
    int *magfilter,  
    int *minfilter);
```

This function returns the magnification and minification filter values of the specified texture bitmap. If the specified bitmap is NULL, then this function returns the default magnification and minification filter values. See *WTtexture_setfilter* above, for more information.

Manipulating Textures

Once you apply a texture, you can modify it using the functions in this section.

The first group of functions, *Texture Rotation, Scaling, and Other Operations*, allows you to modify the texture that is applied to a polygon by using calls to translate, rotate, scale, etc. WTK internally modifies the polygon's texture uv values when these functions are called.

The second group of functions, *Manipulating Texture uv Values Directly*, allows you to modify the texture application by accessing the texture uv information directly.

Texture Rotation, Scaling, and Other Operations

Wtpoly_rotatetexture

```
void Wtpoly_rotatetexture(  
    Wtpoly *poly,  
    float angle);
```

This function rotates the texture on a polygon in 2D (in texture space) on the surface of the polygon to which the texture is applied. The *angle* parameter specifies the amount of relative texture rotation in radians, around the “center of gravity” (arithmetic mean) of the

vertices of the polygon. Positive angles are counterclockwise rotations of the texture when the front face of the polygon is viewed.

Wtpoly_scaletexture

```
void Wtpoly_scaletexture(  
    Wtpoly *poly,  
    float factor);
```

This function scales textures that are applied to a polygon. The *factor* argument specifies the scale factor applied homogeneously to the u and v texture coordinates associated with the polygon vertices. If *factor* > 1.0, the u,v coordinates are scaled up, and the texture bitmap is reduced on the surface of the polygon. When *factor* < 1.0, texture coordinates are scaled down, and the texture bitmap becomes larger on the surface of the polygon. Figure 10-4 shows a scaled texture.

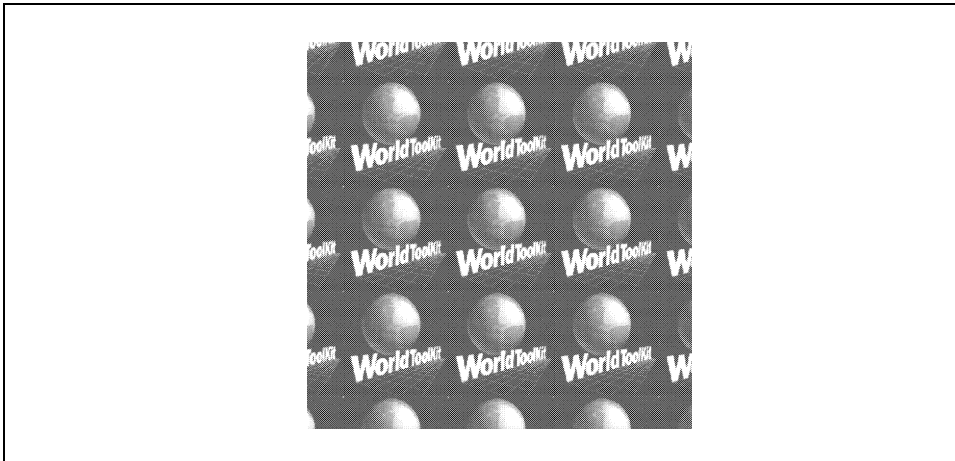


Figure 10-4: A texture after scaling

Wtpoly_translatetexture

```
void Wtpoly_translatetexture(  
    Wtpoly *poly,  
    Wtp2 displacement);
```

This function shifts the origin of the texture bitmap on the polygon surface, to “slide” the texture around. The *displacement* argument is a vector indicating how the applied texture is to be translated in u,v space. Figure 10-5 shows a translated texture.

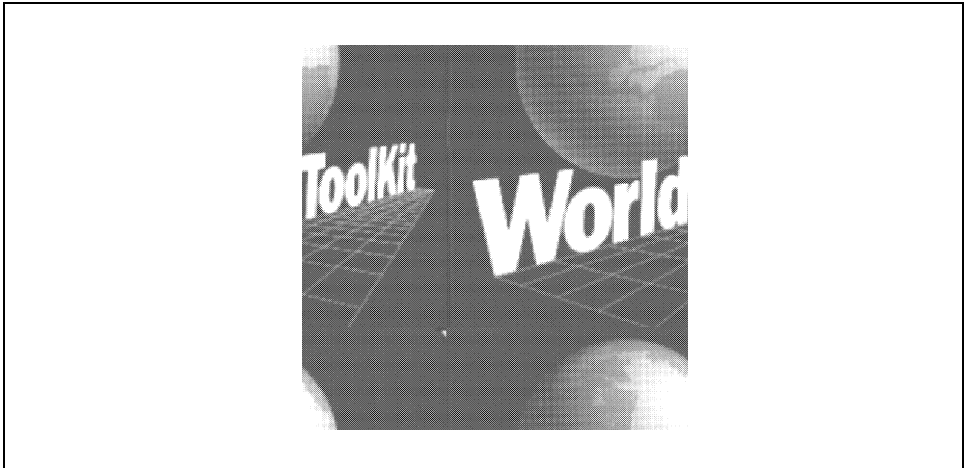


Figure 10-5: A texture after translation.

Wtpoly_mirrortexture

```
void Wtpoly_mirrortexture(  
    Wtpoly *poly);
```

This function “flips” an applied texture in 3D about the v axis of texture space. If you wish to mirror a texture about the u axis, use *Wtpoly_mirrortexture* to mirror it about the v axis, and then rotate the texture through π using *Wtpoly_rotatetexture*. Figure 10-6 shows a “mirrored” texture.

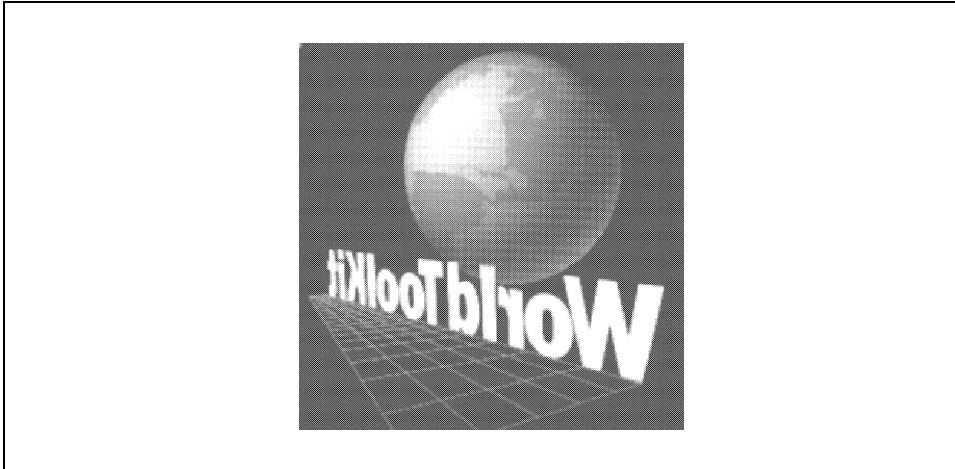


Figure 10-6: A “mirrored” texture.

Wtpoly_stretchtexture

```
void Wtpoly_stretchtexture(  
    Wtpoly *poly,  
    float u,  
    float v);
```

This function stretches a polygon’s texture, with separate scale factors u and v applied to the u and v (horizontal and vertical) texture coordinates associated with the polygon vertices. If $u > 1.0$, then the texture u coordinates (horizontal coordinates in texture space) are scaled up, and the texture bitmap is reduced in the horizontal dimension on the surface of the polygon. When $u < 1.0$, horizontal texture coordinates are scaled down, and the texture bitmap becomes larger in the horizontal dimension on the surface of the polygon. Similarly, values of $v > 1.0$ and $v < 1.0$ scale the texture vertically in texture space so that it appears reduced or enlarged respectively.

If you wish to save out to NFF the precise texture application obtained using *Wtpoly_stretchtexture*, you must write out the file using uv texture coordinates. (See the section *Applying Textures with Explicit uv Values* on page 10-13.) There is no NFF parameter analogous to *rot*, *scale*, *trans*, and *mirror* for texture stretching.

Wtpoly_gettextureinfo

```
FLAG Wtpoly_gettextureinfo(  
    Wtpoly *poly,  
    WTtextureinfo *info);
```

This function retrieves texture information for a specified polygon and places it in the specified *WTtextureinfo* structure. Specifically, it obtains the texture's name (e.g., the filename passed to *Wtpoly_settexture*), and whether it is shaded and/or transparent, the cumulative amounts of rotation, scaling, and translation applied to the texture, and whether the texture is mirrored. If a texture has been mirrored an even number of times (by calls to *Wtpoly_mirrortexture*), it is considered to be not mirrored.

The *info* argument must be a pointer to a declared *WTtextureinfo* structure. The return value TRUE indicates success. If the specified polygon has no texture, then FALSE is returned. The following example demonstrates how to use this function:

```
WTtextureinfo info;  
Wtpoly *poly;  
FLAG success;  
  
WTmessage("poly %p ", poly);  
success = Wtpoly_gettextureinfo(poly, &info);  
if (success) {  
    WTmessage("has texture %s, rotation %f scale %f mirrored %d\n",  
        info.name, info.rotation, info.scale, info.mirrored);  
    WTmessage("translation %f %f\n", info.translation[X], info.translation[Y]);  
    WTmessage("shaded %d transparent %d\n", info.shaded, info.transparent);  
}  
else {  
    WTwarning("has no texture.\n");  
}
```

Manipulating Texture uv Values Directly

Wtpoly_setuv

```
FLAG Wtpoly_setuv(  
    Wtpoly *poly,  
    float *uarray,  
    float *varray);
```

This function changes the way a texture is mapped to a polygon's vertices (on polygons that already have a texture applied). The arrays *uarray* and *varray* are described under *Wtpoly_settextureuv* on page 10-13.

Wtpoly_getuv

```
FLAG Wtpoly_getuv(  
    Wtpoly *poly,  
    float *uarray,  
    float *varray);
```

This function places the uv coordinates of a polygon's texture into the specified arrays. The arrays *uarray* and *varray* must be allocated by the application, and must have at least as many elements as there are vertices in the polygon (which can be obtained using *Wtpoly_numvertices*).

This function returns FALSE if *poly* is NULL or if the polygon does not have a texture, and otherwise returns TRUE.

WTgeometry_setuv

```
FLAG WTgeometry_setuv(  
    WTgeometry *geom,  
    float(*fu)(Wtp3),  
    float(*fv)(Wtp3));
```

This function changes the way textures are mapped to the polygons of a geometry. It does not apply a new texture to the geometry's polygons. Rather, it simply changes the uv values

for the polygons to which textures have already been applied. The arguments *fu* and *fv* are described under *WTgeometry_settextureuv* on page 10-15.

Screen Loading

WTscreen_load

```
FLAG WTscreen_load(  
    char *filename);
```

This function loads an image file to each WTK window. The display occurs immediately (i.e., doesn't wait for the rendering loop). This function returns zero if successful, or a non-zero value if it's not successful.

WTwindow_getimage

```
FLAG WTwindow_getimage(  
    WTwindow *window,  
    int x,  
    int y,  
    int width,  
    int height,  
    unsigned char *image);
```

This function gets an image from the specified window. The *image* parameter returns a pointer to the window image. The *image* data is in a format that can be used by the *WTtexture_replace* function. The *x* and *y* values specify where to start retrieving the image in the window. The (0,0) coordinates specify the lower left corner of the window.

The *width* argument specifies how many pixels per scan line to retrieve and the *height* argument specifies how many scan lines to retrieve. If the *x*, *y* coordinate is outside the window, the function returns FALSE. If either (*x + width*) or (*y + height*) are outside the window, then the function returns FALSE. The *image* argument must be allocated before this function is called and must have a size greater than or equal to *four times* the width value times the height value. For an example of how to use this function, see *A Rear-view Mirror example using WTtexture_replace* on page 10-20.

Introduction

Usually, you use the user-defined universe action function to describe the overall activity of your WTK application. However, you can also use *tasks* to assign behaviors to individual objects. You can specify the behavior of any WTK data structure (or, in fact, any C structure) by assigning tasks to it.

Here are a few examples of the kinds of behavior you can specify:

- Movement
- Change in appearance
- Testing for intersections
- Triggering other behavior
- Attaching a sensor

A WTK “task object” (a *WTtask*) contains a user-defined task function, a pointer to the structure or WTK object with which the task is associated, and a priority value that specifies the order in which the task is executed relative to other tasks as the simulation runs.

You can add, remove, and delete tasks from a simulation. This chapter lists the WTK task functions.

Creation and Deletion Functions

WTtask_new

```
WTtask *WTtask_new(  
    void *objptr,  
    WTtask_function fptr,  
    float priority);
```

This function creates a new *WTtask* and activates it, so that it is automatically executed as the simulation runs. Tasks created by this function are executed in the simulation loop as shown in figure 11-1.

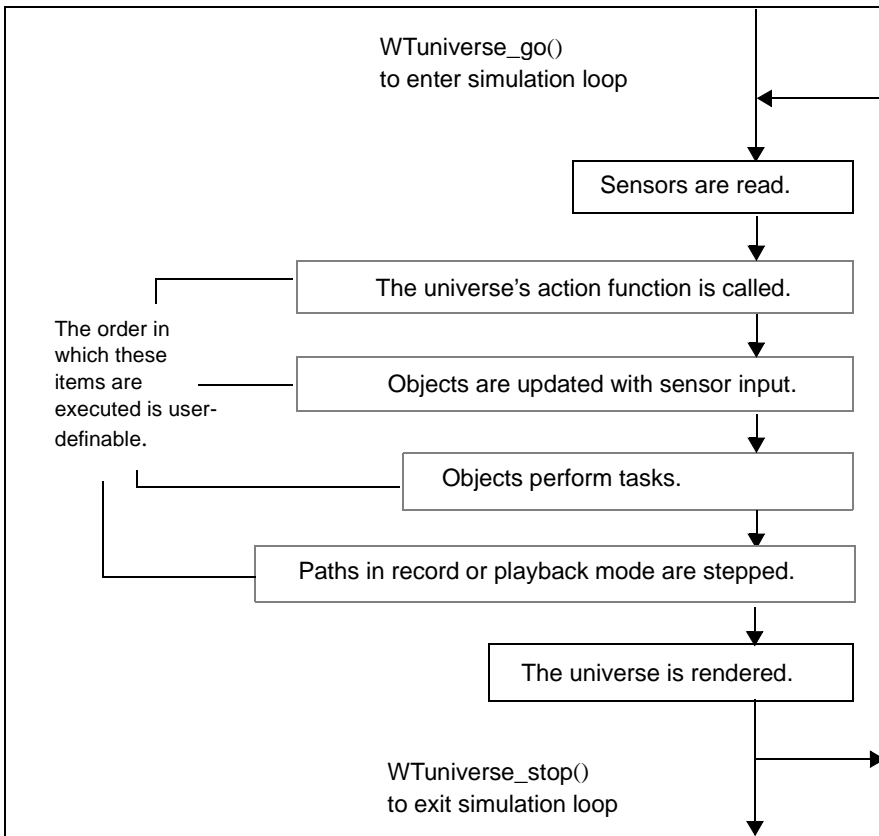


Figure 11-1: The default simulation loop

The argument *objptr* is a pointer to the WTK object or C structure with which the task is associated. The same object can perform more than one task. This can be achieved by calling *WTtask_new* for each task that you wish to associate with *objptr*.

The argument *fptr* is the task function (*WTtask_function*) that is executed as the WTK simulation runs. A *WTtask_function* is typedef'd as a function taking a void* argument and returning a void. You must define the function *fptr* within the WTK application. As the WTK simulation runs, WTK passes in the specified void* pointer *objptr* to the specified *WTtask_function fptr*.

The priority argument specifies the order in which tasks are executed within the tasks slot in the simulation loop shown in figure 11-1. Lower-numbered tasks are executed before higher-numbered ones.

If you call the “delete” function for any of these WTK object types: *WTnode*, *WTpath*, *WTsensor*, *WTviewpoint*, or *WTwindow*, the task for this WTK object is automatically deleted.

For example, to add a task to a light, your application would include code similar to the following:

```
WTnode *light;
WTtask_new(light, light_task,2.5f);
```

where *light_task* is defined as follows:

```
void light_task(WTnode *light) {
    /* code that changes the light */
}
```

In the following example, a C structure is assigned a task that operates on itself:

```
typedef struct mydata {
    /* data declarations */
} mydata;

void mytask(mydata *myptr) {
    /* do something to myptr */
}
```

```
/* in your main program: */  
mydata *myptr;  
WTtask_new(myptr, mytask, 1.0f);
```

See *How Do I Associate A Task With a Particular Object?* on page A-21 for an example of how to associate a task with a particular object.

WTtask_remove

```
FLAG WTtask_remove(  
    WTtask *task);
```

This function removes a task from the simulation (deactivates it) without deleting the *WTtask*. A task which has been deactivated is no longer executed as the simulation runs. A task that has been deactivated can be reactivated by calling *WTtask_add* (see below).

If this function is called from a task function, it affects the current frame, provided the task for which the function is called has not already been executed that frame.

WTtask_add

```
FLAG WTtask_add(  
    WTtask *task);
```

This function adds a task back to the simulation (activates it).

If this function is called from a task function, it affects the current frame, provided the task for which the function is called has not already been executed that frame.

WTtask_delete

```
FLAG WTtask_delete(  
    WTtask *task);
```

This function deletes a task (destroys it). Deleting a task both removes it from the simulation so that it is no longer executed as the simulation runs, and also frees the memory associated with the *WTtask* object passed in. The task pointer passed in is invalid after this function is called.

If this function is called from a task function, it affects the current frame, provided the task for which the function is called has not already been executed that frame.

Other WTask Functions

WTask_setpriority

```
FLAG WTask_setpriority(  
    WTask *task,  
    float priority);
```

This function sets the priority of a task. Tasks with lower-number priority values are executed before tasks with higher values.

If this function is called from a task function, so that the priority of a task (possibly including itself) is changed during execution, the global effect will not take place until the next frame. However, calling *WTask_add*, *WTask_remove*, or *WTask_delete* does affect the current frame, (if the task for which the function is called has not already been executed that frame).

WTask_getpriority

```
float WTask_getpriority(  
    WTask *task);
```

This function returns the priority of a task. The task priority is the value set either when *WTask_new* is called, or by a call to *WTask_setpriority*.

WTask_getfunction

```
WTask_function WTask_getfunction(  
    WTask *task);
```

This function returns a task's function.

WTuniverse_gettaskbypointer

```
WTtask *WTuniverse_gettaskbypointer(  
    void *pointer,  
    int numtask);
```

This function obtains the *WTtask* associated with an object pointer. The argument *numtask* is the number of the task associated with this particular object pointer.

For example, to get the first task assigned to the specified object pointer with *WTtask_new*, pass in 0 (zero) for *numtask*. Pass in 1 (one) for *numtask* to get the second task assigned to the specified object pointer, etc. If three tasks were originally assigned to an object pointer, but the second task was deleted with *WTtask_delete*, then to get the third task assigned to this object pointer, pass in 1 for *numtask*, because the original third task is now the object's second task.

Introduction

Lights include the lights that may be part of a file you load into WTK and the lights you dynamically create in WTK. You can use lights to illuminate some or all of the geometries in a scene. WTK supports several types of lighting: ambient, directed, point, and spot. Each type of light illuminates geometries in a different way. This chapter describes the WTK light nodes and lists their functions.

Light Nodes

A light node is a scene graph node that you use to specify a WTK light (ambient, point, directional, or spot). WTK supports the following four types light nodes:

Ambient light node	A scene graph node that you use to store ambient light. Ambient light is background light that illuminates all surfaces equally regardless of their position or orientation.
Directed light node	A scene graph node that you use to store directed light. Directed light is a light source that has direction but no (finite) position. You can use directed light to emulate the effects of sunlight. Directed light provides illumination as a function of the angle between the light direction and the polygon normal, or, in the case of Gouraud shading, between the light direction and the vertex normals.

Point light node	A scene graph node that you use to store point light. Point light is an omni-directional source of lighting that you can position. It emanates radially from the light position, and may attenuate (drop-off) with distance. Point light provides illumination as a function of the angle between the vector from the light position and the polygon normal, or, in the case of Gouraud shading, between the light direction and the vertex normals.
Spot light node	A scene graph node that you use to store spot light. Spot light is light that illuminates a small area, within a cone of specified angle (e.g., an automobile headlight). Spot light intensity may fall off toward the edge of the light cone (controlled by the exponent value), and attenuate with distance. Spot light provides illumination as a function of the angle between the vector from the light position and the polygon normal, or, in the case of Gouraud shading, between the light direction and the vertex normals.

Light Node Attributes

Other than ambient light nodes, all other light nodes exhibit three types of color: ambient, diffuse, and specular. After you have created a light node, you can set these color attributes for it or accept the defaults. The easiest way of setting the light's color is to specify a diffuse color value, leaving the other color attributes (the ambient and specular components) in the light set to 0 (zero).

There are different attributes available for different types of lights. However, all of these attributes aren't applicable to all light nodes.

This is the full set of attributes available for modifying light nodes:

<i>Position</i>	The location of the light in 3D space, as affected by any existing transformation.
<i>Direction</i>	The direction of the light rays, as affected by any existing transformation.
<i>Intensity</i>	The brightness of the light, with a maximum value of 1.0. See page 12-3 for more information.

<i>Ambient color</i>	The color of the portion of the light that illuminates all surfaces equally regardless of their position or orientation.
<i>Diffuse color</i>	The color of the portion of the light which illuminates polygons as a function of the angle between the light direction and the polygon (or vertex) normal.
<i>Specular color</i>	The color of the portion of light that affects highlights that are reflected off a shiny surface.
<i>Attenuation</i>	The degree to which a point or spot light's intensity decreases with increasing distance from the position of the light.
<i>Angle</i>	The half-angle of the spot light cone. This attribute is used only with spot lights.
<i>Exponent</i>	Specifies how the intensity of a spot light falls off from the center to the edge of the spot light cone. This attribute is used only with spot lights.

Calculating Color

Both a light and the material it illuminates have ambient, diffuse, and specular color values. The precise method of calculating the final perceived material color is explained in the Open GL Specification. Briefly, however, the ambient values for both the light and the material are multiplied together to produce a *term*; similar calculations are also performed to produce terms for diffuse and specular colors. These terms are then added together to achieve the perceived color.

Determining Intensity

The intensity of the color of a polygon is determined by adding the contributions from each of the light sources in the universe. If the result is 0.0, then the polygon will be black, and if the result is 1.0, then the polygon will be of maximum brightness. At maximum brightness, an untextured polygon is rendered with the color assigned to it. At less than maximum brightness, the polygon is rendered with a darker shade of that color. Anything greater than 1.0 is also considered to be maximum brightness. Geometries are dynamically lit, so that shading on a geometry's surfaces is automatically recomputed each frame.

Creating Shadows

Polygons do not cast shadows. Therefore, lighting on a polygon is not affected by polygons which might happen to be between it and a light source. However, the effects of shadowing for a model can be precomputed with what is known as “radiosity preprocessing.” This turns the model surfaces into a mesh and stores shadowing and other lighting information as vertex colors in the new model. See the section *Vertex Colors and Radiosity* on page 6-9 in the *Geometries* chapter.

Using Light Files

WTK supports a keyword-driven light file format. Sample light files containing directed, spot, and point lights are provided in the WTK distribution in the directory called *lightfiles*. You can save your simulation’s current lighting to a file using *WTlightnode_save*. Lights are read in from file using *WTlightnode_load*.

Performance

The maximum number of (non-ambient) lights that can exist in the simulation is eight. However, the greater the number of lights, the greater the performance impact of lighting computations. The time to compute the total effect of all of the lights playing on a geometry’s surfaces is proportional to the number of lights in the simulation. For this reason, if at any time you wish to turn a light off (that is, disable it), it’s better to do so with a call to *WTnode_enable* (with the enable flag set to FALSE), than to set the light’s intensity to 0.0 using *WTlightnode_setintensity*. With *WTnode_enable*, the light is disabled from the simulation and no longer enters into shading computations. With *WTlightnode_setintensity*, however, the light remains part of the simulation and therefore impacts the performance of the simulation. You can also remove a light node from a simulation by detaching the node from the scene graph.

In general, spot lights have the greatest impact on performance, followed by point lights, then directed lights. In addition, attenuated lights have a greater impact on performance than non-attenuated lights (see *WTlightnode_setattenuation* on page 12-17).

Your simulation may contain an unlimited number of ambient light nodes. Unlike spot, point, and directed lights, ambient lights do not significantly increase lighting computations, and hence do not have a significant impact on performance. By default, a

simulation always contains a white ambient light whose intensity is 0.4. Although this default ambient light is inaccessible to you, its effect can be neutralized by adding an ambient light whose intensity is 0.0, so that your simulation effectively has no ambient light.

See also *Vertex Colors and Radiosity* on page 6-9 in the *Geometries* chapter, and the functions *WTpoly_settexture* on page 10-13 and *WTpoly_settexturestyle* on page 10-11 for information about applying shaded textures.

Constructing Light Nodes

WTlightnode_newambient

```
WTnode *WTlightnode_newambient (  
    WTnode *parent);
```

This function creates an ambient light node, and adds it to the scene graph after the last child of the specified parent node. If NULL is specified for the parent argument, then the node is created without a parent. Such nodes can be added to the scene graph by calling *WTnode_addchild* or *WTnode_insertchild*. The default ambient light color is white.

Ambient light illuminates the surfaces of graphical objects regardless of their position or orientation. The intensity and color of the ambient light can be set and retrieved with these functions: *WTlightnode_setintensity*, *WTlightnode_getintensity*, *WTlightnode_setambient*, and *WTlightnode_getambient*.

You can also set ambient light intensity and color using the WTK resource facility — see *Resource Files* on page 2-28. The intensity and color of the ambient light can also be specified in a light file (see *WTlightnode_load* on page 12-9).

The light node you create will have the default values listed below for the red, green, and blue components of its color. This chapter lists several functions that you can use to change an ambient light's properties.

Default values:

Ambient color	1.0, 1.0, 1.0 (white)
---------------	-----------------------

Intensity 0.4

When there are multiple ambient light nodes in a scene graph, successive ambient light nodes in the scene graph traversal replace the previous ambient light node, i.e. the effects of ambient light nodes are not cumulative. By default, a simulation always contains a white ambient light whose intensity is 0.4. Although this default ambient light is inaccessible to you, its effect can be cancelled by adding an ambient light node to the scene graph. By adding an ambient light node whose intensity is 0.0, you can force your scene to have absolutely no ambient light.

WTlightnode_newdirected

```
WTnode*WTlightnode_newdirected (  
    WTnode *parent);
```

This function creates a directed light node, and adds it to the scene graph after the last child of the specified parent node. If NULL is specified for the parent argument, then the node is created without a parent. Such nodes can be added to the scene graph by calling *WTnode_addchild* or *WTnode_insertchild*. Directed lighting can be thought of as parallel rays emanating from a light source at infinity. Once created, a directed light's properties can be modified using the functions described in *Light Properties* on page 12-12.

One of the properties of a light node is color; ambient, diffuse and specular colors combine to create the light produced by a light node. Each of these colors has the default red, green, and blue components listed below. (Values are listed in the order red, green, blue.)

Ambient color	0.0, 0.0, 0.0
Diffuse color	1.0, 1.0, 1.0
Specular color	1.0, 1.0, 1.0

These are the defaults for the other properties of a directed light node:

Intensity	1.0
Direction	0.0, 0.0, 1.0

WTlightnode_newpoint

```
WTnode *WTlightnode_newpoint (  
    WTnode *parent);
```

This function creates a point light node, and adds it to the scene graph after the last child of the specified parent node. If NULL is specified for the parent argument, then the node is created without a parent. Such nodes can be added to the scene graph by calling *WTnode_addchild* or *WTnode_insertchild*. In contrast to a directed light node, whose rays all point in a single direction, the rays of light from a point light are directed radially outward from the light point source.

By default, a point light is a white light which does not attenuate with distance from the light position. Once created, a point light's properties can be modified using the functions described in *Light Properties* on page 12-12.

See *WTlightnode_load* on page 12-9 for information about creating point lights from a file.

One of the properties of a light node is color; ambient, diffuse and specular colors combine to create the light produced by a light node. Each of these colors has the default red, green, and blue components listed below. (Values are listed in the order red, green, blue.)

Ambient	0.0, 0.0, 0.0
Diffuse	1.0, 1.0, 1.0
Specular	1.0, 1.0, 1.0

These are the defaults for the other properties of a point light node:

Intensity	1.0
Position	0.0, 0.0, 0.0
Attenuation	1.0, 0.0, 0.0

WTlightnode_newspot

```
WTnode *WTlightnode_newspot (  
    WTnode *parent);
```

This function creates a spot light node, and adds it to the scene graph after the last child of the specified parent node. If NULL is specified for the parent argument, then the node is created without a parent. Such nodes can be added to the scene graph by calling *WTnode_addchild* or *WTnode_insertchild*.

A spot light node allows you to provide spot light that emanates radially from the spot light source, within a cone of specified angle centered about the spot light direction. Spot lights have an *exponent* value, which specifies how the intensity of the spot light falls off toward the edge of the spot light cone. This value must be between 1.0 and 128.0, or it may be equal to 0.0. The default exponent value is 0.0. If *exponent* is equal to 0.0, then the light does not fall off at all from the center to the edge of the spotlight cone. Increasing values of *exponent* represent sharper fall off toward the edge of the light cone.

A lights's exponent value should not be confused with its attenuation, which determines how light intensity falls off away from the *position* of the light. See figure 12-1.

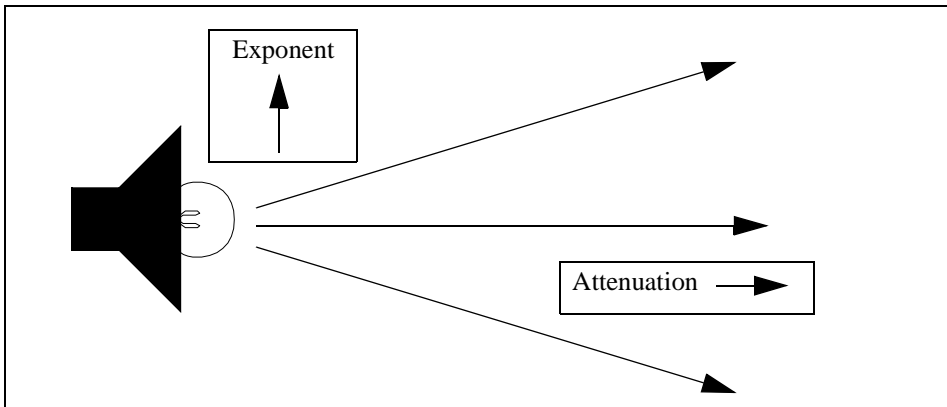


Figure 12-1: A spot light's exponent and attenuation values

By default, a spot light node is a white light that *does not* attenuate with distance from the light position. Once created, a spot light's properties can be modified using the functions in *Light Properties* on page 12-12. Also see *WTlightnode_setattenuation* on page 12-17.

Spot lights have an *angle* value which is the half-angle of the spotlight cone, specified in radians, ranging between 0.0 and $\text{PI}/2$ (90 degrees).

One of the properties of a light node is color; ambient, diffuse and specular colors combine to create the light produced by a light node. Each of these colors has the default red, green, and blue components listed below. (Values are listed in the order red, green, blue.)

Ambient	0.0, 0.0, 0.0
Diffuse	1.0, 1.0, 1.0
Specular	1.0, 1.0, 1.0

These are the defaults for the other properties of a spot light node:

Intensity	1.0
Direction	0.0, 0.0, 1.0
Position	0.0, 0.0, 0.0
Exponent	0.0
Angle	$\text{pi}/8.0$ radians = 22.5 degrees
Attenuation	1.0, 0.0, 0.0

WTlightnode_load

```
FLAG WTlightnode_load(  
    WTnode *parent,  
    char *filename);
```

This function reads a WTK light format file and creates spot, point, directed or ambient light nodes as indicated in the file. It attaches them as children to the specified parent node.

This function can read existing light files from this release of WTK, Release 6, and WTK V2.1. The light format has also been extended to be keyword driven, tolerant of white space, and to support comments and multi-line light specifications.

Directed light parameters are specified as follows:

```
d[irected] [dir <X> <Y> <Z>] [int <V>]  
[amb <R> <G> <B>] [diff <R> <G> <B>] [spec <R> <G> <B>]
```

Point light parameters are specified as follows:

```
p[oint] [pos <X> <Y> <Z>] [ int <V>] [att <X> <Y> <Z>]
    [amb <R> <G> <B>] [diff <R> <G> <B>] [spec <R> <G> <B>]
```

Spot light parameters are specified as follows:

```
s[pot] [pos <X> <Y> <Z>] [dir <X> <Y> <Z>] [int <V>] [angle[rad] <V>] [exp <V>]
    [att <a0> <a1> <a2>] [amb <R> <G> <B>] [diff <R> <G> <B>] [spec <R> <G> <B>]
```

Ambient light parameters are specified as follows:

```
a[mbient] [int <V>] [amb <R> <G> <B>]
```

The square brackets [] represent optional parameters or text. If you leave off any optional parameters, default values are used. Extra white space is ignored. To allow for comments, any text on a line following “//” characters is ignored. Each light does not have to have all its options on the same line. All parameters except the first, which specifies the light type, can be given in any order.

The values in angle brackets <> can be any floating point value, within the following ranges:

- The *int* parameter is the light intensity; it may range from 0.0 to 1.0. This parameter is part of the light specification for all light node types.
- The *angle* parameter is used only for spot light nodes. It is an angular radius in degrees, so it may range from 0.0 to 90.0. If the keyword “anglerad” is used rather than “angle”, the value is taken to be in radians, and so should be between 0.0 and $\text{PI}/2$, or 1.57.
- The *exp* parameter is an exponential factor which must be between 1.0 and 128.0, or be equal to 0.0. This parameter is used only for spot light nodes. See the function *WTlightnode_setexponent* on page 12-19.
- The *att* parameter defines attenuation values. See the functions *WTlightnode_newspot* on page 12-8 and *WTlightnode_newpoint* on page 12-7 for definitions of appropriate values. The default attenuation values are 1.0 0.0 0.0 (representing no attenuation). The attenuation parameters can only be specified for point and spot light nodes; directed and ambient light nodes may not be attenuated.

- The *dir* parameter defines the direction of the light node. The *dir* direction vectors are normalized if they are of non-zero magnitude. This parameter is used only for directed and spot light nodes, not for point and ambient light nodes.
- The *pos* parameter defines the position of the light node. This parameter is used only for point and spot light nodes, not for direct and ambient light nodes.
- Color can be specified by the parameters *amb*, *diff*, and *spec*, which represent ambient, diffuse, and specular colors. Directed, point, and spot light nodes support all the three colors, whereas ambient light nodes support only ambient light color. The <R>, <G>, and for each of the colors represents the red, green, and blue colors. Their values vary from 0.0 to 1.0.

Here's an example file:

```
d dir 0 1 0 int 1.0 amb 0.0 0.0 0.0 diff 1.0 1.0 0.0 spec 1.0 1.0 1.0 // Light 1
p pos -30 0 0 int 0.3 amb 0.0 0.0 0.0 diff 0.0 0.0 1.0 spec 1.0 1.0 1.0 // Light 2
p pos 30 0 0 int 0.3 amb 0.0 0.0 0.0 diff 0.0 1.0 0.0 spec 1.0 1.0 1.0 // Light 3
s pos -20 20 -20 dir 1 1 1 int 0.8 angle 20 exp 1.0
  amb 0.0 0.0 0.0 diff 1.0 1.0 1.0 spec 1.0 1.0 1.0 // Light 4
a int 0.4 amb 1.0 1.0 1.0 // Light 5
```

This file contains five lights. Note that the fourth light has some of its options on a second line. Light one is a yellow directional light pointing straight down with full intensity. Light two and three are dim blue and green point lights to the left and right of the origin. Light four is a spotlight pointing diagonally down toward the origin, with a cone radius of 20 degrees. Light five is a dim white ambient light.

WTlightnode_save

```
FLAG WTlightnode_save(
    WTnode *light,
    char *filename);
```

This function saves out an ambient, directed, point or spot light node to a file with the specified name. The file created has the format described above under *WTlightnode_load*.

Light Properties

You can access and change the properties (e.g., position, direction, color, and intensity) of WTK ambient, directed, point, and spot lights with the functions in this section. When you make changes to lights, the shading on graphical entities is automatically updated.

If you're not concerned about the ambient and specular color components of a particular light, the easiest way of setting the light's color is to specify a diffuse color value, leaving the other color components of the light set to their default values.

Note that a light illuminates a surface only if the light and surface have color components in common. For example, while a white light (which contains all color components) illuminates a surface of any color, and a red light illuminates any surface containing red in it, a red light does not add to the illumination of a surface which is purely blue or green.

WTlightnode_setposition

```
FLAG WTlightnode_setposition(  
    WTnode *light,  
    WTp3 p);
```

This function sets the 3D position of a light to the point passed in (point and spot lights only). The default light position is (0,0,0).

Point and spot lights are affected by light position, as they emanate radially from their source. Directed and ambient lights are unaffected by light position.

WTlightnode_getposition

```
FLAG WTlightnode_getposition(  
    WTnode *light,  
    WTp3 p);
```

This function gets the position of the light node and stores it in the *p* parameter. Use *WTlightnode_setposition* to change a light's position.

WTlightnode_setdirection

```
void WTlightnode_setdirection(  
    WTnode *light,  
    WTP3 dir);
```

This function sets the direction of a light to the vector passed in (spot and directed lights only). Spot and directed lights are affected by light direction as these light types emanate light in a particular direction. Point and ambient lights are unaffected by light direction.

The direction vector passed in (*dir*) does not have to be normalized, that is, it is not required to have a length equal to one. (This function automatically normalizes the direction vector for you.) However, if the three components of the direction vector are all 0 (zero), the function returns without setting the light's direction.

In the following example, a light's direction is set to lie in the X-Z plane, with a specified angle *theta* from the X axis.

```
void orient_light(WTnode *light, double theta)  
{  
    WTP3 dir;  
    dir[Y] = 0.0;  
    dir[X] = cos(theta);  
    dir[Z] = sin(theta);  
    WTlightnode_setdirection(light, dir);  
}
```

WTlightnode_getdirection

```
void WTlightnode_getdirection(  
    WTnode *light,  
    WTP3 dir);
```

This function gets the direction of a light and stores it in the *dir* parameter. This direction vector is normalized to have length equal to 1.0.

WTlightnode_setintensity

```
void WTlightnode_setintensity(  
    WTnode *light,  
    float x);
```

This function sets the intensity of a light to *x*, which should be between 0.0 and 1.0. Since the computer can display colors only between a minimum and maximum intensity (which correspond to 0.0 and 1.0), any intensity values that are requested below or above this range are set to 0.0 and 1.0 respectively.

Examples of using *WTlightnode_setintensity* are given below, under *WTlightnode_getintensity*.

WTlightnode_getintensity

```
float WTlightnode_getintensity(  
    WTnode *light);
```

This function returns a light's intensity value.

The following example uses the functions *WTlightnode_getintensity* and *WTlightnode_setintensity* to increase the intensity of a light by five percent.

```
WTnode *light;  
WTlightnode_setintensity(light, 1.05 * WTlight_getintensity(light));
```

WTlightnode_setambient

```
void WTlightnode_setambient(  
    WTnode *light,  
    float r,  
    float g,  
    float b);
```

This function sets the ambient component of a light's color. The *r,g,b* values must be between 0.0 and 1.0. By default, *r*, *g*, and *b* are each equal to 1.0 for ambient lights, while *r*, *g*, and *b* are each equal to 0.0 for directed, point, and spot lights.

WTlightnode_setdiffuse

```
void WTlightnode_setdiffuse(  
    WTnode *light,  
    float r,  
    float g,  
    float b);
```

This function sets the diffuse component of a light's color (directed, point, and spot lights only). The *r,g,b* values must be between 0.0 and 1.0. By default, *r,g,b* are each equal to 1.0, giving white light.

WTlightnode_setspecular

```
void WTlightnode_setspecular(  
    WTnode *light,  
    float r,  
    float g,  
    float b);
```

This function sets the specular component of a light's color (directed, point, and spot lights only). The *r,g,b* values must be between 0.0 and 1.0. By default, *r,g,b* are each equal to 1.0, giving white light.

WTlightnode_getambient

```
void WTlightnode_getambient(  
    WTnode *light,  
    float *r,  
    float *g,  
    float *b);
```

This function gets the ambient red, green, and blue components of a light's color. Each component is in the range 0.0 and 1.0. When all three components equal 1.0, the light is white.

Example of usage:

```
WTnode *light;
float r, g, b;
WTlightnode_getambient(light, &r, &g, &b);
WTmessage("light ambient color components are: red: %f green: %f blue: %f\n",r,g,b);
```

WTlightnode_getdiffuse

```
void WTlightnode_getdiffuse(
    WTnode *light,
    float *r,
    float *g,
    float *b);
```

This function gets the diffuse red, green, and blue components of a light's color. Each component is in the range 0.0 and 1.0. When all three components equal 1.0, the light is white.

Example of usage:

```
WTnode *light;
float r, g, b;
WTlightnode_getdiffuse(light, &r, &g, &b);
WTmessage("light diffuse color components are: red: %f green: %f blue: %f\n",r,g,b);
```

WTlightnode_getspecular

```
void WTlightnode_getspecular(
    WTnode *light,
    float *r,
    float *g,
    float *b);
```

This function obtains the specular red, green, and blue components of a light's color. Each component is in the range 0.0 and 1.0. When all three components equal 1.0, the light is white.

WTlightnode_setattenuation

```
void WTlightnode_setattenuation(  
    WTnode *light,  
    float atten0,  
    float atten1,  
    float atten2);
```

This function sets the attenuation value for point and spot lights (the rate at which the light falls off as the distance from the light increases). By default, point and spot lights are not attenuated: the default attenuation coefficients are *atten0*=1.0, *atten1*=0.0, *atten2*=0.0.

The light passed in to this function must be either a point light or a spot light. Ambient lights and directed lights (which are located at infinity) can not be attenuated.

This function sets the coefficients of the light attenuation factor, which is computed at each vertex as:

$$1.0 / (\text{atten0} + \text{atten1} * \text{dist} + \text{atten2} * \text{dist} * \text{dist})$$

where *dist* is the distance between a light source and the vertex it is illuminating. The value of the floating point number *atten0* must be greater than *WTFUZZ*, while both *atten1* and *atten2* must be greater than or equal to 0.0.

The value of *atten0* must be greater than *WTFUZZ*. If it is not, then this function returns with no effect. Passing in non-zero values for *atten1* or *atten2* (especially *atten2*) can cause a very rapid decrease of light intensity with increasing distance from the light. This decrease in intensity may be much greater than you might expect. If you call this function, and it seems that your light is no longer illuminating the scene, try decreasing the values of *atten1* and *atten2*.

Also note that using attenuated lights (that is, having non-zero values for *atten1* or *atten2*) may impact performance, with a non-zero value for *atten2* having a greater impact than a non-zero value for *atten1*.

In the following example, light attenuation is set to fall off as $1.0/(1.0 + dist)$:

```
WTnode *light;
WTlight_setattenuation(light, 1.0, 1.0, 0.0);
```

See also *WTlightnode_newpoint* on page 12-7, *WTlightnode_newspot* on page 12-8, and *WTlightnode_gettype* on page 12-18.

WTlightnode_getattenuation

```
void WTlightnode_getattenuation(
    WTnode *light,
    float *atten0,
    float *atten1,
    float *atten2);
```

This function obtains a point or spot light's attenuation coefficients. For example:

```
WTnode *light;
float atten0, atten1, atten2;
WTlightnode_getattenuation(light, &atten0, &atten1, &atten2);
```

WTlightnode_gettype

```
int WTlightnode_gettype(
    WTnode *light);
```

This function determines which light constructor function was used to create the light (i.e., tells you what type of light it is).

A light's type is determined by which light constructor function was used to construct the light: *WTlightnode_newdirected*, *WTlightnode_newspot*, *WTlightnode_newpoint*, or *WTlightnode_newambient*, or by the type specified in the light file if the light was constructed by a call to *WTlightnode_load*.

This function returns one of these values

```
WTLIGHTTYPE_AMBIENT  
WTLIGHTTYPE_DIRECTED  
WTLIGHTTYPE_POINT  
WTLIGHTTYPE_SPOT
```

WTlightnode_setangle

```
void WTlightnode_setangle(  
    WTnode *light,  
    float angle);
```

This function controls the size of the spot light's cone. The angle passed in to this function is specified in radians and must be between 0.0 and $\text{PI}/2$. This radians value represents the half-angle of the spot light cone. The default value of the half-angle of a spot light is $\text{PI}/8$ radians (22.5 degrees).

This function is used only for spot lights, and returns with no effect if another type of light is passed in.

WTlightnode_getangle

```
float WTlightnode_getangle(  
    WTnode *light);
```

This function returns the half-angle of a spot light's cone. If a light that is not a spot light is passed in (see *WTlightnode_gettype* on page 12-18), then -1.0 is returned.

WTlightnode_setexponent

```
void WTlightnode_setexponent(  
    WTnode *light,  
    float val);
```

This function specifies how the intensity of a spot light falls off within the spot light cone. Spot lights have an *exponent* value, which specifies how the intensity of the spot light falls off toward the edge of the spot light cone. This value must be between 1.0 and 128.0, or it

may be equal to 0.0. The default exponent value is 0.0. If *exponent* is equal to 0.0, then the light does not fall off at all from the center to the edge of the spotlight cone. Increasing values of *exponent* represent sharper fall off toward the edge of the light cone.

A light's exponent value should not be confused with its attenuation, which determines how light intensity falls off away from the *position* of the light. See figure 12-1 on page 12-8.

By default, a spot light node is a white light that *does not* attenuate with distance from the light position. See *Wtlightnode_setattenuation* on page 12-17.

Wtlightnode_setexponent is used only for spot lights, and returns with no effect if another type of light is passed in.

Wtlightnode_getexponent

```
float Wtlightnode_getexponent(  
    WtNode *light);
```

This function returns the exponent of a spot light. If a light that is not a spot light is passed in, then -1.0 is returned.

This chapter explains how most sensors work with WTK and shows which sensors are supported (not all sensors work on every platform). Use this chapter to configure sensors with your application.

The sections at the beginning of this chapter apply to all sensors:

- *Introduction to the Sensor Class* – provides a general introduction to sensors and lists the sensors that are currently supported by WTK. (see page 13-2)
- *Sensor Lag and Frame-rate* – describes concepts related to sensors that have a direct impact on the effectiveness of your application’s interactivity. (see page 13-5)
- *Sensor Construction and Destruction* – describes how to create and remove a sensor object. (see page 13-5)
- *Accessing Sensor State* – describes how to access a sensor’s information directly, so you can, for example, set the sensitivity for a sensor or retrieve its rotation information. (see page 13-11)
- *Rotating Sensor Input* – describes how to change the reference frame for a sensor. (see page 13-16)
- *Using Different Baud Rates* – describes how to set a specific baud rate for a sensor. (see page 13-22)
- *Sensor Name* – describes how to set and retrieve the name of a sensor. (see page 13-23)
- *User-specifiable Sensor Data* – describes how to store or retrieve data in a sensor object. (see page 13-23)
- *Custom Sensor Drivers* – describes how to create your own sensor driver, should you wish to use a sensor that is not currently supported by WTK. (see page 13-24) See also Appendix E, *Writing a Sensor Driver*.

The remaining sections provide information on specific WTK-supported drivers. The table on page 13-3 lists the WTK-supported sensors, on which platforms they are supported, and where the relevant information is located in this chapter.

Introduction to the Sensor Class

Sensor objects in WTK generate position, orientation, and other kinds of data by reading inputs that originate in the real world. These inputs can be used to control motion and other behavioral aspects of objects in the simulation. Sensors permit the user of a WTK application to be directly coupled to the viewpoints, graphical objects, and lights in the universe.

Many of the 3D and 6D (position/orientation) sensors that are available are supported by WTK. There are two principal classes of such sensors: desk-based sensors and sensors that are worn on the body. While most desk-based sensors generate relative inputs, that is, *changes* in position and orientation, devices worn on the body typically generate absolute records, that is, values that correspond to their specific spatial location.

Desk-based sensors are conventional devices, like the Mouse, Serial Joystick and isometric balls. CIS Geometry Ball, Jr. and Spacotec IMC's Spaceball are isometric balls that respond to forces and torques applied by the user. Using such devices, a 3D object can be directly manipulated, displaced or rotated — the object acts like it is directly connected to the sensor. Ball sensors are also useful for moving the viewpoint; the applied displacements and rotational forces move and rotate the viewpoint. In this mode of operation, with a ball sensor attached to the viewpoint, the ball operates like a “fly-by-wire” helicopter.

Sensors worn on the body (sensors that generate absolute records) include *electromagnetic 6D trackers* such as the Polhemus FASTRAK and Ascension Bird. This type of sensor can be used for viewpoint tracking when it is attached to a head-mounted display. In addition to electromagnetic devices, a variety of *ultrasonic ranging/triangulation* devices and *optical* devices exist for absolute position and orientation tracking. One example is the ultrasonic Logitech 3D Mouse and Head Tracker.

Regardless of their underlying hardware technology, WTK's sensor objects are treated similarly and can be used interchangeably in an application. Once a sensor object is created, it is automatically maintained by the simulation manager, so you do not have to deal directly with considerations such as whether the sensor is returning relative or absolute records, or whether it is polled or streaming its data.

WTK provides drivers for the devices listed below, making them easy to connect to your computer and use in your applications.

WTK Supported Sensor Devices

Sensor Device	Windows	UNIX	See page...
Any Standard Mouse (two or three buttons)	X	X	13-26
Ascension Bird/Motionstar/6DOF Mouse/Flock of Birds	X	X	13-39
Ascension Extended Range Bird	X	X	13-51
CIS Graphics Geometry Ball, Jr.	X	X	13-53
Fakespace monochrome BOOM, two-color BOOM2C, and full-color BOOM3C (button models and joystick models)	X	X	13-55
Fakespace Pinch Glove System	X	X	13-59
Fifth Dimension Technologies' 5DT Glove	X		13-63
Gameport Joystick	X		13-67
Logitech 3D Mouse (Red Baron)	X	X	13-73
Logitech Head Tracker	X	X	13-77
Logitech Space Control Mouse (Magellan)	X	X	13-81
Polhemus ISOTRAK	X	X	13-85
Polhemus ISOTRAK II	X	X	13-88
Polhemus InsideTRAK	X (only NT 3.51)		13-90
Polhemus FASTRAK	X	X	13-92
Polhemus Stylus	X	X	13-93
Precision Navigation Wayfinder-VR	X		13-96

WTK Supported Sensor Devices (continued)

Sensor Device	Windows	UNIX	See page...
Spacotec IMC Spaceball – Model 2003 and Model 3003 (using only the pick button)	X	X	13-100
Spacotec IMC Spaceball SpaceController	X (only NT 3.51)		13-104
StereoGraphics CrystalEyes and CrystalEyesVR LCD Shutter Glasses	X	X	13-108
ThrustMaster Formula T2 Steering Console	X (only NT 3.51)		13-111
ThrustMaster Serial Joystick (Mark II Flight Control/Weapons Control Systems)	X	X	13-113
VictorMaxx Technologies' CyberMaxx2 HMD	X		13-119
Virtual i-O i-glasses! – monoscopic and stereo (Intergraph only) with head tracking	X	X	13-121
Virtual Technologies Cyberglove	X	X	13-123

Consult your Hardware Guide for platform-specific information on supported sensor devices.

Check the README.1ST file, that was installed with WTK, to see if additional device support became available after this book was printed. You can also contact Technical Support for information about currently supported devices. If you have access to the World Wide Web, check the Technical Support web pages, which show what devices are supported and how to set up the devices correctly. See *Technical Support* on page L-1 for more information.

In addition to the devices shown above, WTK provides functions for easily obtaining input from the keyboard. The keyboard device, which is handled differently from the WTK sensor objects, is described in *Reading the Keyboard* on page 24-1.

WTK also has functions for interfacing with devices that are not currently supported. For information on creating your own sensor driver, see *Custom Sensor Drivers* on page 13-24 and the sensor driver specification in *Writing a Sensor Driver* on page E-1.

Sensor Lag and Frame-rate

WTK is designed so you can interact with computer-generated graphics flexibly and in “real-time.” Sensor objects provide a means of accomplishing this by directly coupling the user of an application to the geometry in the virtual world. The effectiveness of this interaction depends on several factors:

Sensor lag	The time from when the sensor’s state in the real world changes to when the sensor generates a record corresponding to that state; inversely proportional to sensor speed.
Sensor accuracy	The range of values that a sensor may return when in a given state. This is usually specified as something like: “+ or - 0.1 inches within a range of 8 feet.”
Frame-rate	The number of frames per second that the system displays.

Note: Even if your application runs with a high frame-rate, if the sensor lag is very large, then the user’s impression of being able to interact in the virtual world may suffer. For very precise manipulations within the virtual world, the shorter the lag time, the better the user control.

Sensor Construction and Destruction

WTsensor objects can be created with either the generic sensor constructor function *WTsensor_new* (see page 13-7) or with one of WTK’s device-specific constructor macros (*WTmouse_new*, *WTspaceball_new*, *WTpolhemus_new*, *WTbird_new*, etc.).

For example, the device-specific constructor function for the Spacetec IMC Spaceball is a macro defined as follows:

```
#define WTspaceball_new(port) \  
    WTsensord_new(WTspaceball_open, WTspaceball_close,\  
        WTspaceball_update, WTserial_new(...), 1, \  
        WTSENSOR_DEFAULT)
```

(The arguments to *WTserial_new* vary according to the operating system.) To use this macro, you would make the call:

```
WTsensor *spaceball;  
spaceball = WTspaceball_new(SERIAL1);
```

where the constant *SERIAL1* is already defined for all systems (this allows for portability).

All of the device-specific constructors are simply macro calls to *WTsensord_new*. Its first three arguments open, close, and update the particular device by using pointers to WTK functions. See the table on page 13-7 for a listing of the open, close, and update function(s) for a sensor object. As you see, a device has only one open and close function but could have multiple update functions.

If the update function specified in the device-specific constructor function macro is appropriate for your application, just use the macro call. If you want to use an update function other than the default, or if you want to create the sensor object at a different baud rate than the default (see *Using Different Baud Rates* on page 13-22), use *WTsensord_new*.

For example, if you wanted to use your own update function *myspaceball_update* for the Spaceball (consult the sensor driver specification in Appendix E to find out what needs to be in such a function), you would create the sensor object with the call:

```
WTsensor *ball;  
ball = WTsensord_new(WTspaceball_open, WTspaceball_close,  
    myspaceball_update, WTserial_new(...), 1,  
    WTSENSOR_DEFAULT);
```

The next part of this chapter describes the functions that apply generally to WTK sensor objects. Following that, information about the specific devices supported in WTK is provided.

WTsensor_new

```

WTsensor *WTsensor_new(
    int (*openfn)(WTsensor*),
    void (*closefn)(WTsensor*),
    void (*updatefn)(WTsensor*),
    WTserial *serial,
    short unit,
    short location);

```

This function creates a new sensor object and adds it to the universe. If it successfully opens the device, it returns a pointer to the sensor object created. If it's unsuccessful, for example if the device is incorrectly cabled and the device cannot be initialized, NULL is returned.

The first three arguments are pointers to functions to initialize, terminate, and update a sensor. When a particular device is “supported in WorldToolKit”, it means these functions are already provided for that device. The names of the open, close, and update functions for devices supported in WTK are as follows (more than one update function is provided for some devices):

Device	Open, Close, and Update functions
Any Standard Mouse	<i>WTmouse_open; WTmouse_close; WTmouse_drawcursor, WTmouse_moveview1, WTmouse_moveview2, WTmouse_move2D</i>
Ascension Bird, Flock of Birds, Motionstar, and 6DOF Mouse	<i>WTbird_open; WTbird_close; WTbird_update</i>
Ascension Extended Range Bird	<i>WTercbird_open; WTercbird_close; WTercbird_update</i>
CIS Graphics Geometry Ball, Jr.	<i>WTgeoball_open; WTgeoball_close; WTgeoball_update</i>
Fakespace BOOM (all display types)	<i>WTboom_open; WTboom_close; WTboom_update</i> (for BOOMs with buttons), <i>WTboom_joystickupdate</i> (for BOOMs with joysticks).
Fakespace Pinch Glove System	<i>Wtpinch_open; Wtpinch_close; Wtpinch_update</i>

Device	Open, Close, and Update functions
Fifth Dimension Technologies' 5DT Glove	<i>WTglove5dt_open; WTglove5dt_close; WTglove5dt_update, WTglove5dt_updatefingers</i>
Gameport Joystick	<i>WTjoystick_open; WTjoystick_close; WTjoystick_walk, WTjoystick_walk2, WTjoystick_fly</i>
Logitech 3D Mouse (Red Baron)	<i>WTbaron_open; WTbaron_close; WTbaron_update</i>
Logitech Head Tracker	<i>WTlogitech_open; WTlogitech_close; WTlogitech_update.</i>
Logitech Space Control Mouse (Magellan)	<i>WTspacecontrol_open; WTspacecontrol_close; WTspacecontrol_update.</i>
Polhemus ISOTRAK	<i>WTpolhemus_open; WTPolhemus_close; WTPolhemus_update.</i>
Polhemus ISOTRAK II	<i>WTisotrak2_open; WTisotrak2_close; WTisotrak2_update.</i>
Polhemus InsideTRAK	<i>WTinsidetraknt_open; WTinsidetraknt_close; WTinsidetraknt_update.</i>
Polhemus FASTRAK	<i>WTfastrak_open; WTfastrak_close; WTfastrak_update</i>
Precision Navigation Wayfinder-VR	<i>WTprecision_open; WTprecision_close; WTprecision_update</i>
Spacotec IMC Spaceball	<i>WTspaceball_open; WTspaceball_close; WTspaceball_update, WTspaceball_dominant</i>
Spacotec IMC Spaceball SpaceController	<i>WTspaceballSC_open; WTspaceballSC_close; WTspaceballSC_update, WTspaceballSC_dominant</i>
StereoGraphics CrystalEyes and CrystalEyesVR LCD Shutter Glasses	<i>WTcrystaleyesVR_open; WTcrystaleyesVR_update; (WTlogitech_close is used to close the device).</i>
ThrustMaster Formula T2 Steering Console	<i>WTformula_open; WTformula_close; WTformula_drive.</i>

Device	Open, Close, and Update functions
ThrustMaster Serial Joystick	<i>WTjoyserial_open</i> ; <i>WTjoyserial_close</i> ; <i>WTjoyserial_walk</i> , <i>WTjoyserial_walk2</i> , <i>WTjoyserial_fly</i>
VictorMaxx Technologies' CyberMaxx2 HMD	<i>WTcybermaxx2_open</i> ; <i>WTcybermaxx2_close</i> ; <i>WTcybermaxx2_update</i>
Virtual i-O i-glasses!	<i>WTiglasses_open</i> ; <i>WTiglasses_close</i> ; <i>WTiglasses_update</i>
Virtual Technologies Cyberglove	Not Applicable.

To use a device that is not yet supported, you must provide the open, close, and update functions (see Appendix E). Once you have specified these three functions by passing them in as arguments to *WTsensor_new*, WTK takes care of calling these functions at the appropriate times.

The *openfn* is called once when the sensor is created. All predefined WTK device drivers return a NULL or zero value if they couldn't open or initiate communications with the device.

The *closefn* is called once when the sensor is deleted by a call to *WTsensor_delete* (which in turn is called by *WTuniverse_delete* for any sensors that still exist). The *updatefn* is called by the WTK simulation manager once at the beginning of each frame.

The *updatefn* is called each time through the simulation loop and determines how the sensor state (e.g., translational information, relational information, button presses, etc.) is to be updated.

The *serial* argument to *WTsensor_new* is a pointer to an initialized serial port object. Typically a serial port object can be constructed by following the examples provided in the sensor macros in the file *sensor.h*. Also consult your Hardware Guide for information about using serial port devices with WTK on your hardware platform (see also *WTserial_new* on page 23-1). If your device is not a serial port device, then *serial* should be NULL.

For multi-unit devices such as FASTRAK and Flock of Birds, the *unit* argument specifies which unit to open. For all other sensors, *unit* should be 1 (one).

The *location* argument should be set to *WTSENSOR_DEFAULT*.

WTsensor_delete

```
void WTsensor_delete(  
    WTsensor *sensor);
```

This function removes a sensor object from the universe's list of sensors; detaches the sensor from viewpoints, lights, or objects; calls the sensor's close function; deletes the sensor's serial port object (if it has one); and frees the memory used by the sensor object.

WTsensor_next

```
WTsensor *WTsensor_next(  
    WTsensor *sensor);
```

This function returns the next sensor object in the list of sensors maintained by the universe. Use *WTuniverse_getsensors* (see page 2-13) to obtain a pointer to the first sensor in the list. An example of using this function is provided under *WTsensor_setsensitivity* on page 13-11).

WTsensor_setupdatefn

```
void WTsensor_setupdatefn(  
    WTsensor *sensor,  
    void (*updatefn)(WTsensor*));
```

This function allows you to change a sensor's update function. A sensor object's update function is initially set in the generic sensor constructor function *WTsensor_new* (see page 13-7), or the device-specific constructor macro (e.g., *WTmouse_new*). *WTsensor_setupdatefn* should be called if you want to change the update function. The following example illustrates how to set a Mouse sensor's update function to the WTK function *WTmouse_move2D*.

```
WTsensor *mouse;  
/*Create a mouse sensor object using the device-specific macro. This uses the  
WTmouse_moveview2 update function */  
mouse = WTmouse_new()  
/*Change the update function */  
WTsensor_setupdatefn(mouse, WTmouse_move2D);
```

This example assumes the Mouse was originally created as a pointer to a sensor object as described in *The Mouse* on page 13-26.

Accessing Sensor State

WTsensor_setsensitivity

```
void WTsensor_setsensitivity(  
    WTsensor *sensor,  
    float sensitivity);
```

This function sets the sensitivity value for the sensor. The default sensitivity value for all sensors is 1.0. Attempts to set a sensor's sensitivity to a negative value are rejected, with no change to the current sensitivity.

A sensor's sensitivity value defines the maximum magnitude of the translational input from the sensor along each axis (in the same distance units as the 3D geometry making up the virtual world).

For example, suppose you have a Spaceball attached to a viewpoint. The Spaceball's sensitivity determines the maximum distance along each axis that your viewpoint moves when you push on the ball. To move faster, call *WTsensor_setsensitivity* with a larger value than is currently set for the device.

It is frequently desirable to have the sensor's sensitivity scale with the size of the scene, or with some other characteristic distance scale in the virtual world. The example below shows how to accomplish this.

```
WTsensor *sensor;  
float radius;  
  
/* Iterate through all of the sensors in the universe,  
scaling sensor sensitivity with the size of the scene */  
radius = WTnode_getradius(WTuniverse_getrootnodes());  
for ( sensor=WTuniverse_getsensors() ; sensor ;  
      sensor=WTsensor_next(sensor) ) {  
    WTsensor_setsensitivity(sensor, 0.01 * radius);  
}
```

```
}
```

In this example, if the sensor is a Spaceball attached to your viewpoint, then each time through the simulation loop your viewpoint moves a distance equal to at most one hundredth of the scene's radius along each of X, Y, and Z axes. If you do not push on the Spaceball very hard, then you would move less than that.

Not all devices supported in WTK have their translational records scaled in this way. The sensor translational records scaled by *WTsensor_setsensitivity* are described in the corresponding sections of this chapter for each device.

WTsensor_getsensitivity

```
float WTsensor_getsensitivity(  
    WTsensor *sensor);
```

This function returns the sensor's sensitivity value. This value is defined above under *WTsensor_setsensitivity*. The following example uses the function *WTsensor_getsensitivity* to increase a sensor's sensitivity value by 10 percent.

```
WTsensor *sensor;  
WTsensor_setsensitivity(sensor,  
    1.1 * WTsensor_getsensitivity(sensor));
```

WTsensor_setangularrate

```
void WTsensor_setangularrate(  
    WTsensor *sensor,  
    float s);
```

This function sets the scale factor for a sensor's rotation records. The angular rate is the maximum rotation (in radians) around any axis that a sensor returns in any pass through the simulation loop. The default angular rate for all sensors is 0.087266 radians, or 5 degrees. It may be convenient to specify the angular rate in terms of the defined constant *PI*, as in the example below.

Not all devices supported in WTK have their rotation records scaled in this way. You can not set the rotational speed of absolute position and orientation sensing devices, such as the FASTRAK or Bird devices. Some of the devices that are scaled in this way are the

Spaceball, Geometry Ball, Jr., and the Mouse. The sensor rotational records scaled by *WTsensor_setangularrate* are described in the corresponding sections of this chapter for each device.

```
WTsensor *spaceball;

/* create the spaceball sensor object */
spaceball = WTspaceball_new(SERIAL1);

/* set the maximum rotation from the spaceball around any axis
to 22.5 degrees per tick. */
WTsensor_setangularrate(spaceball, PI/8.0);

/* scale translational inputs with the size of the scene */
WTsensor_setsensitivity(spaceball, 0.01 * WTnode_getradius(
    (WTuniverse_getrootnodes()));
```

WTsensor_getangularrate

```
float WTsensor_getangularrate(
    WTsensor *sensor);
```

This function returns the maximum angular rate of change around each axis for a given sensor. Not all devices have their rotation records scaled in this way. Angular rate is specified in radians.

WTsensor_gettranslation

```
void WTsensor_gettranslation(
    WTsensor *sensor,
    WTp3 translation);
```

This function retrieves the current translation record from the sensor and stores it in the *translation* argument. The translation record is affected by the sensor's sensitivity scale factor. (See the function *WTsensor_setsensitivity* on page 13-11.)

If the device is an absolute sensor such as the Polhemus ISOTRAK, then *translation* is the change in sensor position since the last time through the simulation loop.

The following is an example of using a desktop device such as the Spaceball or Geometry Ball, Jr. to interactively stretch a geometry. In this example, the sensor's translation record is obtained and then transformed so that the resulting scale factor for each coordinate lies between 0.0 and 2.0. Values less than 1.0 are used to make the object smaller, while values greater than 1.0 are used to make it larger. The geometry is stretched in its local coordinate frame.

```
WTgeometry *geom;
WTsensor *ball;
WTp3 scalefactor;    /* for WTgeometry_stretch */
WTp3 mid;            /* object geometry */
float sensitivity;

WTsensor_gettranslation(ball, scalefactor);
sensitivity = WTsensor_getsensitivity(ball);

/* transform translation values to be between 0.0 and 2.0.
   (each scalefactor[i] is between -sensitivity and +sensitivity.) */
for ( i=0 ; i<3 ; i++ ) {
    scalefactor[i] = 1.0 + scalefactor[i]/sensitivity;
}

/* stretch the geometry */
WTgeometry_getmidpoint(geom, mid);
WTgeometry_stretch(geom, scalefactor, mid);
```

WTsensor_getrotation

```
void WTsensor_getrotation(
    WTsensor *sensor,
    WTq rotation);
```

This function retrieves the current rotation record from the sensor and stores it as a quaternion in the *rotation* argument. If the device is an absolute sensor such as the Polhemus ISOTRAK, then *rotation* is the change in orientation since the last time through the simulation loop.

See Chapter 25, *Math Library*, for functions that can be used to convert the rotation record into either a matrix or direction vector.

WTsensor_getmiscdata

```
int WTsensor_getmiscdata(  
    WTsensor *sensor);
```

This function returns an integer value in which miscellaneous data pertaining to the sensor, like button press events, are stored. Defined constants are used to interpret the return value of this function. For example, the following code fragment shows how to detect a left-button press on the Mouse:

```
WTsensor *mouse;  
if ( WTsensor_getmiscdata(mouse) & WTMOUSE_LEFTBUTTON )  
    WTmessage("Left button press\n");
```

The WTK defined constants used with *WTsensor_getmiscdata* are described in the corresponding sections of this chapter for each device and are also listed in Appendix C.

WTsensor_getrawdata

```
void *WTsensor_getrawdata(  
    WTsensor *sensor);
```

This function returns the sensor-specific raw data structure. The return needs to be typecast appropriately before the contents of the structure are accessed.

For example, WTK's Mouse raw data structure stores the current Mouse cursor position in screen coordinates in a *WTp2* (2D vector). This position might be passed in to a picking function, as in the following example which selects a polygon located under the Mouse cursor.

```
WTpoly *mouse_pickpoly(WTsensor *mouse)  
{  
    WTmouse_rawdata *raw;  
  
    /* get the mouse raw data struct (note typecasting)*/  
    raw = (WTmouse_rawdata *)WTsensor_getrawdata(mouse);  
  
    /* return the polygon under the mouse cursor */
```

```
    return WTscreen_pickpoly(screen,raw->pos, &nodepath, p3);  
}
```

The sensor raw data structures accessed by *WTsensor_getrawdata* are described in the corresponding sections of this chapter for each device. The *WTscreen_pickpoly* function is described on page 4-91.

WTsensor_getserial

```
WTserial *WTsensor_getserial(  
    WTsensor *sensor);
```

This function returns the serial port object associated with a sensor. The serial port object is the same as that supplied as the *serial* argument to the *WTsensor_new* (see page 13-7) call. This function is used primarily by developers writing their own sensor drivers for devices not already supported in WTK. See Appendix E for examples of using this function. Also consult your Hardware Guide for platform-specific information about using serial ports.

WTsensor_getunit

```
short WTsensor_getunit(  
    WTsensor *sensor);
```

This function retrieves the unit number of the specified sensor. This is useful for multi-unit sensors.

Rotating Sensor Input

Each 6D sensor supported in WTK has a reference frame (that is, a set of coordinate axes) associated with it. This reference frame defines how input from the device generates X,Y, and Z translation and rotation sensor records. The reference frame convention for devices supported in WTK is described for each device in the corresponding sections of this chapter. As shown in the example below, these conventions were chosen for their convenience when controlling a viewpoint or object in the reference frame of the

viewpoint. In some cases, however, it may be necessary to use coordinate axes other than the default ones. For this reason the *WTsensor_rotate* function is provided.

Consider the Geometry Ball, Jr. The coordinate axis convention for this device is such that if it is sitting on your desk with the cord running out the back of the device away from you, then the Z axis points straight back (in the direction of the cord), the X axis points to the right, and the Y axis points straight down (see Figure 13-1 on page 13-18). This coordinate convention has been chosen for its convenience. Let's say that you are using the Geometry Ball, Jr. to control a viewpoint, as set up with the following calls:

```
WTsensor *geoball;  
geoball = WTgeoball_new(SERIAL1);  
WTviewpoint_addsensor(WTuniverse_getviewpoint(), geoball);
```

Then when you apply force or torque to the ball, the viewpoint translates or rotates in the same direction. For example, if you push on the ball from the front (force applied in the positive Z direction on the ball), the viewpoint moves straight ahead, which is the same as the positive Z direction in the viewpoint frame.

Alternatively, the Geometry Ball, Jr. could be attached to a transform node associated with a geometry to control its motion, using this call:

```
WTnode *xform; /* transform node associated with the geometry */  
WTsensor *geoball;  
WTnode_addsensor(xform, geoball);
```

By default the geometry is specified to move in the local frame. Twisting or pushing on the ball causes the geometry to move correspondingly in its local frame.

Now consider an application where coordinate axes other than the default sensor coordinates are needed. We'll use the Geometry Ball, Jr. to control the motion of a graphical car. When we push on the front of the ball (generating input in the positive Z direction), we want the car to drive forward in its local reference frame, no matter which way it is oriented. And twisting the ball about its vertical axis should generate right or left turns of the car. This means that we want to attach the sensor to the graphical car with the following call, where *xform* is the transform node that manipulates the car (which could be a group node):

```
WTnode_addsensor(xform, sensor);
```

This, however, causes the sensor to act on the car in its local frame. The problem we run into is that the reference frame convention for the Geometry Ball, Jr. does not match up with that of the car model. Figure 13-1 illustrates these coordinate frames. Figure 13-2 on page 13-19 further illustrates the concepts involved.

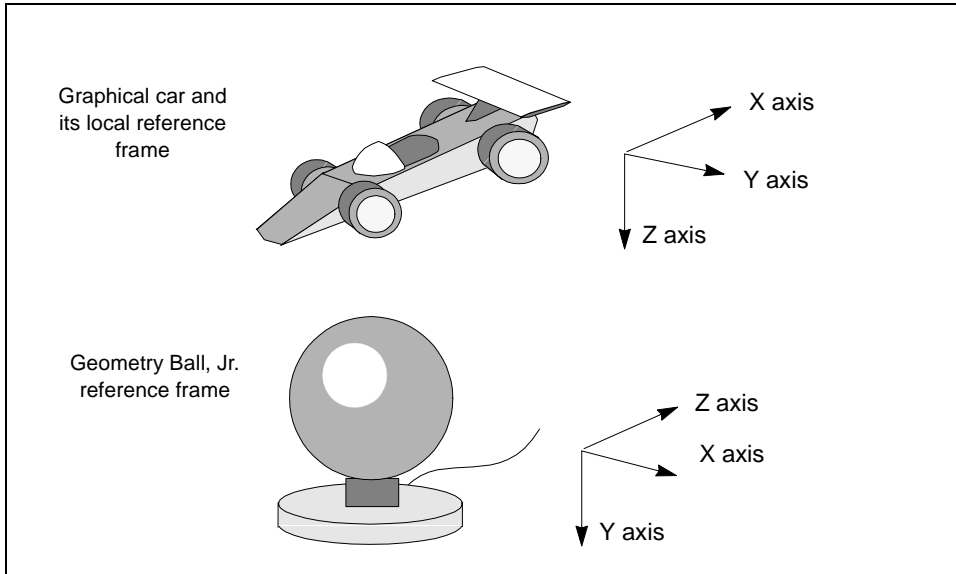


Figure 13-1: Reference frames for Geometry Ball, Jr. and graphical car.

Since the coordinate frames for the Geometry Ball, Jr. and the car do not line up, pushing and twisting the ball would generate inappropriate motion of the car. It is in a case like this that the function `WTsensor_rotate` is useful. This function effectively rotates a sensor's reference frame so that the desired coordinate values are returned.

To understand how to rotate the reference frames, first read the next section, *Geometry Motion Reference Frames*, then read the description of `WTsensor_rotate` on page 13-20, where this example is continued.

Geometry Motion Reference Frames

Many of the functions that let you move geometries within the virtual world take as an argument the reference frame in which the motion is to occur. These reference frames are illustrated in figure 13-2 below.

- *WTFRAME_WORLD* is the world coordinate frame. It is independent of the objects in the universe and is fixed in space.
- *WTFRAME_LOCAL* is the local coordinate frame of the geometry. This is either determined from the location of the geometry's vertices or taken to coincide with the world coordinate frame when the geometry is constructed.
- *WTFRAME_PARENT* is the parent coordinate frame of the geometry. This is similar to the local coordinate frame, except that transforms applied in the parent frame are pre-concatenated instead of post-concatenated.
- *WTFRAME_VPOINT* is the reference frame of a viewpoint (see *WTviewpoint_setposition* on page 16-8). For example, to move a geometry in the direction the viewpoint is looking, move it in the positive Z direction in *WTFRAME_VPOINT*.

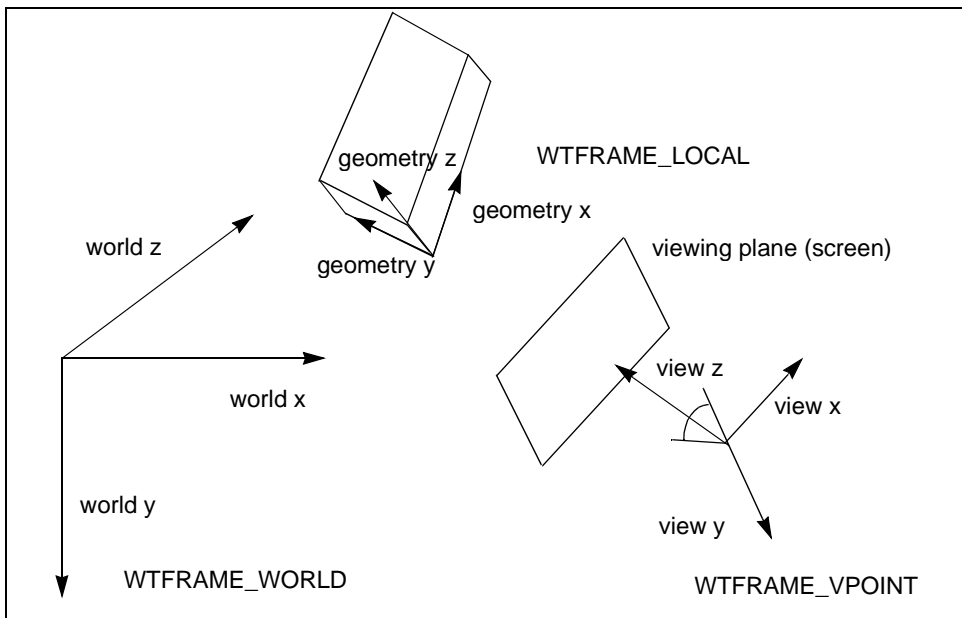


Figure 13-2: Reference frames for geometry motion

WTsensor_rotate

```
void WTsensor_rotate(  
    WTsensor *sensor,  
    WTq rotation);
```

This function rotates a sensor's coordinate frame. The *rotation* argument is a quaternion containing the rotation through which the sensor's coordinate axes are to be rotated to reach the desired coordinate axis orientation. Note that the rotations are always with respect to the world rather than to the local reference frame.

To continue with the example from *Rotating Sensor Input* on page 13-16, we need to rotate the coordinate frame of the Geometry Ball, Jr. so that it coincides with that of the car model. To accomplish this, we first need to rotate the ball's reference frame through minus 90 degrees about the world's Y axis. This causes the ball's X axis to align with the car model's X axis. Then we rotate the ball's reference frame through minus 90 degrees about the world's Z axis (so that the X axes continue to stay aligned). The result is that the two coordinate frames are now aligned. The following example shows how to implement this:

```
WTq qy, qz, qtotal;  
WTsensor *geoball;  
  
/* generate quaternion for -90 degree rotation about Y */  
WTeuler_2q(0.0, -0.5*PI, 0.0, qy);  
  
/* generate quaternion for -90 degree rotation about Z */  
WTeuler_2q(0.0, 0.0, -0.5*PI, qz);  
  
/* obtain combined rotation (note right-to-left multiplication) */  
WTq_mult(qy, qz, qtotal);  
  
/* rotate the Geometry Ball Jr. reference frame */  
WTsensor_rotate(geoball, qtotal);
```

Also see *How Do I Use Orientation-Tracking Sensors (On A Head-Mount-Display) That Are Not Positioned Along The Central Axis Of The HMD?* on page A-36.

Constraining Sensor Input

WTsensor_setconstraints

```
void WTsensor_setconstraints(  
    WTsensor *sensor,  
    short c);
```

This function constrains the values returned by a sensor. This is accomplished by passing in a combination of the following flags separated by the C language bit-wise OR operator “|”.

<i>WTCONSTRAIN_X</i>	constrains X axis translations
<i>WTCONSTRAIN_Y</i>	constrains Y axis translations
<i>WTCONSTRAIN_Z</i>	constrains Z axis translations
<i>WTCONSTRAIN_XROT</i>	constrains rotations about the X axis
<i>WTCONSTRAIN_YROT</i>	constrains rotations about the Y axis
<i>WTCONSTRAIN_ZROT</i>	constrains rotations about the Z axis

For example, to constrain all rotational input so that only translational input is returned:

```
WTsensor *sensor;  
WTsensor_setconstraints(sensor, WTCONSTRAIN_XROT |  
    WTCONSTRAIN_YROT | WTCONSTRAIN_ZROT);
```

The constraints set with *WTsensor_setconstraints* pertain to the values read from the device so objects attached to a sensor may exhibit unexpected behavior because the object's coordinate frame is not aligned with the sensor's reference frame. If you need to constrain an object's motion in a particular coordinate frame, use motion links to connect a sensor to an object and then constrain the motion link. Refer to Chapter 15, *Motion Links* for more information.

In the current version of WTK, rotational constraints applied using *WTsensor_setconstraints* have no effect on the FASTRAK or Bird devices. However, it is possible to simultaneously constrain all rotational input from these devices as described in the sections *Scaling ISOTRAK Records* on page 13-86 and *Scaling Bird Records* on page 13-41.

WTsensor_getconstraints

```
short WTsensor_getconstraints(  
    WTsensor *sensor);
```

This function returns a short describing the constraints currently imposed on the values returned by the sensor. To determine whether a particular constraint has been set, you can use the bit-wise AND operator '&' for the particular constraint. For example:

```
WTsensor *sensor;  
if ( WTsensor_getconstraints(sensor) & WTCONSTRAIN_XROT ) {  
    WTmessage("X rotations are constrained\n");  
}
```

Using Different Baud Rates

As stated under the section *Sensor Construction and Destruction* on page 13-5, *WTsensor* objects can be created with either the generic sensor constructor function – *WTsensor_new* (see page 13-7) or with one of WTK's device-specific constructor macros like *WTspaceball_new*, *WTbird_new*, etc.

The device-specific constructor macros create the sensor object at a specific baud rate. You can see the baud rate at which a sensor is created by looking in the include file *sensors.h* (in the *include* directory).

To use other baud rates you need to do one of the following:

- Create the sensor object with *WTsensor_new* rather than the device-specific macro, passing in a serial port object constructed for that baud rate.
- Use the device-specific macro, but first edit the include file *sensors.h* (in the *include* directory) and change the baud rate setting in the macro definition.

For sensors with DIP switches (i.e., ISOTRAK, ISOTRAK II, FASTRAK, Bird, Flock of Birds, Extended Range Bird, and Pinch Glove) you also need to change the DIP switch settings.

Sensor Name

WTsensor_setname

```
void WTsensor_setname(  
    WTsensor *sensor,  
    const char *name);
```

This function sets the name of the specified sensor. All sensors have a name; by default, a sensor's name is "" (i.e., a NULL string).

WTsensor_getname

```
const char *WTsensor_getname(  
    WTsensor *sensor);
```

This function returns the name of the specified sensor.

User-specifiable Sensor Data

A *void ** pointer is included as part of the structure defining a sensor object, so that you can store whatever data you wish with a sensor. The following functions can be used to set and get this field within any sensor.

WTsensor_setdata

```
void WTsensor_setdata(  
    WTsensor *sensor,  
    void *data);
```

This function sets the user-defined data field in a sensor. Private application data can be stored in any structure. To store a pointer to the structure within the sensor, pass a pointer to it, cast to a *void **, as the *data* argument.

WTsensor_getdata

```
void *WTsensor_getdata(  
    WTsensor *sensor);
```

This function retrieves user-defined data stored within a sensor. You should cast the value returned by this function to the same type that was used to store the data with the *WTsensor_setdata* function.

Custom Sensor Drivers

The following functions are only needed if you are writing your own sensor driver. Consult Appendix E for more on this subject.

WTsensor_setrecord

```
void WTsensor_setrecord(  
    WTsensor *sensor,  
    WTp3 p,  
    WTq q);
```

This function stores the current relative position and orientation record with your sensor. If your sensor returns absolute records, you must first call *WTsensor_relativizerecord* (see below). You may also wish to apply scale factors to the sensor record using *WTsensor_setsensitivity* (see page 13-11) and *WTsensor_getsensitivity* (see page 13-12) before calling *WTsensor_setrecord*, as described in Appendix E.

WTsensor_relativizerecord

```
void WTsensor_relativizerecord(  
    WTsensor *sensor,  
    WTp3 absolute_p,  
    WTq absolute_q,  
    WTp3 relative_p,  
    WTq relative_q);
```

If your sensor returns absolute records, use this function to generate the corresponding relative record. This function is passed the absolute position/orientation record obtained from your device this time through the simulation loop, and returns (in p and q) the change in position and orientation since last time.

Note that you must set the sensor's absolute record (using *WTsensor_setlastrecord*) prior to calling *WTsensor_relativizerecord*.

WTsensor_setlastrecord

```
void WTsensor_setlastrecord(  
    WTsensor *sensor,  
    WTp3 absolute_p,  
    WTq absolute_q);
```

This function sets the absolute record for sensors with absolute position/orientation records. In your sensor update function, after you have set the new sensor record with *WTsensor_setrecord*, store the absolute record with *WTsensor_setlastrecord* so that the next record can be made relative to it the next time through the simulation loop.

WTsensor_getlastrecord

```
void WTsensor_getlastrecord(  
    WTsensor *sensor,  
    WTp3 absolute_p,  
    WTq absolute_q);
```

This function retrieves the position and orientation record most recently set with the *WTsensor_setlastrecord* function and stores them in *absolute_p* and *absolute_q*.

WTsensor_setmiscdata

```
void WTsensor_setmiscdata(  
    WTsensor *sensor,  
    int data);
```

This function stores miscellaneous sensor data, like button press events, with the sensor object. Typically, you do not need to use this function unless you are writing your own sensor driver.

WTsensor_setrawdata

```
void WTsensor_setrawdata(  
    WTsensor *sensor,  
    void *dataptr);
```

This function stores raw sensor data with the sensor object. The data can then be returned with *WTsensor_getrawdata*. The raw data structure for a custom sensor driver is defined by the developer. You do not need to use this function unless you are writing your own sensor driver.

The Mouse

To create a WTK Mouse sensor object, you can call *WTsensor_new*, passing in the driver functions *WTmouse_open*, *WTmouse_close*, and the desired update function — either one of the update functions described on page 13-7 or one that you may have written.

For example, using *WTsensor_new* you might have:

```
WTsensor *mouse;  
mouse = WTsensor_new(WTmouse_open, WTmouse_close,  
    WTmouse_move2D, NULL, 1, WTSENSOR_DEFAULT);
```

Note that the serial port argument for the Mouse sensor is always NULL.

Alternatively, a platform-independent macro *WTmouse_new* is provided for creating a Mouse sensor object.

To use this macro, simply call:

```
WTsensor *mouse;  
mouse = WTmouse_new();
```

This macro makes use of the sensor driver functions *WTmouse_open*, *WTmouse_close*, and *WTmouse_moveview2* (currently the most popular of the Mouse update functions).

Note that to use the *WTmouse_move2D* update function rather than *WTmouse_moveview2*, you could call:

```
WTsensor *mouse;
    WTsensor_setupdatefn(mouse, WTmouse_move2D);
```

Accessing Mouse Raw Data

The Mouse raw data structure stores the raw X,Y screen location of the Mouse. This information is accessed using the *WTsensor_getrawdata* (see page 13-15) function, as in the example below.

The raw data structure for the Mouse is type defined as follows:

```
typedef struct _WTmouse_rawdata {
    WTp2 pos;
} WTmouse_rawdata;
```

and is accessed as follows:

```
WTmouse_rawdata *raw;
/* get raw x and y mouse values in screen coordinates */
raw = (WTmouse_rawdata *)WTsensor_getrawdata(mouse);
WTmessage("Mouse position: %f, %fn", raw->pos[X], raw->pos[Y]);
```

Scaling Mouse Records

Translational and rotational records for the Mouse can be scaled using the functions *WTsensor_setsensitivity* (see page 13-11) and *WTsensor_setangularrate* (see page 13-12) respectively.

Mouse Update Functions

WTmouse_drawcursor

```
void WTmouse_drawcursor(  
    WTsensor *sensor);
```

This function is a Mouse update function that does not allow any movement (neither translational nor rotational). It just updates the X,Y screen location and stores it in the sensor's raw data structure. It is particularly useful when you want to pick a geometry/polygon (corresponding to a button-click) in the scene instead of flying around.

The following function allows toggling between flying and not flying:

```
void FlipMouseMoveMode(WTsensor *mouse)  
{  
    int misc;  
    if (!mouse) return;  
  
    /* global variable */  
    movemouse ^=1;  
  
    if (movemouse)  
        /* enable flying */  
        WTsensor_setupdatefn(mouse, WTmouse_moveview2);  
    else  
        /* disable flying (enable geometry/polygon picking) */  
        WTsensor_setupdatefn(mouse, WTmouse_drawcursor);  
  
    misc = WTsensor_getmiscdata(mouse);  
  
    /* left mouse button to pick geometrys */  
    if ( !movemouse && (misc & WTMOUSE_LEFTBUTTON))  
        /* code for picking geometries */  
  
    /* right mouse button to pick polygons */  
    if ( !movemouse && (misc & WTMOUSE_RIGHTBUTTON))  
        /* code for picking polygons */
```



```
}
```

WTmouse_move2D

```
void WTmouse_move2D(  
    WTSensor *sensor);
```

This function is a Mouse update function that allows Z translation and Y rotation. It is a simple update function useful for controlling vehicle motion on a two-dimensional surface. This update function uses the Mouse position within the screen or window to generate forward and backward motion (translation along Z) and left and right yaw (rotation around Y). Button presses have no effect with this update function.

Translations forward and backward along Z are determined from the vertical displacement (positive or negative) of the Mouse from a horizontal line through the screen or window midpoint. The translation amount scales linearly with this displacement, and is also scaled by the sensor's sensitivity (see *WTSensor_setsensitivity* on page 13-11).

Left and right yaw (rotations about Y) are determined from the horizontal displacement (positive or negative) of the Mouse from a vertical line through the screen or window midpoint. The angular rotation scales linearly with this displacement, and is also scaled by the sensor's angular rate (see *WTSensor_setangularrate* on page 13-12).

WTmouse_moveview1

```
void WTmouse_moveview1(  
    WTSensor *sensor);
```

This function is a Mouse update function that operates in 3D without pitch or roll. It is useful for moving a viewpoint through a 3D environment, but is a fairly simple update function that does not pitch or roll the viewpoint. Use *WTmouse_moveview2* (see below) for a Mouse update function providing control of all six degrees of freedom.

With the Mouse attached to the viewpoint using *WTmouse_moveview1* (as in the example below), manipulating the Mouse has the following effect:

- When the Mouse cursor is centered on the screen and no buttons are pressed, the viewpoint is stationary.

- When the cursor is in the left half of the screen, the viewpoint shifts to the left; in the right half of the screen, the viewpoint shifts to the right.
- When the cursor is in the top half of the screen, the viewpoint moves forward; in the bottom half of the screen, the viewpoint moves backward.
- When the left Mouse button is pressed, the viewpoint yaws to the left; when the right Mouse button is pressed, the viewpoint yaws to the right,
- Pressing both left and right Mouse buttons simultaneously with the cursor in the top half of the screen translates the viewpoint upward; the viewpoint translates downward if the cursor is in the bottom half of the screen.

Here's an example of creating a Mouse sensor object with the `WTmouse_moveview1` update function and attaching it to the viewpoint:

```
main()
{
    WTsensord *mouse;

    /* initialize the universe */
    WTuniverse_new(...);

    /* Create the mouse sensor object. The last argument is NULL,
    since the mouse, although it may be connected to the serial port,
    is not treated by WorldToolKit as a serial port device */
    mouse = WTsensord_new(WTmouse_open, WTmouse_close,
        WTmouse_moveview1, NULL, 1, WTSENSOR_DEFAULT);

    if ( !mouse )
        WTwarning("Warning, couldn't open mouse driver\n");

    /* attach the mouse sensor to the viewpoint */
    WTviewpoint_addsensord(WTuniverse_getviewpoint(), mouse);

    WTuniverse_go();
    WTuniverse_delete();
}
```

WTmouse_moveview2

```
void WTmouse_moveview2(  
    Wtsensor *sensor);
```

This function is a Mouse update function that supports six degrees of freedom movement of a viewpoint through a 3D environment. Unlike *WTmouse_moveview1*, which translates and yaws the viewpoint, with *WTmouse_moveview2* the viewpoint can also be pitched and rolled. Also unlike *WTmouse_moveview1*, which moves the viewpoint whenever the Mouse cursor is away from the center of the screen, *WTmouse_moveview2* only moves the viewpoint while the cursor is away from the center of the screen *and* one or more Mouse buttons are pressed. The farther away from the middle of the screen, the faster the movement. Maximum rotations and translations are scaled as described under the functions *WTsensor_setangularrate* (see page 13-12) and *WTsensor_setsensitivity* (see page 13-11).

When you are using *WTmouse_moveview2*, with the Mouse attached to the viewpoint, manipulating the Mouse has the following effect.

Using the left Mouse button, you can “walk” about your model. When the left button is pressed, and the cursor is in:

- the top half of screen – move forward
- the bottom half of screen – move backward
- the left half of screen – yaw left
- the right half of screen – yaw right

When the right button is pressed, and the cursor is in:

- the top half of screen – move up
- the bottom half of screen – move down
- the left half of screen – pan left
- the right half of screen – pan right

When the left *and* right buttons are pressed, and the cursor is in:

- the top half of screen – pitch up
- the bottom half of screen – pitch down
- the left half of screen – roll left

- the right half of screen – roll right

The viewpoint is stationary when neither the left nor the right Mouse button is pressed.

The macro *WTmouse_new* creates a Mouse sensor object that uses the *WTmouse_moveview2* update function.

Writing Your Own Mouse Update Function

When creating a Mouse sensor object, you can use one of the update functions provided, or you can write your own. This is not difficult. Your update function should first call *WTmouse_rawupdate* (see below) to obtain the Mouse's raw data. It should then specify how the raw data is to be transformed into the 3D position and orientation record. Finally, your update function must store this record with the sensor by calling *WTsensor_setrecord* (see page 13-24). See *Example 1: Update Function for the Mouse* on page E-8.

WTmouse_rawupdate

```
void WTmouse_rawupdate(  
    WTsensor *sensor);
```

This function obtains the X, Y screen location of the Mouse and stores it in the sensor's raw data structure. This information can be accessed with *WTsensor_getrawdata* (see page 13-15). See also *Accessing Mouse Raw Data* on page 13-27.

Mouse button presses are also read by this function and can be accessed with *WTsensor_getmiscdata* (see page 13-15). See also *Mouse Defined Constants* below.

Mouse Defined Constants

Mouse button presses can be detected in WTK using the function *WTsensor_getmiscdata* (see page 13-15), examining the result for the following defined constants (bits). The constants for the Mouse middle button are used only on three-button mice.

- **Button held down.** This is generated each frame that the user holds a button down. These events are defined as: *WTMOUSE_LEFTDOWN*, *WTMOUSE_MIDDLEDOWN*, and *WTMOUSE_RIGHTDOWN*.
- **Button transitioned down.** This generates a single event each time the button moves from up to down. These events are defined as: *WTMOUSE_LEFTBUTTON*, *WTMOUSE_MIDDLEBUTTON*, and *WTMOUSE_RIGHTBUTTON*.
- **Button transitioned up.** This generates a single event each time the button moves from down to up. These events are defined as: *WTMOUSE_LEFTUP*, *WTMOUSE_MIDDLEUP*, and *WTMOUSE_RIGHTUP*. Note that button up events may not be available on all systems.
- **Button double-clicked.** This generates a single event each time the Mouse is double-clicked. These events are defined as: *WTMOUSE_LEFTDBLCLK*, *WTMOUSE_MIDDLEDCLK*, and *WTMOUSE_RIGHTDBLCLK*.

The following is an example of accessing Mouse button events.

```
void read_mouse_record(WTsensor *mouse)
{
    int buttons;
    FLAG leftbutton, rightbutton, bothbuttons;
    FLAG leftdown, rightdown;

    /* get button press data */
    buttons = WTsensor_getmiscdata(mouse);

    /* which buttons were just pressed? */
    leftbutton = buttons & WTMOUSE_LEFTBUTTON;
    rightbutton = buttons & WTMOUSE_RIGHTBUTTON;
    bothbuttons = leftbutton && rightbutton;

    /* which buttons are currently down */
    leftdown = buttons & WTMOUSE_LEFTDOWN;
```

```
        rightdown = buttons & WTMOUSE_RIGHTDOWN;
    }
```

Dragging Objects Using a Mouse

If you wanted to use a mouse to drag picked objects, you will need to write a mouse update function that reports (as XY translation) the movement of the mouse when the button is held down. Here's an example:

```
void mouse_drag(WTsensor *sensor)
{
    static WTp2 last_position;
    WTp2 diff;
    WTmouse_rawdata *raw;
    WTp3 p;
    WTq q;
    WTp3_init(p);
    WTq_init(q);
    WTmouse_rawupdate(sensor);
    raw = (WTmouse_rawdata *)WTsensor_getrawdata(sensor);
    if (WTsensor_getmiscdata(sensor) & WTMOUSE_LEFTDOWN) {
        diff[X] = raw->pos[X] - last_position[X];
        diff[Y] = raw->pos[Y] - last_position[Y];
    }
    WTp2_copy(raw->pos, last_position);
    WTsensor_setrecord(sensor, p, q);
}
```

By using the above mouse update function instead of the standard "WTmouse_moveview2" function, and writing object-picking code to attach the mouse to picked objects, you can use the mouse to pick an object and drag it in the window.

Checking the Input Focus Window for the Mouse

WTmouse_inwindow

```
FLAG WTmouse_inwindow(  
    WTsensor *mouse, WTwindow *w)
```

This function determines whether the mouse is within a WTK window. It returns **TRUE** if the mouse is within the WTK window; otherwise, it returns **FALSE**.

WTmouse_whichwindow

```
WTwindow *WTmouse_whichwindow(  
    WTsensor *mouse)
```

This function determines which WTK window (if any) the mouse is in and returns a pointer to that window. It returns **NULL** if the mouse is not in any WTK window.

The above functions are especially useful in applications having multiple windows. The following example illustrates this concept. It returns **TRUE** if it picks a polygon in a particular window.

```
FLAG Window_Pickpoly(WTwindow *w)  
{  
    int x0, y0, width, height;  
    WTpoly *poly;  
    WTnode *node = NULL;  
    WTp3 p;  
    WTp2 point;  
  
    /* check if mouse is in the window */  
    if (WTmouse_inwindow(mouse, w) {  
        WTwindow *current;  
  
        /* get the current window */  
        current = WTmouse_whichwindow(mouse);  
  
        /* check if the current window is the window that was passed in */
```

```
if (w==current) {
    WTwindow_getposition(current, &x0, &y0, &width, &height);
    point[X] = width/2.f;
    point[Y] = height/2.f;

    /* pick the frontmost polygon in the center of current window */
    poly = WTwindow_pickpolygon(current, point, NULL, p);

    /* if we picked a polygon, return TRUE */
    if ( poly )
        return TRUE
    else
        return FALSE;
}
else
    return FALSE;
}
```

Using the Mouse as a Trackball

WTK provides functions that enable the use of the Mouse sensor object as a trackball. The WTK demo *flipobj.c* uses the Mouse as a trackball.

Following are the update functions for using the Mouse as a trackball:

WTmouse_trackball

```
void WTmouse_trackball(
    WTSensor *sensor);
```


WTmouse_trackballvpoint

```
void WTmouse_trackballvpoint(  
    WTsensor *sensor);
```

Use the function *WTmouse_trackball* if connecting the trackball to the object. If connecting the trackball to the viewpoint, use the function *WTmouse_trackballvpoint*.

To create a Mouse sensor object as a trackball use either the generic sensor constructor function – *WTsensor_new* (see page 13-7) with one of the functions described above as the update function or use the macro *WTmouse_new* and set the update function to one of the functions described above using *WTsensor_setupdatefn* (see page 13-10).

When using the Mouse as a trackball you can control the drift, snap angle, and snap. The default trackball drift value is 1.0. This means that if you provide a rotation to the trackball, it goes into continuous rotation and doesn't stop. A drift value of less than 1.0 means that every frame the rotation speed decreases and finally becomes zero.

The following functions allow you to control the drift of the trackball:

WTmouse_settrackballdrif

```
void WTmouse_settrackballdrif(  
    WTsensor *sensor,  
    float drift);
```

This function is used to set the drift of the trackball. Valid values are between 0.0 and 1.0.

WTmouse_gettrackballdrif

```
float WTmouse_gettrackballdrif(  
    WTsensor *sensor);
```

This function returns the current drift value of the trackball.

The snap angle (specified in radians) allows you to set the increment value of rotations. The default trackball snap angle value is 0.0. This allows for a smooth rotation.

The following functions allow you to control the snap angle of the trackball:

WTmouse_settrackballsnapangle

```
void WTmouse_settrackballsnapangle(  
    WTsensord *sensor,  
    float snapangle);
```

This function is used to set the snap angle of the trackball. When using this function, the trackball drift value must be set to 0.0. If the snap angle is greater than 0.0, rotation is allowed around one axis only at one time.

WTmouse_gettrackballsnapangle

```
float WTmouse_gettrackballsnapangle(  
    WTsensord *sensor);
```

This function returns the current snap angle of the trackball.

The snap allows you to set the increment value of translations. The default trackball snap value is 0.0. This allows for smooth translation.

The following functions allow you to control the snap value of the trackball:

WTmouse_settrackballsnap

```
void WTmouse_settrackballsnap(  
    WTsensord *sensor,  
    float snap);
```

This function is used to set the snap value of the trackball. When using this function, the trackball drift value must be set to 0.0.

WTmouse_gettrackballsnap

```
float WTmouse_gettrackballsnap(  
    WTsensord *sensor);
```

This function returns the current snap value of the trackball.

To reset the trackball use the following function:

WTmouse_trackballreset

```
void WTmouse_trackballreset (  
    WTsensord *sensor);
```

If an object is continuously rotating, you can use this function to stop the rotation.

Ascension Bird

The Bird from Ascension Technology Corporation is an electromagnetic-based six degree-of-freedom sensor that measures absolute position and orientation.

To create a Bird sensor object on serial port 1, you can use the macro call:

```
WTsensord *bird;  
bird = WTbird_new(SERIAL1, 1);
```

This macro makes use of the sensor driver functions *WTbird_open*, *WTbird_close*, and *WTbird_update*. It creates the Bird sensor object running at 9600 baud.

WTbird_new takes a second argument specifying the unit number of the Bird to open. This value is 1 for a single Bird. (Note that in the Ascension documentation a stand-alone unit is number 0, but in WTK it is number 1.)

Following are the DIP switch settings for the Bird sensor objects running at 9600 baud.

Older Birds	OFF ON OFF OFF OFF OFF OFF OFF
Newer Birds	OFF ON ON OFF OFF OFF ON OFF

The three left-most DIP switch settings on each Bird are for the baud rate. For example, 9600 is OFF ON ON and 19200 is ON OFF OFF. The next four DIP switches on each Bird indicate unit number: unit one is OFF OFF OFF ON, unit two is OFF OFF ON OFF, and so on. The right-most DIP switch is normally OFF.

Consult your Bird reference manual if you are uncertain of how to set your Bird DIP switches. A single Bird should be configured with a Bird address of 1, not 0.

When you call *WTbird_new* to construct a new Bird sensor object, the *openfn* for the device is automatically called. Part of the function of the *openfn* for this device is to calibrate the sensor, which consists of obtaining an initial position and orientation record. This takes several seconds, during which the device should not be moved. Records subsequently generated by the *updatefn* are with respect to this initial reference frame. It may be useful in your application to let the user know that the device is about to be calibrated. For example, you might want to have a print statement:

```
WTsensor *sensor;
WTmessage("About to calibrate/initialize Bird...\n");
sensor = WTbird_new(SERIAL1,1);
WTmessage("Initialization complete.\n");
```

The coordinate frame of this sensor is the same as for the ISOTRAK and is defined in the WTK driver functions as follows. If the receiver cube is placed “flat-end down” in front of you with the cable from the cube coming out the back of the cube toward you, then (as illustrated in figure 13-5 on page 13-86 for the Polhemus ISOTRAK) the Z axis of the device points straight ahead, the X axis points to the right, and the Y axis points down. If this coordinate frame is not appropriate for your application, the function *WTsensor_rotate* (see page 13-20) can be used to define another coordinate frame for the device.

If a receiver is plugged into the Bird system, and the receiver is in the fly mode, you must initialize it. This has to be done even if your application does not use all of the receivers. The Bird will not return data for any of the receivers unless you have a call to *WTbird_new* for every connected receiver.

If your bird fails to initialize, you may need to use the *WTBIRDDELAY* environment variable to force WTK to wait longer for the bird to respond before giving up. Refer to the Environment Variables Appendix for more information.

The Ascension 6DOF mouse is used like the standard receiver. It plugs into the back panel of the receiver box and is initialized by a call to *WTbird_new*. The buttons on the mouse are

identified by *WTBIRD_LEFTBUTTON*, *WTBIRD_MIDDLEBUTTON*, and *WTBIRD_RIGHTBUTTON*.

Accessing Bird Raw Data

WTK does not provide a separate raw data structure for this device. The most recent sensor record can be obtained using *WTsensor_getlastrecord* (see page 13-25). This function retrieves the absolute record in WTK coordinates with no scale factors applied. This record is called “absolute” because it describes a location in 3D space rather than a change in location since the last frame. This absolute record is, however, relative to the position and orientation of the device when the device was opened by WTK.

The following function, *WTbird_getabsoluterecord*, retrieves the absolute sensor location relative to the transmitter.

WTbird_getabsoluterecord

```
int WTbird_getabsoluterecord(  
    WTsensor *sensor,  
    WTp3 p,  
    WTq q);
```

This function obtains the most recent absolute position and orientation record for the specified Bird sensor in relation to the Bird transmitter, and places them in *p* and *q*. These values are in WTK coordinates.

Scaling Bird Records

Translation records for the Bird can be scaled using the function *WTsensor_setsensitivity* (see page 13-11). It is often useful, for example, to scale sensor inputs with the size of the scene.

Unlike translation records, however, orientation records from the Bird cannot be scaled in the WTK update functions for this device.

It is possible to turn off all rotational input from these devices by writing your own update function which nullifies the orientation record. *Scaling ISOTRAK Records* on page 13-86 includes an example of how to do this.

Bird Update Function

WTbird_update

```
void WTbird_update(  
    WTsensor * sensor);
```

This update function packages the translation and rotation record from the device into the sensor object's record after relativizing it and then applying any translational scale factor that may have been set with *WTsensor_setsensitivity* (see page 13-11).

The macro *WTbird_new* creates a Bird sensor object that uses the *WTbird_update* function and is recommended for most users.

Syncing the Bird to the Monitor

WTbird_setsync

```
void WTbird_setsync(  
    WTsensor *bird,  
    short synctype);
```

This function sets a Bird to synchronize its scanning frequency with that of a CRT, thereby reducing the amount of interference. To activate this capability, set up your Bird with the appropriate attachment and call this function. The *synctype* argument is typically 2. If the LED display on the Bird indicates an error message, it is probably not picking up the CRT frequency clearly. Consult your Bird documentation for more information about CRT synchronization and about interpreting error messages from the Bird.

Setting the Bird Hemisphere

The functions in this section apply to the Bird, Flock of Birds (see page 13-51), and Bird with Extended Range Controller (see page 13-51).

WTbird_sethemisphere

```
void WTbird_sethemisphere(  
    WTsensort *sensor,  
    int hemisphere);
```

This function tells a Bird sensor in which hemisphere the Bird's receiver will be navigating. The shape of the magnetic field emitted by the Bird transmitter is symmetrical about each of the axes of the transmitter. For this reason, the Bird needs to know which hemisphere the receiver is in to provide correct translation records.

The *hemisphere* argument should be one of the following defined constants:

<i>WTBIRD_FORWARD</i>	side of Bird away from the transmitter cable
<i>WTBIRD_AFT</i>	opposite side from <i>WTBIRD_FORWARD</i>
<i>WTBIRD_UPPER</i>	top side of Bird transmitter
<i>WTBIRD_LOWER</i>	bottom side of Bird transmitter
<i>WTBIRD_LEFT</i>	while facing <i>WTBIRD_FORWARD</i> , the left side of the transmitter
<i>WTBIRD_RIGHT</i>	while facing <i>WTBIRD_FORWARD</i> , the right side of the transmitter

When you call this function, the Bird's receiver must be in the indicated hemisphere.

WTbird_gethemisphere

```
int WTbird_gethemisphere(  
    WTsensort *sensor);
```

This function returns the hemisphere currently set for the specified Bird sensor. The value returned is one of the defined constants listed above under *WTbird_sethemisphere*.

WTbird_autohemisphere

```
void WTbird_autohemisphere(  
    WTsensor *sensor);
```

This function automatically sets the hemisphere for the Bird receiver based on its current location. The Bird transmitter must be located in the forward hemisphere when this function is called.

There are some extreme usage circumstances where WTK may set the hemisphere incorrectly. Therefore, we recommend that you include the ability to reset the Bird hemisphere interactively — for example, using keyboard input as a trigger for calling *WTbird_sethemisphere*.

Streaming-Mode Flock of Birds Driver

WTK also includes a specialized, high performance, streaming-mode Flock of Birds driver that is not intended for generalized usage across multiple platforms. The streaming-mode FOB driver is implemented only for the IRIX 6.X operating system and is designed for the Extended Range Transmitter FOB. This driver utilizes multi-threading and therefore allows the FOB to be run in streaming-mode. This mode of operation eliminates the necessity for frame by frame communication between WorldToolKit and the FOB hardware.

WorldToolKit simply retrieves the most current data available from the FOB thread's buffer. The benefits of this decrease in sensor latency usually far outweighs the two drawbacks associated with the streaming-mode FOB driver. The first drawback is that if you are running on a single-processor machine the operating system must schedule the execution of multiple threads across the same processor; however, this does not necessarily cause a large impact upon your application's performance. The second and largest inconvenience resides in the driver's inability to use automatic hemisphere tracking. Due to the possibly large number of sensor records being sent from the FOB to the driver's buffer between retrievals by the main WorldToolKit application thread, the driver cannot accurately predict when a hemisphere has been crossed. There are two possible solutions for this quandary. (1) Simply restrict your movement to one hemisphere as you would by default in a CAVE or similar Spatially Immersive Display system or, (2) use the *WTFlock_sethemisphere* function during your program's execution to dynamically update the FOB in accordance with your application's possible range of movement.

To create a new Flock sensor object you simply follow the guidelines of the previous bird driver.

```
WTsensor *FOB;
FOB = WTflock_new(SERIAL1,1);
    if (FOB==NULL){
        WTErrror("Could not open flock\n");
    }
}
```

You may of course specify a valid specific serial port if you prefer, for example:

```
FOB = WTflock_new("/dev/ttyd3",1);
```

All of the streaming-mode FOB functions are prefixed with the class name WTflock. The functions provided for use with the streaming-mode driver are described below.

Opening and closing the FOB

WTflock_open

```
int WTflock_open(
    WTsensor *sensor);
```

Corresponds to the *WTbird_open* function.

WTflock_deviceopen

```
WTserial *WTflock_deviceopen(
    char *device,
    int baud,
    char parity,
    int databits,
    int stopbits,
    int buffersize);
```

You should only be making this function call when not using the WorldToolKit default *Wtflock_new* macro. Example arguments for this function can be found in the definition of the *Wtflock_new* macro found below and in *wtk/include/flock.p*.

Wtflock_close

```
void Wtflock_close(  
    WTsensor *sensor);
```

Corresponds to the *WTbird_close* function.

Calibrating/Initializing

Wtflock_getdefaulthemisphere

```
int Wtflock_getdefaulthemisphere(void);
```

Returns the FOB startup default hemisphere.

The returned value represents the currently set hemisphere for ALL FOB receivers upon their initialization. The default setting internal to WTK is *WTFLOCK_FORWARD*. You can change the initialization hemisphere for a receiver with *Wtflock_setdefaulthemisphere*.

Wtflock_setdefaulthemisphere

```
FLAG Wtflock_setdefaulthemisphere(  
    int hemisphere);
```

Returns success in setting the default startup hemisphere (useful with multiple FOB configurations.)

This function is useful in allowing you to initialize multiple receivers in differing hemispheres. For example, if you were using the FOB Streaming Mode driver in a Spatially Immersive Display (such as a CAVE) you would most probably be using two FOB receivers. The first FOB receiver would be used to track the user's head position/orientation whereas the second would be used to track a navigational device. If the receiver being used for head tracking was lying in a different FOB hemisphere in the CAVE than the receiver

for tracking the navigational device, you would initialize them both in different hemispheres. To do so, set the default hemisphere for the first receiver, initialize the receiver, set the default hemisphere for the second receiver and then initialize it as well.

WTflock_gethemisphere

```
int WTflock_gethemisphere(  
    WTsensor *sensor);
```

Returns the current hemisphere setting for a sensor. See *WTbird_gethemisphere*.

WTflock_sethemisphere

```
FLAG WTflock_sethemisphere(  
    WTsensor *sensor,  
    int hemisphere);
```

Returns success in setting the current hemisphere for a sensor. See *WTbird_sethemisphere*. It is important to note that the hemispheric constants for the FOB driver are prefixed with `WTFLOCK_` (not `WTBIRD_`), i.e. `WTFLOCK_UPPER` as opposed to `WTBIRD_UPPER`.

WTflock_getcrtsyncdata

```
FLAG WTflock_getcrtsyncdata(  
    WTsensor *sensor,  
    float *voltage,  
    float *rate);
```

Returns success in retrieving the current CRT synchronization settings.

This function aids you in finding the ‘sweet spot’ for your FOB amid the electro-magnetic noise introduced into the device by operating a receiver or transmitter within a few feet of a magnetically deflected Cathode Ray Tube (a typical monitor). The value returned in *voltage* represents the voltage proportional to your CRT’s vertical scan signal. The value returned in *rate* is the rate at which your monitor is vertically refreshing the screen. These values are very useful in determining what mode of operation to use with the function *WTflock_setcrtsync*.

Note: If the function returns successfully and contains a valid value for voltage but not for rate, your monitor is refreshing at a rate less than 31Hz which precludes your using the syncing features of the Streaming Mode FOB driver.

WTflock_setcrtsync

```
FLAG WTflock_setcrtsync(  
    WTsensor *sensor,  
    int mode);
```

Corresponds to the *WTbird_setsync* function.

WTflock_resetorigin

```
FLAG WTflock_resetorigin(  
    WTsensor *sensor);
```

Returns TRUE if the FOB's current position and orientation values have been set to the FOB's original position and orientation values.

Updating/Getting sensor Position/Orientation

WTflock_update

```
void WTflock_update(  
    WTsensor *sensor);
```

Corresponds to the *WTbird_update* function. Note: orientations are absolute. This update function relativizes the current absolute position of the FOB and then applies the translational scale that may have been set by the function *WTsensor_setsensitivity*. Once the record has been relativized and scaled it is stored in the sensor object's record. Note: If you design and implement your own update function, you must relativize and scale the records yourself.

WTflock_getorgmat

```
FLAG WTflock_getorgmat(  
    WTsensord *sensor,  
    WTm3 mat);
```

This function is used to retrieve the FOB sensor's original position/orientation matrix and place it in mat. This matrix, which is set at FOB initialization, is updated only upon subsequent calls to *WTflock_resetorigin*. The function returns TRUE if successful.

WTflock_getlastmat

```
FLAG WTflock_getlastmat(  
    WTsensord *sensor,  
    WTm3 mat);
```

This function is used to copy the FOB's last stored absolute sensor matrix into mat. When using the *WTflock_update* function, each iteration of the simulation loop updates the FOB's absolute position/orientation matrix with the current, unrelativized, unscaled position/orientation matrix excepting an error case. The function returns TRUE if successful.

WTflock_getlastpos

```
FLAG WTflock_getlastpos(  
    WTsensord *sensor,  
    WTp3 p);
```

This function retrieves the FOB's last stored absolute position and places it in p. When using the *WTflock_update* function, each iteration of the simulation loop updates the FOB's absolute position record with the current, unrelativized, unscaled position/orientation matrix excepting an error case. The function returns TRUE if successful.

Wtflock_getabsmat

```
FLAG Wtflock_getabsmat(  
    WTsensord *sensor,  
    WTm3 mat);
```

This function retrieves the FOB's current absolute position/orientation matrix and stores it in mat. This function will always contain a valid matrix for you to use as during an error case it is not updated. This function returns TRUE if successful.

Wtflock_getabspos

```
FLAG Wtflock_getabspos(  
    WTsensord *sensor,  
    WTp3 p);
```

This function is used to obtain the FOB's current unreletavized, absolute, and unscaled position and store it in p. The WTp3 returned should always be valid as during an error case, the last good matrix is the current matrix from which the position will be obtained. This function returns TRUE if successful.

Wtflock_getabsoluterecord

```
FLAG Wtflock_getabsoluterecord(  
    WTsensord *sensor,  
    WTp3 p,  
    WTq q);
```

This function retrieves the last absolute, unreletavized, unscaled position and orientation values stored in the sensor object's record. This function returns TRUE if successful.

Default sensor creation macro for FOB

```
#define WTflock_new(port,unit) \  
  
WTsensor_new(WTflock_open,WTflock_close,WTflock_update, \  
WTflock_deviceopen(port,38400,'N',8,1,2600),unit,WTSENSOR_DEFAULT)
```

Ascension Flock of Birds

Ascension Technology created Flock of Birds to allow users to attach multiple Birds and communicate with them over a single serial port.

Support for a Flock of Birds is already part of the Bird sensor driver (see page 13-39); the macro *WTbird_new* takes a *unit* argument. With a single Bird, *unit* is always 1, but with a Flock, *unit* specifies which Bird to address. You should always open the birds of a flock in sequential order, like this:

```
WTsensor *b1, *b2, *b3;  
b1 = WTbird_new(SERIAL1, 1);  
b2 = WTbird_new(SERIAL1, 2);  
b3 = WTbird_new(SERIAL1, 3);
```

This creates a flock having 3 birds running at 9600 baud. See page 13-39 for the DIP switch settings. There can be a maximum of 14 birds in a flock.

Additional WTK functions for the Bird—for example, for using the Extended Range Controller or setting the Bird hemisphere—are described in *Ascension Extended Range Bird* below, and *Setting the Bird Hemisphere* on page 13-43.

Ascension Extended Range Bird

The Bird sensors normally have a range of up to 3 feet away from the transmitter. Ascension offers an extended range transmitter (ERT) allowing a range of 8 feet. The ERT works with a special Bird controller called the Extended Range Controller (ERC). The ERC will be the first unit in your bird arrangement. The ERC's serial port is used to communicate

with the whole flock. See your ERC documentation for how to connect the ERC to the other birds in your flock.

When you use the Extended Range Bird, you must call the function *WTercbird_new* instead of *WTbird_new* to initialize a bird. Before you initialize the bird, WTK needs to know which serial port the bird is connected to. You must use the *WTercbird_init* macro (with the port number) to pass this information to WTK.

You must use consecutive addresses (starting at 1) to identify the birds. The Extended Range Controller (ERC) does not have to be the master unit, and is therefore not restricted to an address of 1. For example, if you have the ERC at address 3, and two bird receivers at address 1 and 2 respectively the initialization function calls should be:

```
/* inform WTK that you are using an ERC at port X */
WTercbird_init( SERIAL_X );
/* initialize the receivers at addresses 1 and 2 */
WTercbird_new( SERIAL_X, 1 );
WTercbird_new( SERIAL_X, 2 );
```

If instead, the ERC is set to address 1, the initialization function calls would be:

```
WTercbird_init( SERIAL_X );
WTercbird_new( SERIAL_X, 2 );
WTercbird_new( SERIAL_X, 3 );
```

You may notice that the WTK sensor unit numbers correspond to the bird addresses. As with the regular bird, you must call *WTercbird_new* for all the receiver units in the system, even if you are not using them.

Accessing Extended Range Bird Raw Data

See *Accessing Bird Raw Data* on page 13-41.

Scaling Extended Range Bird Records

See *Scaling Bird Records* on page 13-41.

Extended Range Bird Update Function

See *Bird Update Function* on page 13-42.

CIS Graphics Geometry Ball, Jr.

The CIS Graphics Geometry Ball, Jr. is a 6 degree-of-freedom serial port device that sits on the desktop. It responds to both forces and torques, which can be mapped into translations and rotations in 3D.

To create a Geometry Ball, Jr. sensor object on serial port 1, you can use the macro call:

```
WTsensor *geoball;  
geoball = WTgeoball_new(SERIAL1);
```

This macro makes use of the sensor driver functions *WTgeoball_open*, *WTgeoball_close*, and *WTgeoball_update*. It creates the geoball sensor object running at 9600 baud.

The coordinate frame of this sensor is the same as for the Spaceball and is defined in the WTK driver functions as follows. If the device is placed on a desk or table in front of you with the cable coming out the back of the device oriented away from you, then, (as illustrated in figure 13-6 on page 13-100 for the Spaceball) the Z axis of the device points straight ahead, the X axis points to the right, and the Y axis points down. If this coordinate frame is not appropriate for your application, the function *WTsensor_rotate* (see page 13-20) can be used to define the device's coordinate frame.

Accessing Geometry Ball, Jr. Raw Data

WTK maintains a data structure containing the raw data read from the Geometry Ball, Jr. This information is accessed using *WTsensor_getrawdata* (see page 13-15) as in the example below.

The raw data structure for the Geometry Ball, Jr. is type defined as follows. Note that both *p* and *w* are in the original Geometry Ball, Jr. coordinates and that no scale factors have been applied to the values.

```
typedef struct _WTgeoball_rawdata {
    char p[3];      /* absolute position */
    char w[3];      /* euler angles */
} WTgeoball_rawdata;
```

Geometry Ball, Jr. raw data is accessed as follows:

```
WTsensor *geoball;
WTgeoball_rawdata *raw;

raw = (WTgeoball_rawdata *)WTsensor_getrawdata(geoball);
WTmessage("Position: %c, %c, %c\n", raw->p[X], raw->p[Y], raw->p[Z]);
WTmessage("Angles: %c, %c, %c\n", raw->w[X], raw->w[Y], raw->w[Z]);
```

Scaling Geometry Ball, Jr. Records

Translational and rotational records for the Geometry Ball, Jr. can be scaled using the functions *WTsensor_setsensitivity* (see page 13-11) and *WTsensor_setangularrate* (see page 13-12) respectively.

Geometry Ball, Jr. Update Function

WTgeoball_update

```
void WTgeoball_update(
    WTsensor * sensor);
```

This update function packages the translation and rotation record from the device into the sensor object's record after transforming the record to the WTK coordinate convention, and applying any scale factors that may have been set with *WTsensor_setsensitivity* (see page 13-11) and *WTsensor_setangularrate* (see page 13-12).

The macro *WTgeoball_new* creates a Geometry Ball, Jr. sensor object that uses the function *WTgeoball_update* and is recommended for most users.

See *Example 2: Driver for the Geometry Ball Jr.* on page E-10.

Geometry Ball, Jr. Defined Constants

Button presses for a Geometry Ball, Jr. can be detected using *WTsensor_getmiscdata* (see page 13-15) together with the following defined constants:

- **Button transitioned down.** This event is generated each time the button moves from up to down. These events are defined as: *WTGEOBALL_LEFTBUTTON* and *WTGEOBALL_RIGHTBUTTON*.

Checking the Serial Port for Geometry Ball, Jr.

WTgeoball_present

```
FLAG WTgeoball_present(  
    WTserial * serial);
```

This function checks whether there is a live Geometry Ball, Jr. sensor object at the particular serial port corresponding to the serial port object (which is given as input).

Fakespace BOOM Devices

The monochrome BOOM, two-color BOOM2C (which uses a color space based on red and green), and full-color BOOM3C by Fakespace are CRT-based stereoscopic viewing devices attached to a 6 degree-of-freedom articulated arm. The six joints in the arm permit the BOOM viewer to be moved and oriented within a sphere of about six feet in radius.

WTK's BOOM sensor driver functions *WTboom_open*, *WTboom_close*, and *WTboom_update* manage the serial port communications with the BOOM to generate correct position and orientation records based on the configuration of the BOOM arm.

To create a BOOM sensor object on serial port 1, you can use the macro call:

```
WTsensor *boom;  
WTmessage("About to create BOOM sensor ...\\n");  
boom = WTboom_new(SERIAL1);
```

```
if ( !boom )
    WTwarning("Couldn't open BOOM\n");
else
    WTmessage("Calibration complete.\n");
```

This macro makes use of the sensor driver functions *WTboom_open*, *WTboom_close*, and *WTboom_update* to generate correct position and orientation record based on the configuration of the BOOM arm. It creates the BOOM sensor object running at 9600 baud.

Accessing BOOM Raw Data

WTK maintains a data structure containing the raw data read from the BOOM device. The information is accessed using the function *WTsensor_getrawdata* (see page 13-15). The BOOM raw data structure stores the angular readings from the BOOM arm.

The raw data structure for the BOOM is type defined as follows:

```
typedef struct _WTboom_rawdata {
    short angles[6];
} WTboom_rawdata;
```

and is accessed as follows:

```
WTboom_rawdata *raw;
raw = (WTboom_rawdata *)WTsensor_getrawdata(boom)
```

Scaling BOOM Records

The sensitivity of the BOOM (as set by *WTsensor_setsensitivity* on page 13-11) determines the magnitude of the translation record generated by motion of the BOOM arm. Therefore by setting the sensor sensitivity, you can affect the apparent range of motion within the virtual environment. There is no angular adjustment for the BOOM (i.e., the function *WTsensor_setangularrate* has no effect).

BOOM Update Function

WTboom_update

```
void WTboom_update(  
    WTsensor *sensor);
```

This update function reads the raw angular data from the BOOM arm into the angles array in the *WTboom_rawdata* structure. *angles[0]* refers to the angle of the first joint above the base. Refer to the BOOM documentation for more information about interpreting the data read.

The macro *WTboom_new* creates a BOOM sensor object that uses the *WTboom_update* function and is recommended for most users.

BOOM Defined Constants

The BOOM also supports two push-button switches mounted on the ends of the grip handles. Events from these functions can be accessed using the function *WTsensor_getmiscdata* to determine whether the BOOM buttons are currently pressed:

- **Button held down.** This event is generated each frame that the button is held down. These events are defined as: *WTBOOM_LEFTBUTTON*, and *WTBOOM_RIGHTBUTTON*.

Following is an example of accessing BOOM button events to drive motion through the virtual world:

```
WTsensor *boom;  
WTviewpoint *view;  
WTp3 trans;  
  
/* Obtain the viewpoint direction vector and scale it to a fraction of the  
size of the universe being navigated. */  
view = WTuniverse_getviewpoints();  
WTviewpoint_getdirection(view, trans);  
WTp3_mults(trans, WTnode_getradius(WTuniverse_getrootnodes()));
```

```
/* If the left BOOM button is pressed, translate the viewpoint
along the viewpoint direction */
if ( WTsensor_getmiscdata(boom)&WTBOOM_LEFTBUTTON )
    WTviewpoint_translate(view, trans, WTFRAME_WORLD);

/* Use the right BOOM button to move backwards */
if ( WTsensor_getmiscdata(boom)&WTBOOM_RIGHTBUTTON ) {
    WTp3_invert(trans, trans);
    WTviewpoint_translate(view, trans, WTFRAME_WORLD);
}
```

The defined constants used with BOOMs configured with a joystick are described in the following section.

BOOM Joystick

Some BOOMs are configured with a joystick rather than buttons built into the handle. Use the macro *WTboom_newjoystick* (which references the update function *WTboom_joystickupdate*) for instantiating such a BOOM sensor object, rather than *WTboom_new*.

The defined constants *WTBOOM_LEFT*, *WTBOOM_RIGHT*, *WTBOOM_UP*, *WTBOOM_DOWN* are provided for use with the function *WTsensor_getmiscdata* (see page 13-15) to determine the current state of the joystick. In addition, the defined constant *WTBOOM_RESET* can be used to determine whether the reset button has been pressed. The joystick has nine possible positions that can be determined by calling *WTsensor_getmiscdata*. These positions are: up, down, right, left, up and to the right, up and to the left, down and to the right, down and to the left, and centered (i.e., none of the above). For example, to test the various possibilities, your code might be structured as follows:

```
WTboom *boom;
int data;

data = WTsensor_getmiscdata(boom);
if ( data&WTBOOM_LEFT ) {
    if ( data&WTBOOM_UP )
        WTmessage("joystick is up and to the left\n");
}
```

```
    else if ( data&WTBOOM_DOWN )
        WTmessage("joystick is down and to the left\n");
    else
        WTmessage("joystick is to the left\n");
}
else if ( data&WTBOOM_RIGHT ) {
    if ( data&WTBOOM_UP )
        WTmessage("joystick is up and to the right\n");
    else if ( data&WTBOOM_DOWN )
        WTmessage("joystick is down and to the right\n");
    else
        WTmessage("joystick is to the right\n");
}
else if ( data&WTBOOM_UP )
    WTmessage("joystick is up\n");
else if ( data&WTBOOM_DOWN )
    WTmessage("joystick is down\n");
else
    WTmessage("joystick is centered\n");
```

Fakespace Pinch Glove System

The Pinch Glove provides a way of recognizing natural gestures that have natural meaning to the user. For example a pinching gesture can be used to grab a virtual object.

To create a Pinch Glove sensor object on serial port 1 running at 9600 baud (default baud rate setting), you can use the macro call:

```
WTsensor *pinch;
pinch = WTPinch_new(SERIAL1, 9600);
```

This macro makes use of the sensor driver functions *WTPinch_open*, *WTPinch_close*, and *WTPinch_update*.

The right-most 3 switches are for setting baud rates. Following are the DIP switch settings (for the right-most 3 switches) for the Pinch Glove running at 9600 baud:

Pinch Glove OFF OFF ON

Consult your Pinch Glove reference manual if you are uncertain of how to set the DIP switches.

Accessing Pinch Glove Raw Data

The raw data structure for the Pinch Glove is type defined as follows:

```
typedef struct _WTPinch_rawdata {
    int ntouch;           /* number of touches - maximum 5 */
    short int touch[5];  /* information for each touch */
} WTPinch_rawdata;
```

and accessed as follows:

```
WTSensor *pinch;
WTPinch_rawdata *raw;
short rp,rr,rm,ri,rt,lp,lr,lm,li,lt;
int i;

raw = (WTPinch_rawdata *)WTSensor_getrawdata(pinch);

/* print number of touches */
WTmessage("Number of touches %d\n", raw->ntouch);

/* for each touch print out what fingers touched */
for( i=0;i<raw->ntouch;i++) {
    rp = raw->touch[i]&WTPINCH_RPINKIE;
    rr = raw->touch[i]&WTPINCH_RRING;
    rm = raw->touch[i]&WTPINCH_RMIDDLE;
    ri = raw->touch[i]&WTPINCH_RINDEX;
    rt = raw->touch[i]&WTPINCH_RTHUMB;
    lp = raw->touch[i]&WTPINCH_LPINKIE;
    lr = raw->touch[i]&WTPINCH_LRING;
    lm = raw->touch[i]&WTPINCH_LMIDDLE;
    li = raw->touch[i]&WTPINCH_LINDEX;
```



```
lt = raw->touch[i]&WTPINCH_LTHUMB;
WTmessage("Touch number %d \n", i+1);
if(rp) WTmessage("rpinkie ");
if(rr) WTmessage("rring ");
if(rm) WTmessage("rmiddle ");
if(ri) WTmessage("rindex ");
if(rt) WTmessage("rthumb ");
if(lp) WTmessage("lpinkie ");
if(lr) WTmessage("lring ");
if(lm) WTmessage("lmiddle ");
if(li) WTmessage("lindex ");
if(lt) WTmessage("lthumb ");
WTmessage("\n");
}
```

Scaling Pinch Glove Records

Records cannot be scaled for this sensor object. So the functions *WTsensor_setsensitivity* and *WTsensor_setangularrate* have no effect.

Pinch Glove Update Function

WTpinch_update

```
void WTpinch_update(
    WTsensor *sensor);
```

This update function updates the raw data structure to get contact information between fingers.

The macro *WTpinch_new* creates a Pinch Glove sensor object that uses the *WTpinch_update* function and is recommended for most users.

Pinch Glove Defined Constants

The contact information between fingers of the Pinch Glove can be accessed using the function *WTSensor_getrawdata* (see page 13-15). Also see *Accessing Pinch Glove Raw Data* on page 13-60.

WTK provides defined constants that are common to both left and right fingers as well individual constants for left and right fingers.

The common constants are:

- WTPINCH_THUMB (thumb)
- WTPINCH_INDEX (index finger)
- WTPINCH_MIDDLE (middle finger)
- WTPINCH_RING (ring finger)
- WTPINCH_PINKY (pinky finger)

The individual constants for the left hand fingers are:

- WTPINCH_LTHUMB
- WTPINCH_LINDEX
- WTPINCH_LMIDDLE
- WTPINCH_LRING
- WTPINCH_LPINKY

The individual constants for the right hand fingers are:

- WTPINCH_RTHUMB
- WTPINCH_RINDEX
- WTPINCH_RMIDDLE
- WTPINCH_RRING
- WTPINCH_RPINKY

In addition, the defined constants *WTPINCH_NOTOUCH* (indicating there is no contact between the fingers) and *WTPINCH_FINGERS* (indicating there is contact between fingers) are provided.

Fifth Dimension Technologies' 5DT Glove

The 5DT Glove measures finger flexure and the orientation (roll and pitch) of a user's hand. It can emulate a Mouse as well as a baseless joystick and the user can also type while wearing the glove.

To create a 5DT Glove sensor object on serial port 1, you can use the macro call:

```
WTsensor *glove5DT;  
glove5DT = WTglove5DT_new(SERIAL1);
```

This macro makes use of the sensor driver functions *WTglove5DT_open*, *WTglove5DT_close*, and *WTglove5DT_update*. It creates the 5DT Glove sensor object running at 19200 baud.

Accessing 5DT Glove Raw Data

The raw data structure for the 5DT Glove is type defined as follows:

```
typedef struct _WTglove5DT_rawdata  
{  
    char bSerial;  
    char bFinger[5]; /* finger flex values 0=Thumb 1=Index ...*/  
    char bPitch;    /* -128 to +128, 0 being straight up*/  
    char bRoll;     /* -128 to +128, 0 being straight up*/  
} WTglove5DT_rawdata;
```

and is accessed as follows:

```
WTsensor *glove5DT;  
WTglove5DT_rawdata *raw;  
raw = (WTglove5DT_rawdata *)WTsensor_getrawdata(glove5DT);  
  
WTmessage ("PITCH: %d\tROLL: %d\tFINGERS: %d %d %d %d %d\n", raw->bPitch,  
            raw->bRoll,  
            raw->bFinger[0],  
            raw->bFinger[1],
```

```
raw->bFinger[2],
raw->bFinger[3],
raw->bFinger[4]
);
```

Scaling 5DT Glove Records

Records cannot be scaled for this sensor object. So, the functions *WTsensor_setsensitivity* and *WTsensor_setangularrate* have no effect.

5DT Glove Update Function

WTglove5DT_update

```
void WTglove5DT_update(
    WTsensor *sensor);
```

This update function calls *WTglove5dt_rawupdate* (see page 13-65) to update the raw data structure to get absolute orientation and the current state of the fingers in the hand model. It then applies any rotational constraints to the record. Finally it relativizes the record and stores it with the sensor by calling *WTsensor_setrecord* (see page 13-24).

The macro *WTglove5DT_new* creates a 5DT Glove sensor object that uses the *WTglove5DT_update* function.

WTglove5dt_updatefingers

```
void WTglove5dt_updatefingers(
    WTsensor *sensor);
```

WTglove5dt_updatefingers is provided so that the current state of the fingers in the hand model can be updated independently of the model's orientation.

This function can be used instead of *WTglove5DT_update* if the user does not need the orientation information.

Writing Your Own 5DT Glove Update Function

Your update function should first call *WTglove5DT_rawupdate* (see below) to obtain the absolute pitch and roll data and the finger flex information. It should then specify how the raw data is to be transformed into an orientation record. Finally, your update function must store this record by calling *WTsensor_setrecord* (see page 13-24). See *Example 3: Update Function for Absolute Device (Pseudocode)* on page E-15.

WTglove5dt_rawupdate

```
int WTglove5dt_rawupdate(  
    WTsensor *sensor);
```

This function obtains the absolute pitch and roll data and the finger flex information and stores it in the sensor's raw data structure (*WTglove5DT_rawdata*). This information can be accessed with *WTsensor_getrawdata* (see page 13-15). Also see *Accessing 5DT Glove Raw Data* on page 13-63.

5DT Glove Defined Constants

The current state of the fingers can be accessed using the function *WTsensor_getmiscdata* (see page 13-15) together with the following defined constants:

```
WTGLOVE5DT_OPEN (glove is open)  
WTGLOVE5DT_CLOSED (glove is closed)  
WTGLOVE5DT_THUMB (thumb)  
WTGLOVE5DT_INDEX (index finger)  
WTGLOVE5DT_MIDDLE (middle finger)  
WTGLOVE5DT_RING (ring finger)  
WTGLOVE5DT_PINKY (pinky finger)  
WTGLOVE5DT_ALL (all fingers).
```

Calibrating the 5DT Glove

WTglove5dt_calibrateopen

```
int WTglove5dt_calibrateopen(  
    WTsensor *sensor);
```

This function allows you to set the state and orientation of the glove, which makes up the “open” state. Calibrating the closed position along with the open position allows miscellaneous data to provide an accurate reading of the glove states (i.e., *WTGLOVE5DT_OPEN*, *WTGLOVE5DT_CLOSED*).

WTglove5dt_calibrateclosed

```
void WTglove5dt_calibrateclosed(  
    WTsensor *sensor);
```

This function allows you to set the state and orientation of the glove, which makes up the “closed” state. Calibrating the open position along with the closed position allows miscellaneous data to provide an accurate reading of the glove states (i.e., *WTGLOVE5DT_OPEN*, *WTGLOVE5DT_CLOSED*).

Changing the Hand Model of the 5DT Glove

WTglove5dt_loadhandmodel

```
int WTglove5dt_loadhandmodel(  
    WTsensor *sensor,  
    char *filename,  
    float scale);
```

This function lets you change the hand model while a simulation is running. The name and scale of the model file are specified by *filename* and *scale*, respectively.

Note: The default hand model loaded when the sensor is first initialized is "hand5DT.nff".

Gameport Joystick

Limitations

- Only 2 axes Gameport Joysticks with up to 4 buttons are supported.
- You can only have one joystick attached to the system and it must be on port 1. (This is a limitation of the current NT joystick driver.)

Installing the joystick driver under NT

You must have the NT system driver for the gameport joystick installed to use it under WorldToolKit. If you have not previously installed one before or are unsure, follow the steps below to add the driver to your system.

1. Insert the NT 4.0 CD into your drive.
2. Open the control panel
3. Select 'multimedia'
4. Select 'devices'
5. Select 'add'
6. Choose 'unlisted or updated driver' and press 'ok'
7. Type in 'd:\drvlib\multimed\joystick\x86' where d is the letter of your CD-ROM drive
8. Restart system when prompted

Configuring and calibrating the joystick

WorldToolKit uses the standard NT joystick control panel to calibrate the gameport joystick. You must calibrate before you use a joystick for the first time, and any time your joystick is not behaving correctly. To calibrate your joystick:

1. Open the control panel
2. Select 'joystick'
3. Select the attributes that reflect your joystick
4. Choose 'calibrate' and follow directions
5. Choose 'test' to verify your calibration

Creating a Gameport Joystick Sensor Object

To create a Gameport Joystick sensor object on serial port 1, you can use the macro call:

```
WTsensor *joystick;  
joystick = WTjoystick_new(SERIAL1);  
  
( if !joystick )  
    WTwarning("Could not open gameport joystick\n");
```

This macro makes use of the sensor driver functions *WTjoystick_open*, *WTjoystick_close*, and *WTjoystick_walk*. It creates the Gameport Joystick sensor object running at 19200 baud.

At initialization, WTK searches the current directory for a joystick calibration file named *ajoy.cal*. The calibration file is in ASCII format with six values specifying floating point values for minimum X, maximum X, minimum Y, maximum Y, center X and center Y, respectively. This is a sample calibration file and also represents the default values used by WTK:

```
0.0 255.0 0.0 255.0 128.0 128.0
```

If the calibration file is not found, default values are used for the center and range values of the joystick.

To use an update function other than *WTjoystick_walk*, for example, *WTjoystick_fly*, you can call *WTsensor_new* directly or simply make the following call after using *WTjoystick_new*:

```
WTsensor_setupdatefn( joystick, WTjoystick_fly );
```


Accessing Gameport Joystick Raw Data

WTK maintains a data structure containing the raw data from the Gameport Joystick. This information can be accessed using the function *WTsensor_getrawdata* (see page 13-15) as in the example below. The raw data structure for the Gameport Joystick is type defined as follows:

```
typedef struct _WTjoystick_rawdata
{
    unsigned short x;
    unsigned short y;
} WTjoystick_rawdata;
```

and is accessed as follows :

```
WTsensor *joystick;
WTjoystick_rawdata *raw;
raw = (WTjoystick_rawdata *)WTsensor_getrawdata(joystick);

WTmessage("Roll %d Pitch %d\n", raw->x, raw->y);
```

Scaling Gameport Joystick Records

Translational and rotational records for the Gameport Joystick can be scaled using the functions *WTsensor_setsensitivity* (see page 13-11) and *WTsensor_setangularrate* (see page 13-12) respectively.

Gameport Joystick Update Functions

The WTK update functions for the Gameport Joystick store the position record from the device into the sensor object's record, after applying any scale factors set with *WTsensor_setsensitivity* (see page 13-11) and *WTsensor_setangularrate* (see page 13-12).

WTjoystick_fly

```
void WTjoystick_fly(  
    WTsensor *sensor);
```

This function is an update function that moves a sensor forward along the Z-axis at a constant velocity. It can be used to operate the joystick in a manner familiar to users of flight simulation programs. When this update function is used, the sensor moves forward along Z at a small constant velocity (0.1 times the sensor sensitivity each frame). Moving the joystick forward/backward pitches around X, and moving the joystick right/left rolls around Z.

The macro *WTjoystick_newfly* creates a gameport joystick object that uses the *WTjoystick_fly* update function.

WTjoystick_walk

```
void WTjoystick_walk(  
    WTsensor *sensor);
```

This function initializes a sensor to move in the “walkthrough” mode. The joystick can be used to move a viewpoint or object in the X-Z plane. When no buttons are pressed, moving the joystick forward or backward moves forward or backward along the Z axis. Moving the joystick right or left yaws around the Y axis. When the front button is depressed, moving the joystick forward or backward pitches around the X axis, and moving the joystick right or left rolls around the Z axis.

The macro *WTjoystick_new* creates a Gameport Joystick object that uses the *WTjoystick_walk* update function.

WTjoystick_walk2

```
void WTjoystick_walk2(  
    WTsensor *sensor);
```

This function initializes a sensor to move in a second “walkthrough” mode. The *WTjoystick_walk2* update function is like *WTjoystick_walk* except that holding down the trigger button allows you to raise or lower the viewpoint.

The macro *WTjoystick_newwalk2* creates a gameport joystick object that uses the *WTjoystick_walk2* update function.

Writing your Own Gameport Joystick Update Function

Your update function should first call *WTjoyserial_rawupdate* to obtain the Gameport Joystick's raw data. It should then specify how the raw data is to be transformed into 3D position record. Finally, your update function must store this record by calling *WTsensor_setrecord* (see page 13-24). See *Example 3: Update Function for Absolute Device (Pseudocode)* on page E-15.

WTjoystick_rawupdate

```
int WTjoystick_rawupdate(  
    WTsensor *sensor);
```

This function reads in the raw data from the Gameport Joystick and stores it in the sensor's raw data structure. This information can be accessed with the function *WTsensor_getrawdata* (see page 13-15). Also see *Accessing Gameport Joystick Raw Data* on page 13-69

Gameport Joystick Defined Constants

The Gameport Joystick supports three momentary buttons in addition to the trigger. These values can be accessed by using the *WTsensor_getmiscdata* function with any of the following constants described in the *joystick.h* file in the *include* directory:

```
WTJOYSTICK_TRIGGERDOWN  
WTJOYSTICK_TOPDOWN  
WTJOYSTICK_BUTTON1DOWN  
WTJOYSTICK_BUTTON2DOWN  
WTJOYSTICK_BUTTONNORM  
WTJOYSTICK_BUTTONREVERSE
```

Gameport Joystick Range

WTjoystick_getrange

```
void WTjoystick_getrange(  
    WTsensord *sensor,  
    WTp2 range);
```

This function returns the maximum X and Y values (divided by 2), which can be attained by the joystick.

Gameport Joystick Drift

WTjoystick_setdrift

```
void WTjoystick_setdrift(  
    WTsensord *sensor,  
    float drift_per);
```

This function sets the joystick's drift amount to a percentage of the joystick's range. The *drift_per* parameter specifies the percentage.

WTjoystick_getdrift

```
float WTjoystick_getdrift(  
    WTsensor *sensor);
```

This function returns the drift amount of the specified joystick sensor. See *WTjoystick_setdrift*, above.

Reinitializing the Gameport Joystick

WTjoystick_readcalibrationfile

```
void WTjoystick_readcalibrationfile(  
    void);
```

This function reads the joystick's calibration file (*ajoy.cal*) so that the joystick can be re-initialized. Page 13-68 describes the *ajoy.cal* file.

Logitech 3D Mouse (Red Baron)

An early version of the Logitech 3D Mouse went by the name “Red Baron,” and WTK adopted this name for the sensor driver functions for this device. The Logitech 3D Mouse has two modes of operation. When it is on the desk surface, it functions in a manner very similar to a 2D Mouse, with asynchronous cursor tracking. (Cursor tracking with the Logitech 3D Mouse is not supported on all platforms.) When the device is lifted from the desk, it is driven by software in a mode that tracks 3D positions and orientations. In this mode it is used for direct manipulation of viewpoints or objects in the same way as other position and orientation sensing devices.

To create a 3D Mouse sensor object on serial port 1, you can use the macro call:

```
WTsensor *baron;  
baron = WTbaron_new(SERIAL1);
```

This macro makes use of the sensor driver functions *WTbaron_open*, *WTbaron_close*, and *WTbaron_update*. It creates the 3D Mouse sensor object running at 9600 baud on UNIX platforms and running at 1200 baud on Windows 32-bit platforms.

At initialization, both the transmitter triangle and Mouse should be on the desk surface. Unlike for the head tracker, where the transmitter and receiver triangles point in opposite directions, for desk-based operation the configuration at initialization is for the two triangles to point in the same direction.

When the side button (suspend button) on the 3D Mouse is depressed, position and orientation records for the sensor are frozen at their current values, until the button is released. In this manner, the button can be used as a “clutch” or “ratchet” to be able to traverse large distances or angles by depressing the button while returning the sensor to within range of the ultrasonic speakers.

Accessing 3D Mouse Raw Data

WTK maintains a data structure containing the raw data read from the 3D Mouse. This information is accessed using the function *WTsensor_getrawdata* (see page 13-15) as in the example below.

The raw data structure for the 3D Mouse is type defined as follows:

```
typedef struct _WTbaron_rawdata {
    WTP3 p;          /* absolute position in WTK coordinates*/
    WTP3 w;          /* euler angles in WTK coords, in degrees */
    float x,y;      /* desk-based mouse-like rawdata */
} WTbaron_rawdata;
```

While the values stored in *p* and *w* are updated each frame, the X, Y raw data values are only updated when the unit is on the desk. When the device has been in flying mode (as indicated by *WTLOGITECH_FLYING*) and then is returned to the desk, the X, Y values are re-initialized to the middle of the screen or window.

3D Mouse raw data is accessed as follows:

```
WTsensor *baron;
WTbaron_rawdata *raw;
```

```
/* get raw baron values */
raw = (WTbaron_rawdata *)WTsensor_getrawdata(baron);

/* print position and orientation data if in flying mode */
if ( WTsensor_getmiscdata(baron) & WTLOGITECH_FLYING ) {
    WTp3_print(raw->p, "Baron position: ");
    WTp3_print(raw->w, "Baron angles: ");
}
else {
    WTmessage("x,y coordinates: %f %f\n",x, y);
}
```

Scaling 3D Mouse Records

As for the Logitech Head Tracker, the 3D Mouse translation records can be scaled using the function *WTsensor_setsensitivity* (see page 13-11).

There is no angular speed adjustment for the 3D Mouse, i.e., the function *WTsensor_setangularrate* has no effect.

3D Mouse Update Function

WTbaron_update

```
void WTbaron_update(  
    WTsensor *sensor);
```

This update function updates the raw data structure to get absolute 3D position and euler angles. It also updates the screen coordinates if the unit is on the desk. It then applies any translational constraints and scale factors. Finally, it relativizes the record and stores it with the sensor by calling *WTsensor_setrecord* (see page 13-24).

The macro *WTbaron_new* creates a 3D Mouse sensor object that uses the *WTbaron_update* function and is recommended for most users.

3D Mouse Defined Constants

The 3D Mouse has three buttons (left, middle, and right) similar to a normal Mouse. In addition, it has a button on the side of the Mouse body called the “suspend button”, so named because it is used to suspend motion when pressed (this is the “ratcheting” described above). Events from these buttons are accessed using the function *WTsensor_getmiscdata* (see page 13-15) together with the defined constants:

- **Button transitional down.** This generates a single event each time the button moves from up to down. These events are defined as:
WTLOGITECH_LEFTBUTTON, *WTLOGITECH_MIDDLEBUTTON*,
WTLOGITECH_PEDESTALBUTTON, *WTLOGITECH_RIGHTBUTTON*, and
WTLOGITECH_SUSPENDBUTTON.

In addition, the defined constants *WTLOGITECH_FLAGBIT* and *WTLOGITECH_FLYING* are provided. *WTLOGITECH_FLAGBIT* can be used to detect a bad record. *WTLOGITECH_FLYING* can be used to detect when the 3D Mouse is currently off the desktop, as in the example below. Note that this constant makes sense only if the Mouse was on the desk at initialization. The following code fragment illustrates using this flag to control the viewpoint with the tracker when it is in flying mode:

```
/* Logitech 3D Mouse starts off on the desk */  
FLAG baron_flying = FALSE;
```



```
/* this function tests whether the Logitech 3D Mouse is entering or leaving
 * 6D mode, and if so, attaches or detaches the sensor from the viewpoint. */
baron_control_view(WTsensor *baron) {
    WTviewpoint *view;
    view = WTuniverse_getviewpoints();

    /* If lifted off from desk, attach sensor to viewpoint.*/
    if ( WTsensor_getmiscdata(baron) & WTLOGITECH_FLYING
        && !baron_flying ) {
        WTviewpoint_addsensor(view, baron);
        baron_flying = TRUE;
    }
    /* Else if landing, remove sensor from viewpoint */
    else if ( !(WTsensor_getmiscdata(baron) &
        WTLOGITECH_FLYING ) && baron_flying ) {
        WTviewpoint_removesensor(view, baron);
        baron_flying = FALSE;
    }
}
```

Logitech Head Tracker

The Logitech Head Tracker from Logitech, Inc. is a serial port device that measures absolute position and orientation by using three microphones to triangulate on three ultrasonic speakers. The speakers are mounted in a large triangle, and the microphones are in a smaller triangle, which is either attached to the end of a special Logitech Mouse (see page 13-73) for desk-based use or is mounted on top of a head-mounted display for use as a head tracker. WTK provides sensor drivers for these two different physical configurations. In another configuration, the Head Tracker is built in to the StereoGraphics CrystalEyesVR combined viewing and tracking system. An additional driver is provided in WTK for this system (see page 13-108).

When the Head Tracker sensor moves out of range of the ultrasonic transmitters, records returned from the device are thresholded by WTK so that they cease to change until the sensor returns within range. The tracked area is approximately a two foot cube, with diminished accuracy within a seven foot cube. In using this device, remember that it

operates ultrasonically and therefore, unlike the magnetic ISOTRAK or Bird, the sensor microphones must be within line of sight of the transmitting speakers for stable operation.

To create a Head Tracker on serial port 1, you use the macro call:

```
WTsensor *logitech;  
WTmessage("About to create Logitech head tracker...\n");  
logitech = WTlogitech_new(SERIAL1);  
if ( !logitech )  
    WTwarning("Couldn't open Logitech\n");  
else  
    WTmessage("Calibration complete.\n");
```

This macro makes use of the sensor driver functions *WTlogitech_open*, *WTlogitech_close*, and *WTlogitech_update*. It creates the Head Tracker sensor object running at 19200 baud on UNIX platforms and running at 1200 baud on Windows 32-bit platforms.

At sensor initialization, the transmitter and receiver must be in a particular spatial relationship with one another for its subsequent operation to be correct. The correct orientation is to place both the helmet and transmitter triangles level so that the planes of the triangles are parallel and the transmitter and receiver units face each other (see figure 13-3 on page 13-79). The cables from the two triangles should point in opposite directions. If the transmitter and receiver triangles are not within range of each other at the time of sensor initialization, the *WTlogitech_open* function prompts you to move the receiver within range so that the initialization can be completed.

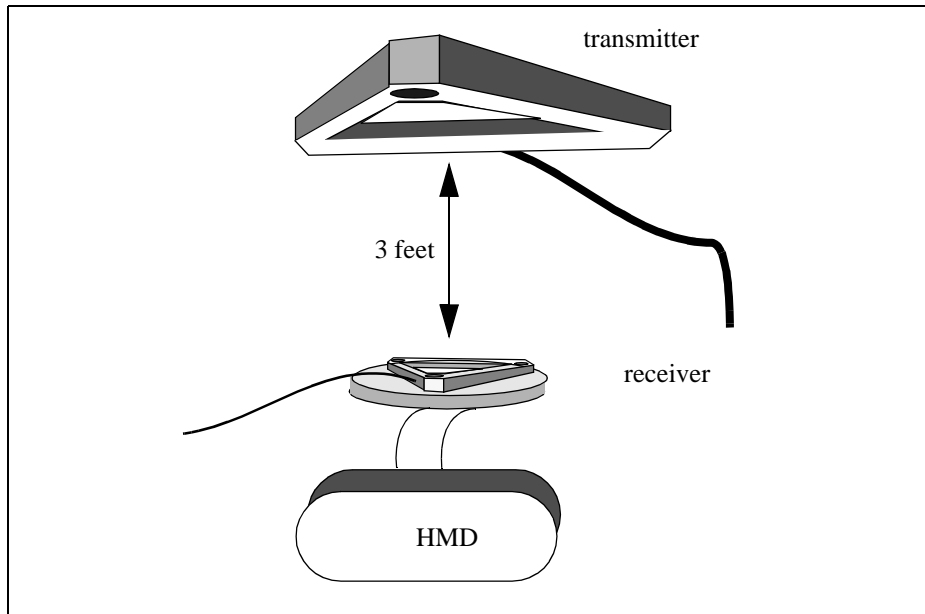


Figure 13-3: Setup for the Logitech Head Tracker.

Accessing Head Tracker Raw Data

WTK maintains a data structure containing the raw data read from the Head Tracker device. This information is accessed using the function *WTsensor_getrawdata* (see page 13-15) as in the example below.

The raw data structure for the Head Tracker is type defined as follows:

```
typedef struct _WTlogitech_rawdata {  
    WTp3 p;          /* absolute position in WTK coordinates */  
    WTp3 w;          /* euler angles in WTK coords, in degrees */  
} WTlogitech_rawdata;
```

and accessed as follows:

```
WTsensor *logitech;  
WTlogitech_rawdata *raw;  
  
/* print out the raw data */  
raw = (WTlogitech_rawdata *)WTsensor_getrawdata(logitech);  
WTP3_print(raw->p, "logitech raw position: ");  
WTP3_print(raw->w, "logitech raw angles: ");
```

Scaling Head Tracker Records

Translational records for the Head Tracker can be scaled using the function *WTsensor_setsensitivity* (see page 13-11).

There is no angular speed adjustment for the Head Tracker, i.e., the function *WTsensor_setangularrate* has no effect.

Head Tracker Update Function

WTlogitech_update

```
void WTlogitech_update(  
    WTsensor *sensor);
```

This update function updates the raw data structure to get absolute 3D position and euler angles. It then applies any translational constraints and scale factors. Finally, it relativizes the record and stores it with the sensor by calling *WTsensor_setrecord* (see page 13-24).

The macro *WTlogitech_new* creates a Head Tracker sensor object that uses the *WTlogitech_update* function and is recommended for most users.

Head Tracker Defined Constants

The defined constants *WTLOGITECH_FLAGBIT* (to detect bad records) and *WTLOGITECH_OUTBIT* (to detect whether the receiver is out of range of the transmitter) are provided and can be used with the function *WTsensor_getmiscdata* (see page 13-15).

Logitech Space Control Mouse (Magellan)

The Space Control Mouse, from Logitech, Inc. is a *six degrees-of-freedom* serial port device that sits on the desktop. It responds to both forces and torques, which can be mapped into translations and rotations in 3D. The WTK update functions package the translation and rotation record from the Space Control Mouse into the sensor object's record, after it transforms the record to the WTK coordinate convention and applies any scale factors set with *WTsensor_setsensitivity* and *WTsensor_setangularrate*.

To create a Space Control Mouse sensor object on serial port 1, you can use the macro call:

```
WTsensor *magellan;  
magellan = WTspacecontrol_new(SERIAL1);
```

This macro makes use of the sensor driver functions *WTspacecontrol_open*, *WTspacecontrol_close* and *WTspacecontrol_update*. It creates the Space Control Mouse sensor object running at 9600 baud.

The coordinate frame of these sensors is defined in the WTK driver functions as follows. If the device is placed on a desk or table in front of you with the cable coming out the back of the device pointing away from you, as illustrated in figure 13-4, then the Z axis of the device points straight ahead, the X axis points to the right, and the Y axis points down. If this coordinate frame is not appropriate for your application, the function *WTsensor_rotate* (see page 13-20) can be used to define the device's coordinate frame.

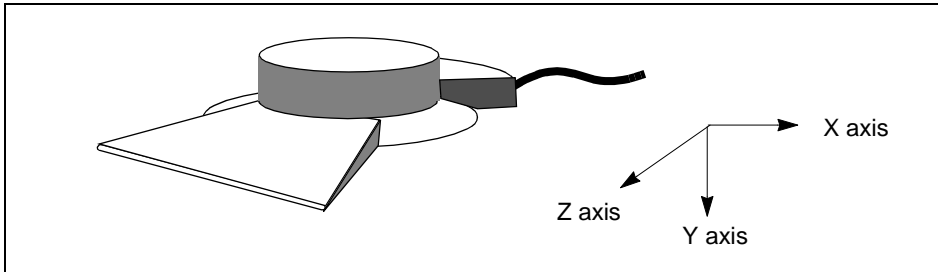


Figure 13-4: Logitech Space Control Mouse (Magellan) reference frame.

Accessing Space Control Mouse Raw Data

WTK maintains a data structure containing the raw data read from the Space Control Mouse. This information is accessed using `WTsensor_getrawdata` (see page 13-15) as in the following example.

The raw data structure for the Space Control Mouse is type defined as follows. Note that both p and w are in the original Space Control Mouse coordinates and that no scale factors or constraints have been applied to the values.

```
typedef struct _WTspacecontrol_rawdata {
    short p[3];      /* absolute position */
    short w[3];      /* euler angles */
} WTspacecontrol_rawdata;
```

Space Control Mouse raw data is accessed as follows:

```
WTsensor *spacecontrol;
WTspacecontrol_rawdata *raw;

/* get the raw spacecontrol values and print them out */
raw = (WTspacecontrol_rawdata *)WTsensor_getrawdata(spacecontrol);
WTmessage("Position: %d, %d, %d\n", raw->p[X], raw->p[Y], raw->p[Z]);
WTmessage("Angles: %d, %d, %d\n", raw->w[X], raw->w[Y], raw->w[Z]);
```

Scaling Space Control Mouse Records

Translational and rotational records for the Space Control Mouse can be scaled using the functions *WTsensor_setsensitivity* (see page 13-11) and *WTsensor_setangularrate* (see page 13-12) respectively.

Space Control Mouse Update Function

WTspacecontrol_update

```
void WTspacecontrol_update(  
    WTsensor *sensor);
```

This update function calls *WTspacecontrol_rawupdate* (see below) to update the raw data structure to get the 3D position and euler angle. It then applies any scale factors set with *WTsensor_setsensitivity* (see page 13-11) and *WTsensor_setangularrate* (see page 13-12). Finally it converts the euler to a quaternion and stores the record with the sensor by calling *WTsensor_setrecord* (see page 13-24).

The macro *WTspacecontrol_new* creates a Space Control Mouse sensor object that uses the *WTspacecontrol_update* function.

Writing your own Space Control Mouse Update Function

Your update function should first call *WTspacecontrol_rawupdate* (see below) to obtain the sensor's raw position and orientation. It should then specify how the raw data is to be transformed into 3D position and orientation record. Finally, your update function must store this record by calling *WTsensor_setrecord* (see page 13-24).

WTspacecontrol_rawupdate

```
void WTspacecontrol_rawupdate(  
    WTsensor *sensor);
```

This function obtains raw position, orientation, and button-press data for a Space Control Mouse device. *WTspacecontrol_rawupdate* should be called at the beginning of the sensor

user's update function to obtain the raw Space Control Mouse position, orientation and button-press data. It obtains the relative translation and orientation information from the Space Control Mouse and stores it in the sensor's raw data structure (*WTspacecontrol_rawdata*). This information can be accessed with *WTsensor_getrawdata* (see page 13-15). Also see *Accessing Space Control Mouse Raw Data* on page 13-82.

This function also reads the SpaceController button presses, which can be accessed with *WTsensor_getmiscdata* (see page 13-15). Also see *Space Control Mouse Defined Constants* below.

Space Control Mouse Defined Constants

There are nine user-programmable buttons on the Space Control Mouse. All of these are positioned on the top edge of the Space Control Mouse frame. The button marked with a "*" is called the "pick button" (to maintain compatibility with the Spaceball).

Events from these buttons can be accessed using *WTsensor_getmiscdata* (see page 13-15) together with the following defined constants:

- **Button held down.** This event is generated each frame that the button is held down. These events are defined as: *WTSPACECONTROL_BUTTONx* (where x is a number between 1 and 8. For example, *WTSPACECONTROL_BUTTON4*) and *WTSPACECONTROL_BUTTONA* (the "pick button").

Special Space Control Mouse Modes

As indicated in the Space Control Mouse documentation, you can set the device into a special "dominant" mode where only the largest of the six DOF values is returned. This makes the device easier to operate for new users. This and other control modes are accessed through a special combination of key presses. WTK should work fine with any of these settings.

Polhemus ISOTRAK

The ISOTRAK tracker from Polhemus, Inc. is an electromagnetic-based six degree-of-freedom sensor that measures absolute position and orientation.

To create an ISOTRAK sensor object on serial port 1, you can use the macro call:

```
WTsensor *isotrak;  
isotrak = WTPolhemus_new(SERIAL1);
```

This macro makes use of the sensor driver functions *WTPolhemus_open*, *WTPolhemus_close*, and *WTPolhemus_update*. It creates the ISOTRAK sensor object running at 9600 baud.

Following are the DIP switch settings for the ISOTRAK sensor objects running at 9600 baud.

ISOTRAK ON ON OFF ON OFF OFF OFF OFF

Consult your ISOTRAK reference manual if you are uncertain of how to set your ISOTRAK DIP switches.

When you call *WTPolhemus_new* to construct a new ISOTRAK sensor object, the *openfn* for the device is automatically called. Part of the function of the *openfn* for this device is to calibrate the sensor, which consists of obtaining an initial position and orientation record. This takes several seconds, during which the device should not be moved. Records subsequently generated by the *updatefn* are with respect to this initial reference frame. It may be useful in your application to let the user know that the device is about to be calibrated. For example, you might want to have a print statement:

```
WTsensor *sensor;  
WTmessage("About to calibrate/initialize ISOTRAK...\n");  
sensor = WTPolhemus_new(SERIAL1);  
WTmessage("Initialization complete.\n");
```

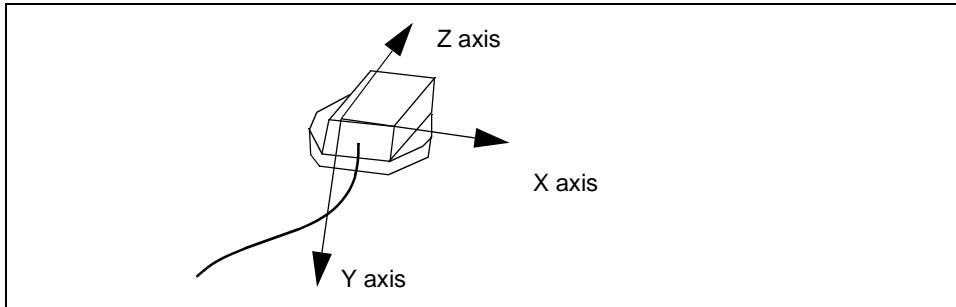


Figure 13-5: ISOTRAK sensor reference frame

The coordinate frame of this sensor is defined in the WTK driver functions as follows. If the receiver cube is placed “flat-end down” in front of you with the cable from the cube coming out the back of the cube toward you, then (as illustrated in figure 13-5) the Z axis of the device points straight ahead, the X axis points to the right, and the Y axis points down. If this coordinate frame is not appropriate for your application, the function *WTsensor_rotate* (see page 13-20) can be used to define another coordinate frame for the device.

Accessing ISOTRAK Raw Data

WTK does not provide a separate raw data structure for this device. The most recent sensor record can be obtained using *WTsensor_getlastrecord* (see page 13-25). This function retrieves the absolute record in WTK coordinates with no scale factors applied. This record is called “absolute” because it describes a location in 3D space rather than a change in location since the last frame. This absolute record is, however, relative to the position and orientation of the device when the device was opened by WTK.

Scaling ISOTRAK Records

Translation records for the ISOTRAK can be scaled using the function *WTsensor_setsensitivity* (see page 13-11). It is often useful, for example, to scale sensor inputs with the size of the scene.

Unlike translation records, however, orientation records from the ISOTRAK cannot be scaled in the WTK update function for this device. For example, if the ISOTRAK is used to track head motion (the sensor object is attached to the viewpoint), then a 360 degree rotation of the ISOTRAK device in the real world generates a 360 degree rotation in the virtual world.

It is possible to turn off all rotational input from this device by writing your own update function which nullifies the orientation record. The following is a simple update function that accomplishes this for the ISOTRAK:

```
/* update function which turns off all rotational input from the ISOTRAK */
void polhemus_myupdate(WTsensor *sensor)
{
    WTp3 p;
    WTq q;

    /* call the WTK-supplied update function */
    WTp3_update(sensor);

    /* use the translation record as is */
    WTsensor_gettranslation(sensor, p);

    /* nullify the orientation record */
    WTq_init(q);

    /* reset the ISOTRAK sensor record */
    WTsensor_setrecord(sensor, p, q);
}
```

This update function could be set as follows:

```
WTsensor_setupdatefn(sensor, polhemus_myupdate);
```

or by passing in *polhemus_myupdate* to *WTsensor_new*.

ISOTRAK Update Function

WTPolhemus_update

```
void WTPolhemus_update(  
    WTSensor * sensor);
```

This update function packages the translation and rotation record from the device into the sensor object's record after relativizing it and then applying any translational scale factor that may have been set with *WTSensor_setsensitivity* (see page 13-11). The macro *WTPolhemus_new* creates an ISOTRAK sensor object that uses the *WTPolhemus_update* function and is recommended for most users.

Polhemus ISOTRAK II

The ISOTRAK II tracker is very similar to the ISOTRAK (see page 13-85), except that it supports two sensors (receivers) instead of one. In fact, an ISOTRAK II with only one receiver attached operates exactly like an ISOTRAK, and from WTK you should treat it just like an ISOTRAK (however, use *WTPolhemus_new* rather than *WTisotrak2_new* if you are using just one receiver).

To create an ISOTRAK II sensor object having two receivers on serial port 1, you can use the macro call:

```
WTSensor *i1, *i2;  
i1 = WTisotrak2_new(SERIAL1, 1);  
i2 = WTisotrak2_new(SERIAL1, 2);
```

This macro makes use of the sensor driver functions *WTisotrak2_open*, *WTisotrak2_close*, and *WTisotrak2_update*. It creates the ISOTRAK II sensor object having two receivers running at 9600 baud.

The second argument to *WTisotrak2_new* is the unit number (1 or 2). Following are the DIP switch settings for the ISOTRAK II running at 9600 baud:

ISOTRAK II ON ON OFF ON OFF OFF OFF OFF

Consult your ISOTRAK II reference manual if you are uncertain of how to set your DIP switches. You should try to use the ISOTRAK II at 19200 baud as this may dramatically improve response time. To do so, change the DIP switches on the ISOTRAK II and change your *WTisotrak2_new* macro (in the *sensor.h* file in the *include* directory) to use the higher baud rate.

Following are the DIP switch settings for the ISOTRAK II running at 19200 baud:

ISOTRAK II OFF OFF ON ON OFF OFF OFF OFF

Consult your ISOTRAK II reference manual if you are uncertain of how to set your ISOTRAK II DIP switches.

When you call *WTisotrak2_new* for the first receiver (unit 1), the *openfn* for the device is called, which calibrates the sensor. Calibration consists of obtaining an initial position and orientation record, which takes several seconds, during which the device should not be moved. Records subsequently generated by the *updatefn* are with respect to this initial reference frame. As with other 6D sensors, it may be useful in your application to let the user know that the device is about to be calibrated (see the example under *Polhemus ISOTRAK* on page 13-85).

The coordinate frame of the ISOTRAK II is the same as for the ISOTRAK (see figure 13-5 on page 13-86).

Accessing ISOTRAK II Raw Data

See *Accessing ISOTRAK Raw Data* on page 13-86.

Scaling ISOTRAK II Records

See *Scaling ISOTRAK Records* on page 13-86.

ISOTRAK II Update Function

WTisotrak2_update

```
void WTisotrak2_update(  
    WTsensor * sensor);
```

This update function packages the translation and rotation record from the device into the sensor object's record after relativizing it and then applying any translational scale factor that may have been set with *WTsensor_setsensitivity* (see page 13-11).

The macro *WTisotrak2_new* creates an ISOTRAK II sensor object that uses the *WTisotrak2_update* function and is recommended for most users.

Polhemus InsideTRAK

The InsideTRAK tracker from Polhemus, Inc. is very similar to the Polhemus ISOTRAK (see page 13-85), but it is only available on Intel-based workstations with ISA bus slots. This six-degree-of-freedom electromagnetic tracking system supports one transmitter and two receivers that measure absolute position and orientation.

To create an InsideTRAK sensor object having two receivers you can use the macro call:

```
WTsensor *i1, *i2;  
i1 = WTinsidetrak_new(1);  
i2 = WTinsidetrak_new(2);  
  
if (!i1 || !i2)  
    WTwarning("Could not open InsideTRAK receivers\n");
```

This macro makes use of the sensor driver functions *WTinsidetraknt_open*, *WTinsidetraknt_close*, and *WTinsidetraknt_update*. It creates the InsideTRAK sensor object having two receivers.

Note: Unlike other sensor objects, you do not need to specify the serial port for the InsideTRAK. So, if using the generic sensor constructor function - *WTsensor_new* (see page 13-7), the serial port argument should always be *NULL*.

When you call *WTinsidettrak_new* for the first receiver (unit 1), the *openfn* for the device is called, which calibrates the sensor. Calibration consists of obtaining an initial position and orientation record, which takes several seconds, during which the device should not be moved. Records subsequently generated by the *updatefn* are with respect to this initial reference frame. As with other 6D sensors, it may be useful in your application to let the user know that the device is about to be calibrated (see the example under *Polhemus ISOTRAK* on page 13-85).

The coordinate frame of the InsideTRAK is the same as for the ISOTRAK (see figure 13-5 on page 13-86).

Accessing InsideTRAK Raw Data

See *Accessing ISOTRAK Raw Data* on page 13-86.

Scaling InsideTRAK Records

See *Scaling ISOTRAK Records* on page 13-86.

InsideTRAK Update Function

WTinsidettrak_update

```
void WTinsidettrak_update(  
    WTsensor * sensor);
```

This update function packages the translation and rotation record from the device into the sensor object's record after relativizing it and then applying any translational scale factor that may have been set with *WTsensor_setsensitivity* (see page 13-11).

The macro *WTinsidetrak_new* creates an InsideTRAK sensor object that uses the *WTinsidetrak_update* function and is recommended for most users.

Polhemus FASTRAK

The FASTRAK is similar to previous Polhemus sensors, except that it supports multiple trackers (up to four) and has much reduced sensor lag and increased accuracy.

To create a FASTRAK sensor object on serial port 1, you can use the macro call:

```
WTsensor *fastrak;  
fastrak = WTfastrak_new(SERIAL1, unit);
```

This macro makes use of the sensor driver functions *WTfastrak_open*, *WTfastrak_close*, and *WTfastrak_update*. It creates the FASTRAK sensor object running at 9600 baud on UNIX platforms and running at 19200 baud on Windows 32-bit platforms.

unit is a unit number, from 1 to 4, specifying which FASTRAK receiver to open. If you are using multiple receivers, open them in sequential order. For example, if you are using 3 FASTRAK receivers, open them like this:

```
WTsensor *f1, *f2, *f3;  
f1 = WTfastrak_new(SERIAL1, 1);  
f2 = WTfastrak_new(SERIAL1, 2);  
f3 = WTfastrak_new(SERIAL1, 3);
```

Following are the DIP switch settings for the FASTRAK running at 9600 baud:

FASTRAK ON ON OFF OFF ON OFF OFF ON

You should try to use the FASTRAK at 19200 baud as this will dramatically improve response when used with multiple sensors. To do this, you must first power down your FASTRAK, change the DIP switch settings to 19200 baud and then repower the device. With WTK, you will need to modify the *sensors.h* include file to specify a baud rate of 19200.

Following are the DIP switch settings for the FASTRAK running at 19200 baud:

FASTRAK OFF OFF ON ON ON OFF OFF ON

Consult your FASTRAK reference manual if you are uncertain of how to set your FASTRAK DIP switches.

The first sensor you open *must* be the sensor connected to the FASTRAK's sensor number one port. For example, you cannot start up using just sensor number two. In addition, if you connect multiple sensors to the FASTRAK, but tell WTK to open only a single sensor, you may get erratic results because the FASTRAK is returning multiple records. If you are only using a single sensor with WTK, then you should also configure the FASTRAK accordingly.

Polhemus Stylus

Using the Polhemus Stylus in your WTK applications is no different than using the Polhemus FASTRAK. However, you must use the Stylus as the first unit of the FASTRAK. You must plug the Stylus into the back panel of the FASTRAK where, if you were not using a Stylus, the FASTRAK receiver 1 would have been plugged in. (Refer to your FASTRAK/ Stylus manual for more information.)

Following is an example of accessing the Stylus button event.

```
WTsensor *stylus,*f2;
stylus= WTfastrak_new(SERIAL1,1);
f2= WTfastrak_new(SERIAL1,2);
/*Additional units may be connected*/
if(WTsensor_getmiscdata(stylus) & WTFASTRAK_STYLUSBUTTON_DOWN)
    WTmessage("Stylus button is down\n");
else
    WTmessage("Stylus button is up\n");
```

Accessing FASTRAK Raw Data

See *Accessing ISOTRAK Raw Data* on page 13-86.

Scaling FASTRAK Records

See *Scaling ISOTRAK Records* on page 13-86.

FASTRAK Update Function

WTfastrak_update

```
void WTfastrak_update(  
    WTsensor * sensor);
```

This update function packages the translation and rotation record from the device into the sensor object's record after relativizing it and then applying any translational scale factor that may have been set with *WTsensor_setsensitivity* (see page 13-11).

The macro *WTfastrak_new* creates a FASTRAK sensor object that uses the *WTfastrak_update* function and is recommended for most users.

Filtering the FASTRAK

The default operation of the FASTRAK doesn't use any filtering of its signal inputs. In some environments, this may lead to a jumpy or erratic signal. Polhemus supplies two functions for adaptive filtering of either the position or orientation information returned by the FASTRAK. These filtering operations are performed by the FASTRAK hardware and they will add a noticeable sensor lag into the measurement process.

For more detailed information about the use of these filters, please consult your Polhemus FASTRAK manual.

WTfastrak_afilter

```
void WTfastrak_afilter(  
    WTsensord *ftrak,  
    float sensitivity,  
    float flow,  
    float fhigh,  
    float factor);
```

This function controls the amount of adaptive filtering applied to the orientation or altitude values returned by the FASTRAK device.

WTfastrak_afilteroff

```
void WTfastrak_afilteroff(  
    WTsensord *ftrak);
```

This function turns off the filtering of orientation or altitude values previously set by *WTfastrak_afilter*.

WTfastrak_pfilter

```
void WTfastrak_pfilter(  
    WTsensord *ftrak,  
    float sensitivity,  
    float flow,  
    float fhigh,  
    float factor);
```

This function controls the amount of adaptive filtering applied to the position values returned by the FASTRAK device.

WTfastrak_pfilteroff

```
void WTfastrak_pfilteroff(  
    WTsensord *ftrak);
```

This function turns off the filtering of position values previously set by *WTfastrak_pfilter*.

Precision Navigation Wayfinder-VR

The Precision Navigation Wayfinder-VR head tracker is a serial device used to track the orientation of the wearer using inertial and compass technologies. This tracker provides 360 degrees of yaw rotation, and about +/- 45 degrees of pitch and roll rotation.

To create a Wayfinder-VR sensor object on serial port 1, you can use the macro call:

```
WTsensor *precision;  
precision = WTprecision_new(SERIAL1);
```

This macro makes use of the sensor driver functions *WTprecision_open*, *WTprecision_close*, and *WTprecision_update*. It creates the Wayfinder-VR sensor object running at 38400 baud.

Accessing Wayfinder-VR Raw Data

WTK maintains a data structure containing the raw data read from the Wayfinder-VR. This information can be accessed using the function *WTsensor_getrawdata* (see page 13-15) as in the example below. The Wayfinder-VR raw data structure stores the absolute rotation of the tracker as an euler.

The raw data structure for the Wayfinder-VR is type defined as follows:

```
typedef struct _WTprecision_rawdata {  
    WTP3 e;  
} WTprecision_rawdata;
```

and is accessed as follows:

```
WTsensor *precision  
WTprecision_rawdata *raw;  
raw = (WTprecision_rawdata *) WTsensor_getrawdata (precision);  
WTP3_print (raw->e, "Raw euler: ");
```

Scaling Wayfinder-VR Records

Records cannot be scaled for this sensor object. So the functions *WTsensor_setsensitivity* and *WTsensor_setangularrate* have no effect.

Wayfinder-VR Update Function

WTprecision_update

```
void WTprecision_update(  
    WTsensor *sensor);
```

This update function calls *WTprecision_rawupdate* (see below) to update the raw data structure, convert it to a quaternion, and relativize it with the previous record. The macro *WTprecision_new* creates a Wayfinder-VR sensor object that uses the *WTprecision_update* function.

Writing your Own Wayfinder-VR Update Function

Your update function should first call *WTprecision_rawupdate* (see below) to obtain the sensor object's raw data as an absolute euler. It should then specify how the raw data is to be transformed into an orientation record. Finally, your update function must store this record by calling *WTsensor_setrecord* (see page 13-24). See *Example 3: Update Function for Absolute Device (Pseudocode)* on page E-15.

WTprecision_rawupdate

```
int WTprecision_rawupdate(  
    WTsensor *sensor);
```

This function reads the tracker input and puts it in the raw data structure as an absolute euler rotation. This information can be accessed with *WTsensor_getrawdata* (see page 13-15). Also see *Accessing Wayfinder-VR Raw Data* on page 13-96.

Special Notes on Wayfinder-VR

This section provides some tips on what you can do if your Wayfinder-VR is not working. Before you proceed, *make sure the battery is not dead.*

The device keeps track of the previous baud rate at which it was being operated. This means that suppose it was being used at 9600 baud rate before being disconnected, then the next time that it is used it would still be running at 9600 baud rate. WTK uses 38400 baud rate when trying to open the device. Thus, if the device was being used at any other baud rate, it fails to open.

If you encounter the above problem do the following:

Connect the Precision Navigation Wayfinder-VR to one of the serial ports.

On Windows NT 3.51:

1. From Program Manager/Accessories, double-click on Terminal.
A window displays from which you can either see erroneous records being output or nothing being output.
2. Click on Settings and select Terminal Preferences. From the dialog box that displays, check the Local Echo option.
3. Click on Settings and select Communications. From the dialog box that displays, choose the required COM port. Set Data Bits to 8, Stop Bits to 1 and Parity to None.
4. If erroneous records are being output, choose the baud rate (from the same dialog box as above) one by one (i.e., 300/1200/2400/4800/9600/19200/38400) until the device starts outputting correct records (i.e., of type `$C<compass>P<pitch>R<roll>*checksum<cr><lf>`).
5. If nothing is being output, the device is probably in halt mode. Choose the baud rate one by one (as above) but additionally type *go* and press Enter (try to get the device into continuous mode). If the device is on the correct baud rate, records of type `$C<compass>P<pitch>R<roll>*checksum<cr><lf>` will start being output.
6. Once correct records have started being output, type *h* and press Enter to bring the device into halt mode. Now set the baud rate to 38400 by typing *b=7*. The device is now set to baud rate 38400 and can be opened by WTK.

7. Close the terminal program (no need to save settings). Disconnect the device (this is *important*). The effect of changing the baud rate does not take place unless the device is disconnected. Now reconnect it and open it with WTK.

On Windows 95/NT 4.0:

1. Double-click on Hypertrm.exe from Program Files/Accessories/Hyper Terminal.
The Connection Description dialog box displays.
2. Choose an icon (any icon) and type a name (any name). Click OK.
The Phone Number dialog box displays.
3. From the Connect Using pull-down menu, choose the required COM port. Click OK.
The COM Properties dialog box displays.
4. Set Data Bits to 8, Parity to None, Stop Bits to 1, and Flow Control to Hardware. Click OK.
5. From the File menu, click Properties. From the dialog box, choose the Settings tab. Click on ASCII Setup. Check the Echo Typed Characters Locally option.
6. If erroneous records are being output, choose the baud rate (from the same dialog box as above) one by one (i.e., 300/1200/2400/4800/9600/19200/38400) until the device starts outputting correct records (i.e., of type \$C<compass>P<pitch>R<roll>*checksum<cr><lf>).
7. If nothing is being output, the device is probably in halt mode. Choose the baud rate one by one (as above) but additionally type *go* and press Enter (try to get the device into continuous mode). If the device is on the correct baud rate, records of type \$C<compass>P<pitch>R<roll>*checksum<cr><lf> will start being output.
8. Once correct records have started being output, type *h* and press Enter to bring the device into halt mode. Now, set the baud rate to 38400 by typing *b=7*. The device is now set to baud rate 38400 and can be opened by WTK.
9. Close the terminal program (no need to save settings). Disconnect the device (this is *important*). The effect of changing the baud rate does not take place unless the device is disconnected. Now reconnect it and open it with WTK.

Spacotec IMC Spaceball

The Spacotec IMC Spaceball is a 6 degree-of-freedom serial port device that sits on the desktop. It responds to both forces and torques, which can be mapped into translations and rotations in 3D.

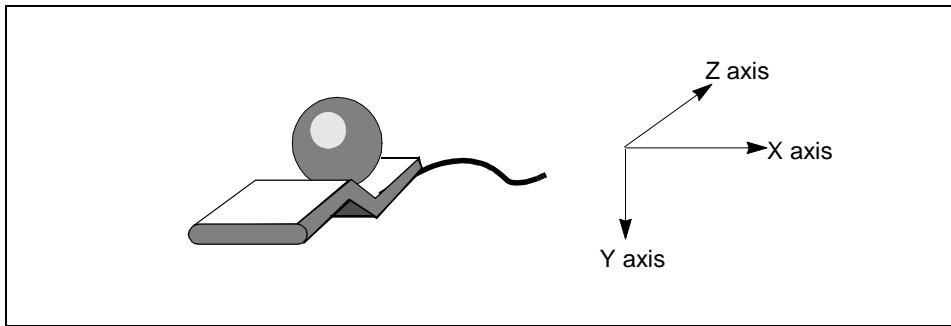


Figure 13-6: Spaceball and its reference frame

To create a Spaceball sensor object on serial port 1, you can use the macro call:

```
WTsensor *spaceball;  
spaceball = WTspaceball_new(SERIAL1);
```

This macro makes use of the sensor driver functions *WTspaceball_open*, *WTspaceball_close*, and *WTspaceball_update*. It creates the Spaceball sensor object running at 9600 baud.

The coordinate frame of this sensor is defined in the WTK driver functions as follows. If the device is placed on a desk or table in front of you with the cable coming out the back of the device oriented away from you, then, (as illustrated above) the Z axis of the device points straight ahead, the X axis points to the right, and the Y axis points down. If this coordinate frame is not appropriate for your application, the function *WTsensor_rotate* (see page 13-20) can be used to define the device's coordinate frame.

Accessing Spaceball Raw Data

WTK maintains a data structure containing the raw data read from the Spaceball. This information is accessed using *WTsensor_getrawdata* (see page 13-15) as in the example below.

The raw data structure for the Spaceball is type defined as follows. Note that both *p* and *w* are in the original Spaceball coordinates and that no scale factors have been applied to the values.

```
typedef struct _WTspaceball_rawdata {
    short p[3];      /* absolute position */
    short w[3];      /* euler angles */
} WTspaceball_rawdata;
```

Spaceball raw data is accessed as follows:

```
WTsensor *spaceball;
WTspaceball_rawdata *raw;

/* get the raw spaceball values and print them out */
raw = (WTspaceball_rawdata *)WTsensor_getrawdata(spaceball);
WTmessage("Position: %d, %d, %d\n", raw->p[X], raw->p[Y], raw->p[Z]);
WTmessage("Angles: %d, %d, %d\n", raw->w[X], raw->w[Y], raw->w[Z]);
```

Scaling Spaceball Records

Translational and rotational records for the Spaceball can be scaled using the functions *WTsensor_setsensitivity* (see page 13-11) and *WTsensor_setangularrate* (see page 13-12) respectively.

Spaceball Update Functions

WTspaceball_update

```
void WTspaceball_update(  
    WTsensord * sensor);
```

This update function packages the translation and rotation record from the device into the sensor object's record after transforming the record to the WTK coordinate convention, and applying any scale factors that may have been set with *WTsensor_setsensitivity* (see page 13-11) and *WTsensor_setangularrate* (see page 13-12).

The macro *WTspaceball_new* creates a Spaceball sensor object that uses the *WTspaceball_update* function and is recommended for most users.

WTspaceball_dominant

```
void WTspaceball_dominant(  
    WTsensord *sensor);
```

This update function restricts the Spaceball movement to just one axis. It is an update function for the Spaceball that can be used in place of *WTspaceball_update*.

In “dominant” mode, the largest single input value from the Spaceball is considered the only input. So, out of the six possible values only the largest value is kept, and the other five values are filtered out (reduced to zero). This can be very useful for avoiding unwanted motion. For example, if you push the Spaceball forward intending to cause a forward motion, the Spaceball is likely to detect small forces in other directions or rotations. In dominant mode, these small values are filtered out and the viewpoint would move forward along a single axis only. This can be a useful technique for “rookie” users or for more precise control over positioning objects.

To toggle the use of dominant mode, simply use the *WTsensor_setupdatefn* to switch between the functions *WTspaceball_dominant* and *WTspaceball_update*.

The macro *WTspaceball_newdominant* creates a Spaceball sensor object that uses the *WTspaceball_dominant* function.

Spaceball Defined Constants

There are nine user-programmable buttons on the Spaceball. Eight of these are positioned on the top edge of the Spaceball frame. One other button, called the “pick button,” is mounted on the forward face of the ball itself. Events from these buttons can be accessed using *WTsensor_getmiscdata* (see page 13-15) together with the following defined constants:

- **Button held down.** This event is generated each frame that the button is held down. These events are defined as: *WTSPACEBALL_BUTTON x* where x is a number between 1 and 8 (for example, *WTSPACEBALL_BUTTON4*) and *WTSPACEBALL_PICKBUTTON*. In addition, there is a mask called *WTSPACEBALL_BUTTONS* that can be used to see if any button is currently held down.
- **Button transitioned down.** This event is generated each time the button moves from up to down. These events are defined as: *WTSPACEBALL_BUTTON x _DOWN*, where x is a number between 1 and 8, and *WTSPACEBALL_PICKBUTTON_DOWN*. In addition, there is a mask called *WTSPACEBALL_BUTTONS_DOWN* that can be used to see if any button transitioned down.
- **Button transitioned up.** This generates a single event each time the button moves from down to up. These events are defined as: *WTSPACEBALL_BUTTON x _UP*, where x is a number between 1 and 8, and *WTSPACEBALL_PICKBUTTON_UP*. In addition, there is a mask called *WTSPACEBALL_BUTTONS_UP* that can be used to see if any button transitioned up.

Redefining the Center for the Spaceball

WTspaceball_rezero

```
WTspaceball_rezero(  
    WTsensor *spaceball);
```

This function redefines the Spaceball’s center value at its current position. Spaceballs may become slightly inaccurate with use and may “drift” when their default “center” value has changed. If you have a drifting Spaceball, you can call this function to redefine the

Spaceball's "center" as its current position. The Spaceball should not have any forces applied to it when this function is called.

Special Notes on Spaceball Model 3003

If you are using the Spacotec Spaceball Model 3003, be aware that the WTK driver has not been rewritten for the Model 3003. The 2003 driver works for both units, with a couple of differences. The ball controls translation and rotation in 6 degrees in real time for both models, but the 3003 only has one button (whereas the 2003 has 8 plus a pick button) that WTK supports. The button on the right side of the 3003 acts as the pick button. Future drivers for the 3003 may include additional support.

Spacotec IMC Spaceball SpaceController

The Spacotec IMC Spaceball SpaceController is a 6 degree-of-freedom serial port device that sits on the desktop. It responds to both forces and torques, which can be mapped into translations and rotations in 3D.

To create a Spaceball SpaceController sensor object on serial port 1, you can use the macro call:

```
WTsensor *spacecontrol;  
spacecontrol = WTspaceballSC_new(SERIAL1);
```

This macro makes use of the sensor driver functions *WTspaceballSC_open*, *WTspaceballSC_close*, and *WTspaceballSC_update*. It creates the Spaceball SpaceController sensor object running at 9600 baud.

The coordinate frame of this sensor is the same as for the Spaceball and is defined in the WTK driver functions as follows. If the device is placed on a desk or table in front of you with the cable coming out the back of the device oriented away from you, then, (as illustrated in figure 13-6 on page 13-100 for the Spaceball) the Z axis of the device points straight ahead, the X axis points to the right, and the Y axis points down. If this coordinate frame is not appropriate for your application, the function *WTsensor_rotate* (see page 13-20) can be used to define the device's coordinate frame.

Accessing Spaceball SpaceController Raw Data

WTK maintains a data structure containing the raw data read from the Spaceball SpaceController. This information is accessed using *WTsensor_getrawdata* (see page 13-15) as in the example below.

The Spaceball SpaceController raw data structure is type defined as follows. Note that both *p* and *w* are in the original Spaceball SpaceController coordinates and that no scale factors have been applied to the values.

```
typedef struct _WTspaceballSC_rawdata {
    WTp3 p;          /* absolute position */
    WTp3 w;          /* euler angles */
} WTspaceballSC_rawdata;
```

Spaceball SpaceController raw data is accessed as follows:

```
WTsensor *spacecontrol;
WTspaceballSC_rawdata *raw;

raw = (WTspaceballSC_rawdata *)WTsensor_getrawdata(spacecontrol);
WTmessage("Position: %f, %f, %f\n", raw->p[X], raw->p[Y], raw->p[Z]);
WTmessage("Rotation: %f, %f, %f\n", raw->w[X], raw->w[Y], raw->w[Z]);
```

Scaling Spaceball SpaceController Records

Translational and rotational records for the Spaceball SpaceController can be scaled using the functions *WTsensor_setsensitivity* (see page 13-11) and *WTsensor_setangularrate* (see page 13-12) respectively.

Spaceball SpaceController Update Functions

WTspaceballSC_update

```
void WTspaceballSC_update(  
    WTsensord *sensor);
```

This update function packages the translation and rotation record from the device into the sensor object's record after transforming the record to the WTK coordinate convention, and applying any scale factors that may have been set with *WTsensor_setsensitivity* (see page 13-11) and *WTsensor_setangularrate* (see page 13-12).

The macro *WTspaceball_new* creates a Spaceball sensor object that uses the function *WTspaceballSC_update* and is recommended for most users.

WTspaceballSC_dominant

```
void WTspaceballSC_dominant(  
    WTsensord *sensor);
```

This update function restricts Spaceball SpaceController movement to just one axis. It is an update function for the Spaceball SpaceController that can be used in place of *WTspaceballSC_update*.

In “dominant” mode, the largest single input value from the Spaceball SpaceController is considered the only input. So, out of the six possible values only the largest value is kept, and the other five values are filtered out (reduced to zero). This can be very useful for avoiding unwanted motion. For example, if you push the Spaceball SpaceController forward intending to cause a forward motion, the Spaceball SpaceController is likely to detect small forces in other directions or rotations. In dominant mode, these small values are filtered out and the viewpoint would move forward along a single axis only. This can be a useful technique for “rookie” users or for more precise control over positioning objects.

To toggle the use of dominant mode, simply use the *WTsensor_setupdatefn* (see page 13-10) to switch between the functions *WTspaceballSC_dominant* and *WTspaceballSC_update*.

Spaceball SpaceController Defined Constants

There are two buttons on the Spaceball SpaceController, one on each side. Events from these buttons can be accessed using *WTsensor_getmiscdata* (see page 13-15) together with the following defined constants:

- **Button held down.** This event is generated each frame the button is held down. These events are defined as: *WTSPACEBALLSC_BUTTONx* where *x* is either 1 and 2 (for example, *WTSPACEBALLSC_BUTTON1*). In addition, there is a mask called *WTSPACEBALLSC_BUTTONS* that can be used to see if either button is currently held down.

Redefining the Center for the Spaceball SpaceController

WTspaceballSC_rezero

```
WTspaceballSC_rezero(  
    WTsensor *spacecontrol);
```

This function redefines the Spaceball SpaceController's center value at its current position. Spaceball SpaceControllers may become slightly inaccurate with use and may "drift" when their default "center" value has changed. If you have a drifting Spaceball SpaceController, you can call this function to redefine the Spaceball SpaceController's "center" as its current position. The Spaceball SpaceController should not have any forces applied to it when this function is called.

Changing the Input Focus Window for the Spaceball SpaceController

WTspaceballSC_setwindow

```
FLAG WTspaceballSC_setwindow(  
    WTsensor *sensor,  
    WTwindow *window);
```

The Spaceball SpaceController has a WTK window associated with it to which it sends messages. By default, this is the WTK window created by the call to *WTuniverse_new* (see page 2-2). So, you need to call this function only if you want to change the WTK window having the input focus associated with the Spaceball Spacecontrollers.

StereoGraphics CrystalEyes and CrystalEyesVR LCD Shutter Glasses

WTK supports StereoGraphics CrystalEyes and the CrystalEyesVR display system. The CrystalEyes provide stereo viewing, whereas the CrystalEyesVR display system is actually a specialized usage of the Logitech position tracker together with stereo viewing. The system consists of LCD-shutter glasses synchronized with a high-frequency monitor and imbedded Logitech ultrasonic receivers. Since orientation and initialization differ from a generic Logitech sensor, WTK provides a separate sensor driver for CrystalEyesVR.

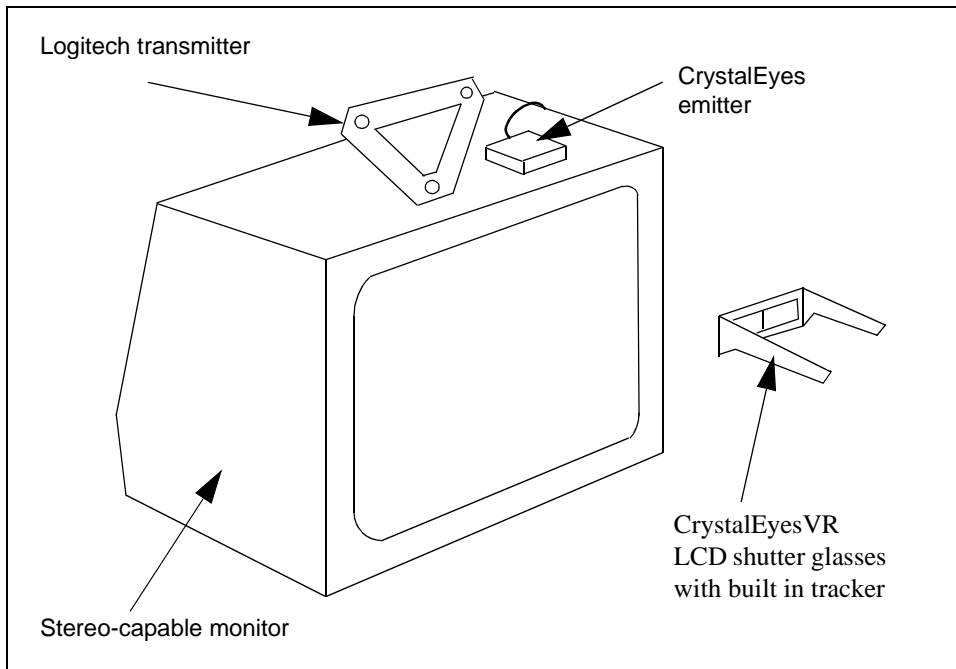


Figure 13-7: CrystalEyesVR sensor reference frame.

To create a CrystalEyesVR sensor object on serial port 1, you can use the macro call:

```
WTsensor *ceyesvr;  
WTmessage("About to open CrystalEyes VR...\n");  
ceyesvr =WTcrystaleyesVR_new(SERIAL1);  
if (!ceyesvr)  
    WTwarning("Warning, couldn't open CrystalEyesVR\n");  
else  
    WTmessage("Calibration complete.\n");
```

This macro makes use of the sensor driver functions *WTcrystaleyesVR_open*, *WTlogitech_close*, and *WTcrystaleyesVR_update*. It creates the CrystalEyesVR sensor object running at 9600 baud on UNIX platforms and running at 1200 baud on Windows 32-bit platforms.

At sensor initialization, the transmitter and receiver must be in a particular spatial relationship with one another for everything to work correctly. The correct orientation is with the goggles and transmitter triangle facing each other directly as shown in figure 13-7 on page 13-108. If the transmitter and receiver are not within range of each other at the time of sensor initialization, the *WTcrystaleyesVR_open* function prompts you to move the receiver within range so that the initialization process can be completed.

Accessing CrystalEyesVR Raw Data

The raw data structure for the CrystalEyesVR is the same raw data structure used for the Logitech Head Tracker device (see page 13-79).

Scaling CrystalEyesVR Records

As for the Logitech Head Tracker, CrystalEyesVR translation records can be scaled by using the function *WTsensor_setsensitivity* (see page 13-11).

There is no angular adjustment for the CrystalEyesVR (i.e., the function *WTsensor_setangularrate* has no effect).

CrystalEyesVR Defined Constants

The defined constants *WTLOGITECH_FLAGBIT* (to detect bad records) and *WTLOGITECH_OUTBIT* (to detect whether the receiver is out of range of the transmitter) are provided and can be used with the function *WTsensor_getmiscdata* (see page 13-15).

CrystalEyesVR Update Function

WTcrystaleyesVR_update

```
void WTcrystaleyesVR_update(  
    WTsensor *sensor);
```

The update function provided in WTK follows the approach recommended by StereoGraphics (the vendor of the CrystalEyesVR system). This update function is appropriate for a user who is sitting or standing in front of the monitor on which the scene is displayed. Most likely the transmitter triangle is fixed to the top of the monitor.

The function *WTcrystaleyesVR_update* uses only the X and Y translation values returned by the ultrasonic tracker. In other words, only the side-to-side motion and up-and-down motion of the user, as returned by the head-tracker, is used by the update function. This information is turned into translation and rotation records by the update function as follows:

- *Horizontal and vertical (X and Y) translations* are generated by this update function based on the X and Y input values. As with other WTK sensor drivers, the translation amounts are scaled by the sensor's sensitivity, which can be set using *WTsensor_setsensitivity* (see page 13-11), and is by default equal to 1.0. No Z translations are generated.
- *Rotations* are computed as a function of the X, Y translation values of the device (while rotational inputs from the device are ignored). The viewpoint is yawed (i.e., rotated about Y) when the user shifts left or right. The yaw angle (in radians) is computed by scaling the user's X location, relative to the user's original location, so that it is between plus and minus the sensor's angular rate value, as set with *WTsensor_setangularrate* (see page 13-12). Similarly, the angle of pitch is computed based on the user's Y location relative the user's original position. This update function does not generate roll, that is, rotations corresponding to tilting the head.

The effect is that when you translate your head to the right, your viewpoint is also rotated (by an amount controlled by the sensor's angular rate) to the left. If you translate your head to the left, your viewpoint is rotated to the right. If you move your head upward, your viewpoint translates upward within the scene, and is also pitched down. If you move your head down, your viewpoint is translated down in the scene and is also pitched upward.

With this approach, as you move your head from side-to-side and up and down, you are able to see around to the sides of objects as well as above and below them, enhancing the sense of 3D.

The macro *WTcrystaleyesVR_new* creates a CrystalEyesVR sensor object that uses the *WTcrystaleyesVR_update* function and is recommended for most users.

ThrustMaster Formula T2 Steering Console

The Formula T2 provides a natural driving experience around your virtual world.

To create a Formula T2 sensor object you can use the macro call:

```
WTsensor *formula;  
formula = WTformula_new(unit);
```

This macro makes use of the sensor driver functions *WTformula_open*, *WTformula_close*, and *WTformula_update*. It creates the Formula T2 sensor object having two steering consoles. The unit argument is unused and can be set to 1.

At initialization, WTK searches the current directory for a Formula T2 calibration file named *formula.cal*. The calibration file is in ASCII format with six values specifying integer values for wheel center, wheel range, wheel drift, pedal center, pedal range, and pedal drift. This is a sample calibration file and also represents the default values used by WTK:

```
11 24 4 20 21 4
```

If the calibration file is not found, the default values are used.

Note: Unlike other sensor objects, you do not need to specify the serial port for the Formula T2. So, if using the generic sensor constructor function – *WTsensor_new* (see page 13-7), the serial port argument should always be *NULL*.

Accessing Formula T2 Raw Data

The raw data structure for the Formula T2 is type defined as follows:

```
typedef struct _WTformula_rawdata {
    unsigned short wheel;
    unsigned short pedal;
} WTformula_rawdata;
```

and accessed as follows:

```
WTsensor *formula;
WTformula_rawdata *raw;
raw = (WTformula_rawdata *)WTsensor_getrawdata(formula);
WTmessage(" Wheel %d Pedal %d\n", raw->wheel, raw->pedal);
```

Scaling Formula T2 Records

Translational and rotational records for the Formula T2 can be scaled using the functions *WTsensor_setsensitivity* (see page 13-11) and *WTsensor_setangularrate* (see page 13-12) to change the speed and turning radius respectively.

Formula T2 Update Function

WTformula_drive

```
void WTformula_drive(
    WTsensor *sensor);
```

This function calls *WTformula_rawupdate* (see below) to obtain the raw wheel and pedal information. It then applies any scale factors that may have been set with *WTsensor_setsensitivity* (see page 13-11) and *WTsensor_setangularrate* (see page 13-12). Finally it stores the record with the sensor by calling *WTsensor_setrecord* (see page 13-24).

The macro *WTformula_new* creates a Formula T2 sensor object that uses the *WTformula_update* function.

Writing Your Own Formula T2 Update Function

Your update function should first call *WTformula_rawupdate* (see below) to obtain the raw wheel and pedal information. It should then specify how the raw data is to be transformed into a 3D position and orientation record. Finally, your update function must store this record by calling *WTsensor_setrecord* (see page 13-24).

WTformula_rawupdate

```
void WTformula_rawupdate(  
    WTsensor *sensor);
```

This function obtains the raw wheel and pedal information. It stores the raw wheel and pedal information in the sensor's raw data structure (*WTformula_rawdata*). This information can be accessed with *WTsensor_getrawdata* (see page 13-15). Also see *Accessing Formula T2 Raw Data* on page 13-112.

ThrustMaster Serial Joystick

The ThrustMaster Mark II Flight Control System is similar to a number of game-port joysticks available for Intel-based personal computers, although this version has a DB9 serial connector which makes it usable on virtually all platforms supported by WTK. In addition to left and right analog actuators, this joystick has three buttons, one trigger, and a four-way hat switch. For more inputs, you can attach a ThrustMaster Mark II Weapons Control System directly to the Mark II Flight Control System.

To create a Serial Joystick sensor object on serial port 1, you can use the macro call:

```
WTsensor *joyserial;  
joyserial = WTjoyserial_new(SERIAL1);  
  
( if !joyserial )  
    WTwarning("Could not open serial joystick\n");
```

This macro makes use of the sensor driver functions *WTjoyserial_open*, *WTjoyserial_close*, and *WTjoyserial_walk*. It creates the Serial Joystick sensor object running at 19200 baud.

At initialization, WTK searches the current directory for a joystick calibration file named *joystick.cal*. The calibration file is in ASCII format with six values specifying floating point values for minimum X, maximum X, minimum Y, maximum Y, center X and center Y, respectively. This is a sample calibration file and also represents the default values used by WTK:

```
0.0 255.0 0.0 255.0 128.0 128.0
```

If the calibration file is not found, default values are used for the center and range values of the joystick.

To use an update function other than *WTjoyserial_walk*, for example, *WTjoyserial_fly*, you can call *WTsensor_new* directly or simply make the following call after using *WTjoyserial_new*:

```
WTsensor_setupdatefn( joyserial, WTjoyserial_fly );
```

Accessing Serial Joystick Raw Data

WTK maintains a data structure containing the raw data from the Serial Joystick. This information can be accessed using the function *WTsensor_getrawdata* (see page 13-15) as in the example below.

The raw data structure for the Serial Joystick is type defined as follows:

```
typedef struct _WTjoyserial_rawdata
{
    unsigned short x,y;           /* roll & pitch */
    unsigned short throttleleft; /* weapons control system throttle (wcs) */
    unsigned short throttleright;
    unsigned short throttletrim;
    unsigned short rudder;
} WTjoyserial_rawdata;
```

and is accessed as follows :

```
WTsensor *joyserial;
WTjoyserial_rawdata *raw;
raw = (WTjoyserial_rawdata *)WTsensor_getrawdata(joyserial);

WTmessage("Roll %d Pitch %d\n", raw->x, raw->y);
WTmessage("Throttleleft %d ThrottleRight %d Rudder %d\n", raw->throttleleft,
raw->throttleright, raw->rudder);
```

Scaling Serial Joystick Records

Translational and rotaional records for the Serial Joystick can be scaled using the functions *WTsensor_setsensitivity* (see page 13-11) and *WTsensor_setangularrate* (see page 13-12) respectively.

Serial Joystick Update Functions

The WTK update functions for the Serial Joystick store the position record from the device into the sensor object's record, after applying any scale factors set with *WTsensor_setsensitivity* (see page 13-11) and *WTsensor_setangularrate* (see page 13-12).

WTjoyserial_fly

```
void WTjoyserial_fly(  
    WTsensor *sensor);
```

This function is an update function that moves a sensor forward along the Z-axis at a constant velocity. It can be used to operate the joystick in a manner familiar to users of flight simulation programs. When this update function is used, the sensor moves forward along Z at a small constant velocity (0.1 times the sensor sensitivity each frame). Moving the joystick forward/backward pitches around X, and moving the joystick right/left rolls around Z.

The macro *WTjoyserial_newfly* creates a serial joystick object that uses the *WTjoyserial_fly* update function.

WTjoyserial_walk

```
void WTjoyserial_walk(  
    WTsensor *sensor);
```

This function initializes a sensor to move in the “walkthrough” mode. The joystick can be used to move a viewpoint or object in the X-Z plane. When no buttons are pressed, moving the joystick forward or backward moves forward or backward along the Z axis. Moving the joystick right or left yaws around the Y axis. When the front button is depressed, moving the joystick forward or backward pitches around the X axis, and moving the joystick right or left rolls around the Z axis.

The macro *WTjoyserial_new* creates a Serial Joystick object that uses the *WTjoyserial_walk* update function.

WTjoyserial_walk2

```
void WTjoyserial_walk2(  
    WTsensor *sensor);
```

This function initializes a sensor to move in a second “walkthrough” mode. The *WTjoyserial_walk2* update function is like *WTjoyserial_walk* except that holding down the trigger button allows you to raise or lower the viewpoint.

The macro `WTjoyserial_newwalk2` creates a serial joystick object that uses the `WTjoyserial_walk2` update function.

Writing your Own Serial Joystick Update Function

Your update function should first call `WTjoyserial_rawupdate` to obtain the Serial Joystick's raw data. It should then specify how the raw data is to be transformed into 3D position record. Finally, your update function must store this record by calling `WTsensor_setrecord` (see page 13-24). See *Example 3: Update Function for Absolute Device (Pseudocode)* on page E-15.

WTjoyserial_rawupdate

```
int WTjoyserial_rawupdate(  
    WTsensor *sensor);
```

This function reads in the raw data from the Serial Joystick and stores it in the sensor's raw data structure. This information can be accessed with the function `WTsensor_getrawdata` (see page 13-15). Also see *Accessing Serial Joystick Raw Data* on page 13-114.

Serial Joystick Defined Constants

The ThrustMaster Mark II Flight Control System supports three momentary buttons in addition to the trigger and a hat switch. The Mark II Weapons Control System adds an additional six momentary switches as well as a three position rocker switch. These values can be accessed by using the `WTsensor_getmiscdata` function with any of the following constants described in the `joyserial.h` file in the `include` directory:

```
WTJOYSERIAL_TRIGGERDOWN  
WTJOYSERIAL_TOPDOWN  
WTJOYSERIAL_SIDEDOWN  
WTJOYSERIAL_BOTTOMDOWN  
WTJOYSERIAL_HATRIGHT  
WTJOYSERIAL_HATLEFT  
WTJOYSERIAL_HATDOWN  
WTJOYSERIAL_HATUP
```

```
WTJOYSERIAL_WCS1  
WTJOYSERIAL_WCS2  
WTJOYSERIAL_WCS3  
WTJOYSERIAL_WCS4  
WTJOYSERIAL_WCS5  
WTJOYSERIAL_WCS6  
WTJOYSERIAL_WCS7  
WTJOYSERIAL_WCSTOGGLEA  
WTJOYSERIAL_WCSTOGGLEB
```

Serial Joystick Range

WTjoyserial_getrange

```
void WTjoyserial_getrange(  
    WTsensor *sensor,  
    WTp2 range);
```

This function returns the maximum X and Y values (divided by 2), which can be attained by the joystick.

Serial Joystick Drift

WTjoyserial_setdrift

```
void WTjoyserial_setdrift(  
    WTsensor *sensor,  
    float drift_per);
```

This function sets the joystick's drift amount to a percentage of the joystick's range. The *drift_per* parameter specifies the percentage.

WTjoyserial_getdrift

```
float WTjoyserial_getdrift(  
    WTsensor *sensor);
```

This function returns the drift amount of the specified joystick sensor. See *WTjoyserial_setdrift*, above.

Reinitializing the Serial Joystick

WTjoyserial_readcalibrationfile

```
void WTjoyserial_readcalibrationfile(  
    void);
```

This function reads the joystick's calibration file so *joystick.cal* can be re-initialized. Page 13-113 talks about the *joystick.cal* file.

VictorMaxx Technologies' CyberMaxx2 HMD

VictorMaxx Technologies' CyberMaxx2 HMD is a serial device used to track the orientation of the wearer using inertial and compass technologies. This tracker provides 360 degrees of yaw rotation, and about +/- 60 degrees of pitch and roll rotation.

To create a CyberMaxx2 HMD sensor object on serial port 1, you can use the macro call:

```
WTsensor *cybermaxx2;  
cybermaxx2 = WTcybermaxx2_new(SERIAL1);
```

This macro makes use of the sensor driver functions *WTcybermaxx2_open*, *WTcybermaxx2_close*, and *WTcybermaxx2_update*. It creates the CyberMaxx2 HMD sensor object running at 9600 baud.

Accessing CyberMaxx2 HMD Raw Data

WTK maintains a data structure containing the raw data read from the CyberMaxx2 HMD. This information can be accessed using the function *WTsensor_getrawdata* (see page 13-15) as in the example below. The CyberMaxx2 HMD raw data structure stores the absolute rotation of the tracker as an euler.

The raw data structure for the CyberMaxx2 HMD is type defined as follows:

```
typedef struct _WTcybermaxx2_rawdata {  
    float e[3];  
} WTcybermaxx2_rawdata;
```

and is accessed as follows:

```
WTsensor *cybermaxx2;  
WTp3 p3;  
WTcybermaxx2_rawdata *raw;  
raw = (WTcybermaxx2_rawdata *) WTsensor_getrawdata (cybermaxx2);  
WTp3_copy(raw->e, p3);  
WTp3_print (p3, "Raw euler: ");
```

Scaling CyberMaxx2 HMD Records

Records cannot be scaled for this sensor object. So the functions *WTsensor_setsensitivity* and *WTsensor_setangularrate* have no effect.

CyberMaxx2 HMD Update Function

WTcybermaxx2_update

```
void WTcybermaxx2_update(  
    WTsensor *sensor);
```

This update function calls *WTcybermaxx2_rawupdate* (see below) to update the raw data structure, convert it to a quaternion, and relativize it with the previous record. The macro *WTcybermaxx2_new* creates a CyberMaxx2 HMD sensor object that uses the *WTcybermaxx2_update* function.

Writing your Own CyberMaxx2 HMD Update Function

Your update function should first call *WTcybermaxx2_rawupdate* (see below) to obtain the sensor object's raw data as an absolute euler. It should then specify how the raw data is to be transformed into an orientation record. Finally, your update function must store this record by calling *WTsensor_setrecord* (see page 13-24). See *Example 3: Update Function for Absolute Device (Pseudocode)* on page E-15.

WTcybermaxx2_rawupdate

```
FLAG WTcybermaxx2_rawupdate(  
    WTsensor *sensor);
```

This function reads the tracker input and puts it in the raw data structure as an absolute euler rotation. This information can be accessed with *WTsensor_getrawdata* (see page 13-15). Also see *Accessing CyberMaxx2 HMD Raw Data* (see page 13-120).

Virtual i-O i-glasses!

Virtual i-O i-glasses! is a serial device used to track the orientation of the wearer using inertial and compass technologies. This tracker provides 360 degrees of yaw rotation, and about +/- 60 degrees of pitch and roll rotation.

To create an i-glasses! sensor object on serial port 1, you can use the macro call:

```
WTsensor *iglasses;  
iglasses = WTiglasses_new(SERIAL1);
```

This macro makes use of the sensor driver functions *WTiglasses_open*, *WTiglasses_close*, and *WTiglasses_update*. It creates the i-glasses! sensor object running at 19200 baud.

Accessing i-glasses! Raw Data

WTK maintains a data structure containing the raw data read from the i-glasses!. This information can be accessed using the function *WTsensor_getrawdata* (see page 13-15) as in the example below. The i-glasses! raw data structure stores the absolute rotation of the tracker as an euler.

The raw data structure for the i-glasses! is type defined as follows:

```
typedef struct _WTiglasses_rawdata {
    WTP3 e;
} WTiglasses_rawdata;
```

and is accessed as follows:

```
WTsensor *iglasses;
WTiglasses_rawdata *raw;
raw = (WTiglasses_rawdata *) WTsensor_getrawdata (iglasses);
WTP3_print (raw->e, "Raw euler: ");
```

Scaling i-glasses! Records

Records cannot be scaled for this sensor object. So the functions *WTsensor_setsensitivity* and *WTsensor_setangularrate* have no effect.

i-glasses! Update Function

WTiglasses_update

```
int WTiglasses_update(
    WTsensor *sensor);
```

This update function calls *WTiglasses_rawupdate* (see below) to update the raw data structure, convert it to a quaternion, and relativize it with the previous record.

The macro *WTiglasses_new* creates an i-glasses! sensor object that uses the *WTiglasses_update* function.

Writing your Own i-glasses! Update Function

Your update function should first call *WTiglasses_rawupdate* (see below) to obtain the sensor object's raw data as an absolute euler. It should then specify how the raw data is to be transformed into an orientation record. Finally, your update function must store this record by calling *WTsensor_setrecord* (see page 13-24). See *Example 3: Update Function for Absolute Device (Pseudocode)* on page E-15.

WTiglasses_rawupdate

```
void WTiglasses_rawupdate(  
    WTsensor *sensor);
```

This function reads the tracker input and puts it in the raw data structure as an absolute euler rotation. This information can be accessed with *WTsensor_getrawdata* (see page 13-15). Also see *Accessing i-glasses! Raw Data* on page 13-122.

Virtual Technologies CyberGlove

The Virtual Technologies CyberGlove is a popular serial port device for direct manipulation of objects in virtual worlds. The CyberGlove comes in an 18-sensor model and a 22-sensor model. WorldToolKit's CyberGlove driver automatically handles CyberGloves with either number of sensors, and also supports both left and right gloves. WTK's CyberGlove device driver is different from the other WTK device drivers and uses a *WTcybglove* structure which is different from the *WTsensor* structure. The reason for this difference is to be able to provide functions for calibrating and graphically representing this compound device. WorldToolKit's CyberGlove driver enables you to do the following:

- Instantiate up to two *WTcybglove* entities.
- Optionally specify CyberGlove parameters in the *VirtualHand* resource file.
- Calibrate the CyberGlove.

- Create a graphical hand model which is automatically updated by input from the CyberGlove.
- Set the visibility of the hand model.
- Access the movable objects comprising the hand model and the angle information from the glove.
- Perform collision detection between the hand model objects and the other objects in the scene graph.

Note: This device is supported on the SGI and Windows NT platforms only. The subsection - "For Windows NT Users" - lists some characteristics of the driver that are specific to its functioning on the Windows NT platform.

Initializing the CyberGlove

WTcybglove_new

```
WTcybglove *WTcybglove_new(  
    int baud,  
    char *device,  
    char *calibrationfilename);
```

Call `WTcybglove_new` to initialize a CyberGlove and obtain a pointer to a new `WTcybglove` object. `WTcybglove_new` must be called separately for each CyberGlove you wish to use. For example, to create an application which uses both a left and a right glove, you must call `WTcybglove_new` twice. In addition, you must use a different serial port device for each CyberGlove.

Note: The `WTcybglove` structure is different from the `WTsensor` structure and may not be passed in to the `WTsensor` functions.

Currently WTK supports the use of up to two CyberGloves in an application. Once two WTK CyberGlove objects have been constructed, subsequent calls to `WTcybglove_new` return `NULL`.

The first argument to `WTcybglove_new` is the baud rate, which must be one of 1200, 2400, 4800, 9600, 19200, 38400 or 57600. The second argument is the serial port device name, which is, for example, `SERIAL1` for serial port 1. You can find out more about serial ports

by consulting your hardware guide. The third argument to `WTcybglove_new` is the name of a calibration file for the glove. This file must be located in the same directory as the application executable. A default calibration file (`default.cal`) is supplied with the CyberGlove product.

Note: You must have a CyberGlove calibration file to call `WTcybglove_new`. If the specified calibration file is not found in the directory in which your executable is located, then `WTcybglove_new` produces a warning message and returns `NULL`.

On the Windows platform, you must have the calibration panel resource file, `panel.res`, included in the project makefile. The file "`panel.res`" is located in the CyberGlove distribution provided by Virtual Technologies. The CyberGlove driver will not be able to create the calibration panel without this file, and will exit with an "unhandled exception" if the file is not present. If your application has a resource file of its own, you must merge the resources specified in `panel.res` into your application's resource file. This may be done by dragging and dropping the resources from one file into the other, from within the Visual C++ development environment. Even if you do not wish to open and use the calibration panel, `WTcybglove_new` initializes the panel window and requires this file to be present. There is no equivalent calibration panel resource file on the UNIX platform.

The baud rate, serial port selection, and calibration file name can alternatively be specified in a configuration file. The configuration (or application resource) file is called `VirtualHand`. Refer to the CyberGlove User's Manual for a description of the format that must be used in the `VirtualHand` resource file. Apart from the initialization parameters, there are various other application specific options that can be set using the resource file. These, and their respective field definitions are discussed in the CyberGlove User's Manual. On UNIX platforms, you may use the `xrdb` command to merge the `VirtualHand` resource file into the X server's resource database. (Type `xrdb -merge VirtualHand` at the command prompt). On Windows platforms the `VirtualHand` resource file must exist in the directory that contains the executable. The resource file will not be read otherwise. By using the `VirtualHand` file you can make changes to the values of the initialization parameters without having to recompile your application.

If you wish to use the resource file to specify the baud rate for your glove, simply pass in 0 for the baud argument to `WTcybglove_new`. If you pass in 0 as the baud rate argument, but no value for the baud rate is found in the configuration file, then the baud rate defaults to 38400.

Similarly, to use the `VirtualHand` resource file to specify the device name and/or calibration file name, pass in `NULL` for the corresponding argument to `WTcybglove_new`. The default value for the device name is `/dev/ttyd1` on UNIX platforms and `COM1` on Windows

platforms. You must use a system-specific device designation in the configuration file and not one of WTK's cross-platform serial device constants. The default value for the glove calibration file is default.cal.

Following are example entries such as might appear in the VirtualHand resource file. Note that "glove1" refers to the first CyberGlove activated with WTcybglove_new. You can add similar lines substituting "glove2" for "glove1" if two CyberGloves are in use.

```
VirtualHand*glove1device: /dev/ttyd2
VirtualHand*glove1speed: 38400
VirtualHand*glove1calFile: default.cal
VirtualHand*handModel: hires_hand.vnf
```

(On Windows platforms, a device name of COM2 is specified by entering:

```
VirtualHand*glove1device: COM2)
```

Calibrating the CyberGlove

WorldToolKit enables you to interactively recalibrate your CyberGlove while your WorldToolKit application is running. Please note though that an initial calibration file is still required to be present at the time that WTcybglove_new is called, as described in the previous section.

WTcybglove_showcalibrationpanel

```
void WTcybglove_showcalibrationpanel(
    FLAG on);
```

Call WTcybglove_showcalibrationpanel passing in TRUE to display a panel from which you can alter the calibration settings loaded from your calibration file. If you do not call this function, your calibration remains as originally loaded from your calibration file. To close the calibration panel, call WTcybglove_showcalibrationpanel passing in FALSE. This function may not be called until at least one CyberGlove has been initialized by calling WTcybglove_new.

The adjustments made using the calibration panel will be used in the current WTK session. If you wish to save the adjustments, you may do so by clicking on the "save" button in the

calibration panel. The default name of a saved calibration file is "untitled.cal." This can be changed by editing the name field in the panel. You will not be warned about overwriting an existing calibration file.

The calibration panel can be used to calibrate multiple gloves. To do so, you must enter the desired glove number into the field next to the "Show" button on the calibration panel, and then click the "Show" button. Make sure that all CyberGloves to be calibrated have been activated using `WTcybglove_new` before trying to calibrate them.

The calibration file written out by this function may be used in a new WTK session by specifying the new file name in the configuration file.

Creating a Graphical Hand Model for CyberGlove

WTcybglove_usehandmodel

```
WTnode *WTcybglove_usehandmodel(  
    WTcybglove *glove,  
    char *handmodelname,  
    float scale,  
    WTnode *parent);
```

`WTcybglove_usehandmodel` builds a hand model from a multi-object NFF file. The order and naming of the objects in this file is described in the CyberGlove User's Manual.

The first argument to `WTcybglove_usehandmodel` specifies the CyberGlove sensor object by which the hand model is to be controlled. Once this call has been made, the hand model is automatically updated by WTK with the latest CyberGlove input once per frame.

The second argument is the name of the file which contains the hand model. The file must follow the Virtual Technologies guidelines for a hand model file. You may supply `NULL` as an argument here, which will result in a default to the value specified in your `VirtualHand` resource file. If the hand model file is not found, a very simple hand model is generated automatically without a model file. WTK supplies a more complex hand model for use with the CyberGlove, if you so desire. Please see the `Readme.txt` file in the `models` directory of the WTK distribution.

WTK assigns the following names to the hand model objects constructed by this call: "cyforearm1", "cypalm1", "cythumbbase1", "cythumbmedial1", "cythumbtip1", "cyindexbase1", "cyindexmedial1", "cyindextip1", "cymiddlebase1", "cymiddlemedial1", "cymiddletip1", "cyringbase1", "cyringmedial1", "cyringtip1", "cypinkiebase1", "cypinkiemedial1", "cypinkietip1", for the first WTcybglove object constructed. If a second WTcybglove is constructed, the names are "cyforearm2", etc. These are the

names that are returned by the function `WTnode_getname`.

The third argument specifies the scale of the hand model. The hand model may not be scaled after it is created by the `WTcybglove_usehandmodel` function. If you use any of the geometry scaling functions provided by WTK on the hand model objects, the hand will become distorted.

The fourth argument, `parent`, is a pointer to a `WTnode` that indicates the node below which the CyberGlove hand model structure will be attached in the scene graph. If you do not wish to insert the CyberGlove hand model into the scene graph, you may pass in `NULL` for this argument.

This function returns the top most node in the CyberGlove hand model structure. The hand model is created as a hierarchy of movable nodes and attachments. If you passed in `NULL` for the `parent` argument, you may use the returned node to add the CyberGlove model structure anywhere in the scene graph by using `WTnode_addchild` or `WTnode_insertchild`. Use the function `WTnode_print` (with the node returned by this function) to get a listing of the hierarchy of nodes in the hand model structure.

Once constructed, WTK updates the graphical hand's position each frame using input from the CyberGlove device. The forearm object is the only object in the hand model which you may attach another sensor to or alter the orientation of. Any sensor, such as a 6-D tracker, can be attached to the forearm object, and the rest of the hand model will move along with the forearm. In the WTK event loop, the CyberGlove finger and wrist objects are updated immediately after all other objects (including the forearm object) have been updated by all active `WTsensors` in the simulation. The input from the CyberGlove determines the position and orientation of all of the finger and wrist objects relative to the forearm. For this reason, attempts to alter the orientation or position of any of the hand model objects other than the forearm will have no effect.

You may not delete the nodes which comprise the hand model objects using `WTnode_delete` or `WTnode_deletechild`; your program will crash if any node belonging to the CyberGlove hand model structure is deleted. To delete the hand model, use `WTcybglove_deletehandmodel`. The forearm node (which is the top most node in the hand

model hierarchy) may be removed and attached elsewhere in the scene graph. The entire hand model will automatically be moved along with the forearm node.

If there is already a hand model associated with the `WTcybglove` object at the time `WTcybglove_usehandmodel` is called, it is deleted and a new hand model is built corresponding to the current args to `WTcybglove_usehandmodel`. Note that when the new hand model is created, it is positioned at the universe origin and oriented along the Y-axis.

You may wish to obtain the position and orientation of the forearm object before creating the new hand model, so that you can position and orient the new hand model where the old one was. Also, if a WTK sensor had been attached to the original forearm object, you may wish to attach the sensor to the new forearm object if you want it to have the same behavior as the original hand model.

In order to get the current position and orientation of the forearm node, you must create a `nodepath` from the root node of the scene graph to the forearm node. A `nodepath` is necessary to obtain the cumulative transformation matrix from the root node to the hand model. Pass this `nodepath` to the function `WTnodepath_gettransform` to get a 4x4 matrix containing the position and orientation information of the forearm node in world coordinates.

WTcybglove_deletehandmodel

```
void WTcybglove_deletehandmodel(  
    WTcybglove *glove);
```

`WTcybglove_deletehandmodel` deletes and frees all of the objects in the hand model. The calibration of the CyberGlove remains the same, so that if you call `WTcybglove_usehandmodel` again, the current calibration data will be used for the new model.

You must use this function only to delete the CyberGlove object and hand model nodes. The hand model nodes must not be deleted using `WTnode_delete`.

Setting the Visibility of the Hand Model

WTcybglove_setvisibility

```
void WTcybglove_setvisibility(  
    WTcybglove *glove,  
    FLAG visible);
```

WTcybglove_setvisibility sets the visibility of all of the objects in the hand model associated with the given CyberGlove. The second argument should be TRUE to make the hand model visible, that is, to have WTK render the hand model each frame, or FALSE to make it invisible. The hand model is visible by default.

To set the visibility of an individual object in the hand model, such as the forearm or the thumb, you must first access the relevant node. Accessing individual nodes in the hand model is discussed in the next section. Once you get a pointer to the node, you may then call WTnode_enable passing in TRUE or FALSE to turn the visibility on or off respectively. Note that since the hand model is organized as a hierarchy of movable nodes, turning off the visibility of the palm causes the fingers also to be invisible. This behavior is programmed to be different for the forearm, in that, calling WTnode_enable on the forearm disables/enables the forearm only, even though the rest of the hand is hierarchically below the forearm. This allows you to choose not to render the forearm and display the hand as just the palm and the fingers.

Use the function WTnode_print to get a listing of the hierarchy of nodes in the hand model structure.

Accessing Hand Model Objects

The graphical objects making up the hand model can be accessed through the functions described in this section.

CWTcybglove_getforearm

```
WTnode *WTcybglove_getforearm(  
    WTcybglove *glove);
```

WTcybglove_getforearm returns a pointer to the forearm object associated with the specified CyberGlove. This is a pointer to a movable node. The forearm object may be moved in any way that you wish, and the rest of the hand will follow this orientation. It is permitted to set the visibility of this or any of the other hand model objects individually with WTnode_enable.

If you haven't called WTcybglove_usehandmodel before calling this function, NULL is returned.

You may not delete this node. To delete the CyberGlove hand model structure use WTcybglove_deletehandmodel.

WTcybglove_getpalm

```
WTnode *WTcybglove_getpalm(
    WTcybglove *glove);
```

WTcybglove_getpalm returns a pointer to the palm object constructed by WTcybglove_usehandmodel. If you haven't called WTcybglove_usehandmodel before calling this function, NULL is returned.

You may not move this object in relation to the forearm; this relationship is controlled by the CyberGlove. The only way to affect the position and orientation of a CyberGlove object (other than the forearm) with respect to the object it is connected to is by changing the angle information present in the 2-D array of floats returned by WTcybglove_getanglearray.

You may not delete this node. To delete the CyberGlove hand model structure use WTcybglove_deletehandmodel.

WTcybglove_getfingers

```
WTnode **WTcybglove_getfingers(
    WTcybglove *glove);
```

WTcybglove_getfingers returns a 5x3 array of WTnode pointers. These pointers could be useful if you wish to do collision detection or change the color of the finger objects.

Indexing into the 2d array is accomplished using multiplication and addition as shown in the example below, which sets the visibility of the finger objects.

If you haven't called `WTcybglove_usehandmodel` before calling this function, `NULL` is returned.

```
WTnode  **fingers;
WTcybglove *glove;
int      finger, joint;
glove = WTcybglove_new(19200, SERIAL1, "default.cal");
WTcybglove_usehandmodel(glove, "hires_hand.vnf", 1.0f, NULL);
fingers = WTcybglove_getfingers(glove);
for (finger = WTCG_THUMB; finger < WTCG_FINGERS; finger++) {
    for (joint = WTCG_BASE; joint < WTCG_FINGER_SEGMENTS; joint++) {
        WTnode_enable(fingers[WTCG_FINGER_SEGMENTS * finger + joint],
                      FALSE);
    }
}
```

(Note that the above example is only intended to show how to index into the finger array. In actuality, if you want to disable all the fingers, you would have to call `WTnode_enable` on the bases of the fingers only. The medials and tips will be automatically disabled as they are arranged below the bases in the node hierarchy.)

Refer to the section 'Defined Constants for the CyberGlove Hand Model' for a list of the constants `WTCG_THUMB`, `WTCG_BASE`, etc, that identify the different hand model parts.

You may not delete the finger objects. To delete the CyberGlove hand model structure use `WTcybglove_deletehandmodel`.

Accessing the CyberGlove Bend Angle Data

CyberGlove bend angle data can be obtained with a call to `WTcybglove_getanglearray`. The angles returned reflect the current calibration settings for the glove, and are the angles used when the CyberGlove hand model is rendered if you have called `WTcybglove_usehandmodel`. All of the angle information is specified in radians.

WTcybglove_getanglearray

```
float *WTcybglove_getanglearray(
```



```
WTcybglove *glove);
```

WTcybglove_getanglearray returns a 6x4 array of floating point values. These values are bend angles represented in radians. The first index in this array corresponds to the finger, starting with 0 for the thumb to 4 for the pinkie. Index 5 is used for the palm and wrist - the palm arch, wrist pitch and wrist yaw are stored from [5][0] through [5][2]. The second index into this array refers to the joints of each finger. Index 0 is the base, 1 is the medial, and 2 is the tip. Index 3 holds the abduction angle for each finger.

The following example shows how to access the information in this array.

```
float *anglearray;
int finger, joint;
anglearray = WTcybglove_getanglearray(glove);
for (finger = WTCG_THUMB; finger < WTCG_FINGERS; finger++) {
    for (joint = WTCG_BASE; joint < WTCG_FINGER_ANGLES; joint++) {
        printf("anglearray[%d][%d] = %f\n", finger, joint,
            anglearray[ WTCG_FINGER_ANGLES * finger + joint ]);
    }
}

finger = WTCG_WRIST;
for (joint = WTCG_PALM_ARCH; joint < WTCG_WRIST_ANGLES; joint++) {
    printf("anglearray[%d][%d] = %f\n", finger, joint,
        anglearray[ WTCG_FINGER_ANGLES * finger + joint ]);
}
```

If you wish to impose constraints on the movement of the hand model, you can do so by modifying the contents of the angle array. If you choose to do this, you will have to examine and alter the joint angle values every frame.

Refer to the section 'Defined Constants for the CyberGlove Hand Model' for a list of the constants WTCG_THUMB, WTCG_BASE, etc, that identify the different hand model parts.

Note that in the angle array, the following elements:

```
array[WTCG_FINGER_ANGLES * WTCG_WRIST + 3]
array[WTCG_FINGER_ANGLES * WTCG_INDEX + WTCG_ABDUCT]
```

are not used or updated by the CyberGlove, and the following element:

```
array[WTCG_FINGER_ANGLES * WTCG_WRIST + WTCG_PALM_ARCH]
```

is not considered during the rendering process.

Defined Constants for the CyberGlove Hand Model

The following constants are used for the fingers and the wrist.

```
WTCG_THUMB  
WTCG_INDEX  
WTCG_MIDDLE  
WTCG_RING  
WTCG_PINKIE  
WTCG_WRIST  
WTCG_FINGERS
```

The first five of the above are used for the thumb, index, middle, ring and pinkie fingers respectively. WTCG_WRIST is used to identify the wrist joint. WTCG_FINGERS is provided as a delimiter for the set of finger constants.

The following constants are used to identify the joint angles for each finger, from the base to the tip (for the three joints in a finger) and the abduction angle for the finger as a whole.

```
WTCG_BASE  
WTCG_MEDIAL  
WTCG_TIP  
WTCG_ABDUCT  
WTCG_FINGER_ANGLES
```

WTCG_FINGER_ANGLES is a delimiter for the finger angles.

The following constants are used to identify the joint angles for the wrist -the arch of the palm, the pitch and yaw of the wrist. WTCG_WRIST_ANGLES is a delimiter for the wrist angles.

WTCG_PALM_ARCH
WTCG_WRIST_PITCH
WTCG_WRIST_YAW
WTCG_WRIST_ANGLES

For Windows NT Users:

- The resource file panel.res must be included in the Visual C++ project makefile. If this file is not present, the CyberGlove driver will not be able to create the calibration panel, and will exit because of an "unhandled exception". (panel.res exists in the CyberGlove distribution provided by Virtual Technologies.)
- The glove defaults resource file, VirtualHand, must be present in the current working directory. (VirtualHand exists in the CyberGlove distribution provided by Virtual Technologies.)

Introduction

A WorldToolKit path stores a series of position and orientation records in absolute world coordinates. These paths can be used to guide the viewpoint or move other entities in the scene. Paths can be dynamically recorded, edited, saved and loaded, and played back in a variety of ways. You can also use interpolation to smooth a roughly defined path.

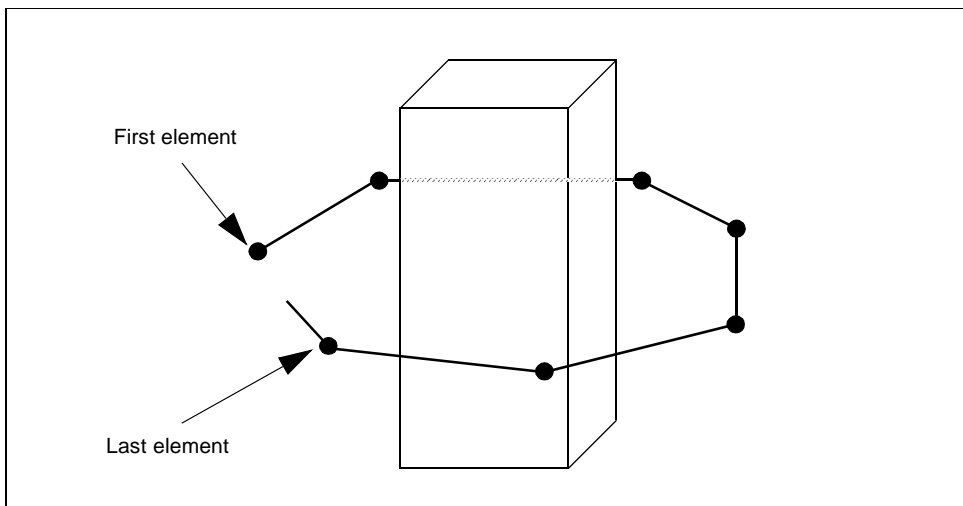


Figure 14-1: A path around an object

As shown in figure 14-1, paths are made up of a set of discrete *elements*, where each element stores an absolute position and orientation. A path may be constructed by recording the position and orientation of the viewpoint each frame, creating one element each time through the simulation loop or at a specified sample rate.

Paths are useful for a variety of applications. For example, if you are creating a demonstration program, you can record an optimal path through the virtual environment before the actual demonstration. Viewpoint paths are useful for any application in which it may be important for the user to see certain aspects of the virtual world. Viewpoint paths can also be used whenever an application requires that a viewpoint be moved from one location to another and you want to provide a smooth transition.

Similarly, there are many uses for paths associated with other entities in the scene. Consider a simple case in which you want to have a door swing open and shut. One approach is to create a task function, in which the door is rotated a specified amount each frame. The task function would also include a test to determine when the door was fully open so that it could be made to rotate in the opposite direction. An alternative approach is to use pathing to record the motion of the door while it is interactively swung open and shut. For example, you could attach a sensor such as the Spaceball to the door, and while twisting the Spaceball to open and close the door, record the door's path. Then, whenever the door needed to be opened and closed in the simulation, the path could be replayed. If the path's playback mode were set to oscillate, then you would only need to record the motion of the door as it opened to have it both open and shut on playback.

Path Construction and Destruction

There are five ways to create or define a new path. You can:

- Record it
- Construct it element by element
- Interpolate an existing path
- Copy an existing path
- Load a path from a file

TO RECORD A PATH:

1. Call *WTpath_new* to obtain a pointer to a new, empty path.
2. Call *WTpath_record* to start recording your current viewpoint's location and orientation.

3. Call *WTuniverse_go* to start the simulation loop if it is not already running. One element will be recorded to the path each frame.
4. Call *WTpath_stop* to stop recording.

TO CONSTRUCT A PATH ELEMENT BY ELEMENT:

1. Call *WTpath_new* to obtain a pointer to a new, empty path.
2. Call *WTpathelement_new* to create a new element at the desired location.
3. Call *WTpath_insertelement* or *WTpath_appendelement* to add the element to the path.

TO INTERPOLATE AN EXISTING PATH

To construct a path which is an interpolated (“smoothed”) version of an existing path, use the function *WTpath_interpolate*.

TO COPY AN EXISTING PATH:

- See *WTpath_copy* on page 14-5.

TO LOAD A PATH FROM A FILE

- See *WTpath_load* on page 14-11.

Functions

WTpath_new

```
WTpath *WTpath_new(  
    NULL);
```

This function creates and returns a pointer to a new path. The path is initially empty, that is, it contains no elements. *NULL* is passed in as the only argument. (The *NULL* parameter is a redundancy necessitated by earlier releases of WTK.) necessary

A path has a variety of state parameters, summarized in the following list. The default values listed here are the values set when a new path is constructed with *WTpath_new*:

<i>Visibility</i>	TRUE (on) or FALSE (off). The default is FALSE (off). See <i>WTpath_setvisibility</i> on page 14-8 and <i>WTpath_setmarker</i> on page 14-9.
<i>Direction</i>	<i>WTDIRECTION_FORWARD</i> or <i>WTDIRECTION_BACKWARD</i> . The default is <i>WTDIRECTION_FORWARD</i> . See <i>WTpath_setdirection</i> on page 14-19.
<i>Play mode</i>	<i>WTPLAY_TOEND</i> , <i>WTPLAY_CONTINUOUS</i> , and/or <i>WTPLAY_OSCILLATE</i> . The default is <i>WTPLAY_TOEND</i> . See <i>WTpath_setmode</i> on page 14-21.
<i>Speed</i>	The default playback speed is 1 (one) element per frame. See <i>WTpath_setplayspeed</i> on page 14-22.
<i>Sample rate</i>	The default sample rate is 1 (one) element recorded each frame. See <i>WTpath_setsamples</i> on page 14-22.
<i>Constraints</i>	The default is none. See <i>WTpath_setconstraints</i> on page 14-20.

By default, a path created with *WTpath_record* is associated with the motion of the current viewpoint. Any viewpoint's motion can be recorded in a path by first assigning that viewpoint to be the current viewpoint (if it isn't already) using *WTuniverse_setviewpoint* (see page 2-15). To associate a path with an entity other than the viewpoint (and which could be controlled by a sensor) use *WTpath_setrecordlink* (see page 14-15).

To visually represent the path, a marker is displayed at each element of the path. This marker (a geometry) can be set using *WTpath_setmarker* (see page 14-9). As no default marker will be used, you must set the marker before setting the visibility of a path with *WTpath_setvisibility* (see page 14-8).

WTpath_delete

```
void WTpath_delete(  
    WTpath *path);
```

This function deletes the path specified by the *path* argument. If the path is playing, it is stopped. All elements belonging to the path are deleted, as are all markers used to display the path if it is visible. The path is removed from the universe's list of paths, and all memory used by the path is released.

WTpath_copy

```
WTpath *WTpath_copy(  
    WTpath *path);
```

This function copies an existing path. It creates a new path with a sequence of path elements with the same position and orientation values as in the original path. No other information is copied from the original path to the new path, so the new path's play direction, visibility, and other state values are the same as those of a path just constructed with *WTpath_new*.

If successful, a pointer to the copy of the path is returned, otherwise NULL is returned.

Note: If you plan to visually display the copied path, you must call *WTpath_setmarker* for the new copied path, as in the example below.

```
WTpath *path, *copy;  
WTgeometry *newmarker;  
  
copy = WTpath_copy(path);  
WTpath_setmarker(copy, newmarker);
```

WTpath_interpolate

```
WTpath *WTpath_interpolate(  
    WTpath *path,  
    int nsectors,  
    int method);
```

This function creates a new path by interpolating between the elements of the specified path. The new path's play direction, visibility, and other state values are the same as those of a path just constructed with *WTpath_new*.

If successful, the new path is returned, otherwise NULL is returned. The original path is unaffected by this operation.

The *nsectors* argument specifies the number of sectors to be created between each of the elements of the original path. This number must be 1 (one) or greater and the original path must have at least two elements for the interpolation to be successful.

The *method* argument indicates the approach to be used to generate the *positions* of the interpolated points (see figure 14-2). The possible values for *method* are as follows:

<i>WTPATH_LINEAR</i>	For a straight line path between elements.
<i>WTPATH_BEZIER</i>	For a Bezier curve.
<i>WTPATH_BSPLINE</i>	For a B-spline curve.

The *orientations* of the elements are also interpolated, however the method used to interpolate orientations is always linear, independent of the method chosen to interpolate positions.

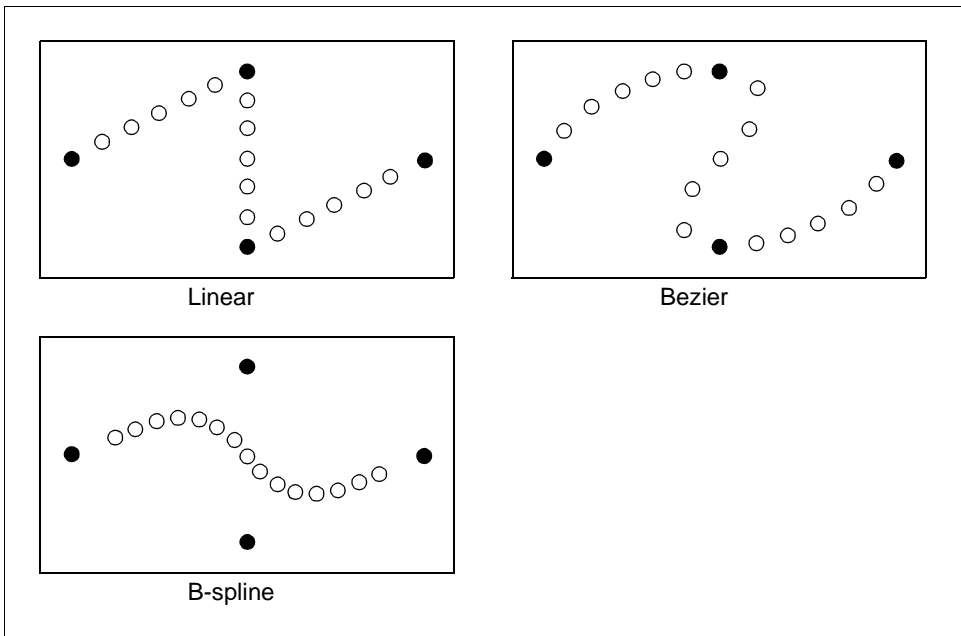


Figure 14-2: Path Interpolation methods

The Linear path interpolation option places the interpolated points along a straight line between each pair of points in the original path.

The Bezier option may be the most generally useful since it gives a smooth curve that passes through the elements of the original path. WTK sets the control points for the Bezier interpolation so that the tangent vector to the curve at any point on the original path is parallel to the vector from the previous point on the path to the next one.

The B-spline option produces a curve which is the “smoothest” of all the options, but which does not in general pass through the elements of the original path.

An example of calling *WTpath_interpolate* is the following. Note that after calling this function, you can delete the original path if it is no longer needed.

```
WTpath *oldpath, *newpath;  
newpath = WTpath_interpolate(oldpath, 6, WTPATH_BEZIER);
```

The new path created by the example above has six elements between every pair of elements in the original path, or 7 times as many elements as *oldpath* (plus one). The elements of the new path lie on a Bezier curve through the elements of the original path.

Path Management

There can be many multiple active paths in the universe, each in a different state. To find the list of all paths in the universe, use *WTuniverse_getpaths* (see page 2-13), then *WTpath_next* (see page 14-10) to iterate through the list. By default, paths are associated with the universe's current viewpoint (see *WTuniverse_setviewpoint* on page 2-15). You can use the function *WTpath_setrecordlink* (see page 14-15) to associate a path with any other entity.

Paths can be made visible by calling *WTpath_setvisibility* (see below). When visible, the path's "marker," passed in as the argument to *WTpath_setmarker* (see page 14-9), is replicated at each path element location. The effect is that you can see the entire path as a string of geometries through space.

WTpath_setvisibility

```
void WTpath_setvisibility(  
    WTpath *path,  
    FLAG flag);
```

This function toggles the visibility of a path's graphical representation. When visible, a copy of the path's marker appears at each element of the path (with the path element's position and orientation). The *flag* argument should be TRUE to make the path visible and FALSE to make it invisible.

By default, paths are invisible. Once a path is made visible for the first time, subsequent calls to `WTpath_setvisibility` adds or removes the marker replicas from the universe as needed. See the example below, under `WTpath_getvisibility`.

Also see `WTpath_setmarker` on page 14-9 and `WTpath_getmarker` on page 14-10.

WTpath_getvisibility

```
FLAG WTpath_getvisibility(  
    WTpath *path);
```

This function returns TRUE if a path is currently visible, otherwise FALSE. In the following example, a path's visibility is toggled, so that if it was invisible it becomes visible and vice versa:

```
WTpath *path;  
WTpath_setvisibility(path, !WTpath_getvisibility(path));
```

WTpath_setmarker

```
void WTpath_setmarker(  
    WTpath *path,  
    WTgeometry *marker);
```

This function sets the geometry that will be used to display a path element, when the path's visibility is TRUE. The function takes in the relevant path and the geometry as arguments.

This function is provided to visualize a path that has been recorded, loaded or created. The requested geometry is displayed at every path element position. It is best to use a very simple geometry model (with very few polygons) for the marker since potentially hundreds of copies of it could be visible in the simulation. If either the path or the geometry is NULL, the function returns without any effect.

Note: This function should only be called for a path that has never been made visible.

WTpath_getmarker

```
WTgeometry *WTpath_getmarker(  
    WTpath *path);
```

This function returns a pointer to the geometry that is currently being used to represent path elements when the path is made visible.

The function returns NULL when the path passed in is void. This function should be used only after a call to *WTpath_setmarker*, otherwise NULL will be returned since there is no default geometry to make the path elements visible.

WTpath_getelements

```
WTpathelement *WTpath_getelements(  
    WTpath *path);
```

This function returns a pointer to the first element in the specified path's list of elements. Use *WTpathelement_next* to iterate through the path's list of elements. If the path does not contain any elements, for example if the path was just created with *WTpath_new*, NULL is returned. See also *WTpath_getcurrentelement* on page 14-18.

WTpath_numelements

```
int WTpath_numelements(  
    WTpath *path);
```

This function returns the number of elements in a path's element list.

WTpath_next

```
WTpath *WTpath_next(  
    WTpath *path);
```

This function iterates through the universe's list of paths. If the *path* argument is NULL, or if the specified path is the last path on the list, then NULL is returned.

The following example uses *WTpath_next* to iterate through the universe's list of paths, turning off the visible representation of each path:

```
WTpath *path;
for ( path=WTuniverse_getpaths() ; path ; path=WTpath_next(path) ) {
    WTpath_setvisibility(path, FALSE);
}
```

Loading and Saving Paths

Paths can be saved to files and loaded back again. The file created when a path is saved contains a simple ASCII listing of the path elements' coordinates. No other information about the path is saved.

WTpath_load

```
WTpath *WTpath_load(
    char *filename,
    NULL);
```

This function creates a new path by loading in path data from the file specified by the *filename* argument. The new path consists of one element for each position and orientation record in the path file. Aside from the elements constructed from the file, the state of the new path is the same as that of a path constructed using *WTpath_new*. If successful, a complete, new path is returned, otherwise NULL is returned.

See *WTpath_new* on page 14-12 for more information about the default state of a newly-created path, and how to set a marker to visualize it.

WTpath_save

```
FLAG WTpath_save(  
    WTpath *path,  
    char *filename);
```

This function saves a path to the file specified by *filename*. The file that is written contains a sequential list of the positions and orientations of the elements making up the path. Success is indicated by the return value; TRUE indicates success, otherwise FALSE is returned.

Path File Format

WTK path files are usually given a .pth extension. (Note however, that this is not necessary). A WTK path file stores position and orientation records in the ASCII format. An example path file is shown below:

```
path record v.2  
8  
Posi -10.60 651.98 875.80  
Orie -0.33 0.01 -0.004 0.94  
Posi -6.14 646.44 3885.47  
Orie -0.33 0.011 -0.004 0.94  
Posi -1.680267 640.893738 3895.134277  
Orie -0.334686 0.011820 -0.004199 0.942246  
Posi 2.781840 635.345947 3904.798096  
Orie -0.334686 0.011820 -0.004199 0.942246  
Posi 7.243946 629.798096 3914.461914  
Orie -0.335399 0.014391 -0.005124 0.941952  
Posi 11.024731 624.091980 3923.803711  
Orie -0.335399 0.014391 -0.005124 0.941952  
Posi 14.805515 618.385864 3933.145508  
Orie -0.335399 0.014391 -0.005124 0.941952  
Posi 18.586300 612.679749 3942.487305  
Orie -0.335399 0.014391 -0.005124 0.941952
```


The first line is used by WTK to identify that the file is indeed a path file in the WTK format. If the path file is generated by WTK (using `WTpath_save`), this line is inserted for you. If you are generating the file using an external editor, make sure the first line says "path record v.2".

The second line indicates the number of position and orientation pairs contained in the file. In the above example, this number is 8, which means that there are 8 position records and 8 orientation records in the file. If the number of records in the file is less than this value, WTK will not load the file. If the number of records in the file is greater than this number, WTK will quit reading the file after the specified number of records have been read. A path can contain any (non-negative) number of position and orientation pairs.

The path data follows next, as a sequence of alternating position and orientation records. Each pair of position and orientation records constitutes a 'path element'.

A position record begins with the keyword "Posi", and consists of three floats representing the x,y and z values of that position respectively. An orientation record begins with the keyword "Orie", and consists of four floats that represent the orientation in the form of a quaternion.

Recording and Playback

Many options are provided for recording and playing back WTK paths. It may help to think of WTK paths as analogous with a common tape deck or VCR, since similar functions are available: play, record, stop, rewind, etc. One difference between using a WTK path and using a VCR is that the path only affects the viewpoint or any other associated entity when it is being played – you can rewind the path or change the current element setting without actually moving the viewpoint there.

By default, a path plays (either forward or backward) until the end (or beginning) of the path is reached, and then it stops. Using alternative playback modes (set with *WTpath_setmode*) a path can be played back continuously and can be made to play backwards and forwards between its two ends.

WTpath_play

```
void WTpath_play(  
    WTpath *path);
```

This function begins the playback of the indicated path starting from the path's current element. Prior to calling this function, a motion link connecting the specified path with a target object must have been created. (See *WTmotionlink_new* on page 15-3.) When a path plays, the target of the motion link associated with the path is moved from element to element along the path. At any given path element, the target of the motion link (viewpoint, transform or movable) is given the position and orientation stored with the path element.

Once *WTpath_play* is called, the path continues to play until either *WTpath_stop* is called or the conditions for stopping, as determined by the path's play mode, are met.

You cannot simultaneously play and record a path. If the path you wish to play is currently recording, call *WTpath_stop* before calling *WTpath_play*.

See also *WTpath_stop*, *WTpath_setplayspeed*, *WTpath_setdirection*, *WTpath_setmode*, and *WTpath_setcurrentelement*.

WTpath_play1

```
void WTpath_play1(  
    WTpath *path);
```

This function begins the playback of the indicated path starting from the path's current element, but plays for one frame only. Depending on the path's play speed, the viewpoint, or any associated entity may or may not advance when *WTpath_play1* is called.

WTpath_record

```
FLAG WTpath_record(  
    WTpath *path);
```

This function starts recording the position and orientation of the current viewpoint (default) or the position and orientation of the target of a motion link. (See *WTpath_setrecordlink* on page 14-15.) By default, position and orientation are recorded once per frame, however this sample rate can be changed by calling *WTpath_setsamples*.

Each position/orientation record obtained while recording is stored in a new path element that is added to the end of the specified path. In this way, you can use *WTpath_record* to build a completely new sequence of path elements for a newly constructed path or to add new path elements to the end of an existing path.

To stop recording, call *WTpath_stop*. You can not simultaneously play and record a path. If the path you wish to record is currently playing, you must either call *WTpath_stop* first or wait until the path finishes playing. The return value indicates success or failure.

WTpath_record1

```
FLAG WTpath_record1(  
    WTpath *path);
```

This function starts recording the position and orientation of the current viewpoint (default) or the position and orientation of the target of a motion link, but only one frame is recorded.

WTpath_setrecordlink

```
FLAG WTpath_setrecordlink(  
    WTpath *path,  
    Wtmotionlink *link)
```

Use this function to record the motion of the target of a motion link. The motion link is expected to have been created with a valid source (a sensor or another path), and a valid target (a viewpoint, a transform node, a node path, or a movable node). If the path does not already exist, *WTpath_new* must be called to create a new path prior to calling this function.

The path should be stopped (i.e., not playing or recording) at the time *WTpath_setrecordlink* is called. If the path you wish to record is currently playing, you must either call *WTpath_stop* first or wait until the path finishes playing.

WTpath_setrecordlink returns TRUE if it is able to begin recording or FALSE if either the path or the motion link is void, or if the path is already playing or recording.

If this function is not called first, then *WTpath_record* will record the position and orientation of the current viewpoint. To record the position and orientation of an entity other than the current viewpoint, you must call *WTpath_setrecordlink* prior to calling *WTpath_record* or *WTpath_record1*. To begin recording, call *WTpath_record* or *WTpath_record1* after calling this function. Once the path has been recorded, you can create

a motion link between this newly created path and any target for playback. See *WTmotionlink_new* on page 15-3.

WTpath_stop

```
void WTpath_stop(  
    WTpath *path);
```

This function stops a path that is either playing or recording.

WTpath_rewind

```
void WTpath_rewind(  
    WTpath path);
```

This function sets a path's current pointer to the path's first element. Only the path's pointer and, not the current viewpoint (or other entity associated with the path), is moved by this call. To move the current viewpoint (or other entity associated with the path), to the current element, call *WTpath_showcurrentelement*.

WTpath_isplaying

```
FLAG WTpath_isplaying(  
    WTpath *path);
```

This function returns TRUE if the specified path is currently playing, otherwise it returns FALSE.

WTpath_isrecording

```
FLAG WTpath_isrecording(  
    WTpath *path);
```

This function returns TRUE if the specified path is currently being recorded, otherwise it returns FALSE.

WTpath_showcurrentelement

```
void WTpath_showcurrentelement(  
    WTpath *path);
```

This function moves the current viewpoint (or other entity associated with the path) to the position and orientation of the path's current element. In the following example, the viewpoint is moved to the first element of a path:

```
WTpath *path;  
WTpath_rewind(path);  
WTpath_showcurrentelement(path);
```

WTpath_setcurrentelement

```
FLAG WTpath_setcurrentelement(  
    WTpath *path,  
    WTpathelement *element);
```

This function sets the current element of a path. The current element is the element from which play begins, when *WTpath_play* or *WTpath_play1* is called. It is also the element after which a new element is inserted when *WTpath_insertelement* is called. This function affects only the current element setting, not the location of the current viewpoint. To move the current viewpoint (or an entity associated with the path) to the current element location after calling *WTpath_setcurrentelement*, call *WTpath_showcurrentelement* as in the following example:

```
WTpath *path;  
WTpathelement *element;  
WTpath_setcurrentelement(path, telement);  
WTpath_showcurrentelement(path);
```

If successful, TRUE is returned. Otherwise, for example if the specified path element does not belong to the path, then FALSE is returned.

WTpath_getcurrentelement

```
WTpathelement *WTpath_getcurrentelement(  
    WTpath *path);
```

This function returns a path's current element. If the path has no elements, for example, if the path was just created with *WTpath_new*, then NULL is returned.

WTpath_seek

```
FLAG WTpath_seek(  
    WTpath *path,  
    int offset,  
    int where);
```

This function moves a path's current element pointer forward or backward in the path's element list. The *offset* value, which can be either positive or negative, specifies the number of elements to move. The *where* argument specifies the starting point from which the offset is made. Valid values of *where* are:

- *WTPATH_FIRST*
- *WTPATH_CURRENT*
- *WTPATH_LAST*

The return value is TRUE if successful and FALSE if the seek is invalid, that is, if an attempt is made to seek to a non-existent position in the list.

For example, to move the element position backward by one, call:

```
WTpath_seek(path, -1, WTPATH_CURRENT);
```

To move the element position forward by two, call:

```
WTpath_seek(path, 2, WTPATH_CURRENT);
```

To move the element position to the third element in the list (two ahead of the first element), call:

```
WTpath_seek(path, 2, WTPATH_FIRST);
```

To move the element position to three before the last element, call:

```
WTpath_seek(path, -3, WTPATH_LAST);
```

If there were 10 elements in the list, after the above call to *WTpath_seek* the current element position would be at the 7th element.

Additional examples of using *WTpath_seek* are provided below under *WTpath_setdirection*.

WTpath_setdirection

```
void WTpath_setdirection(  
    WTpath *path,  
    FLAG flag);
```

This function sets the play direction of a path. The *flag* argument should be *WTDIRECTION_BACKWARD* for backward or *WTDIRECTION_FORWARD* for forward. The default play direction for a path is forward. In the following example, a path is made to play back and forth between its fifth and tenth elements. This example assumes that a path with at least this many elements has been constructed.

```
WTpath *path;  
WTpathelement *element, *element5, *element10;  
/* make sure we actually have this many elements */  
if ( WTpath_numelements(path)<10 )  
    WTwarning("Don't proceed\n");  
/* get pointers to the 5th and 10th elements */  
WTpath_seek(path, 9, WTPATH_FIRST);  
element10 = WTpath_getcurrentelement(path);  
WTpath_seek(path, 4, WTPATH_FIRST);  
element5 = WTpath_getcurrentelement(path);  
  
/* set the play direction to forward and start playing from the  
5th element. */  
WTpath_setdirection(path, WTDIRECTION_FORWARD);  
WTpath_play(path);  
  
/*..... the simulation is run..... */
```

```
/* reverse the path playback direction when the 5th and 10th elements
are reached while the simulation runs. */
element = WTPath_getcurrentelement(path);
if ( element==element5 )
    WTPath_setdirection(path, WTDIRECTION_FORWARD);
else if ( element==element10 )
    WTPath_setdirection(path, WTDIRECTION_BACKWARD);
```

WTPath_getdirection

```
FLAG WTPath_getdirection(
    WTPath *path);
```

This function returns a path's play direction, either *WTDIRECTION_BACKWARD* or *WTDIRECTION_FORWARD*.

WTPath_setconstraints

```
void WTPath_setconstraints(
    WTPath *path,
    short constraints);
```

This function constrains the position and orientation information played back by a path. This is accomplished by passing in a combination of the flags listed below separated by the C language bitwise OR operator “|”.

One particularly useful application of this function is to provide a guided tour around a simulation for someone wearing a head-mounted display. In this case it is often desirable to have the viewpoint follow the path, while leaving orientations under the complete control of the user as their head motion is tracked. The following line of code constrains the playback of path orientations (rotations):

```
WTPath *path;
WTPath_setconstraints(path, WTCONSTRAIN_XROT |
    WTCONSTRAIN_YROT | WTCONSTRAIN_ZROT);
```

It is not possible to constrain path rotations about the individual coordinate axes independently. Turning on any of the rotational constraints (*WTCONSTRAIN_XROT*, *WTCONSTRAIN_YROT*, or *WTCONSTRAIN_ZROT*) effectively turns all of them on.

Similarly, it is not possible to constrain path translations along the individual coordinate axes independently. Turning on any of the translational constraints (*WTCONSTRAIN_X*, *WTCONSTRAIN_Y*, or *WTCONSTRAIN_Z*) effectively turns all of them on.

Also see *WTsensor_setconstraints* on page 13-21.

WTpath_getconstraints

```
short WTpath_getconstraints(  
    WTpath *path);
```

This function returns a path's constraints, as set by *WTpath_setconstraints*. The default value is 0 (zero), meaning no constraints are applied.

A restriction on the use of path constraints is described under *WTpath_setconstraints*.

WTpath_setmode

```
void WTpath_setmode(  
    WTpath *path,  
    short mode);
```

This function sets a path's playback mode. The following list summarizes the possible values of the *mode* argument.

<i>WTPLAY_TOEND</i>	The path plays in its current direction until it reaches either end of the path, then it stops.
<i>WTPLAY_CONTINUOUS</i>	The path plays in its current direction until it reaches either end of the path, then it repeats continuously. For example, when a forward-playing path reaches the end of the path, it starts playing again from the beginning of the path.
<i>WTPLAY_OSCILLATE</i>	When a playing path reaches either end of the path, it stops, but its direction is reversed.
<i>WTPLAY_OSCILLATE</i> / <i>WTPLAY_CONTINUOUS</i>	The path plays continuously backward and forward between the ends of the path.

If the fourth option above is set, as in the following example, the path will both change direction *and* keep going when it reaches either end of the path:

```
short mode;
WTpath *path;
mode = WTPLAY_CONTINUOUS | WTPLAY_OSCILLATE;
WTpath_setmode(path, mode);
```

WTpath_getmode

```
short WTpath_getmode(
    WTpath *path);
```

This function returns a path's play mode, as set by *WTpath_setmode*. The return value is either *WTPLAY_TOEND*, *WTPLAY_CONTINUOUS*, *WTPLAY_OSCILLATE* or *WTPLAY_CONTINUOUS|WTPLAY_OSCILLATE*.

The following code fragment calls *WTpath_getmode* to determine whether the mode *WTPLAY_OSCILLATE* has been set for the path:

```
WTpath *path;
if ( WTpath_getmode(path) & WTPLAY_OSCILLATE )
    WTmessage("Path set to oscillate\n");
else
    WTwarning("Path not set to oscillate\n");
```

WTpath_setplayspeed

```
void WTpath_setplayspeed(
    WTpath *path,
    int speed);
```

This function sets the playback speed for a path. The speed is the number of path elements advanced each frame of the simulation. The *speed* argument must be an integer greater than or equal to 1 (one). The default speed is 1 (one).

WTpath_getplayspeed

```
int WTpath_getplayspeed(  
    WTpath *path);
```

This function returns the playback speed of a path. The default value is 1 (one).

WTpath_setsamples

```
void WTpath_setsamples(  
    WTpath *path,  
    int frames_per_element);
```

This function sets a path's sample rate, that is, the number of frames of the simulation which elapse for each recorded element. For example, if *frames_per_element* is 10, an actively recording path will record position and orientation information once every 10 frames. The *frames_per_element* argument must be an integer greater than or equal to 1 (one). The default sample rate is 1 (one), meaning that one element is created each frame.

This function allows you to save memory by recording fewer elements. This is especially useful for long paths and/or high frame rates.

WTpath_getsamples

```
int WTpath_getsamples(  
    WTpath *path);
```

This function returns the sample rate of a path. The default value is 1 (one).

Path Element Management

The WTpathelement Class

The individual elements in a path are a WTK class of their own, the *WTpathelement* class. With this class you can create a path element by element or edit an existing path. There are functions for creating, deleting, and copying path elements, and functions for adding and removing path elements from paths. You can also set and get the locations of path elements directly. Once a path element is created, it can be added to a path with either *WTpath_insertelement* or *WTpath_appendelement*.

WTpathelement_new

```
WTpathelement *WTpathelement_new(  
    WTPq *location);
```

This function creates and returns a pointer to a new path element, which is initialized to the specified position and orientation.

Position and orientation are specified in the *location* structure. The path element does not belong to any path until specifically added to one with *WTpath_insertelement* or *WTpath_appendelement*.

A path element can belong to only one path at a time. If a path element is currently in a path and you wish to insert it in another path, it must first be removed from the path it is in using *WTpathelement_remove*.

WTpathelement_delete

```
void WTpathelement_delete(  
    WTpathelement *element);
```

This function deletes a path element and frees the memory used. If the path element is a member of a path, it is first removed from the path and then deleted.

WTpathelement_remove

```
void WTpathelement_remove(  
    WTpathelement *element);
```

This function removes a path element from the path that references it but does not delete it. If the path element does not belong to a path, this function has no effect.

WTpathelement_copy

```
WTpathelement *WTpathelement_copy(  
    WTpathelement *element);
```

This function creates a copy of the path element pointed to by the *element* argument. The copy is a new path element with the same position and orientation as the original one. If successful, a pointer to the path element copy is returned, otherwise NULL is returned. The new path element does not belong to any path.

WTpathelement_setposition

```
void WTpathelement_setposition(  
    WTpathelement *element,  
    WTp3 pos);
```

This function sets the position of a single path element to the location specified in *pos*. Path element positions are the positions to which the viewpoint (or an entity associated with the path) is moved as a path is played back.

WTpathelement_getposition

```
void WTpathelement_getposition(  
    WTpathelement *element,  
    WTp3 pos);
```

This function retrieves the position of the specified path element and places it in *pos*.

WTpathelement_setorientation

```
void WTpathelement_setorientation(  
    WTpathelement *element,  
    WTq q);
```

This function sets the orientation of a single path element to the orientation specified by *q*.

WTpathelement_getorientation

```
void WTpathelement_getorientation(  
    WTpathelement *element,  
    WTq q);
```

This function retrieves the orientation of a single path element and places it in *q*.

WTpathelement_getpath

```
WTpath *WTpathelement_getpath(  
    WTpathelement *element);
```

This function returns a pointer to the path to which a path element belongs. If the path element does not belong to any path, NULL is returned. Path elements are assigned to a path either automatically when a path is in record mode or with the functions *WTpath_appendelement* or *WTpath_insertelement*.

WTpathelement_next

```
WTpathelement *WTpathelement_next(  
    WTpathelement *element);
```

This function returns the next element in a list of path elements. Use this function to iterate through the list of elements in a path, as in the following example.

```
WTp3 p;  
WTpathelement *element;  
WTpath *path;
```

```
/* Display the positions of the elements in a path */
for ( element=WTpath_getelements(path) ; element ;
      element=WTpathelement_next(element)) {
    WTpathelement_getposition(element, p);
    WTp3_print(p, "element position");
}
```

Path Editing

These path-editing functions let you add elements to the end of the path or insert elements at the current element position. Elements can be removed from a path and/or deleted with *WTpathelement_remove* and *WTpathelement_delete*, which are described in the previous section.

WTpath_appendelement

```
void WTpath_appendelement(
    WTpath *path,
    WTpathelement *element);
```

This function appends a path element onto a specified path's list of elements, making it the last element of the path. The *element* argument is a pointer to an existing path element object. A path element can only belong to one path at a time. If the path element pointed to by the *element* argument is already in a path, then this function has no effect. To append this element to the new path, first call *WTpathelement_remove* to remove it from the old path.

WTpath_insertelement

```
void WTpath_insertelement(
    WTpath *path,
    WTpathelement *element);
```

This function inserts a path element into a path's list of elements at the path's current position. The element is inserted immediately after the path's current element.

An element can only belong to one path at a time. If the path element pointed to by the *element* argument is already in a path, then this function has no effect. To insert this element into the new path, first call *WTpathelement_remove* to remove it from the old path.

The *element* argument is either a pointer to an existing path element or it may be NULL. If *element* is NULL, then a new path element is created and inserted into the path, and the position and orientation of this new path element are taken from the current viewpoint (or any other entity associated with the path).

For example, suppose that you wish to insert an element in a path so that the path passes through the world coordinate origin with the same orientation as the path element just before the inserted element. The following example shows how to create such an element between the fifth and sixth elements in a path:

```
WTpath *path;
WTpathelement *element;
WTpq location;

/* go to the 5th element in the path (the 4th element after the first one) */
WTpath_seek(path, 4, WTPATH_FIRST);

/* get the orientation of that element and store it in location */
WTpathelement_getorientation(WTpath_getcurrentelement(path),
    location.q);

/* construct a pathelement at the world origin with the same orientation
as the 5th element */
WTp3_init(location.p);
element = WTpathelement_new(&location);

/* insert the element in the path after the 5th (current) element */
WTpath_insertelement(path, element);
```


Path Name

WTpath_setname

```
void WTpath_setname(  
    WTpath *path,  
    const char *name);
```

This function sets the name of the specified path. All paths have a name; by default, a path's name is "" (i.e., a NULL string).

WTpath_getname

```
const char *WTpath_getname(  
    WTpath *path);
```

This function returns the name of the specified path.

User-specifiable Path Data

A *void* pointer is included as part of the structure defining a path, so that you can store whatever data you wish with a path. The following functions can be used to set and get this field within any path.

WTpath_setdata

```
void WTpath_setdata(  
    WTpath *path,  
    void *data);
```

This function sets the user-defined data field in a path. Private application data can be stored in any structure. To store a pointer to the structure within the path, pass in a pointer to the structure, cast to *void**, as the *data* argument.

WTpath_getdata

```
void *WTpath_getdata(  
    WTpath *path);
```

This function retrieves user-defined data stored within a path. Cast the value returned by this function to the same type used to store the data with the *WTpath_setdata* function.

Introduction

Sensors and paths allow you to interact with a virtual world by providing you with control over the motion of objects or the viewpoint. To associate a sensor (or a path) with an entity in a world, use motion links. A motion link connects a *source* of position and orientation information with a *target* that moves to correspond with that changing set of information.

Motion Link Sources and Targets

The motion link source can be a path or a sensor. Motion link targets include the following:

- **viewpoint:** Use this as your target when you want to control your viewpoint by the source you've specified.
- **transform node:** Use this as your target when you want your source to affect a specific transformation in the scene graph, such as the one that controls wrist movement in a human figure.
- **node path:** Use this as your target when you want your source to affect the cumulative set of transformations used for a specific node, as when you want to control the position of a human figure in the world coordinate frame. Note that the leaf node of the node path must be either a transform node or a movable node.
- **movable node:** Use this as your target when you want your source to affect a movable node (with or without attachments). Refer to the *Movable Nodes* chapter (starting on page 5-1) for more information about movable nodes.

Figure 15-1 illustrates the targets that can be attached to a sensor using a motion link. Although a sensor is shown on one end (the source) of the motion link. A path can also be used as the source that is connected to a target via a motion link.

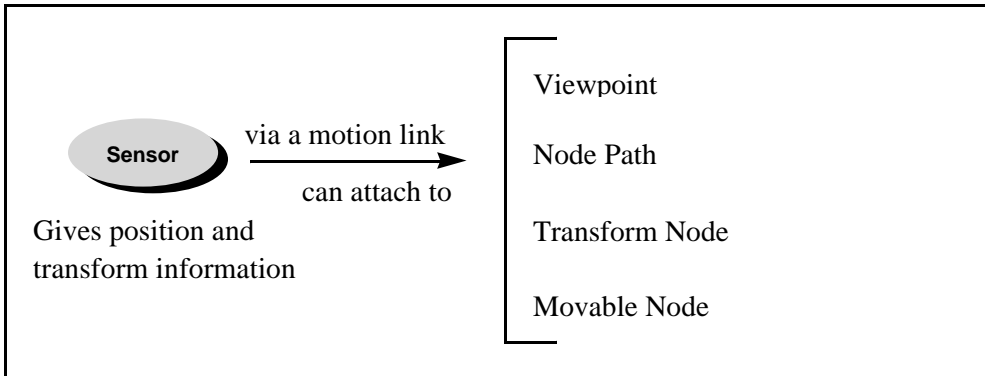


Figure 15-1: Some ways to use motion links

Once a motion link is created (*WTmotionlink_new* on page 15-3), position and orientation records from the motion link source automatically cause corresponding translation and rotation of the motion link's target. If the target has more than one motion link associated with it, each of these motion links contributes to the motion of the target.

You can also use a path as the source of position and orientation information which can then be directed to some object by a motion link. This would be an advantage if you want to move a viewpoint through your scene along a defined path. If you have a Grand Canyon simulation, for example, you can define a path through the best parts of the canyon, then attach the path to the viewpoint using a motion link.

Additionally, you can use functions like *WTpath_setrecordlink* (see page 15-14) to create a path from the position and orientation information being transmitted by a motion link. This path can then be used as a source of position and orientation information to some target using another motion link.

Reference Frames

When you create a new motion link, the source affects the position and orientation of the target relative to a particular reference frame. The default reference frame used for a newly created motion link is dependent upon the target type. The target types and their default reference frames are as listed in table 15-1 on page 15-4.

It is possible to change the reference frame in which the source's position and orientation information is applied to the motion link's target by using the function *WTmotionlink_setreferenceframe* (see page 15-8). For example, if you have created a motion link which connects a sensor to a movable, the sensor's position and orientation information will, by default, affect the movable in its local frame. By calling *WTmotionlink_setreferenceframe*, you could apply a sensor's position and orientation information to the movable in a coordinate frame other than the default.

Constraints

WTK lets you add control to a motion link so that the position and/or orientation of the motion link's target is constrained. You can add the constraint along any degree of freedom (DOF) or any combination of DOFs using the *WTmotionlink_addconstraint* (see page 15-11) function.

Motion Link Functions

WTmotionlink_new

```
WTmotionlink *WTmotionlink_new(  
    void *source,  
    void *target,  
    int from_type,  
    int to_type);
```

Arguments:

<i>source</i>	Pointer to either a sensor or a path.
<i>target</i>	Pointer to either a viewpoint, a movable, a transform node, or a node path leading to either a transform node or a movable node.
<i>from_type</i>	The type of the source — one of the following pre-defined constants: WTSOURCE_SENSOR

to_type WTSOURCE_PATH

The type of the target — one of the following pre-defined constants:

WTTARGET_VIEWPOINT

WTTARGET_MOVABLE

WTTARGET_TRANSFORM

WTTARGET_NODEPATH

This function creates a new motion link whose source will affect the position and orientation of the target relative to a particular reference frame. The default reference frame used for a newly created motion link is dependent upon the target type and is shown in table 15-1.

The possible reference frames are *WTFRAME_LOCAL*, *WTFRAME_PARENT*, *WTFRAME_VPOINT*, and *WTFRAME_WORLD*.

Table 15-1: Default Motion Link Reference Frames

Target type	Default reference frame
WTTARGET_VIEWPOINT	WTFRAME_LOCAL
WTTARGET_TRANSFORM	WTFRAME_LOCAL
WTTARGET_NODEPATH	WTFRAME_WORLD
WTTARGET_MOVABLE	WTFRAME_LOCAL

To change the reference frame of a motion link from its default value, use the function *WTmotionlink_setreferenceframe*. For sensors that return absolute records (e.g., FASTRAK, ISOTRAK, InsideTRAK, and Flock of Birds), to either a transform or movable node, you must set the reference frame of the corresponding motion link to *WTFRAME_PARENT* in order to get the expected behavior.

WTmotionlink_delete

```
void WTmotionlink_delete(  
    WTmotionlink *link);
```

This function deletes the specified motion link from the universe's list of motion links, and releases all memory used by the motion link.

WTuniverse_deletelink

See *WTuniverse_deletelink* on page 2-17 for a description.

WTmotionlink_enable

```
void WTmotionlink_enable(  
    WTmotionlink *link,  
    FLAG flag);
```

If the flag is TRUE, this function enables the specified motion link. If the flag is FALSE, this function disables the specified motion link. When disabled, a motion link has no effect on its target. By default, a motion link is enabled, meaning that it is active.

WTmotionlink_isenabled

```
FLAG WTmotionlink_isenabled(  
    WTmotionlink *link);
```

This function returns TRUE if the specified motion link is enabled (i.e., active), and returns FALSE if the motion link is disabled. If a motion link is disabled, it ceases to have effect on its target.

WTmotionlink_setdata

```
void WTmotionlink_setdata(  
    WTmotionlink *link,  
    void *data);
```

This function sets the user-defined data field for the specified motion link. You will have to type cast *data* to a VOID pointer. Use the data field if you need to store any application information that is specific to a motion link.

WTmotionlink_getdata

```
void *WTmotionlink_getdata(  
    WTmotionlink *link);
```

This function retrieves the user-defined data field for the specified motion link. This function returns NULL if you did not set the data field with non-NULL data, by way of *WTmotionlink_setdata*.

WTmotionlink_getsource

```
FLAG WTmotionlink_getsource(  
    WTmotionlink *link,  
    void **source,  
    int *type);
```

Use this function to retrieve the source and source type of the specified motion link. The return value is TRUE if successful. See *WTmotionlink_gettarget* below for an example of usage.

WTmotionlink_gettarget

```
FLAG WTmotionlink_gettarget(  
    WTmotionlink *link,  
    void **target,  
    int *type);
```

Use this function to retrieve the target and target type of the specified motion link. The return value is TRUE if successful.

Example:

```
WTmotionlink *link;
void *from, *to;
int from_type, to_type;
WTmotionlink_getsource(link, &from, &from_type);
WTmotionlink_gettarget(link, &to, &to_type);
switch (from_type){
    case WTSOURCE_SENSOR:
        WTmessage("From a sensor");
        break;
    case WTSOURCE_PATH:
        WTmessage("From a path");
        break;
}
switch (to_type){
    case WTTARGET_VIEWPOINT:
        WTmessage(" to a viewpoint.\n");
        break;
    case WTTARGET_MOVABLE:
        WTmessage(" to a movable.\n");
        break;
    case WTTARGET_TRANSFORM:
        WTmessage(" to a transform.\n");
        break;
    case WTTARGET_NODEPATH:
        WTmessage(" to a nodepath.\n");
        break;
}
```

WTuniverse_getmotionlinks

See *WTuniverse_getmotionlinks* on page 2-17 for a description.

WTmotionlink_next

```
WTmotionlink *WTmotionlink_next(
    WTmotionlink *link);
```

This function returns the next motion link in the universe’s list of motion links. A pointer to the first link is obtained with a call to *WTuniverse_getmotionlinks*. You can then iterate through the list of existing motion links using *WTmotionlink_next*.

WTmotionlink_setreferenceframe

```
FLAG WTmotionlink_setreferenceframe(
    WTmotionlink *link,
    int frame,
    WTviewpoint *vpoint);
```

Use this function to set the reference frame in which the indicated motion link will operate. A reference frame, (not to be confused with a constraint frame) is the coordinate frame in which motion of the motion link’s target is expected. Depending on the type of the motion link’s target, only certain coordinate frames are valid reference frames. Table 15-2 lists the valid motion link reference frames.

Table 15-2: Valid Motion Link Reference Frames

Target	Valid reference frames
WTTARGET_VIEWPOINT	WTFRAME_LOCAL, (equivalent to WTFRAME_VPOINT) and WTFRAME_WORLD
WTTARGET_TRANSFORM	WTFRAME_PARENT, (equivalent to WTFRAME_WORLD), WTFRAME_LOCAL and WTFRAME_VPOINT
WTTARGET_NODEPATH	WTFRAME_WORLD, WTFRAME_VPOINT and WTFRAME_LOCAL
WTTARGET_MOVABLE	WTFRAME_PARENT, (equivalent to WTFRAME_WORLD), WTFRAME_VPOINT and WTFRAME_LOCAL

If a motion link is to be applied in a viewpoint frame, then a pointer to the pertinent viewpoint is passed in as the third argument, *vpoint*. In this case, if this pointer is invalid the function returns FALSE. In all other cases the *vpoint* argument should be NULL.

This function returns FALSE if an invalid motion link is passed in, or if the requested reference frame is not a valid one, otherwise TRUE is returned. If this function returns FALSE, the reference frame of the specified motion link remains unchanged. When a motion link is created, the reference frame is assigned to a default value, depending upon the target type. See table 15-1 for a list of the default motion link reference frames.

Note: For sensors that return absolute records (e.g., *FASTRAK*, *ISOTRAK*, *InsideTRAK*, and *Flock of Birds*), to either a transform or movable node, you must set the reference frame of the corresponding motion link to *WTFRAME_PARENT* in order to get the expected behavior.

WTmotionlink_getreferenceframe

```
int WTmotionlink_getreferenceframe(  
    WTmotionlink *link);
```

This function returns the frame (*WTFRAME_LOCAL*, *WTFRAME_WORLD*, *WTFRAME_PARENT*, or *WTFRAME_VPOINT*), in which the indicated motion link is applied. If the specified motion link is invalid, -1 is returned.

Constraints on Motion links

Use the following functions to set and manipulate constraints on motion links. You can constrain translation along and rotation about any axis, to either prevent motion entirely or to restrict motion to a specified range.

For ease of use, this release supports the functions *WTsensor_setconstraints* and *WTsensor_getconstraints* (see Chapter 13, *Sensors*). Remember, however, that these functions constrain the values returned by a sensor so they affect all the targets (or entities) that are controlled by that sensor. Constraints on motion links, on the other hand, apply only on the target of the motion link and are not associated with a sensor. That is why they provide better flexibility.

WTmotionlink_setconstraintframe

```
FLAG WTmotionlink_setconstraintframe(  
    WTmotionlink *link,  
    int constraintframe);
```

Use this function to set the *constraint frame* of a motion link. A constraint frame is the coordinate frame in which the constraints on a motion link are applied. If the constraints are to be applied in a frame different from the default one, the new frame is passed in as the argument *constraintframe*. Depending on the motion link's target type, only certain constraint frames are valid. Table 15-3 lists the valid motion link constraint frames.

Table 15-3: Valid Motion Link Constraint Frames

Target	Valid constraint frames
WTTARGET_VIEWPOINT	WTFRAME_LOCAL, (equivalent to WTFRAME_VPOINT) and WTFRAME_WORLD.
WTTARGET_TRANSFORM	WTFRAME_PARENT, (equivalent to WTFRAME_WORLD) and WTFRAME_LOCAL.
WTTARGET_NODEPATH	WTFRAME_WORLD and WTFRAME_VPOINT.
WTTARGET_MOVABLE	WTFRAME_PARENT, (equivalent to WTFRAME_WORLD) and WTFRAME_LOCAL.

This function returns `FALSE` if an invalid motion link is passed in, or if the requested constraint frame is an invalid one, otherwise `TRUE` is returned. When a motion link is created, the constraint frame is set to a default value. If there are no constraints applied upon the motion link, then the constraint frame assigned to this motion link has no significance. The default constraint frame assigned to a motion link is dependent upon the motion link's target type and are listed in table 15-4.

Table 15-4: Default Motion Link Constraint Frames

Target	Default constraint frames
WTTARGET_VIEWPOINT	WTFRAME_LOCAL
WTTARGET_TRANSFORM	WTFRAME_LOCAL
WTTARGET_NODEPATH	WTFRAME_WORLD
WTTARGET_MOVABLE	WTFRAME_LOCAL

WTmotionlink_getconstraintframe

```
int WTmotionlink_getconstraintframe(
    WTmotionlink *link);
```

This function returns the frame (*WTFRAME_LOCAL*, *WTFRAME_WORLD*, *WTFRAME_PARENT* or *WTFRAME_VPOINT*) in which the constraints on the specified motion link are applied. If the motion link is invalid, -1 is returned. Even if no constraints have been applied on the motion link, a *WTFRAME_* value (the default value, if not set) is returned which indicates the frame in which constraints, if added, would be in effect.

See also *WTmotionlink_setconstraintframe* on page 15-10.

WTmotionlink_addconstraint

```
FLAG WTmotionlink_addconstraint(
    WTmotionlink *link,
    int dof,
    float min,
    float max);
```

Use this function to add a constraint to a motion link so that the position and/or orientation of the motion link's target is constrained. The constraint is added along the degrees of freedom (DOF) specified by the *dof* argument (*WTCONSTRAIN_X*, *WTCONSTRAIN_Y*, *WTCONSTRAIN_Z*, *WTCONSTRAIN_XROT*, *WTCONSTRAIN_YROT* or *WTCONSTRAIN_ZROT*). The *min* and *max* arguments specify the range within which the target of the motion link is constrained (within that DOF). When constraining a

translational DOF, *min* and *max* specify coordinates, and when constraining a rotational DOF *min* and *max* represent angles specified in radians.

This function returns FALSE if an invalid motion link is passed in or if the min and max values are unacceptable for the indicated DOF. Valid *min* and *max* values for the different DOFs and target types are discussed below. When constraining a translational DOF, regardless of the motion link's target type, *min* must be less than or equal to *max*. When constraining a rotational DOF with a target type of either *WTTARGET_TRANSFORM*, *WTTARGET_NODEPATH* or *WTTARGET_MOVABLE*, min must be less than or equal to max. When constraining a rotational DOF with a target type of *WTTARGET_VIEWPOINT*, the following rules apply:

- For constraining rotations about the x-axis or the y-axis, *min* and *max* could assume either positive or negative values, with *min* being less than or equal to *max*. The absolute values of *min* and *max* should individually be less than two times Pi. Also, the sum of the absolute values of *min* and *max* should be less than two times Pi.
- For constraining rotation about the z-axis (or twist) apart from min having to be less than or equal to *max*, *min* and *max* must each be between -Pi and Pi.

WTmotionlink_removeconstraint

```
FLAG WTmotionlink_removeconstraint(  
    WTmotionlink *link,  
    int dof);
```

This function removes a particular constraint, if applied, from the specified motion link. The constraint is specified by the degree of freedom (*dof*) argument. For example, *dof* could be *WTCONSTRAIN_X*, *WTCONSTRAIN_YROT* or some other *WTCONSTRAIN_* value. (See *WTmotionlink_addconstraint* on page 15-11.)

If the motion link passed in is invalid, or if the specified constraint does not exist in the motion link's list of constraints, FALSE is returned, otherwise, the specified constraint is removed and TRUE is returned.

Example of Constraining a Motion Link

```
/* Program segment to demonstrate the use of constraints on a motion link between
/* a sensor and a transform node */
```

```
WTmotionlink *link;
WTsensor *sensor;
WTnode *pos_xform, *sens_xform;
WTnode *root;
WTnode *sep;
WTnode *door_node;
WTgeometry *door_geom;
root = WTuniverse_getrootnodes();
sep = WTsepnodes_new( root);
/* Create and set a transform node that sets the door in its global position */
pos_xform = WTxfnodes_new( sep);
WTnode_translate( pos_xform, 4.0f, 0.0f, 4.0f);
/* Create a transform node that will be linked to the sensor to allow sensor
control of the door */
sens_xform = WTxfnodes_new( sep);
door_geom = WTgeometry_newblock( 2.4f, 4.8f, 0.4f, TRUE);
door_node = WTgeometrynode_new( sep, door_geom);
sensor = WTmouse_new();
link = WTmotionlink_new(sensor, sens_xform,
                        WTSOURCE_SENSOR, WTTARGET_TRANSFORM);
/* Set constraints on the motion link to the door, to allow only restricted
rotation around Y-axis. */
WTmotionlink_addconstraint( link, WTCONSTRAIN_X, 0.0f, 0.0f );
WTmotionlink_addconstraint( link, WTCONSTRAIN_Y, 0.0f, 0.0f );
WTmotionlink_addconstraint( link, WTCONSTRAIN_Z, 0.0f, 0.0f );
WTmotionlink_addconstraint( link, WTCONSTRAIN_XROT, 0.0f, 0.0f );
WTmotionlink_addconstraint( link, WTCONSTRAIN_YROT, 0.0f, 1.4f );
WTmotionlink_addconstraint( link, WTCONSTRAIN_ZROT, 0.0f, 0.0f );
```

The above code segment constrains the motion of a door such that it is allowed to rotate only about its Y axis between 0.0 and 1.4 radians. (Translation along all three axes and rotation about the X axis and Z axis is completely restrained.)

WTpath_setrecordlink

See *WTpath_setrecordlink* on page 14-15 for a description.

Introduction

A WorldToolKit viewpoint defines the position and orientation from which all of the geometries associated with a simulation are rendered and projected to the computer screen. Each WTK window has a viewpoint associated with it, and it is from this viewpoint that the scene graph associated with the window is drawn.

When you create a universe with *WTuniverse_new*, WTK automatically creates a viewpoint for it. For many applications, this one viewpoint is sufficient. WTK also lets you construct additional viewpoints and switch between them. For example, you may wish to create a “birds-eye view,” an “out-the-window view,” or a “rear view.” Changing viewpoints in this way is like cutting between different cameras in a movie. (The *Animating Textures* section on page 10-18 of the *Textures* chapter discusses rear-view mirrors.) To create additional viewpoints, or to copy or delete existing viewpoints, see *Basic Viewpoint Management* on page 16-3.

To display several viewpoints simultaneously, you create multiple windows and then use the *WTwindow_setviewpoint* function (see page 17-11) to specify the viewpoint from which the scene is rendered into each window. Each of these windows is associated with a scene graph; alternate views of the same scene would use the same scene graph, while windows depicting different scenes would use different scene graphs. Unlike some systems (such as Open Inventor), viewpoints aren’t nodes in the WTK scene graph; the viewpoint is determined before a scene is rendered.

You can set the position and orientation of a viewpoint through function calls like *WTviewpoint_setposition* and *WTviewpoint_setorientation* (see *Accessing Viewpoint Position and Orientation* on page 16-8). Or, you can control a viewpoint’s position and orientation using a sensor, which you attach to it (see *Linking a Sensor to a Viewpoint* on page 16-6). For example, if a mouse sensor object is constructed and attached to a viewpoint (see *WTmotionlink_new* on page 15-3), you can translate and rotate the viewpoint using mouse motion and button clicks.

You can also manage a viewpoint's motion through viewpoint *pathing*. Using the functions in the *Paths* chapter, you can record a suitable path through a virtual world. You can then play back the path such that the viewpoint moves smoothly along the path. Refer to the *Paths* chapter (starting on page 14-1) and the *Motion Links* chapter (starting on page 15-1) for more information.

Apart from position and orientation, a viewpoint is characterized by other parameters such as aspect ratio, parallax, convergence and convergence distance. These parameters are defined in detail in the description of the function *WTviewpoint_new* on page 16-3. The WTK functions that manipulate these parameters are presented in the sections *Viewpoint Aspect Ratio* on page 16-18 and *Stereo Viewing* on page 16-19.

Figure 16-1 and Figure 16-2 illustrate monoscopic and stereoscopic viewing geometries for symmetric window projections. (For information on the different stereoscopic viewing modes, see page 2-34). The view angle and the hither and yon values are set using functions described in the *Windows* chapter (starting on page 17-1). (These terms are explained in detail on page 17-5.) Note that the view position and orientation is relative to the global (i.e., world) coordinate frame.

In figure 16-1, the view position is the origin of the viewpoint coordinate frame. The view direction is the same as the Z axis of the viewpoint frame. Although the Y axes in the viewpoint frame and the world coordinate frame happen to be parallel, this is not generally the case. The yon clipping plane, which truncates the view pyramid defining its far end, is not shown.

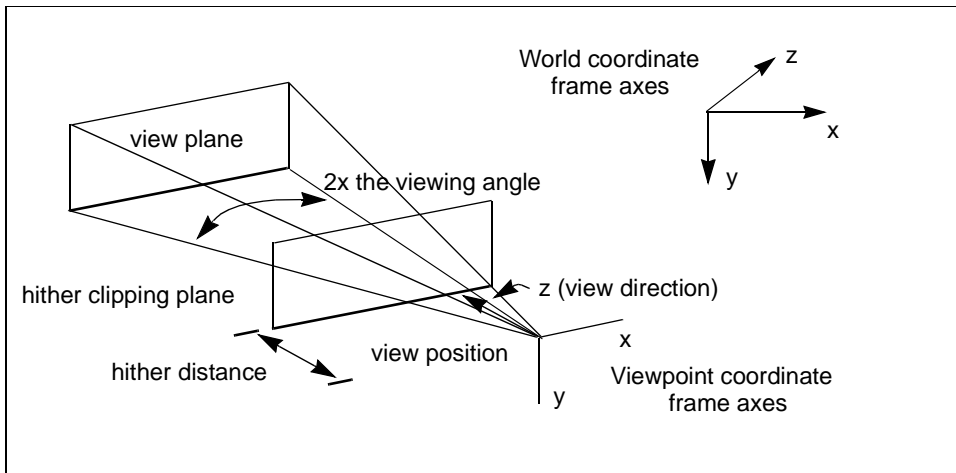


Figure 16-1: Monoscopic viewing geometry

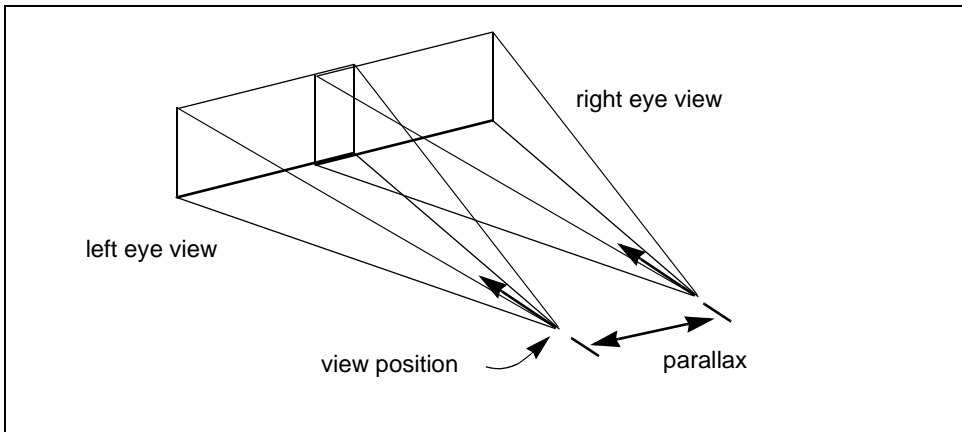


Figure 16-2: Stereoscopic viewing

Figure 16-2 illustrates how stereoscopic viewing has the same parameters as monoscopic viewing, except that there are *two* view pyramids, linearly offset by the parallax distance.

Basic Viewpoint Management

WTviewpoint_new

```
WTviewpoint *WTviewpoint_new(
    void);
```

This function creates and returns a pointer to a new viewpoint object with the following default parameter values:

Position	The origin of the world coordinate frame: (0.0, 0.0, 0.0).
Orientation	Looking straight down the Z axis, with no twist about this axis. From this orientation, the world X axis points to the right, the world Y axis points straight down, and the world Z axis points straight ahead. The corresponding quaternion is (0.0, 0.0, 0.0, 1.0), and the corresponding orientation matrix is the identity matrix.

Direction	Looking straight down the Z axis: (0.0, 0.0, 1.0).
Aspect ratio	1.0. This is a vertical scale factor applied to the screen image. You can use this value to correct for any monitor or pixel distortions that cause spherical or square objects to look flattened.
Parallax	0.0. Both right and left eye views are from the same position. Parallax is the distance between the right and left eye views in the simulation.
Convergence	0. Convergence is a horizontal offset in pixels, which is applied to both the left and right eye images. This offset is subtracted from the left eye and added to the right eye.
Convergence distance	100.0. For asymmetric window projections only. The distance at which a stereoscopic image is perceived to exist. For example, with StereoGraphics CrystalEyesVR LCD Shutter Glasses, this parameter determines the perceived location of an object relative to the plane of the computer screen.

If only one viewpoint is needed for your application, you do not need to call *WTviewpoint_new* because *WTuniverse_new* automatically constructs a viewpoint and adds it to the universe. *WTuniverse_new* (which must be called at the beginning of any WTK application) also creates a window, which by default uses the automatically created viewpoint when the scene is rendered.

The *WTwindow_setviewpoint* function is used to set the viewpoint for a window. The *WTwindow_seteye* function is used to specify whether the view is rendered as seen from the left or right eye.

The universe maintains a list of all viewpoints created with *WTviewpoint_new*. This list can be accessed with *WTuniverse_getviewpoints* (see page 2-15), which returns a pointer to the first viewpoint. You can then iterate through the list using the *WTviewpoint_next* function, which returns the next viewpoint in the list.

WTviewpoint_delete

```
void WTviewpoint_delete(  
    WTviewpoint *viewpoint);
```

This function deletes the specified viewpoint, and frees the memory it uses. WTK does not delete the viewpoint if that is the universe's current viewpoint. You can, however, delete any other viewpoint. All viewpoints are deleted when *WTuniverse_delete* is called.

WTviewpoint_copy

```
WTviewpoint *WTviewpoint_copy(  
    WTviewpoint *old_viewpoint);
```

This function copies an existing viewpoint and returns a pointer to a new viewpoint. The new viewpoint's state is initialized to the values of the original viewpoint. The entire state of the original viewpoint is copied, except for any sensors that may be attached to it. The new viewpoint has no sensors attached to it.

WTviewpoint_next

```
WTviewpoint *WTviewpoint_next(  
    WTviewpoint *viewpoint);
```

This function returns the next viewpoint in the universe's list of viewpoints. A pointer to the first viewpoint is obtained with a call to *WTuniverse_getviewpoints*. You can then iterate through the list of existing viewpoints using *WTviewpoint_next*.

Linking a Sensor to a Viewpoint

It is possible to attach a sensor to a viewpoint, so that the sensor's position and orientation records automatically cause a corresponding translation and rotation of the viewpoint. The easiest way to attach a sensor to a viewpoint is by calling `WTviewpoint_addsensor` as shown in the example below. Motion links, which are described in the *Motion Links* chapter (starting on page 15-1), are a more powerful and general-purpose mechanism for attaching sensors to viewpoints or other entities in the scene graph.

Motion links cause position and orientation information generated by a sensor or a path to be applied to the link's target. A viewpoint is one such target.

Once you've linked a sensor or a path to a viewpoint, translation and rotation of the viewpoint can be controlled by the sensor. If a viewpoint is linked to more than one sensor, each sensor contributes to the motion of the viewpoint.

In the following example, Polhemus ISOTRAK and Spacetec IMC Spaceball sensor objects are created and attached to the viewpoint. This is a useful sensor configuration in setups where head tracking with an absolute sensor such as the ISOTRAK is desired, but where you also want to independently control the viewpoint with a joystick-like device such as the Spaceball.

```
#include "wt.h"

main()
{
    WTSensor *polhemus, *spaceball; /* sensor objects */
    WTnode *root, *scene;

    /* initialize the universe */
    WTuniverse_new(WTDISPLAY_DEFAULT, WTWINDOW_DEFAULT);

    /* create some graphics */
    root = WTuniverse_getrootnodes();
    scene = WTnode_load(root, "myscene", 1.0);

    /* create a polhemus sensor object on serial port SERIAL1 */
    polhemus = WTPolhemus_new(SERIAL1);
```

```
/* create a spaceball sensor object on serial port SERIAL2 */
spaceball = WTspaceball_new(SERIAL2);

/* attach the polhemus and spaceball to the universe's viewpoint */
WTviewpoint_addsensor(WTuniverse_getviewpoints(), polhemus);
WTviewpoint_addsensor(WTuniverse_getviewpoints(), spaceball);

/* prepare to enter the simulation */
WTuniverse_ready();

/* start the simulation */
WTuniverse_go();

/* clean up */
WTuniverse_delete();

return 0;
}
```

This example uses an absolute device and a relative device to control the viewpoint, and is a fairly intuitive configuration to work with. It can be interesting to experiment with different sensor configurations. Not all work equally well, although what works well depends on your particular application. Linking more than one absolute sensor to the same viewpoint, for example, can lead to non-intuitive results if the devices generate input simultaneously. Refer to *Constraints on Motion links* on page 15-9 in the *Motion Links* chapter to constrain the effect of a sensor on the motion of a viewpoint.

WTviewpoint_addsensor

```
void WTviewpoint_addsensor(
    WTviewpoint *viewpoint,
    WTsensor *sensor);
```

This function attaches a sensor to a viewpoint.

WTviewpoint_removesensor

```
void WTviewpoint_removesensor(  
    WTviewpoint *viewpoint,  
    WTsensor *sensor);
```

This function detaches a sensor from a viewpoint object, so that input from the sensor no longer affects the motion of the viewpoint.

Accessing Viewpoint Position and Orientation

When sensors are linked to a viewpoint, the viewpoint moves automatically with input from the sensors. The functions in this section provide additional means for specifying the motion or placement of viewpoints.

Several of the functions in this section take a reference frame as the final argument. If *WTFRAME_LOCAL* or *WTFRAME_VPOINT* is specified, then the viewpoint is translated and/or rotated with respect to its own reference frame.

WTviewpoint_setposition

```
void WTviewpoint_setposition(  
    WTviewpoint *viewpoint,  
    WTp3 p);
```

This function moves the viewpoint to the specified 3D position. The position is specified in the *p* parameter (in world coordinates).

WTviewpoint_getposition

```
void WTviewpoint_getposition(  
    WTviewpoint *viewpoint,  
    WTp3 p);
```


This function retrieves the 3D position of the viewpoint and places it in p . In the case of stereo viewing with non-zero parallax, this is the position of the left eye, as shown in figure 16-2 on page 16-3.

WTviewpoint_getlastposition

```
void WTviewpoint_getlastposition(  
    WTviewpoint *vpoint,  
    WTp3 pos);
```

This function gets a viewpoint's position when the last frame was rendered. The viewpoint is passed in as $vpoint$ and the position is returned in pos .

Technically speaking, this is the viewpoint's position after the completion of the last frame. Before the viewpoint moves in the current frame, *WTviewpoint_getposition* and *WTviewpoint_getlastposition* return the same position. For example, suppose the viewpoint is being controlled by a sensor. Now, working with the default event order, a call to *WTviewpoint_getlastposition* in the actions function would return the same value as would a call to *WTviewpoint_getposition* because the sensor updates have not occurred yet. To effectively use this function, you should change the event order (using *WTuniverse_seteventorder*) such that sensor updates occur before the actions function is called. This way, the sensor updates the viewpoint's position, and *WTviewpoint_getposition* returns the new position, while *WTviewpoint_getlastposition* returns the previous position.

This function is especially useful if you are implementing a collision detection algorithm to prevent the viewpoint from bumping into objects in the universe. After the viewpoint is updated by the sensor, if you detect a collision with any object, you can reset it with the value returned by *WTviewpoint_getlastposition*.

WTviewpoint_translate

```
void WTviewpoint_translate(  
    WTviewpoint *viewpoint,  
    WTp3 p,  
    short frame);
```

This function translates a viewpoint by the specified vector in the world, local/viewpoint frame. The parameter p is the specified vector. The world, local or viewpoint frame are specified by *WTFRAME_WORLD*, *WTFRAME_LOCAL*, or *WTFRAME_VPOINT*. Note that

`WTFRAME_LOCAL` and `WTFRAME_VPOINT` both refer to the reference frame of the viewpoint in this case, and produce the same result when used.

The following code fragment shows how to shift a viewpoint to the right in its own reference frame by one unit. Recall that for any reference frame, the X axis points to the right, the Y axis points straight down, and the Z axis points straight ahead (see figure 16-3).

```

WTviewport *viewport;
WTp3 p;

p[X] = 1.0; p[Y] = p[Z] = 0.0;
WTviewport_translate(view, p, WTFRAME_VPOINT);
    
```

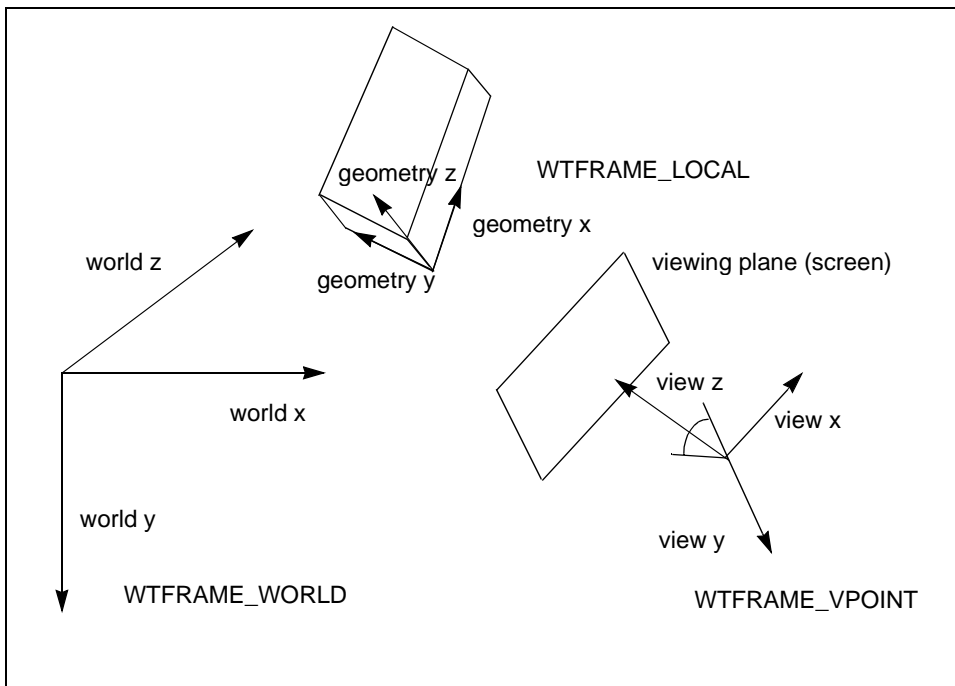


Figure 16-3: Reference frames for geometry motion

WTviewpoint_setorientation

```
void WTviewpoint_setorientation(  
    WTviewpoint *viewpoint,  
    WTq q);
```

This function sets the viewpoint's orientation to the specified quaternion. The *q* parameter is the specified quaternion.

If orientations are represented as 3x3 matrices in your program, the conversion function *WTm3_2q* can be used to generate the corresponding quaternion, which can then be passed in to *WTviewpoint_setorientation*.

WTviewpoint_getorientation

```
void WTviewpoint_getorientation(  
    WTviewpoint *viewpoint,  
    WTq q);
```

This function returns the orientation of the viewpoint, specified as a quaternion (in the *q* parameter). To convert this to a 3x3 matrix representation, use the function *WTq_2m3*.

WTviewpoint_getlastorientation

```
void WTviewpoint_getlastorientation(  
    WTviewpoint *view,  
    WTq q);
```

This function gets a viewpoint's orientation in the last frame. The viewpoint is passed in as *view* and the orientation is returned in *q*.

Similar to the *WTviewpoint_getlastposition* function, the way you use this function depends on the universe event order – whether the sensor updates are done before the actions function is called. See *WTviewpoint_getlastposition* (on page 16-9) for more information about how and where you can use this function.

WTviewpoint_rotate

```
void WTviewpoint_rotate(  
    WTviewpoint *viewpoint,  
    short axis,  
    float angle,  
    short frame);
```

This function rotates a viewpoint on a specified axis about the viewpoint's position in the world, local/viewpoint frame (see figure 16-3 on page 16-10). The *axis* parameter is one of the defined constants X, Y, or Z, and pertains to the specified reference frame (*WTFRAME_WORLD*, *WTFRAME_LOCAL*, or *WTFRAME_VPOINT*). Note that *WTFRAME_LOCAL* and *WTFRAME_VPOINT* both refer to the reference frame of the viewpoint in this case, and produce the same result when used. The *angle* parameter represents the amount of rotation (in radians) to perform about the specified axis.

The following example shows how to roll a viewpoint by 90 degrees to the right:

```
WTviewpoint *viewpoint;  
WTviewpoint_rotate (view, Z, 0.5*PI, WTFRAME_VPOINT);
```

WTviewpoint_move

```
void WTviewpoint_move(  
    WTviewpoint *viewpoint,  
    WTpq *moveby,  
    short frame);
```

This function moves a viewpoint by the translation and rotation values specified in *moveby*. The argument *moveby* is a pointer to a *WTpq* structure (which contains both a *WTp3* and a *WTq*), and is applied to the viewpoint in the specified reference frame *WTFRAME_WORLD*, *WTFRAME_LOCAL* or *WTFRAME_VPOINT*. (The latter two reference frames are the same when referring to a viewpoint.) This function causes a translation and a rotation of the viewpoint, because the *WTpq* structure contains both translation and rotational information. *WTviewpoint_move* is a relative move, compared to *WTviewpoint_moveto* (see below), which is an absolute move.

The rotational component of the *moveby* parameter (the *q* portion of the *WTpq* structure) is applied about the viewpoint position (see figure 16-1 on page 16-2).

WTviewpoint_moveto

```
void WTviewpoint_moveto(  
    WTviewpoint *viewpoint,  
    WTpq *newviewat);
```

This function moves a viewpoint to the position and orientation specified in *newviewat*. The argument *newviewat* is a pointer to a *WTpq* structure (which contains both a *WTp3* and a *WTq*). *WTviewpoint_moveto* moves the viewpoint to the absolute position and orientation contained in them. *WTviewpoint_moveto* is an absolute move, compared to *WTviewpoint_move* (see above), which is a relative move.

Since a *WTpq* is a structure, only a pointer to it can be passed in to a function. Structures should not be directly passed in to functions.

WTviewpoint_setdirection

```
void WTviewpoint_setdirection(  
    WTviewpoint *viewpoint,  
    WTp3 dir);
```

This function sets the viewpoint direction to the view specified by the *dir* parameter. The *view* direction represents the *Z* axis of the local viewpoint frame.

The *WTviewpoint_rotate* function can be used after a call to *WTviewpoint_setdirection* to specify the amount of twist (rotation) around the new view direction.

WTviewpoint_getdirection

```
void WTviewpoint_getdirection(  
    WTviewpoint *viewpoint,  
    WTp3 dir);
```

This function returns the direction of the viewpoint. The *dir* vector points along the *Z* axis of the viewpoint's local coordinate frame.

WTviewpoint_getaxis

```
void WTviewpoint_getaxis(  
    WTviewpoint *viewpoint,  
    short axis,  
    WTp3 vector);
```

This function returns the unit *vector* in the direction of the specified viewpoint's *axis* in the world frame, which is specified by the *axis* parameter. Valid values for *axis* are X, Y, and Z (which represent axes).

WTviewpoint_alignaxis

```
void WTviewpoint_alignaxis(  
    WTviewpoint *viewpoint,  
    short axis,  
    WTp3 dir);
```

This function rotates the viewpoint so the specified axis aligns with the specified direction. The axis is specified by the *axis* parameter. Valid values of *axis* are X, Y, and Z (which represent axes). Direction is specified by the *dir* vector. The *dir* vector should be specified relative to the world frame axis.

Using a Specified Reference Frame

The first function in this section, *WTviewpoint_getframe*, is used to obtain reference frame information (a *WTpq*) which can then be passed in to any of the other “frame” functions.

The functions in this section are just like the correspondingly named functions without the final “frame” at the end of the function name, except that the positions (or 3D vectors) and orientations passed in to these functions are interpreted as being relative to the specified reference frame.

WTviewpoint_getframe

```
void WTviewpoint_getframe(  
    WTviewpoint *viewpoint,  
    WTPq *frame);
```

This function returns the specified viewpoint's position and orientation and places it in the *frame* parameter.

WTviewpoint_setpositionframe

```
void WTviewpoint_setpositionframe(  
    WTviewpoint *viewpoint,  
    WTP3 pos,  
    WTPq *frame);
```

This function moves the viewpoint to the specified 3D position in the specified frame. It is like *WTviewpoint_setposition* (see page 16-8) but takes an additional argument *frame*.

WTviewpoint_getpositionframe

```
void WTviewpoint_getpositionframe(  
    WTviewpoint *view,  
    WTP3 pos,  
    WTPq *frame);
```

This function returns the 3D position of the viewpoint relative to the specified frame. It is like *WTviewpoint_getposition* (see page 16-8) but takes an additional argument *frame*.

WTviewpoint_translateframe

```
void WTviewpoint_translateframe(  
    WTviewpoint *view,  
    WTP3 p,  
    WTPq *frame);
```

This function translates a viewpoint by the specified vector in the specified frame. It is like *WTviewpoint_translate* but takes an additional argument *frame*.

WTviewpoint_setorientationframe

```
void WTviewpoint_setorientationframe(  
    WTviewpoint * view,  
    WTq q,  
    WTPq *frame);
```

This function sets the viewpoint's orientation in the specified frame to the specified quaternion. It is like *WTviewpoint_setorientation* (see page 16-11) but takes an additional argument *frame*.

WTviewpoint_getorientationframe

```
void WTviewpoint_getorientationframe(  
    WTviewpoint * view,  
    WTq q,  
    WTPq *frame);
```

This function returns the orientation of the viewpoint relative to the specified frame, specified as a quaternion. It is like *WTviewpoint_getorientation* (see page 16-11) but takes an additional argument *frame*.

WTviewpoint_rotateframe

```
void WTviewpoint_rotateframe(  
    WTviewpoint * view,  
    short axis,  
    float angle,  
    WTPq *frame);
```

This function rotates a viewpoint around a given axis around the viewpoint's position in the specified frame. It is like *WTviewpoint_rotate* (see page 16-12) but takes an additional argument *frame*, which can be any coordinate frame (i.e., the specified *WTPq*).

WTviewpoint_moveframe

```
void WTviewpoint_moveframe(  
    WTviewpoint * view,  
    WTPq *pq,  
    WTPq *frame);
```

This function moves a viewpoint *by* the specified translation and rotation values in the specified frame. It is like *WTviewpoint_move* (see page 16-12) but takes an additional argument *frame*, which can be any coordinate frame (i.e., the specified WTPq). This is a relative move, compared to *WTmovetoframe* (see below), which is an absolute move.

WTviewpoint_movetoframe

```
void WTviewpoint_movetoframe(  
    WTviewpoint * view,  
    WTPq *pq,  
    WTPq *frame);
```

This function moves a viewpoint *to* the specified position and orientation in the specified frame. It is like *WTviewpoint_moveto* (see page 16-13) but takes an additional argument *frame*, which can be any coordinate frame (i.e., the specified WTPq). This is an absolute move, compared to *WTmoveframe* (see above), which is a relative move.

WTviewpoint_setdirectionframe

```
void WTviewpoint_setdirectionframe(  
    WTviewpoint * view,  
    WTP3 dir,  
    WTPq *frame);
```

This function rotates the viewpoint to the specified view direction in the specified frame. It is like *WTviewpoint_setdirection* but takes an additional argument *frame*.

WTviewpoint_getdirectionframe

```
void WTviewpoint_getdirectionframe(  
    WTviewpoint * view,  
    WTp3 dir,  
    WTpq *frame);
```

This function returns the direction of the viewpoint relative to the specified frame. It is like *WTviewpoint_getdirection* but takes an additional argument *frame*.

For more information about the use of reference frames, please see the discussion in *Geometry Motion Reference Frames* on page 13-19.

Viewpoint Aspect Ratio

WTviewpoint_setaspect

```
void WTviewpoint_setaspect (  
    WTviewpoint *viewpoint,  
    float aspect);
```

This function sets the viewpoint's aspect ratio. This function can be used to correct for any monitor or pixel distortion that causes round objects to look elliptical or square objects to look rectangular. If the horizontal and vertical extents of pixels in the display are equal, then no correction should be needed. Otherwise, call this function with *aspect* set to the ratio of the horizontal pixel extent to the vertical pixel extent. Increasing values of *aspect* make objects appear taller on the screen (without affecting their apparent width).

For example, if the pixels in your display are twice as wide as they are tall, then an object which was modeled as a perfect square would appear on the screen to be only half as tall as it was wide when rendered with the default viewpoint aspect ratio of *1.0*. You could compensate for this by using the following call.

```
WTviewpoint *viewpoint;  
WTviewpoint_setaspect(view, 2.0);
```

This effectively stretches objects vertically by a factor of two, making the particular object appear square. See also *WTwindow_setviewangle* on page 17-19, and *WTviewpoint_getaspect*, below.

WTviewpoint_getaspect

```
float WTviewpoint_getaspect(  
    WTviewpoint *viewpoint);
```

This function returns the viewpoint's current aspect ratio. This value is specified as a ratio of the horizontal and vertical drawing dimensions. See also *WTwindow_setviewangle*, *WTviewpoint_setaspect* above.

Stereo Viewing

The functions in this section are used to set and get the parameters used for stereo viewing. Please refer to the function *WTviewpoint_new* on page 16-3 for parameter definitions and default values.

WTviewpoint_setparallax

```
void WTviewpoint_setparallax(  
    WTviewpoint *viewpoint,  
    float parallax);
```

This function sets the parallax value for stereo viewing. Parallax is the distance between the left and right eye views in the simulation (see figure 16-2 on page 16-3).

It is often desirable to set the parallax value to some fraction of the typical range of units of interest in your application. For example, you might use the radius of the volume defined by the scene graph, as in the following:

```
WTviewpoint *viewpoint;  
WTviewpoint_setparallax(view, 0.05 *  
    WTnode_getradius(WTuniverse_getrootnodes()));
```

In some applications, the volume defined by the scene graph may be very large compared to the size of typical objects in the scene. For example, consider a driving simulation over a very large terrain containing trees and buildings that are considerably smaller in extent than the terrain. In this case, it may be preferable to scale the viewpoint parallax relative to the extents of just a portion of the scene graph, as in the following:

```
WTnode*node;
WTviewpoint *viewpoint;
WTviewpoint_setparallax(view, 0.1 * WTnode_getradius(node));
```

By increasing the parallax value, you can achieve an enhanced stereo effect (sometimes called “hyper-stereo”). However, as parallax increases, it may become difficult for your eyes to fuse the stereo pair of images into a single 3D image.

WTviewpoint_getparallax

```
float WTviewpoint_getparallax(
    WTviewpoint *viewpoint);
```

This function returns the viewpoint’s parallax value, which is the distance in world coordinates between the left and right eyes.

The following example uses the viewpoint’s parallax value to determine the location of the viewpoint’s left and right eyes in the world coordinate frame.

```
WTviewpoint *viewpoint;
WTp3 pleft, pright;

/* retrieve the position of the viewpoint’s left eye */
WTviewpoint_getposition(view, pleft);

/* set pright to the position of the right eye
   in the viewpoint frame */
pright[X] = WTviewpoint_getparallax(view);
pright[Y] = pright[Z] = 0.0;

/* convert pright to world coordinates */
WTviewpoint_local2world(view, pright, pright);
```

```
/* print out eye positions in world coordinates */  
WTP3_print(pleft, "left eye");  
WTP3_print(pright, "right eye");
```

WTviewpoint_setconvergence

```
void WTviewpoint_setconvergence(  
    WTviewpoint *viewpoint,  
    short convergence);
```

This function sets the horizontal offset (in pixels) that is applied to both the left and right eye images. The offset is subtracted from the left eye and added to the right eye.

This function can be used to achieve stereo fusion in head-mounted displays where the display screens are not exactly centered in front of the user's eyes. A negative convergence value moves the images for the eyes closer together, a positive value moves them further apart.

WTviewpoint_getconvergence

```
short WTviewpoint_getconvergence(  
    WTviewpoint *viewpoint);
```

This function returns the viewpoint's stereo convergence value in screen pixel units. This value should not be confused with the convergence distance value used with asymmetric viewing projections, shown in figure 16-4.

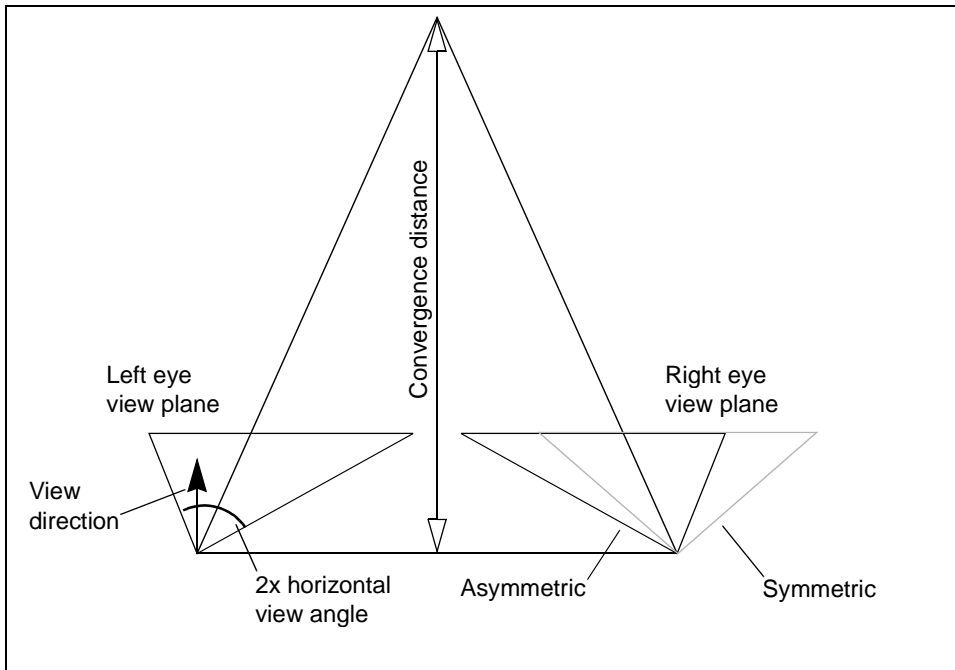


Figure 16-4: Top view of stereoscopic viewing pyramid with asymmetric projection

WTviewpoint_setconvdistance

```
void WTviewpoint_setconvdistance(
    WTviewpoint *viewpoint,
    float val);
```

This function sets the convergence distance of the specified viewpoint. This value has no effect on the scene that is drawn unless asymmetric viewing has been turned on by calling *WTwindow_setprojection* and specifying *WTPROJECTION_ASYMMETRIC* for the projection type.

When asymmetric viewing is on, the scene is drawn from the same view position, view direction, and view angle as when asymmetric viewing is off. However, with asymmetric viewing, the scene appears horizontally skewed. The amount of skew is determined by the value of the viewpoint's convergence distance parameter, as illustrated in figure 16-4. As the convergence distance decreases, the amount of skew increases.

Asymmetric viewing is useful in some stereo viewing configurations. By changing the convergence distance, geometries can be made to appear either in front of or behind the display device (e.g., the screen). A geometry in the 3D world closer to the viewpoint than the convergence distance appears to be in front of the screen, while a geometry that is farther from the viewpoint than the convergence distance appears to be behind the screen.

The most dramatic stereo effect is often achieved when part of your scene appears to be in front of the screen, and part appears to be behind the screen. You may wish to experiment with setting the value of the convergence distance to the distance from the viewpoint to the midpoint of your scene, as in the following example. *This example assumes that the asymmetric projection type has already been set for the specified window:*

```
void adjustconvergencedistance(WTwindow *w)
{
    WTviewpoint *viewpoint;
    float distance;
    WTp3 p, midpt;

    /* find distance from viewpoint to midpoint of scene graph*/
    view = WTwindow_getviewpoint(w);
    WTnode_getmidpoint(WTwindow_getrootnode(), midpt);
    WTviewpoint_getposition(view, p);
    distance = WTp3_distance(midpt, p);

    /* set viewpoint convergence distance to that value */
    WTviewpoint_setconvdistance(view, distance);
}
```

WTviewpoint_getconvdistance

```
float WTviewpoint_getconvdistance(
    WTviewpoint *viewpoint);
```

This function returns the value of the viewpoint's convergence distance parameter.

See also *WTviewpoint_setconvdistance* above and *WTwindow_setprojection* on page 17-14.

Coordinate Transformations

WTviewpoint_world2local

```
void WTviewpoint_world2local(  
    WTviewpoint *viewpoint,  
    WTP3 pin,  
    WTP3 pout);
```

This function takes the specified 3D point *pin* in the world coordinate frame, and determines the location of that point in relation to the specified viewpoint's reference frame. The result is stored in the *pout* parameter.

WTviewpoint_local2world

```
void WTviewpoint_local2world(  
    WTviewpoint *viewpoint,  
    WTP3 pin,  
    WTP3 pout);
```

This function takes a 3D point *pin* in the coordinate frame of the specified viewpoint (specified in the *viewpoint* parameter), and determines the location of that point in relation to the world coordinate frame. The result is stored in *pout*.

The following example uses *WTviewpoint_local2world* to place a geometry in front of a viewpoint. Another example is provided under *WTviewpoint_getparallax* on page 16-20.

```
WTnode *geom;  
WTviewpoint *viewpoint;  
WTP3 pos_local;    /* position in viewpoint's frame */  
WTP3 pos_world;   /* position in world frame */  
/* place the object in front of the viewpoint.  
The object's orientation is not considered in this example. */  
pos_local[X] = pos_local[Y] = 0.0;  
pos_local[Z] = 5.0 * WTnode_getradius(geom);  
WTviewpoint_local2world(viewpoint, pos_local, pos_world);  
/* move the object to the world-coordinate location */  
WTnode_settranslation(geom, pos_world);
```


Viewpoint Name

WTviewpoint_setname

```
void WTviewpoint_setname(  
    WTviewpoint *viewpt,  
    const char *name);
```

This function sets the name of the specified viewpoint. All viewpoints have a name; by default, a viewpoint's name is "" (i.e., a NULL string).

WTviewpoint_getname

```
const char *WTviewpoint_getname(  
    WTviewpoint *viewpt);
```

This function returns the name of the specified viewpoint.

User-specifiable Viewpoint Data

A *void ** pointer is included as part of the structure defining a viewpoint, so that you can store whatever data you want with a viewpoint. The following functions can be used to set and get this field within any viewpoint.

WTviewpoint_setdata

```
void WTviewpoint_setdata(  
    WTviewpoint *viewpoint,  
    void *data);
```

This function sets the user-defined data field in a viewpoint. Private application data can be stored in any structure. To store a pointer to a structure within the viewpoint, pass in a pointer to the structure, cast to a *void**, as the *data* argument.

WTviewpoint_getdata

```
void *WTviewpoint_getdata(  
    WTviewpoint *viewpoint);
```

This function retrieves private data stored within a viewpoint. You should cast the value returned by this function to the same type used to store the data with the *WTviewpoint_setdata* function.

Viewpoint Intersection Test

WTviewpoint_intersectpoly

For information on this function, see page 4-89. Also see *How Do I Test For Objects Intersecting With Other Objects In The Universe?* on page A-25.

Introduction

A WTK window object corresponds to a region of the screen in which a view of the graphical universe is displayed. With the window class, multiple views can be displayed simultaneously and flexibly to different parts of the screen.

Included in this chapter are WTK functions that let you do the following:

- create a window with system-specific characteristics (such as border type) and delete it
- reposition and resize a window
- define the way in which the scene is viewed in a window when rendered
- define the way in which the scene is projected to the window when rendered
- picking and ray-casting in a window
- set the rendering properties of a window (such as background color and texture backdrop)
- assign user-specifiable data to a window
- get the system-specific ID of a window
- create multiple viewports within a window

Window Construction and Destruction

WTwindow_new

```
WTwindow *WTwindow_new(  
    int x0,  
    int y0,  
    int xsize,  
    int ysize,  
    int flags);
```

This function creates a new WTK window object and displays it on the screen using the host system window manager. If successful, a pointer to the window object is returned; otherwise NULL is returned.

The values in the *x0* and *y0* arguments are the minimum X, Y screen coordinates of the window. The values in the *xsize* and *ysize* arguments are the width and height of the window, not including the window border. The parameter *flags* is a constant defining the window's characteristics. (For information on the different stereoscopic viewing modes, see page 2-34).

These are the possible values for *flags*:

WTWINDOW_DEFAULT

Creates a window with no special attributes. The window has a border unless *WTWINDOW_NOBORDER* is used in combination with this constant (via the bitwise OR operator).

WTWINDOW_STEREO

Creates a stereo window on systems that have hardware support for stereo. On systems without hardware stereo support, this option will create 2 images in the window (one on the top with the left eye view, the other on the bottom with the right eye view). On Windows platforms, if this option is selected and the *WTDISPLAY_NEEDSTENCIL* option is selected in the *display_config* parameter when *WTuniverse_new* is called, the

<i>WTWINDOW_STEREOVSPLIT</i>	behavior you will obtain is that of <i>WTWINDOW_STEREOVSPLIT</i> . This constant can be combined with the <i>WTWINDOW_STEREO</i> option by using the bitwise OR operator (), to create 2 images in the window (one on the top with the left eye view, the other on the bottom with the right eye view) even if your system has hardware stereo support. In essence, this option will cause WTK to disable your system's stereo hardware and to create a "vertically split" stereo window instead.
<i>WTWINDOW_RBSTEREO</i>	Creates a window with red/blue stereo.
<i>WTWINDOW_INTERLACEEVENODD</i>	Creates an interlaced stereo window whose even numbered scanlines correspond to the left eye view and whose odd numbered scanlines correspond to the right eye view. This option requires that the <i>WTDISPLAY_NEEDSTENCIL</i> option be selected in the <i>display_config</i> parameter when <i>WTuniverse_new</i> is called.
<i>WTWINDOW_INTERLACEODDEVEN</i>	Creates an interlaced stereo window whose odd numbered scanlines correspond to the left eye view and whose even numbered scanlines correspond to the right eye view. This option requires that the <i>WTDISPLAY_NEEDSTENCIL</i> option be selected in the <i>display_config</i> parameter when <i>WTuniverse_new</i> is called.
<i>WTWINDOW_NOBORDER</i>	This constant can be combined with any of the above listed options by using the bitwise OR operator (), to create a window without a border.
<i>WTWINDOW_SCREENn</i>	Where n is a number from 0 to 8. In the multi-processor version of WTK, this constant can be combined with any of the above listed options by using the bitwise OR

operator (|), to specify which screen the window is to be placed on.

If the `window_config` parameter is set to any of the stereo options (`WTWINDOW_STEREO`, `WTWINDOW_RBSTEREO`, `WTWINDOW_INTERLACEEVENODD`, or `WTWINDOW_INTERLACEODDEVEN`), you will need to adjust the viewpoint's parallax and convergence values. See `WTviewpoint_setparallax` and `WTviewpoint_setconvergence`.

You only need to call `WTwindow_new` to create additional windows besides those created by the call to `WTuniverse_new`. The windows created by `WTuniverse_new` have viewpoints associated with them, while windows created by calling `WTwindow_new` are assigned a NULL viewpoint. A window's viewpoint is set using `WTwindow_setviewpoint` (see page 17-11).

As windows are created, they are added to the end of the universe's list of windows. A pointer to the front window in this list is returned by the `WTuniverse_getwindows` (see page 2-13) function. When the window is created, the following parameters are set:

<i>projection type</i>	This defines how the scene is projected into the window. The default projection is symmetric (<code>WTPROJECTION_SYMMETRIC</code>). See <code>WTwindow_setprojection</code> on page 17-14.
<i>viewpoint</i>	This is the viewpoint from which the scene is projected into the window. By default, this viewpoint is NULL. See <code>WTwindow_setviewpoint</code> on page 17-11.
<i>eye</i>	By default, the scene projected into the window is viewed from the left eye (<code>WTEYE_LEFT</code>) of the window's viewpoint. To have the scene rendered from the right eye, use <code>WTwindow_seteye</code> on page 17-12. As explained in the <i>Viewpoints</i> chapter, the viewpoint's left eye position is obtained when <code>WTviewpoint_getposition</code> (on page 16-8) is called. The right eye position is obtained by a translation from the left eye along the viewpoint's X axis by the parallax distance.
<i>background color</i>	Default value: blue (rgb=0, 0, 255). See <code>WTwindow_setbgrgb</code> on page 17-22.
<i>view angle (in radians)</i>	The default view angle (half the total horizontal viewing angle) is 0.698131 radians (40 degrees). Given the horizontal view angle, the vertical view angle is determined

from the window's aspect ratio, which is the ratio of the vertical view angle tangent to the horizontal view angle tangent. See *WTwindow_setviewangle* on page 17-19. and *WTviewpoint_setaspect* on page 16-18. For general and orthographic window projections (*WTPROJECTION_GENERAL* and *WTPROJECTION_ORTHOGRAPHIC*), the view angle is not used. See *WTwindow_setparams* on page 17-16.

hither clipping value

The distance (along the viewpoint direction) from the viewpoint position to the hither clipping plane. Graphical entities are clipped at this plane; only things on the opposite side of the hither plane from the viewpoint are drawn. The default hither clipping value is 1.0. See *WTwindow_sethithervalue* on page 17-18.

yon clipping value

The distance (along the viewpoint direction) from the viewpoint position to the yon clipping plane. Graphical entities are clipped at this plane; only things on the side of the yon clipping plane closest to the viewpoint are drawn. The default yon clipping value is 65536.0. See *WTwindow_setyonvalue* on page 17-19.

Figure 17-1 illustrates the relationship of the viewpoint to the window parameters. The view plane is a slice through the view frustum (pyramid) determined by the size of the window and the view angle. The yon clipping plane, which truncates the view pyramid defining its far end, is not shown.

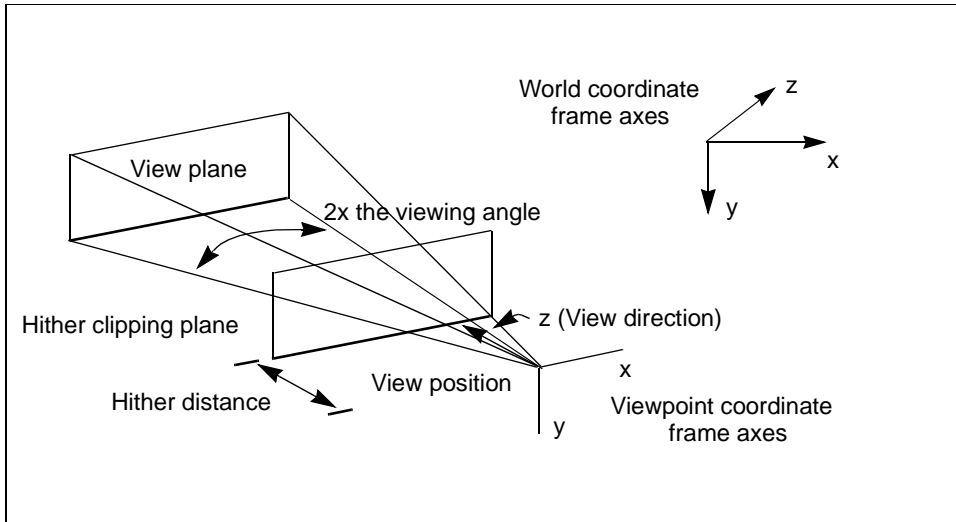


Figure 17-1: Monoscopic viewing geometry

WTinit_usewindow

Prototype for Windows:

```
void WTinit_usewindow(
    HWND *parent);
```

Prototype for UNIX:

```
void WTinit_usewindow(
    Widget *parent)
```

Argument:

parent The ID of the parent host-specific window that will enclose the WTK window.

This function integrates WTK windows with host-specific windows. This function must be called *before* *WTuniverse_new* (see page 2-2). See *How Do I Integrate A WTK Rendering Window With A Host-Specific Window?* on page A-35.

WTwindow_newuser

Prototype for Windows

```
WTwindow *WTwindow_newuser(  
    HWND *parent,  
    int window_config);
```

Prototype for UNIX:

```
WTwindow *WTwindow_newuser(  
    Widget *parent,  
    int window_config);
```

Argument:

<i>parent</i>	The ID of the parent host-specific window that would enclose the WTK window.
<i>window_config</i>	This value is the same as the flag's value in the <i>WTwindow_new</i> (see page 17-2) function.

This function integrates WTK windows with host-specific windows. This function must be called *after* *WTuniverse_new* (see page 2-2). See *How Do I Integrate A WTK Rendering Window With A Host-Specific Window?* on page A-35.

WTwindow_delete

```
void WTwindow_delete(  
    WTwindow *window);
```

This function deletes a WTK window object. The *window* argument may be a pointer to a window object, created by an explicit call to *WTwindow_new*, or a pointer to a window that was created implicitly by a call to *WTuniverse_new*.

Accessing Universe's Windows

WTwindow_next

```
WTwindow *WTwindow_next(  
    WTwindow *window);
```

This function returns the next window in the universe's list of WTK window objects. A pointer to the first window in this list is obtained by calling *WTuniverse_getwindows* (see page 2-13).

For example, suppose that your application uses a stereo display created by calling *WTuniverse_new* with the *WTDISPLAY_STEREO* option, and that you wish to obtain pointers to the windows in which the left and right-eye views are displayed. This can be accomplished as follows:

```
WTwindow *wleft, *wright;  
  
/* WTuniverse_new is first WTK call in program */  
WTuniverse_new(WTDISPLAY_STEREO, WTWINDOW_NOBORDER);  
  
/* left eye view is displayed in first window created by WTuniverse_new */  
wleft = WTuniverse_getwindows();  
/* right eye view displayed in second window created. */  
wright = WTwindow_next(wleft);
```

Associating Scene Graphs with Windows

WTwindow_setrootnode

```
void WTwindow_setrootnode(  
    WTwindow *window,  
    WTnode *rootnode);
```

This function associates a scene graph with a specified window by passing in the root node of the scene graph. Once this is done, the scene graph will be rendered into the specified window.

The *WTuniverse_new* (see page 2-2) function automatically associates the default scene graph (i.e., the root node constructed by *WTuniverse_new*) with each of the windows it creates. So, if your application uses only the default scene graph and does not create any additional windows, then you do not need to call this function.

WTwindow_getrootnode

```
WTnode *WTwindow_getrootnode (  
    WTwindow *window);
```

This function returns the root node of the scene graph associated with the specified window.

WTwindow_enable

```
void *WTwindow_enable (  
    WTwindow *window,  
    FLAG enable);
```

This function allows you to enable or disable rendering to a specified window. By default, each window is enabled. This is useful when your simulation contains multiple windows and one or more windows are not active, i.e., they do not need to be updated in the simulation loop. Using this function to disable rendering to inactive windows can substantially improve performance.

WTwindow_isenabled

```
FLAG WTwindow_isenabled (  
    WTwindow *window);
```

This function returns TRUE if the specified window is enabled and returns FALSE if the specified window is disabled. See *WTwindow_enable* above.

Window Size and Placement

A WTK window object's initial size and location on the screen are set at the time it is created. For a window created with *WTwindow_new* (see page 17-2), the initial size and location are specified as arguments of the function. Size and location depend on the display option chosen, and can also be set using WTK's resource facility (discussed in *Resource Files* on page 2-28).

After the windows objects are created, you can obtain their current size and location, resize them, or move them using the functions described in this section.

WTwindow_setposition

```
void WTwindow_setposition(  
    WTwindow *window,  
    int x0,  
    int y0,  
    int width,  
    int height);
```

This function changes a window's size and/or location on the screen. The *x0* and *y0* parameters are the minimum X and Y screen coordinates of the window, while *width* and *height* specify the width and height of the window in screen coordinates (not including the window border).

WTwindow_getposition

```
void WTwindow_getposition(  
    WTwindow *window,  
    int *x0,  
    int *y0,  
    int *width,  
    int *height);
```

This function returns the location and size of a window. It places the minimum X and Y window coordinates into the values *x0* and *y0*, and the width and height of the window (in pixels) into the values *width* and *height* (not including the window border).

Windows and Viewpoints

Each WTK window object has, associated with it, a viewpoint, as well as the eye (either left or right) from which the scene is rendered in that window (see figure 17-1 on page 17-6). For example, it may be useful to have a window that provides a bird's-eye view of the simulation, or a close-up view, or an out-the-window view. It may even be useful to have a window with a NULL viewpoint, which simply displays status information with user-defined drawing functions.

The functions in this section allow you to control your viewpoints.

WTwindow_setviewpoint

```
void WTwindow_setviewpoint(  
    WTwindow *window,  
    WTviewpoint *view);
```

This function sets the viewpoint to be displayed in the specified window. The window or stereo-pair of windows created when *WTuniverse_new* (see page 2-2) is called is assigned the viewpoint which is automatically created by the *WTuniverse_new* call. However, when a window is created with *WTwindow_new* (see page 17-2), the viewpoint set for it is NULL. To have a viewpoint displayed in a window other than the windows put up by the call to *WTuniverse_new*, you must explicitly set the viewpoint for the window by calling *WTwindow_setviewpoint*.

To have a window in your WTK application which does not display a view of the WTK graphical universe, but which only displays the results of user-defined drawing functions, set the viewpoint for the window to NULL.

WTwindow_getviewpoint

```
WTviewpoint *WTwindow_getviewpoint(  
    WTwindow *window);
```

This function returns the viewpoint currently set for the specified window. See *WTwindow_setviewpoint*, above.

WTwindow_seteye

```
void WTwindow_seteye(  
    WTwindow *window,  
    short eye);
```

This function specifies whether the scene displayed in the specified window should be rendered from the left or right eye of the viewpoint. The left and right eyes are separated by the viewpoint's parallax value.

The value of *eye* must be one the defined constants *WTEYE_LEFT* or *WTEYE_RIGHT*. The default value for a window is *WTEYE_LEFT*.

This function is useful in the case where two distinct windows are created for stereo viewing and when it is desirable to use a common viewpoint for each window. Even though each window shares a common viewpoint, *WTviewpoint_setparallax* (see page 16-19) can be used to set the distance in the 3D virtual world between the points from which the left and right eye views are drawn. You can then use *WTwindow_seteye* to specify which eye view to use for each window of the stereo window pair.

WTwindow_geteye

```
short WTwindow_geteye(  
    WTwindow *window);
```

This function determines which eye the window's viewpoint is set to display. It returns either *WTEYE_LEFT* or *WTEYE_RIGHT*. If the pointer passed in is not in the universe's list of windows, it returns -1. See *WTwindow_seteye* above, and *WTviewpoint_setparallax* on page 16-19.

WTwindow_setviewpoint2

```
void WTwindow_setviewpoint2(  
    WTwindow *window,  
    WTviewpoint *view);
```

This function sets the second viewpoint to display for the specified window when using a stereo window. Use this function in special situations, when it is necessary to perform view projections into a stereo window using two completely different viewpoints.

If the viewpoints used for the left and right eye view only differ in their parallax value, then you do not need to use this function. You can instead use a single viewpoint and set the viewpoint's parallax value to an appropriate value. See *WTwindow_setviewpoint* on page 17-11 and *WTviewpoint_setparallax* on page 16-19.

WTwindow_getviewpoint2

```
WTviewpoint *WTwindow_getviewpoint2(  
    WTwindow *window);
```

This function returns the second viewpoint associated with the specified window. See *WTwindow_setviewpoint2* above.

WTwindow_getscreen

```
int WTwindow_getscreen(  
    WTwindow *window);
```

This function returns the screen number upon which the window appears.

Zooming the Window Viewpoint

WTwindow_zoomviewpoint

```
void WTwindow_zoomviewpoint(  
    WTwindow *window);
```

This function zooms the viewpoint of the given window so that all geometries in the scene graph (associated with that window) are visible. The orientation of the viewpoint is preserved. This is useful when you associate a new scene graph with the window and require orientation.

Wtwindow_zoomviewtonode

```
void Wtwindow_zoomviewtonode(  
    Wtwindow *window,  
    Wtnode *node,  
    int which);
```

This function zooms the viewpoint of the specified window so that all geometries in the node (and the node's sub-tree) are visible. The orientation of the viewpoint is preserved. This is useful when you associate a new scene graph with the window and require orientation. The *which* parameter indicates which instance of the node to zoom to, since there may be many instances of the node in the scene graph.

Window-projection Functions

In WTK Version 2.0, the *Wtviewpoint* class contained all parameters pertaining to the viewing pyramid or frustum. Beginning with WTK 2.1 and including the current release, functions are provided so that the view frustum parameters can instead be stored with the window. This gives more flexibility, because with this capability, you are able to render the scene from the same viewpoint into different windows using different viewing projections into each window.

Wtwindow_setprojection

```
void Wtwindow_setprojection(  
    Wtwindow *window,  
    int type);
```

This function sets one of the following projection types for the specified window:

WTPROJECTION_SYMMETRIC

Commonly used projection, especially for monoscopic and flat screen displays. This is the default projection type.

WTPROJECTION_ASYMMETRIC

Useful for stereoscopic displays. By varying the viewpoint convergence distance, objects can be made to appear in front of or behind

the projection plane. See the discussion under *WTviewpoint_setconvdistance* on page 16-22.

WTPROJECTION_GENERAL

Provides the greatest flexibility; useful when the viewer is not always perpendicular to the display surface, for example in CAVE environments. *See note below.*

WTPROJECTION_ORTHOGRAPHIC

Useful for plan views or anytime a perspective distortion is not desired; parallel lines remain parallel regardless of viewpoint position. Translations in the X and Y directions work as before, but translations along the Z-axis do not affect the scene (except when either the hither or yon clipping planes interact with the scene's geometries). *See note below.*

Note: When using either *WTPROJECTION_GENERAL* or *WTPROJECTION_ORTHOGRAPHIC*, you must specify any aspect of the window's viewing frustum using *WTwindow_setparams*. The functions *WTwindow_setviewangle*, *WTwindow_sethithervalue*, and *WTwindow_setyonvalue*, do not affect the window's view frustum when using orthographic or general projections.

The default projection type is *WTPROJECTION_SYMMETRIC*. However, when *WTuniverse_new* is called with *WTDISPLAY_CRYSTALEYES*, the window projection is set to *WTPROJECTION_ASYMMETRIC*.

If you intend to use a projection type other than the default one (*WTPROJECTION_SYMMETRIC*), the call to *WTwindow_setprojection* must precede calls to any other window functions. For example, if you call *WTwindow_zoomviewpoint* and then set the projection to orthographic, it will look as if the zoom didn't work.

WTwindow_getprojection

```
int WTwindow_getprojection  
    (WTwindow *window);
```

This function returns the projection type for the specified window. The projection type is one of the following:

```
WTPROJECTION_SYMMETRIC  
WTPROJECTION_ASYMMETRIC  
WTPROJECTION_GENERAL  
WTPROJECTION_ORTHOGRAPHIC.
```

WTwindow_setparams

```
void WTwindow_setparams(  
    WTwindow *window,  
    FLAG eye,  
    float left,  
    float right,  
    float bottom,  
    float top,  
    float near,  
    float far);
```

This function specifies the parameters describing the window's viewing frustum used for the specified eye (*WTEYE_LEFT* or *WTEYE_RIGHT*). This function only works when a window's projection type has been set to either *WTPROJECTION_GENERAL* or *WTPROJECTION_ORTHOGRAPHIC*.

The *near* parameter defines the distance to the near (hither) clipping plane. The *far* parameter defines the distance to the far (yon) clipping plane. The parameters *top* and *left* are the distances along the X and Y axes in the viewpoint coordinate frame which define the top-left corner of the view pyramid where it intersects the hither clipping plane. The parameters *bottom* and *right* are the distances along the X and Y axes in the viewpoint coordinate frame which define the bottom-right corner of the view pyramid where it intersects the hither clipping plane. For orthographic projections the viewing "pyramid" is not a pyramid at all, but a box with all of its walls mutually perpendicular. The view pyramid is shown in figure 17-1 on page 17-6.

With this function, you can create windows that are not directly in front of the viewpoint, depending on the left, right, bottom, and top coordinates. If left and right are positive, then the window is off to the right. If bottom and top are both negative, then the window is below the viewpoint.

To keep the viewpoint within the window boundaries, make left negative, right positive, bottom negative, and top positive. For example, to create a window like the default WTK window (with the viewpoint directly in the center) use *WTPROJECTION_GENERAL*, make *left* a negative number, *right* a positive number of the same magnitude as *left*, *top* a positive number, and *bottom* a negative number of the same magnitude as *top*.

When this function is called, the values specified for *near* and *far* override the hither and yon values of the window.

WTwindow_getparams

```
void WTwindow_getparams(  
    WTwindow *window,  
    FLAG eye,  
    float *left,  
    float *right,  
    float *bottom,  
    float *top,  
    float *near,  
    float *far);
```

This function obtains the current window parameters describing the viewing frustum for a window with general or orthographic projection. The eye parameter can be either *WTEYE_LEFT* or *WTEYE_RIGHT*. See *WTwindow_setprojection* on page 17-14.

Other Window-projection Functions

The functions described in this section are *WTwindow* class functions that had corresponding functions in the *WTviewpoint* class in releases of WTK prior to Release 6 and this current release. These functions have been moved to the *WTwindow* class to give greater flexibility in associating viewpoints and viewing parameters with windows.

For backward compatibility, the corresponding *WTviewpoint* class functions will continue to be supported, and applications created with WTK 2.1 should behave in exactly the same way if recompiled using the current release. However, if you call one of the functions in this section, it will override any calls to the corresponding *WTviewpoint* function that you previously made for the viewpoint associated with that window. In future development, it is recommended that you use the functions in this section rather than the old corresponding *WTviewpoint* functions.

The functions described in this section have no effect when the window projection type is *WTPROJECTION_GENERAL* or *WTPROJECTION_ORTHOGRAPHIC*, because in those cases all view frustum parameters are set with *WTwindow_setparams* (see page 17-16).

WTwindow_sethithervalue

```
void WTwindow_sethithervalue(  
    WTwindow *window,  
    float val);
```

This function sets the distance of the window's hither clip plane in front of the viewpoint. The default hither clipping value is 1.0

The new value specified in the *val* argument must be greater than the floating point "fuzz" value *WTFUZZ* (0.004) used by WTK, or the function will leave the hither clipping plane as close to the viewpoint as it can.

WTwindow_gethithervalue

```
float WTwindow_gethithervalue(  
    WTwindow *window);
```

This function returns the window's hither clipping value.

WTwindow_setyonvalue

```
void WTwindow_setyonvalue(  
    WTwindow *window,  
    float val);
```

This function sets the window's yon clipping value. This is the distance in front of the viewpoint beyond which the scene is not rendered in that window. For example, if you pass in a value of 100.0 to this function, then those geometries or portions of geometries which are beyond 100.0 distance units from the eye are not rendered.

The default yon clipping value is 65536.0.

WTwindow_getyonvalue

```
float WTwindow_getyonvalue(  
    WTwindow *window);
```

This function returns the window's yon clipping value.

WTwindow_setviewangle

```
void WTwindow_setviewangle(  
    WTwindow *window,  
    float angle);
```

This function sets the specified window's horizontal view angle. The view angle is defined as half the horizontal angular field of view (in radians). The angle specified must be between 0.0 and $\text{PI}/2.0$ or the function has no effect. The default view angle (half the total horizontal viewing angle) is 0.698131 radians (40 degrees).

When the horizontal view angle is set with this function, the vertical view angle is automatically set as well, based on the dimensions of the window.

WTwindow_getviewangle

```
float WTwindow_getviewangle(  
    WTwindow *window);
```

This function returns the window's view angle in radians. The view angle is half the horizontal angular field of view.

Picking and Ray Casting

WTwindow_pickpoly

```
WTPoly *WTwindow_pickpoly(  
    WTwindow *window,  
    WTP2 point,  
    WTnodepath **nodepath,  
    WTP3 p);
```

This function obtains a pointer to the frontmost polygon at the specified 2D point in the specified window. The 2D point argument is specified in window coordinates, not screen coordinates, where (0.0, 0.0) represents the top-left corner of the window and the bottom-right corner of the window is represented by (window width - 1, window height - 1). If the specified point does not lie within the specified window, or if there is no polygon at that coordinate, then NULL is returned.

The WTP3 value obtained is the 3D world coordinate point of the picked polygon which projects to the specified 2D point.

This function also fills in the value of the *WTnodepath* pointer, indicating the node path to which the selected polygon belongs. The node path returned begins at the root node of the specified window. If the polygon selected is in a *WTgeometry* node which is referenced more than once in the scene graph, it may be useful to know for which occurrence of the *WTgeometry* node the polygon was selected. You are allowed to pass in NULL for the *nodepath* argument. If you do pass in NULL, then the function does not provide the *WTnodepath* pointer information to you and does not create a *WTnodepath* for you. If a *WTnodepath* is created, you are responsible for deleting this *WTnodepath*, when you no longer need it. To do so, call *WTnodepath_delete*.

The following example shows how to pick the frontmost polygon in the center of a window.

```
WTpoly *pick_center_poly(WTwindow *w)
{
    int width, height, x0, y0;
    WTpoly *pickedpoly;
    WTp2 point;
    WTp3 p;

    WTwindow_getposition(w, &x0, &y0, &width, &height);
    point[X] = width/2.0;
    point[Y] = height/2.0;

    pickedpoly = WTwindow_pickpoly(w, point, NULL, p);
    return pickedpoly;
}
```

WTwindow_getray

```
FLAG WTwindow_getray(
    WTwindow *window,
    WTp2 point,
    WTp3 rayorigin,
    WTp3 ray);
```

This function determines the ray that emanates from the view position (where the scene in that window is rendered), which passes through the specified point. For the *point* argument, the point (0.0, 0.0) corresponds to the upper-left corner of the window. This ray is normalized (that is, has a length equal to 1.0) and is in world coordinates; it is placed in the parameter *ray* by this function. The view position is placed in *rayorigin* by this function.

WTwindow_projectpoint

```
FLAG WTwindow_projectpoint(
    WTwindow *window,
    int eye,
    WTp3 pos,
    WTp2 point);
```

This function computes the 2D screen point relative to the window position where a 3D world coordinate projects. If the 3D point projects to a 2D screen point that is outside of the window, this function still returns the 2D point relative to the window but it will also return `FALSE`.

If the specified window is a stereo window, then the eye parameter identifies whether the projection is computed from the left (`WTEYE_LEFT`) eye of the window's viewpoint or the right (`WTEYE_RIGHT`) eye. For non-stereo windows, the eye parameter is ignored.

Window-rendering Properties

WTwindow_setbgrgb

```
void WTwindow_setbgrgb(  
    WTwindow *window,  
    unsigned char r,  
    unsigned char g,  
    unsigned char b);
```

This function sets the 24-bit background color for the specified window. Valid values for *r*, *g*, and *b* are 0 to 255. The default value is blue (`rgb = 0, 0, 255`).

The following example sets the background color of the first window created by the WTK application to yellow:

```
WTwindow_setbgrgb(WTuniverse_getwindows(), 255, 255, 0);
```

WTwindow_getbgrgb

```
void WTwindow_getbgrgb(  
    WTwindow *window,  
    unsigned char *r,  
    unsigned char *g,  
    unsigned char *b);
```

This function obtains the background color of the specified window.

WTwindow_setdrawfn

```
void WTwindow_setdrawfn(  
    WTwindow *window,  
    void (*drawfn)(WTwindow *win, FLAG eye));
```

This function specifies a function containing calls to 3D drawing routines (see *3D Drawing* on page 19-8 in the *Drawing Functions* chapter). For example, you could use this function to incorporate a 3D grid, or other objects, into the simulation.

Your user-defined 3D drawing function *drawfn* is invoked by WTK during the simulation loop. If the specified window is a stereo window then *drawfn* will be invoked twice, once for each eye. In this case the eye parameter that WTK passes to *drawfn* will be *WTEYE_RIGHT* and then *WTEYE_LEFT*. If the specified window is not a stereo window then WTK will pass in *WTEYE_LEFT* as the eye parameter.

Before *drawfn* is called within the WTK simulation loop, a copy of the current view matrix is pushed on top of the model view stack, so that your drawing elements can be drawn from this viewpoint if desired. Don't forget that the WTK coordinate convention differs from the OpenGL convention. (The WTK convention has X pointing to the right, Y pointing down, and Z pointing straight ahead. WTK coordinates are obtained by simply negating Y and Z OpenGL coordinate values.)

The current view matrix in WTK incorporates the transformation from OpenGL to WTK coordinates. So, if your *drawfn* uses the current view matrix, you must specify your drawing coordinates such as values passed into the OpenGL function *glVertex* using the WTK coordinate convention.

If using OpenGL drawing routines in the *drawfn* function, you must pop all matrices, and only those matrices, that you pushed onto the stack.

Note that no WTK function calls (other than math library calls, *WTwindow_set3D...*, *WTwindow_draw3D...*, or *WTwindow_loadimage*) may be used in the user-defined draw function, *drawfn*.

It is recommended that you not use this function because 3D drawing calls made in the user-specified *drawfn* are platform specific and hence make your application non-portable.

WTwindow_setfgactions

```
void WTwindow_setfgactions(  
    WTwindow *window,  
    void (*fgdrawfn)(WTwindow* win,  
        FLAG eye));
```

This function specifies a function containing calls to 2D drawing routines (drawn in the foreground). (See *2D Drawing* on page 19-1 in the *Drawing Functions* chapter.) The drawing routines are incorporated as an overlay onto the WTK scene. WTK handles the overlaying of the drawing elements onto the WTK scene; you do not have to manage this yourself. WTK does not actually draw into overlay bitplanes. Instead it uses the normal draw bitplanes, so that you can use the full image depth of the normal draw buffer. For example, you could use this function to create a “heads-up display” including text or other 2D graphical entities.

Your user-defined 2D drawing function *fgdrawfn* is invoked by WTK during the simulation loop. If the specified window is a stereo window, then *fgdrawfn* is invoked twice, once for each eye; the eye parameter that WTK passes to *fgdrawfn* will first be *WTEYE_RIGHT* and then *WTEYE_LEFT*. If the specified window is not a stereo window then WTK passes in *WTEYE_LEFT* as the eye parameter.

Before the function *fgdrawfn* is called, the matrix stack is initialized so that all the 2D functions use a normalized window coordinate system. A value of:

<i>0.0 for X</i>	Specifies the left edge of the window
<i>1.0 for X</i>	Specifies the right edge of the window
<i>0.0 for Y</i>	Specifies the bottom edge of the window
<i>1.0 for Y</i>	Specifies the top edge of the window

Note: This coordinate convention is unique within WTK.

If using OpenGL drawing routines in the function *fgdrawfn*, you must pop all matrices and only those matrices that you push onto the stack.

Note that no WTK function calls (other than math library calls, *WTwindow_set2D...*, *WTwindow_draw2D...*, or *WTwindow_loadimage*) may be used in the user-defined draw function, *fgdrawfn*. In fact, the *WTwindow_set2D...* and *WTwindow_draw2D...* functions can only be called from the *fgdrawfn* specified in *WTwindow_setfgactions*.

It is recommended that you do not use this function because 2D drawing calls made in the user-specified *drawfn* are platform specific and hence make your application non-portable.

WTwindow_numpolys

```
int WTwindow_numpolys(  
    WTwindow *window);
```

This function returns the number of polygons sent to the graphics pipeline associated with the specified window.

WTwindow_loadimage

```
FLAG WTwindow_loadimage(  
    WTwindow *window,  
    char *filename,  
    float zval,  
    FLAG swapbuf,  
    FLAG bitmapdel);
```

This function loads an image (bitmap) file to the specified window so that it fills the window. This function can be called from the universe action function, or from a user-specified draw function (see *WTwindow_setdrawfn* and *WTwindow_setfgactions* above).

This function draws the image at depth *zval* in a view frustum for which depth values are scaled to lie between $z = -0.999$ and $z = 1.0$. To create a texture backdrop for your WTK scene, call *WTwindow_loadimage* from a user-defined draw function specifying *zval* = -0.999. If you want the image to be placed on top of the WTK scene, you must call *WTwindow_loadimage* with *zval* = 1.0.

The parameter *swapbuf* (either TRUE or FALSE) is used to specify whether the image buffer should be swapped immediately after the image is drawn. If *WTwindow_loadimage* is called from an action function, then this value should be TRUE, so that the image is displayed immediately. (In this case, you will probably want to put in a delay after calling *WTwindow_loadimage* so that the image is visible for a specified time.) If *WTwindow_loadimage* is called from a user-defined draw function, then you should pass in FALSE for *swapbuf*, so that the image can be incorporated into the WTK scene.

The value of *bitmapdel* specifies whether you want the bitmap that is created when the image is loaded to be deleted after the call to *WTwindow_loadimage*. If you will only be displaying this bitmap once, and not using it as a surface texture, then you should call *WTwindow_loadimage* with *bitmapdel* set to TRUE. However, if this bitmap will be reused in your program, you should call *WTwindow_loadimage* with *bitmapdel* set to FALSE, which saves time when it is reused.

It returns TRUE if it successfully loads and draw the image in the window. It returns FALSE if the specified pointer is not a valid window pointer, or if the bitmap specified by *filename* could not be loaded.

WTwindow_saveimage

```
FLAG WTwindow_saveimage(  
    WTwindow *window,  
    int x,  
    int y,  
    int width,  
    int height,  
    char *filename);
```

Use this function to save a part or all of the display in a WTK window into a file. The argument *window* is a pointer to the WTK window. (See *WTuniverse_getwindows* on page 2-13 to get a pointer to your WTK window.) The image is saved in the TARGA (.tga) format. The last argument, *filename* is your name for the image file.

The arguments *x*, *y*, *width* and *height*, determine what area of the display you want captured. *x* = 0 and *y* = 0 corresponds to the lower left corner of the window. The *width* and *height* arguments indicate the extents in the X and Y axes respectively, that is, the extents over which the image will be saved. The lower-left corner of the captured part will have the window-coordinates *x*, *y* and the upper-right corner of the captured part will have the window-coordinates *x* + *width*, *y* + *height*. For example, if you need to capture your entire window:

```
WTwindow_saveimage(w, 0, 0, width, height, "file.tga");
```

where *width* and *height* are determined with the following call:

```
WTwindow_getposition(w, &xpos, &ypos, &width, &height);
```

Note that the second and third parameters in the *WTwindow_saveimage* call are 0 (zero), indicating that you want to capture from the lower left corner of the window. This function only works if your system is set to true color (24 bit) mode. It will not work in 16 bit or 8 bit mode. See also *WTwindow_getimage*.

WTwindow_getimage

See page 10-33 for a description of *WTwindow_getimage*.

Window Name

WTwindow_setname

```
void WTwindow_setname(  
    WTwindow *win,  
    const char *name);
```

This function sets the name of the specified window. All windows have a name; by default, a window's name is "" (i.e., a NULL string).

WTwindow_getname

```
const char *WTwindow_getname(  
    WTwindow *win);
```

This function returns the name of the specified window.

User-specifiable Window Data

WTwindow_setdata

```
void WTwindow_setdata(  
    WTwindow *window,  
    void *data);
```

This function sets the user-defined data field for the specified window. Private application data can be stored in any structure. To store a pointer to this structure within the window, pass in a pointer to the structure, cast to a *void**, as the *data* argument.

The following example stores a pointer to a WTK graphical object in the window's user-defined data field:

```
WTnode *geo;  
WTwindow *window;  
WTwindow_setdata(window, (void *) geo);
```

WTwindow_getdata

```
void *WTwindow_getdata(  
    WTwindow *window);
```

This function retrieves a pointer to the user-defined data stored within a window. Cast the value returned by this function to the same type that was used to store the data in the window with *WTwindow_setdata* (see above).

In the following example, the user-defined data field set in the example under *WTwindow_setdata* is retrieved.

```
WTnode *geo;  
WTwindow *window;  
  
/* retrieve pointer to the geometry node that was associated with the window */  
geo = (WTnode *) WTwindow_getdata(window);
```

System-specific Window ID

WTwindow_getidx

```
WTwinditype WTwindow_getidx(  
    WTwindow *window);
```

This function returns the system-specific window ID for the specified window. The return value's type is host-system specific:

UNIX platforms	The return type is Widget (i.e., an XID)
Windows platforms	The return type is HWND.

See also *WTuniverse_getwindows* on page 2-13 and *WTuniverse_getcurrwindow* on page 2-14.

WTwindow_getwidget

```
Widget WTwindow_getwidget(  
    WTwindow *w);
```

This function gets the X Widget that corresponds to a WTK window, *w*. If *w* is invalid, that is if the pointer is not recognized to be a valid WTK window, NULL is returned. (Available only on UNIX systems.)

This is a very useful function when you need to make Xt calls that require WTK's pointer to the X11 Display. (Display = XtDisplay(Widget);)

See *On UNIX Platforms, How Do I Get A Pointer To The Display That WTK Is Using?* on page A-38 for an example of how to use this function.

Viewports

Every WorldToolKit window contains, by default, a single viewport which covers the entire area of the window and wherein the scene is rendered. Additional viewports can be created for each WTK window so that multiple views of one or more scenes can be rendered inside a single WTK window.

There are two advantages to creating and using multiple viewports in a single window instead of creating and using multiple windows. The first advantage is that performance is improved when using multiple viewports in a WTK window instead of using multiple (single viewport) WTK windows. The reason for this is that the rendering buffers are cleared and swapped only once for the single window, rather than having to clear and swap for several windows. The second advantage is that the rendering of each viewport is frame synchronized, i.e. all viewports are rendered on the screen at the same time in a given frame. In contrast, using multiple windows means that WTK must process and render the geometry associated with the first WTK window before it can process and render the geometry associated with succeeding WTK windows and if your application's frame rate is low, there will be a discernable time lag between the updates of each window within one frame.

It is also possible to create a rear-view mirror effect by using multiple viewports in a window. Refer to the `Rv_mirror.c` example program in the examples sub-directory of the WTK distribution for an example of how viewports can be used to achieve a rear-view mirror effect.

Applications which do not require multiple viewports within a window, can ignore the concept of a viewport entirely, because viewports are not directly exposed like other WTK objects such as `WTwindow`, `WTnode`, etc. There are no objects such as a `WTviewport` object because the viewport concept is embedded into the `WTwindow` type. By embedding viewports into the `WTwindow` type, all of the functionality pertaining to viewports can be accessed via three functions: `WTwindow_setviewport`, `WTwindow_getviewport`, and `WTwindow_newviewport`. The `WTwindow_setviewport` function is used to position and size a window's default viewport, `WTwindow_newviewport` is used to create, position, and size additional viewports within a window while `WTwindow_getviewport` is used to access a viewport's position and size.

Each of these functions operates on a `WTwindow` object type. For example, to reposition or resize the default viewport of a window, you would pass a pointer to the `WTwindow` whose viewport is to be modified into `WTwindow_setviewport`. To create additional viewports in a window, you would pass a pointer to the `WTwindow` to which additional viewports are to

be added, to the *WTwindow_newviewport* function which will return a pointer to another *WTwindow*. In this situation, you would have two *WTwindow* pointers, one which points to the original window (the one which contained the default viewport), and the other which points to the newly created second viewport. All of the *WTwindow* functions can be used on either of these two *WTwindow* objects. The only distinction between these two *WTwindow* objects is the fact that one of them is the base window and the second represents the additional viewport contained in the base window. In essence, there are only two kinds of *WTwindows*: base windows (which represent the default viewport of a window) and additional viewports (which are added to base windows). The only real distinction between these two types of *WTwindows* is the fact that only a base type of window can be passed into *WTwindow_newviewport* to create additional viewport windows. You cannot create additional viewports within a viewport window; trying to do so will result in an error.

The background color for all the viewports within a window is the same as that of the window. You cannot set a different background color for each viewport within a window. Each viewport can be independently disabled. However, if the base window is disabled, all the viewports it contains are also disabled.

As with windows, the position of a viewport is specified by the position of its top left corner. The viewport's position is, however, relative to the window, and is represented as x and y offsets from the top left corner of the window. For instance, if a window is positioned at (50,50) in screen coordinates, the default viewport's position is (0,0), since it is relative to the top left corner of the window.

WTwindow_setviewport

```
FLAG WTwindow_setviewport(  
    WTwindow *win,  
    int xoff,  
    int yoff,  
    int xsize,  
    int ysize);
```

WTwindow_setviewport is used to set the position and size of the default viewport of a window. The x and y positions of the top left corner of the viewport are specified by *xoff* and *yoff* respectively. Since the viewport's position is relative to the top left corner of the window to which it belongs, *xoff* and *yoff* must represent offsets from the top left corner of the window.

The width and height of the viewport are specified by *xsize* and *ysize* respectively. *xsize* and *ysize* are not relative to the window; they must be specified in pixel values. For instance, if you want the height of the viewport to be half the window's height (where the window's height is 480), *ysize* must be 240, and NOT 0.5.

WTwindow_setviewport returns TRUE if it succeeds in setting the position and size of the viewport. It returns FALSE if the pointer to the window is invalid.

WTwindow_getviewport

```
FLAG WTwindow_getviewport(  
    WTwindow *win,  
    int *xoff,  
    int *yoff,  
    int *xsize,  
    int *ysize);
```

Use this function to retrieve the position and size of a viewport. The argument *win* is used to specify the viewport whose position and size properties are to be obtained. Note that the handle of a viewport is a pointer to a *WTwindow*. The x and y positions of the top left corner of the viewport are returned in *xoff* and *yoff* respectively. Since the viewport's position is relative to the position of the window to which it belongs, *xoff* and *yoff* represent the offsets from the top left corner of the window that contains the viewport. Use *WTwindow_getposition* to obtain the window's position in screen coordinates. The viewport's width and height are returned in *xsize* and *ysize* respectively.

This function returns TRUE if it succeeds in retrieving the viewport's position and size. It returns FALSE if the viewport handle *win* is invalid.

WTwindow_newviewport

```
WTwindow *WTwindow_newviewport(  
    WTwindow *basewin,  
    int xoff,  
    int yoff,  
    int xsize,  
    int ysize);
```

WTwindow_newviewport creates a new viewport in a window. *basewin* must point to the base window wherein to create the new viewport. Note that a window has a viewport associated with it by default. Use *WTwindow_setviewport* to either reposition or resize the default viewport. Use *WTwindow_newviewport* only to create an additional viewport in the window.

The x and y positions of the top left corner of the new viewport are specified as *xoff* and *yoff* respectively. Since the viewport's position is relative to the position of the window, *xoff* and *yoff* must be offsets from the top left corner of the window.

The width and height of the viewport are specified using *xsize* and *ysize* respectively. *xsize* and *ysize* are not relative to the window; they must be specified in pixel values. For instance, if you want the height of the viewport to be half the window's height (where the window's height is 480), *ysize* must be 240, and NOT 0.5.

If this function succeeds, it returns a pointer to a *WTwindow* structure. This pointer must be used as a handle to the newly created viewport. For example, if at a later time you want to get the position and size of this viewport, you must pass in this pointer to *WTwindow_getviewport*. Since the viewport's handle is a pointer to a window, all window functions are applicable to it.

This function returns NULL if it fails. *WTwindow_newviewport* will fail if the argument *basewin* is an invalid pointer, or if WTK cannot create any more viewports. (WTK places a limit (8) on the number of windows and viewports in an application.)

Adding User Interface (UI) Objects

WTK provides a set of User Interface (*WTui*) functions for you to create a standard graphical user interface (GUI). The *WTui* functions are a set of generic high-level functions for creating common UI objects for both UNIX and Microsoft Windows environments using X/Motif widgets and Microsoft Foundation Classes (MFC). You can use *WTui* functions to make a GUI that runs on both X Windows and MS Windows systems, while preserving the look and feel of each native environment.

In addition to making common UI objects, you can also easily add functionality to certain UI objects by associating callback handler functions with them. Differences in the way Motif and MFC handle events and messages are automatically taken care of by WTK's underlying UI framework.

Although the *WTui* functionality provided by WTK is not powerful enough to create independent GUI applications (to do that, you must either use Motif or MFC directly), it is useful in creating GUIs for virtual reality applications developed with WTK. Applications developed with WTK using *WTui* functionality are portable across UNIX and Microsoft Windows platforms.

Note that on Windows platforms, you can only create a windows application using WTK's *WTui* functionality. You cannot create a console application when using WTK's *WTui* functionality. However, WTK can simulate and display a console window if your application wants to use `WTmessages` to display text in a console window. See *WTui_showconsole* on page 18-41.

Creating a UI Application

Here's the basic procedure to create a UI application using WTK's UI:

1. Create the toplevel application shell by calling *WTui_init*. (See *Step 1: Creating the Toplevel Application Shell* on page 18-5.)
2. Create the main form by calling *WTuiform_new*. (See *Step 2: Creating the Main Form* on page 18-6.)
3. Create the required UI objects (such as menus, pushbuttons, scrolled lists, etc.). (See *Step 3: Creating Other UI Objects* on page 18-7.)
4. Add functionality (if required) to individual UI objects by associating a callback handler function using *WTui_setcallback* or *WTui_settoolbarcallback*, for tool bars. (See *Step 4: Adding Functionality to the Required UI Objects* on page 18-8.)
5. Create the WorldToolKit rendering window by calling *WTuiwtkwindow_new* (to integrate the WTK rendering window and the GUI window) or *WTwindow_new* (to have the WTK rendering window separate from the GUI window). (See *Step 5: Creating the WorldToolKit Rendering Window* on page 18-11.)
6. Manage each form created by calling *WTui_manage*. (See *Step 6: Managing Each Container Object* on page 18-11.)
7. Call *WTui_go* to start the main platform specific application loop. (See *Step 7: Starting The Application Loop* on page 18-12.)

The following example shows these seven steps.

Example: A Complete GUI Application Using WTK's UI

/** The following program creates an application consisting of the WTK rendering window and two pushbuttons. Clicking the 'Load' pushbutton alternates between loading the file clown.nff and deleting it. You can exit the application by clicking on the 'Exit' pushbutton */

```
#include <stdio.h>
#include <stdlib.h>
#include "wt.h"
```

```
/** global variables */
WTui      *toplevel;

/** callback handler function - called when pushbuttons are clicked */
static void LoadFile(WTui *pStruct, void *pData);
static void Exit(WTui *pStruct, void *pData);

int main(int argc, char *argv[])
{
    WTui    *shell, *windowform, *pushbutton1, *pushbutton2;
    FLAG    toload = TRUE;

    /** Step 1: Creating the toplevel application shell */
    toplevel = WTui_init(&argc,argv);

    /** Step 2: Creating the main form */
    shell = WTuiform_new(toplevel, "WTK GUI Application",
                        WTUIATT_LEFT, 200, WTUIATT_TOP, 200,
                        WTUIATT_WIDTH, 500, WTUIATT_HEIGHT, 500, NULL);

    /** Step 3: Creating other UI objects */
    pushbutton1 = WTuipushbutton_new(shell, "Load",
                                     WTUIATT_LEFT, 100, WTUIATT_TOP, 400,
                                     WTUIATT_WIDTH, 100, WTUIATT_HEIGHT, 50, NULL);

    pushbutton2 = WTuipushbutton_new(shell, "Exit",
                                     WTUIATT_LEFT, 300, WTUIATT_TOP, 400,
                                     WTUIATT_WIDTH, 100, WTUIATT_HEIGHT, 50, NULL);

    windowform = WTuiform_new(shell, NULL,
                              WTUIATT_LEFT, 50, WTUIATT_TOP,50, WTUIATT_WIDTH, 400,
                              WTUIATT_HEIGHT, 300, NULL);

    /** Step 4: Adding functionality to the pushbuttons */
    WTui_setcallback(pushbutton1, WTUIEVENT_ACTIVATE, LoadFile,
                    (void *)&toload);
    WTui_setcallback(pushbutton2, WTUIEVENT_ACTIVATE, Exit, NULL);

    WTuniverse_new(WTDISPLAY_NOWINDOW, WTWINDOW_DEFAULT);
}
```

```
    /** Step 5: Creating the WTK rendering window */
    WTuiwtkwindow_new(windowform, WTWINDOW_DEFAULT);

    /** Step 6: Managing each container object */
    WTui_manage(windowform);
    WTui_manage(shell);

    WMotionlink_new(WTmouse_new(), WTuniverse_getviewpoints(),
        WTSOURCE_SENSOR, WTTARGET_VIEWPOINT);
    WTuniverse_ready();

    /** Step 7: Starting the application loop */
    WTui_go(toplevel, TRUE);
    return 0;
}

static void LoadFile(WTui *pStruct, void *pData)
{
    if ((*FLAG *)pData) {
        WTnode_load(WTuniverse_getrootnodes(), "clown.nff", 1.f);
        WTwindow_zoomviewpoint(WTuniverse_getwindows());
        (*(FLAG *)pData) = FALSE;
    }
    else {
        WTnode_deletechild(WTuniverse_getrootnodes(), 0);
        (*(FLAG *)pData) = TRUE;
    }
}

static void Exit(WTui *pStruct, void *pData)
{
    WTui_delete(toplevel);
    WTuniverse_delete();
    exit(0);
}
```

The rest of this section details these seven steps and describes the appropriate functions for each of the steps.

STEP 1: CREATING THE TOPLEVEL APPLICATION SHELL

WTui_init

```
WTui *WTui_init(  
    int *argc,  
    char **argv);
```

This function performs the necessary platform-specific initialization and creates the top-level application shell. You must call this function before you create any UI objects.

Arguments:

<i>argc</i>	Specifies the number of command-line arguments passed.
<i>argv</i>	Specifies a pointer to an array of command line argument strings.

The arguments above are used to process command-line options in the normal C language style, and these are the same arguments passed to the application's main function. These are particularly useful in the UNIX environment, where you can specify a particular resource for the application from the command line. This resource value overrides other definitions (for the same resource) specified in standard resource files (the app-defaults file, for example). Thus, if the name of the application is *gui*, the user can set the default background color for this application using the following command:
gui -bg blue.

On UNIX platforms, if you are using WTK's support for X Resources, calling *WTui_init* is sufficient since it creates the X Resource database. You do not need to make a call to *WTinit_defaults*, as this function is called internally.

Note: The *WTui* returned by this function is passed as the first parameter to *WTuiform_new* when creating the main form as explained in Step 2.

STEP 2: CREATING THE MAIN FORM

WTuiform_new

```
WTui* WTuiform_new(  
    WTui* parent,  
    char *title,  
    ...);
```

This function creates a form (container) object and returns a pointer to it.

Arguments:

<i>parent</i>	The parent UI object, the toplevel application shell (when creating the main form) or another form. When you use <i>WTuiform_new</i> to create the main form for your application, the <i>parent</i> parameter is always the toplevel application shell created by the call to <i>WTui_init</i> . (See Step 1 on page 18-5.)
<i>title</i>	Text to go in the title bar. The <i>title</i> parameter is equivalent to NULL for a form that is a child of another form. In other words, the title does not show for a form that is a child of another form, and therefore should be NULL.
...	Variable argument list for size and position of the form object (see below).

A form is a container object for other UI objects. The main window of the GUI application must always be a form object. The *WTUIEVENT_ACTIVATE* event (see *WTui_setcallback* on page 18-8) of a form object occurs whenever the form is resized by the end-user.

STEP 3: CREATING OTHER UI OBJECTS

WTK provides a set of generic high-level functions for creating common UI objects.

Many of the UI objects make use of the following UI resources, when created. These resources are used to set the position and size of a UI object with respect to its parents.

<i>WTUIATT_LEFT</i>	X offset of the UI object with respect to the parent UI object, defaults to 0 (zero).
<i>WTUIATT_TOP</i>	Y offset of the UI object with respect to the parent UI object, defaults to 0 (zero).
<i>WTUIATT_WIDTH</i>	Width of the UI object with respect to the parent UI object, defaults to 100.
<i>WTUIATT_HEIGHT</i>	Height of the UI object with respect to the parent UI object, defaults to 100.

You can set these resources through the variable argument list. They apply to all UI objects having a variable argument list parameter in their creation function.

Example: Using Variable Argument List Resources

```

/** A form and a textfield object */
WTui *form, *textfield;
textfield = WTuitextfield_new(form, "Testing",
    WTUIATT_LEFT, 150,    /** X offset of the textfield with respect to the form */
    WTUIATT_TOP, 150,    /** Y offset of the textfield with respect to the form */
    WTUIATT_WIDTH, 50,    /** Width of the textfield with respect to the form */
    WTUIATT_HEIGHT, 20,  /** Height of the textfield with respect to the form */
    NULL);

```

SCALED POSITION AND SIZE

The above resources - *WTUIATT_LEFT*, *WTUIATT_TOP*, *WTUIATT_WIDTH*, and *WTUIATT_HEIGHT* are scaled internally. They are scaled by a factor of 1000/x-resolution of the screen. So, if you are running your application at a resolution of 1024x768, the scale factor is 1000/1024 (0.98 approximately). Now if you specify the *WTUIATT_LEFT*, *WTUIATT_TOP*, *WTUIATT_WIDTH*, and *WTUIATT_HEIGHT* to be 100, 100, 200, and 200 respectively, the scaled values would be 98, 98, 196, and 196 (approximately). Thus WTK

divides the screen into 1000 units along the X and Y axis. This scaling is to ensure cross-platform portability so that an application looks the same on UNIX and Windows platforms.

Note: Although you should be able to take advantage of WTK's cross-platform functionality, you might be required to change the values of the above resources when you port an application from UNIX to Windows and vice-versa. This is because the actual size of a UI object on UNIX and Windows is different. For example, a pushbutton of width 100 on Windows may display the string "File Load" fully, whereas you might have to increase the width on UNIX to display the same string fully.

See *User Interface Objects* on page 18-13 for a complete list of UI objects which can be created using WTK's *WTui* functionality. See *User Interface Object's Utility Functions* on page 18-29 for a list of functions which operate on the different types of UI objects.

STEP 4: ADDING FUNCTIONALITY TO THE REQUIRED UI OBJECTS

You can add functionality to the UI objects that you have created by associating a callback handler function with the particular UI object. For example, you can add functionality to a pushbutton or menu item so that something happens when the user clicks on it.

WTui_setcallback

```
void WTui_setcallback(  
    WTui *ui,  
    int eventtype,  
    UICBP cb,  
    void *cbdata);
```

This function sets the callback handler function for a UI object.

Arguments:

<i>ui</i>	The UI object in question. Valid UI objects are forms, file selection boxes, text-input dialogs, checkbuttons, pushbuttons, radio boxes, scales, scrolled lists, and menu items.
<i>eventtype</i>	This parameter should always be set to <i>WTUIEVENT_ACTIVATE</i> .

<i>cb</i>	Name of the callback handler function.
<i>cbdata</i>	Pointer to the callback data to be passed to the callback handler function. You can pass any data into the callback handler function by typecasting it to a void pointer. It should be NULL if no data is to be passed to the callback handler function.

UICBP (User Interface Call Back Procedure) is a type define for a callback handler function. In general a callback handler function has the following signature:

```
void callback_function_name(WTui *pStruct, void *pData)
```

where,

<i>callback_function_name</i>	is the name of the call back handler function;
<i>pStruct</i> :	is the UI object for which the callback has been set;
<i>pData</i>	is a void pointer to the callback data that has been passed to the callback handler function. You can access this data after it has been typecast to the proper type. For example, if the callback data was of the type <i>char *</i> , inside the callback handler function, you access it by type casting it to a <i>char *</i> , i.e., (<i>char *</i>) <i>pData</i> . See the example <i>Example: A Complete GUI Application Using WTK's UI</i> on page 18-2 to see how data is passed to a callback handler function (by the call <i>WTui_setcallback(pushbutton, WTUIEVENT_ACTIVATE, Loadfile, (void*)&toload)</i> ; and how it is accessed in the callback handler function (in the function <i>LoadFile</i>).

WTui_settoolbarcallback

```
void WTui_settoolbarcallback(  
    WTui *ui,  
    int id,  
    int eventtype  
    UICBP cb,  
    void *cbdata);
```

This function sets a callback handler function for a particular button on the tool bar.

Arguments:

<i>ui</i>	The tool bar UI object in question.
<i>id</i>	The index of the desired pushbutton in the tool bar. Indices start from 0 for the left-most button.
<i>eventtype</i>	This parameter should always be set to <i>WTUIEVENT_ACTIVATE</i> .
<i>cb</i>	Name of the callback handler function.
<i>cbdata</i>	Pointer to the callback data, as explained in the description for the <i>WTui_setcallback</i> function on page 18-8.

Example: Creating a Tool Bar and Associating Callbacks

This example code creates a tool bar with five buttons and associates callbacks with each of the toolbar buttons.

```
/** Specify the bitmap files *****/

char *bmap[] = { {xm_hour16},
                 {keyboard16},
                 {scales},
                 {icon},
                 {xlogo16}
               };

/** Create the toolbar **/
toolbar = WTui_newtoolbar(shell, 5, bmap);

/** set callbacks to each toolbar button **/
WTui_settoolbarcallback(toolbar, 0, WTUIEVENT_ACTIVATE, FileOpen, NULL);
WTui_settoolbarcallback(toolbar, 1, WTUIEVENT_ACTIVATE, DoLights, NULL);
WTui_settoolbarcallback(toolbar, 2, WTUIEVENT_ACTIVATE, DoFlying, NULL);
WTui_settoolbarcallback(toolbar, 3, WTUIEVENT_ACTIVATE, DoFlying, NULL);
WTui_settoolbarcallback(toolbar, 4, WTUIEVENT_ACTIVATE, FileExit, NULL);
```

STEP 5: CREATING THE WORLDTOOLKIT RENDERING WINDOW

WTuiwtkwindow_new

```
WTwindow *WTuiwtkwindow_new(  
    WTui *form),  
    int window_config;
```

This function integrates a WTK rendering window with a user-defined GUI window, so that WTK rendering takes place in the GUI window instead of a separate window. Either this function or *WTwindow_new* (see page 17-2) must be called to create the WTK rendering window. (*WTwindow_new* sets the WTK rendering window separate from the user-defined GUI window.)

Arguments:

<i>form</i>	The form object in which the WTK rendering window is to be created. This is usually the main form object created with the first call to <i>WTuiform_new</i> .
<i>window_config</i>	This is the same as the flags parameter used with <i>WTwindow_new</i> .

Note: You must call *WTuniverse_new* with the *display_config* parameter set to *WTDISPLAY_NOWINDOW* before calling this function.

STEP 6: MANAGING EACH CONTAINER OBJECT

WTui_manage

```
void WTui_manage(  
    WTui *ui);
```

This function manages a form object.

Argument:

<i>ui</i>	The form UI object.
-----------	---------------------

In GUI applications, a form object is a Manager object, i.e., it manages its children (controlling their size and location). This is called Geometry Management and is accomplished by an X Toolkit or Windows function call after all its children are created. To comply with this, WTK has a function, *WTui_manage*, that should be called for each form object after all the children of the form object have been created.

For form objects, you should call this function for the child first and then the parent. For example, if form A is the parent of form B, then you should call *WTui_manage*(B) before *WTui_manage*(A).

STEP 7: STARTING THE APPLICATION LOOP

WTui_go

```
void WTui_go(  
    WTui *toplevel,  
    FLAG startwtk);
```

This function enters the main platform-specific application loop and continuously processes events and messages. *WTui_go* should be the last statement in your application's main function.

Arguments:

<i>toplevel</i>	The toplevel application shell created by a call to <i>WTui_init</i> .
<i>startwtk</i>	Controls starting of the WTK simulation loop. If the <i>startwtk</i> parameter is TRUE, the WTK simulation loop is automatically started. If it is FALSE, you must call <i>WTui_start</i> (see page 18-38) from within a callback handler function to start the WTK rendering.

WTui_go is analogous to *WTuniverse_go* (which is used for WTK applications not using WTK's UI functionality) except that it performs the additional functionality of processing windowing system events. So, when using this function, you do not need to call *WTuniverse_go*. Note that WTK does not have a *WTui_go1* function which corresponds to *WTuniverse_go1*. Example: *Simulating WTuniverse_go1* on page 18-38 shows how you can accomplish the functionality equivalent to *WTuniverse_go1* when using WTK's UI.

User Interface Objects

Forms

Forms are described in *Step 2: Creating the Main Form* on page 18-6. Forms are the only container (user interface) objects available in WTK. The only end-user triggered event which applies to forms is a resize event, and so a callback handler function for a form need only concern itself with resize events.

File-selection Boxes

WTuifileselection_new

```
WTui *WTuifileselection_new(  
    WTui *parent,  
    char *title,  
    char *file,  
    char *pat);
```

This function creates a modal file selection box and returns a pointer to it.

Arguments:

<i>parent</i>	The parent UI object.
<i>title</i>	The title text of the file selection box object.
<i>file</i>	The default file
<i>pat</i>	The pattern for file selections

A file selection box is a selection dialog that presents the user with a list of directories and files. It normally pops up when the user clicks on the File Open or File Save As menu items. During creation, a file pattern can be specified using standard wild-card characters (e.g., *.c) and the file selection box will display all files fitting that pattern. A default file can also be specified.

You are responsible for writing the appropriate handler function to call when the user clicks OK after selecting a file. Again, identifying the selected file in the callback handler function is very easy; just set a callback for the file selection box object using *WTui_setcallback* (see page 18-8). A pointer to the pathname of the selected file is obtained from the callback data argument, *pData* (see page 18-8) of the callback handler function.

The following example illustrates how file selection works.

EXAMPLE: ACCESSING PATHNAME OF SELECTED FILE

```
/** The following code segment shows how you can access the pathname of the
selected file from a file selection box. It assumes the file selection box is created in the
callback handler function of a menu item (already created). The file selected can be
accessed in the callback handler function of the file selection box, when you click OK
***/
```

```
/** Callback handler function for the menu item (already created) */
static void FileOpen(WTui *pStruct, void *pData)
{
    WTui *file_select;

    file_select = WTuifileselection_new(pStruct, "File Open", "clown.nff", "*.nff");
    WTui_setcallback(file_select, WTUIEVENT_ACTIVATE, OnFileOpenOK, NULL);
}

/** Callback handler function for the file selection box */
static void OnFileOpenOK(WTui *pStruct, void *pData)
{
    char *file;

    file = (char *)pData;
    WTnode_load(root,file,1.0f);
}
```

Message Boxes

WTuimessagebox_new

```
WTui *WTuimessagebox_new(  
    WTui *parent,  
    char *message,  
    char *title);
```

This function creates a modal message box, and returns a pointer to it.

Arguments:

<i>parent</i>	The parent UI object.
<i>message</i>	The message text
<i>title</i>	The box title text.

A message box object is a simple message dialog. It displays an appropriate message and the dialog disappears when the user clicks on OK or Cancel. You cannot associate a callback handler function with a message box object.

Text-input Dialogs

WTuitextinput_new

```
WTui *WTuitextinput_new(  
    WTui *parent,  
    char *title);
```

This function creates a simple text input box and returns a pointer to it. The text input box is composed of a label and a text field.

Arguments:

<i>parent</i>	The parent UI object.
---------------	-----------------------

title The prompt message.

A text input dialog box is a prompt dialog that displays an appropriate message and prompts the user for input. This dialog box allows the user to enter a text string. You can access the string entered by associating a callback handler function with the text input dialog that gets called when the user presses the enter key or clicks on the OK button. In the callback handler function, *WTui_gettext* (see page 18-30) can be used to retrieve the text string which was entered.

Checkbuttons

WTuicheckbutton_new

```
WTui* WTuicheckbutton_new(  
    WTui *parent,  
    char *label,  
    ...);
```

This function creates a checkbutton object and returns a pointer to it.

Arguments:

parent	The parent UI object.
label	Text on the checkbutton.
...	Variable argument list for size and position of the checkbutton object. The resources WTUIATT_WIDTH, and WTUIATT_HEIGHT have no effect on UNIX platforms, whereas these must be set on Windows platform for the checkbutton to be made visible.

A checkbutton is a simple object that can be checked or unchecked by clicking on it. Checkbuttons are normally used in a group, where the user can select any number of checkbuttons at a given time. Use *WTui_isChecked* on page 18-35 to get the state of a particular checkbutton.

Note: *This function creates a single checkbutton. Thus, for example, if you want a group of five checkbuttons, you need to call this function five times.*

Labels

WTuiabel_new

```
WTui *WTuiabel_new(  
    WTui *parent,  
    char *label,  
    FLAG labeltype,  
    ...);
```

This function creates a simple, non-editable, static text label and returns a pointer to it.

Arguments:

<i>parent</i>	The parent UI object, normally a form object.
<i>label</i>	When the <i>labeltype</i> parameter is <i>WTUI_TEXT</i> , this represents the text which goes into the label. When the <i>labeltype</i> parameter is <i>WTUI_FILE</i> , this represents the name of a image file (*.bmp for Windows applications and *.xpm for UNIX applications). The file name should be specified without extension to ensure cross-platform portability (e.g., if the filename is <i>name.bmp</i> or <i>name.xpm</i> , the label should be just <i>name</i>).
<i>labeltype</i>	This parameter should be set to either <i>WTUI_TEXT</i> (to load a text string) or <i>WTUI_FILE</i> (to load an image file).
...	Variable argument list for size and position of the label object.

Labels are basic objects that do not permit user interaction. Labels normally create non-editable text. However, an image may also be loaded into a label as described above. You cannot associate a callback handler function with a label object.

Pushbuttons

WTuipushbutton_new

```
WTui *WTuipushbutton_new(  
    WTui *parent,  
    char *label,  
    ...);
```

This function creates a pushbutton object and returns a pointer to it.

Arguments:

<i>parent</i>	The parent UI object.
<i>label</i>	Text on the pushbutton.
...	Variable argument list for size and position of the pushbutton object.

A pushbutton is a simple object that responds to a mouse click. It normally has a text label indicating its functionality. You can write a callback handler function and associate it with the pushbutton so that it will be called whenever the user clicks the pushbutton. This connection is accomplished with the *WTui_setcallback* function (see page 18-8).

Radioboxes

WTuiradiobox_new

```
WTui* WTuiradiobox_new(  
    WTui *parent,  
    int num,  
    char **labels,  
    ...);
```

This function creates a radiobox object and returns a pointer to it. The radiobox consists of togglebuttons, only one of which can be selected at a given time.

Arguments:

<i>parent</i>	The parent UI object.
<i>num</i>	The number of toggle buttons in the radiobox.
<i>labels</i>	Pointer to an array of text strings, holding the text for the togglebuttons.
...	Variable argument list for size and position of the radiobox object. The resources <i>WTUIATT_WIDTH</i> , and <i>WTUIATT_HEIGHT</i> have no effect on UNIX platforms, whereas these must be set on Windows platform for the radiobox to be made visible.

A radiobox is normally used when the user is required to make a single selection from a set of choices. You can associate a callback handler function with a radiobox object and use *WTui_getselecteditem* on page 18-31 to get the index number of the togglebutton selected in a radiobox.

Scales

WTuiscale_new

```
WTui *WTuiscale_new(  
    WTui *parent,  
    char *label,  
    int minimum,  
    int maximum,  
    int decimal_points,  
    int value,  
    ...);
```

This function creates a scale object and returns a pointer to it.

Arguments:

<i>parent</i>	The parent UI object.
<i>label</i>	Text to appear below the scale object.

<i>minimum</i>	The minimum value (lower bound) of the scale.
<i>maximum</i>	The maximum value (upper bound) of the scale.
<i>decimal_points</i>	The number of decimal points (if any) to be considered, as described below.
<i>value</i>	The initial value to be displayed on the scale.
...	Variable argument list for size and position of the scale object.

A scale object is a slider that displays a numeric value depending on the position of the slider. The value displayed is between an upper and lower bound, which you specify when you create the object. Thus the user can interactively change the value displayed by using the slider mechanism. Scale objects should be used, whenever possible, in place of having the user input a numeric value using the keyboard.

One important aspect to be noted for scale objects is that the scale's values can be stored only as an integer. This, however, does not keep you from representing fractional values. When creating the scale object, you specify an integer argument called *decimal_points*. This value indicates the number of places to move the decimal point to the left of the displayed value. WTK takes care of calculating the actual fractional value that is displayed. The fractional value (float) can be obtained from the callback data of the callback handler function.

Note: On Windows platforms the actual height of the scale object is 2/3rd of the height specified using *WTUIATT_HEIGHT* variable, whereas 1/3rd of the height is the text that appears below the scale object. So if *WTUIATT_HEIGHT* is 60, the actual scale height is 40 and the caption height is 20.

The example below illustrates how to create and use a scale object.

Example: Creating a Scale Object

```
WTui *scale, *form;
/** Make a new scale object inside the Form (which is already created). The following
call creates a scale object at the desired location. The minimum value is 0 as specified.
The maximum value is 1 although 100 is specified. This is because the number of
decimal points is specified to be 2.**/
scale = WTuiscale_new(form, "Ambient Light",
    0,          /** the minimum value          **/
```



```
    100,    /** the maximum value          **/
    2,      /** the # of decimal points    **/
    40,     /** the initial value to be displayed **/
    WTUIATT_LEFT, 20,
    WTUIATT_TOP, 40,
    NULL);
/** set callback for the scale **/
WTui_setcallback(scale, WTUIEVENT_ACTIVATE, SetLightIntensity, NULL );

/** The callback routine **/
void SetLightIntensity(WTui *pStruct, void *pData)
{
    float *value;
    value = (float *)pData;
    /** Note: pointer pData automatically points to the actual fraction value of the
    scale**/
    WTlightnode_setintensity(light, *value);
}
```

Scrolled Lists

WTuiscrolledlist_new

```
WTui* WTuiscrolledlist_new(
    WTui *parent,
    char *label,
    char *items[],
    int nitems,
    ...);
```

This function creates a scrolled list of strings and returns a pointer to it.

Arguments:

parent The parent UI object.
label The text to appear above the scrolled list object.

- items* Array of text strings to be put in the scrolled list object.
- nitems* The number of text strings in the scrolled list object.
- ... Variable argument list for size and position of the scrolled list object.

A scrolled list is very convenient for displaying a list of text choices. To select an item in the list, the user double-clicks on it. Only one item can be selected at a time. Additionally, these objects have scroll bars attached to them; the scroll bars show up whenever it is not possible to simultaneously display all items in the list.

What happens after a particular item is selected is up to the callback handler function, which must be written by you. The callback handler function gets called when the user selects a particular text item by double-clicking on it. Inside the callback handler routine you can get the index number of the item selected by calling *WTui_getselecteditem* on page 18-31 or the actual string selected by calling *WTui_gettext* on page 18-30). See also *WTui_insertitem* and *WTui_deleteitem* to insert or delete items from the scrolled list.

The following example illustrates how to use scrolled list objects.

Example: Accessing Selected Items in a Scrolled List

```
WTui *scroll_list;
scroll_list = WTuiscrolledlist_new(...

WTui_setcallback(scroll_list, WTUIEVENT_ACTIVATE, SelectedItem, NULL);
void SelectedItem(WTui *pStruct, void *pData)
{
    int index;
    char *string;

    /** get the index of item selected **/
    index = WTui_getselecteditem(pStruct);

    /** get the string selected **/
    string = WTui_gettext(pStruct);
}
```

Scrolled Text

WTuisrolledtext_new

```
WTui *WTuisrolledtext_new(  
    WTui *parent,  
    char *text,  
    FLAG editable,  
    ...);
```

This function creates a text box (with a scroll bar) and returns a pointer to it.

Arguments:

<i>parent</i>	The parent UI object, normally a form.
<i>text</i>	Text to go in the box.
<i>editable</i>	This parameter should be set to either <i>WTUI_EDITABLE</i> (indicating the text can be edited) or <i>WTUI_NOTEDITABLE</i> (indicating the text cannot be edited).
...	Variable argument list for size and position of the scrolled text object.

This is a simple text object that can be used for text entry by the user. You can use it anywhere the user types in free-form text. You can also use it to display non-editable text as in a popup help box. Use *WTui_gettext* on page 18-30 to retrieve the text from the text box. You cannot associate a callback handler function to a scrolled text object.

Note: When the text box is used to display non-editable text, a newline character in the text to be displayed is given by `\r\n` on Windows and by `\n` on UNIX. `\r\n` also works on UNIX.

Text Fields

WTUITextfield_new

```
WTui *WTUITextfield_new(  
    WTui *parent,  
    char *text,  
    ...);
```

This function creates a new editable text-field object and returns a pointer to it.

Arguments:

<i>parent</i>	The parent UI object.
<i>text</i>	Text to go in the field.
...	Variable argument list for size and position of the text field object.

A text field is a simple object that allows a user to enter text using the keyboard. Text-field objects are normally used as a single line data entry field. It allows the user text editing capabilities and also provides the point and click functionality expected of GUI applications. Use *WTui_gettext* (see page 18-30) to retrieve the text from the text field. You cannot associate a callback handler function with a text field object.

Menus

WTuiMenubar_new

```
WTui *WTuiMenubar_new(  
    WTui *parent);
```

This function creates a new menu bar and returns a pointer to it. Menu pop-up buttons and menu items must be created separately with the functions listed below.

Argument:

<i>parent</i>	The parent UI object, usually a form.
---------------	---------------------------------------

You cannot associate callback handler functions with a menubar object.

Menus are important objects in a GUI application. WTK allows you to create custom pull-down menu systems for an application. A menu system in WTK consists of three parts:

- The menu bar object, created by a call to the function *WTuimenubar_new* (see above).
- Pop-up buttons as children of the menu bar (or pop-up buttons can be children of other pop-up buttons to create cascading menus), created by a call to the function *WTuimenupopup_new* (see below).
- Menu items as children of pop-up buttons, created by a call to the function *WTuimenuitem_new* (see page 18-26). You can associate callback handler functions to menu items only.

Almost any menu system can be developed using these three objects. When a pop-up button is a child of the menu bar, the menu drops down below the button when the user clicks on it. When the pop-up button is a child of another pop-up button, the new menu pops up to the right of the item when the user clicks on it.

The example on page 18-26 illustrates how easy it is to create a menu system using these functions.

Note: On Windows platforms, if your application has a menu bar, it affects the width and height of the main form. It reduces the width by 8, and the height by 46 units.

WTuimenupopup_new

```
WTui *WTuimenupopup_new(  
    WTui *parent,  
    char *label);
```

This function creates a new menu pop-up button, and returns a pointer to it.

Arguments:

<i>parent</i>	The parent UI object (always a menu bar or another pop-up button. The latter allows for cascading menus).
<i>label</i>	Text label to appear on the pop-up button.

You cannot associate a callback handler function to a menu pop-up button object.

WTuimenuitem_new

```
WTui *WTuimenuitem_new(  
    WTui *parent,  
    char *label);
```

This function creates a new menu item button and returns a pointer to it.

Arguments:

parent The parent UI object (always a pop-up menu button).
label The menu item text.

You can associate a callback handler function which gets called every time the user clicks on the menu item. For example, you can pop-up a file selection dialog when the user clicks on a menu item. See *Example: Accessing Pathname of Selected File* on page 18-14.

The text on a menu item can be accessed or changed by calls to *WTui_settext* on page 18-29 and *WTui_gettext* on page 18-30. The functions *WTui_enable* on page 18-34 and *WTui_isenabled* on page 18-34 can be used to enable/disable a menu item and to check if a menu item is enabled respectively.

Example: Simple Menu System Creation

```
WTui *shell, *mmenu;  
WTui *pop1, *pop2, *pop3;  
WTui *pb1, *pb2, *pb3, *pb4;  
WTui *pb5, *pb6, *pb7, *pb8, *pb9;  
  
/** Construct the menu system inside the main form (shell - already created)**/  
/** Create the menu bar **/  
mmenu = WTuimenubar_new(shell);  
  
/** Create the pop-up buttons **/  
pop1 = WTuimenupopup_new(mmenu, File);  
pop2 = WTuimenupopup_new(mmenu, Options);
```

```
pop3 = WTuimenupopup_new(mmenu, WTK);

/** Create the menu items */
pb1 = WTuimenuitem_new(pop1,New);
pb2 = WTuimenuitem_new(pop1,Open);
pb3 = WTuimenuitem_new(pop1,Save);
pb4 = WTuimenuitem_new(pop1,Exit);
pb5 = WTuimenuitem_new(pop2,WireFrame);
pb6 = WTuimenuitem_new(pop2,Picking);
pb7 = WTuimenuitem_new(pop2,Lights);
pb8 = WTuimenuitem_new(pop3,WTK Start);
pb9 = WTuimenuitem_new(pop3,WTK Stop);
/** Register callbacks for each of above buttons in the menu */
WTui_setcallback(pb1, WTUIEVENT_ACTIVATE, WTUI_FileNew, NULL);
WTui_setcallback(pb2, WTUIEVENT_ACTIVATE, WTUI_FileOpen, NULL);
WTui_setcallback(pb3, WTUIEVENT_ACTIVATE, WTUI_FileSave, NULL);
WTui_setcallback(pb4, WTUIEVENT_ACTIVATE, WTUI_FileExit, NULL);
WTui_setcallback(pb5, WTUIEVENT_ACTIVATE, WTUI_DoWireFrame, NULL);
WTui_setcallback(pb6, WTUIEVENT_ACTIVATE, WTUI_DoPicking, NULL);
WTui_setcallback(pb7, WTUIEVENT_ACTIVATE, WTUI_DoLights, NULL);
WTui_setcallback(pb8, WTUIEVENT_ACTIVATE, WTUI_DoWTKStart, NULL);
WTui_setcallback(pb9, WTUIEVENT_ACTIVATE, WTUI_DoWTKStop, NULL);
```

The above code generates a menu system having the following attributes:

- The main menu bar with three pop-up buttons named File, Options and WTK.
- Each of the above is associated with a popup menu that has several menu items associated with it. For example, if the user clicks on the File popup button, a menu pops up with the following items: New, Open, Save, and Exit.
- Each menu item has a callback handler function associated with it.

It is your responsibility to write the callback handler functions named above (i.e., DoPicking, DoLights etc.). That's all there is to creating a menu system. It can be as simple or as complex as the application demands.

Tool Bars

WTUIToolbar_new

```
WTui *WTUIToolbar_new(  
    WTui *parent,  
    int items,  
    char **bitmap_files);
```

This function creates a new tool bar and returns a pointer to it.

Arguments:

<i>parent</i>	The parent UI object, usually a form.
<i>items</i>	The number of items required in the tool bar.
<i>bitmap_files</i>	A pointer to an array of character strings, each holding the filename of the image file for above items. For UNIX the file format is .xbm; for Windows the format is .bmp. However, when you declare the array of character strings, you do not have to specify the filename extension. This ensures cross-platform portability.

Tool bars allow users to quickly accomplish a task instead of selecting the appropriate menu item and clicking on it. A tool bar object is essentially a group of push buttons, each painted with an appropriate bitmap.

During creation, you must specify the number of items required in the tool bar and pass a pointer to an array of character strings, each specifying the name of the bitmap file to be displayed on each button of the tool bar.

To get the best visual results, all bitmaps in the tool bar should of the same size. As with other UI objects, callback handler functions can be associated with each button of the tool bar. This can be accomplished using *WTui_settoolbarcallback* (see page 18-9), which is similar to the more generic *WTui_setcallback* function, with the difference being that you can pass the integer index of the desired button in the tool bar as one of the arguments.

User Interface Object's Utility Functions

Accessing the Scale Factors

WTui_setscalefactor

```
void WTui_setscalefactor(  
    float x,  
    float y);
```

This function adjusts the scaling factors that WTK uses when positioning a UI object.

WTui_getscalefactor

```
void WTui_getscalefactor(  
    float *x,  
    float *y);
```

This function retrieves the scaling factors used by WTK to position UI objects.

Accessing the Text for Text UI Objects

WTui_settext

```
FLAG WTui_settext(  
    WTui *ui,  
    char *text);
```

This function sets the text of a menu item, scrolled list, scrolled text, text input dialog box, or text field. It returns TRUE if successful, or FALSE otherwise.

Arguments:

<i>ui</i>	The menu item, scrolled list, scrolled text, text input dialog box, or text field UI object in question.
<i>text</i>	Text to display on the UI object.

This function replaces the Release 7 function *WTui_setmenutext*.

WTui_gettext

```
char *WTui_gettext(  
    WTui *ui);
```

This function returns the text of a menu item, scrolled list, scrolled text, text input dialog box, or text field UI object, or NULL if the UI object is not one of listed UI objects.

Argument:

<i>ui</i>	The UI object in question. Valid UI objects are menu items, scrolled lists, scrolled texts, text input dialog boxes and text fields.
-----------	--

For menu items, scrolled lists and text-input dialogs, this can be called inside the respective callback handler function, to access the selected string or string input, respectively. For scrolled texts and text fields this function can be called from the action function by associating it to a keyboard input.

This function replaces the Release 7 function *WTui_getmenutext*.

Accessing the Position of a Selection (Scrolled Lists and Radioboxes)

WTui_setselecteditem

```
void WTui_setselecteditem(  
    WTui *ui,  
    int item);
```

This function selects the *item* numbered text string in a scrolled list object or the *item* numbered togglebutton in a radiobox object. Items are numbered from 0 to n-1, where n is the number of items in the scrolled list or radio box. 0 refers to the first item and n-1 refers to the last item.

Argument:

ui	The scrolled list or radiobox UI object in question.
item	The item number of the scrolled list or radiobox which is to be selected.

WTui_getselecteditem

```
int WTui_getselecteditem(  
    WTui *ui);
```

This function retrieves the position of the selected text string in a scrolled list object or the position of the selected togglebutton in a radiobox object.

Argument:

ui	The scrolled list or radiobox UI object in question.
----	--

The values returned vary from 0 to n-1, where n is the number of items in the scrolled list or radio box. 0 refers to the first item and n-1 refers to the last item. A value of -1 is returned if the UI object is not a scrolled list or a radiobox. This function replaces the Release 7 function *WTui_getselected*. Use *WTui_getnumitems* on page 18-32 to determine how many items are in a scrolled list or radiobox.

Accessing the Number of Items (Scrolled Lists and Radioboxes)

WTui_getnumitems

```
int WTui_getnumitems(  
    WTui *ui);
```

This function returns the number of items contained in the specified scrolled list or radiobox UI object.

Accessing Text of Scrolled List Items

WTui_setitemtext

```
void WTui_setitemtext(  
    WTui *ui,  
    char *text,  
    int item);
```

This function assigns the text string to the *item* numbered element of the specified scrolled list. Items are numbered from 0 to n-1, where n is the number of items in the scrolled list. Use *WTui_getnumitems* on page 18-32 to determine the number of items contained in a scrolled list.

WTui_getitemtext

```
const char* WTui_getitemtext(  
    WTui *ui,  
    int item);
```

This function returns the text string associated with the *item* numbered element of the specified scrolled list. Items are numbered from 0 to n-1, where n is the number of items in the scrolled list. Use *WTui_getnumitems* on page 18-32 to determine the number of items contained in a scrolled list.

Inserting or Deleting Items (Scrolled Lists)

WTui_insertitem

```
void WTui_insertitem(  
    WTui *ui,  
    int index,  
    char *text);
```

This function inserts a new text string into an existing scrolled list object.

Arguments:

<i>ui</i>	The scrolled list UI object in question.
<i>index</i>	The position where the text string is to be inserted (0 to n , where n is the number of items in the scrolled list. 0 inserts before all other strings, n inserts at the end of the list).
<i>text</i>	The text string to be inserted.

WTui_deleteitem

```
void WTui_deleteitem(  
    WTui *ui,  
    int item);
```

This function deletes a text string from an existing scrolled list object.

Arguments:

<i>ui</i>	The scrolled list UI object in question.
<i>item</i>	The position where the text string is to be deleted (0 to $n-1$, where n is the number of items in the scrolled list. 0 deletes the first item and $n-1$ deletes the last item).

Accessing Status of UI Objects

WTui_enable

```
FLAG WTui_enable(  
    WTui *ui,  
    FLAG flag);
```

This function enables or disables (dims or undims) the specified menu item or radiobox. It returns TRUE if the UI object is enabled, or FALSE otherwise. This function replaces the Release 7 function *WTui_dimitem*.

Arguments:

<i>ui</i>	The menu item or radiobox UI object in question.
<i>flag</i>	This parameter should be set to either <i>TRUE</i> (indicating the menu item or radiobox should be enabled) or <i>FALSE</i> (indicating the menu item or radiobox should be disabled).

WTui_isenabled

```
FLAG WTui_isenabled(  
    Wtui *ui);
```

This function returns TRUE if the menu item or radiobox is enabled, or FALSE otherwise. This function replaces the Release 7 function *WTui_isdimmed*.

Argument:

<i>ui</i>	Specifies the menu item or radiobox UI object in question.
-----------	--

Accessing State of UI Objects (Menu Items and Checkbuttons)

WTui_check

```
int WTui_check(  
    WTui *ui,  
    int flag);
```

This function is used to enable or disable the checkmark on a checkbutton or menu item. If *flag* is TRUE, the checkbutton's or menu item's checkmark will be enabled (displayed). If *flag* is FALSE, the checkbutton's or menu item's checkmark will be disabled (not displayed).

Arguments:

ui	The checkbutton or menu item UI object in question.
flag	Specifies whether to check the checkmark in a checkbutton or menu item.

It returns 1 if the checkbutton is checked, or 0 otherwise. A value of -1 is returned if the UI object is not a checkbutton or a menu item.

WTui_isChecked

```
int WTui_isChecked(  
    WTui *ui);
```

This function returns TRUE if the specified checkbutton's or menu item's checkmark is checked and FALSE if it is not checked. A value of -1 is returned if the UI object is not a checkbutton or menu item. This function replaces the Release 7 function *WTui_checkbuttonstate*.

Arguments:

ui	The checkbutton or menu item UI object in question.
----	---

Accessing the Position of UI objects

WTui_setposition

```
void WTui_setposition(  
    WTui *ui,  
    int left,  
    int top,  
    int width,  
    int height);
```

This function sets the scaled position and size of the UI object. See *Scaled Position and Size* on page 18-7. This function is applicable to all UI objects.

Arguments:

<i>ui</i>	The UI object in question.
<i>left</i>	Upper-left X coordinate
<i>top</i>	Upper-left Y coordinate.
<i>width</i>	Width of object.
<i>height</i>	Height of object.

WTui_getposition

```
void WTui_getposition(  
    WTui *ui,  
    int *left,  
    int *top,  
    int *width,  
    int *height);
```

This function returns the scaled position and size of the UI object. See *Scaled Position and Size* on page 18-7. This function is applicable to all UI objects.

Arguments:

<i>ui</i>	The UI object in question.
<i>left</i>	Upper-left X coordinate
<i>top</i>	Upper-left Y coordinate.
<i>width</i>	Width of object.
<i>height</i>	Height of object.

Extending The UI Functionality of Your Application

WTui_getid

```
WTwinidtype WTui_getid(  
    WTui *ui)
```

This function returns a platform-specific ID. On Windows platforms, the return type is `HWND`. On UNIX platforms, the return type is `Widget`.

Argument:

<i>ui</i>	The UI object in question.
-----------	----------------------------

As mentioned in the beginning of the chapter, *WTui* does not provide comprehensive GUI functionality. It provides the basic functionality to create a GUI. However, you can extend the functionality of your application by adding UI objects directly using Motif or MFC (depending on the platform). This function allows access to the platform-specific ID to do that.

Controlling the WorldToolKit Simulation Loop

A WTK UI application must process the window system events while the WorldToolKit simulation loop is running. Sometimes an application may be required to start/stop the WTK simulation loop, while still being able to process the window events. In other words,

an application may require the user to interact with the UI objects whether or not the WTK simulation is running. The following three functions give you this control.

WTui_wtkstart

```
void WTui_wtkstart(
    void);
```

This function starts the WTK simulation loop. It should always be called from a callback handler function.

WTui_wtkstop

```
void WTui_wtkstop(
    void);
```

This function stops the WTK simulation loop. It is a complementary function to *WTui_start* and is used to stop the WTK simulation loop, while window system events are still processed. It can be called from the universe action function or from a callback handler function.

Note: This function should never be called after *WTuniverse_delete*.

WTui_iswtkrunning

```
FLAG WTui_iswtkrunning(
    void);
```

This function returns TRUE if the WTK simulation loop is running at the time this function is called, or FALSE otherwise.

Example: Simulating WTuniverse_go1

```
/** The following code segment simulates WTuniverse_go1. It calls WTui_go as usual
from the main function to start the WTK simulation loop. In the action function a global
counter keeps track of the number of frames the simulation is run - 50 times in this
example. The simulation can be started again by calling WTui_wtkstart from a callback
function of a pushbutton or menu item (not shown in this code segment)***/
```

```
/** global variables */
FLAG go1 = TRUE;
int framecount = 0;

int main(int argc, char *argv[])
{
    ....
    ....
    ....
    ....
    WTuniverse_setactions(actions);

    WTui_go(toplevel, TRUE);
    return 0;
}
void actions(void)
{
    if (go1) {
        if (framecount != 50) {
            /** tasks to perform in the first 50 frames */
            framecount++;
        }
        else {
            WTui_wtkstop();
            go1 = FALSE;
        }
    }
    /** rest of the tasks */
}
```

Miscellaneous Functions

WTui_delete

```
void WTui_delete(  
    WTui *ui);
```

This function deletes the UI object and its children.

Argument:

ui Specifies the UI object to delete.

WTui_getparent

```
WTui *WTui_getparent (  
    WTui *ui);
```

This function returns a pointer to the UI object's parent.

Argument:

ui Specifies the UI object in question.

WTui_unmanage

```
void WTui_unmanage(  
    WTui *ui);
```

This function is used to hide a UI object.

Argument:

ui Specifies the UI object in question.

Note: *This function is not applicable for the toplevel application shell, menus (menubar, pop-up menus, and menu items) and toolbars.*

WTui_getcallback

```
UICBP WTui_getcallback(  
    WTui *ui,  
    int eventtype);
```

This function returns a pointer to the callback handler function associated with the specified UI object.

Argument:

<i>ui</i>	Specifies the UI object in question.
<i>eventtype</i>	This parameter should always be set to <i>WTUIEVENT_ACTIVATE</i> .

Use this function to dynamically switch callbacks between UI objects. For example, to switch callbacks of two pushbuttons, you would get their callbacks using this function and then set the appropriate callbacks using *WTui_setcallback* (see page 18-8.)

WTui_showconsole

```
void WTui_showconsole(  
    int flag);
```

This function can be used on Windows platforms to show or hide the console window. Pass in *TRUE* as the *flag* parameter to make the console window visible, and *FALSE* to make the console window invisible.

WTui_isconsolevisible

```
int WTui_isconsole(void);
```

This function returns *TRUE* if the console window is visible, and *FALSE* if it is not visible. This function can only be used on Windows platforms.

Drawing Functions

User-defined Drawing Functions

WTK lets you embed your own OpenGL drawing routines into your WTK application. SENSE8 created this capability because there will probably always be a greater number of rendering capabilities of your OpenGL workstation than can be supported within WTK. With WTK's user-defined drawing functions, you can combine the full rendering capabilities of your workstation with the functionality of WTK. The drawing functions are classified into two categories: 2D (see below) and 3D (see page 19-8).

2D Drawing

WTwindow_setfgactions

See *WTwindow_setfgactions* on page 17-24 for a description.

Pre-defined 2D Drawing Functions

WTwindow_set2Dcolor

```
void WTwindow_set2Dcolor(  
    WTwindow *window,  
    unsigned char r,  
    unsigned char g,  
    unsigned char b);
```

This function specifies the color to be used by subsequent 2D drawing (*WTwindow_draw2D...*) functions. The default color is white. This function should only be

called from within the user-defined *fgdrawfn* function, which is specified in *WTwindow_setfgactions* (see page 19-1).

WTwindow_set2Dlinestyle

```
void WTwindow_set2Dlinestyle(  
    WTwindow *window,  
    int style);
```

This function sets the 2D line style of the specified window to the stipple pattern represented by the *style* parameter. The default line style is solid.

The *style* parameter's 16 least significant bits represent a bitmask where a 1 bit indicates that drawing occurs while a 0 bit indicates that drawing does not occur. If the style value is:

<i>0xaaaa</i> ,	The line style will be solid
<i>0x3333</i>	Results in dashed lines

This function should only be called from within the user-defined *fgdrawfn* function, which is specified in *WTwindow_setfgactions* (see page 19-1).

WTwindow_set2Dlinewidth

```
void WTwindow_set2Dlinewidth(  
    WTwindow *window,  
    float width);
```

This function sets the 2D line width (in pixels) for the specified window to the value specified by the *width* argument. The default line width is 1.0.

This function should only be called from within the user-defined *fgdrawfn* function, which is specified in *WTwindow_setfgactions* (see page 19-1).

WTwindow_draw2Dcircle

```
void WTwindow_draw2Dcircle(  
    WTwindow *window,  
    float xc,  
    float yc,  
    float radius,  
    int mode);
```

This function draws a circle whose center is specified by the *xc*, *yc* parameters. The *xc*, *yc*, and *radius* values are in normalized window coordinates (i.e., 0.0 -1.0). The discussion on the function *WTwindow_setfgactions* (see page 19-1) talks about normalized window coordinates. A radius of 1.0 will be the smaller of the window height or width.

The *mode* parameter indicates the drawing style to be used. If the mode is:

```
WTDRAW2D_HOLLOW    Draw outline.  
WTDRAW2D_SOLID     Solid fill.
```

This function should only be called from within the user-defined *fgdrawfn* function, which is specified in *WTwindow_setfgactions* (see page 19-1).

Also see, *WTwindow_set2Dcolor* (page 19-1), *WTwindow_set2Dlinestyle* (page 19-8), and *WTwindow_set2Dlinewidth* (page 19-2).

WTwindow_draw2Drectangle

```
void WTwindow_draw2Drectangle(  
    WTwindow *window,  
    float x1,  
    float y1,  
    float x2,  
    float y2,  
    int mode);
```

This function draws a rectangle whose bottom left point is specified by the *x1*, *y1* values and the upper right point is specified by the *x2*, *y2* values. The *x1*, *y1*, *x2*, and *y2* values are in normalized coordinates (i.e., 0.0-1.0). The discussion on the function *WTwindow_setfgactions* (see page 19-1) talks about normalized window coordinates.

The *mode* parameter indicates the drawing style to be used. If the mode is:

`WTDRAW2D_HOLLOW` Draw outline.

`WTDRAW2D_SOLID` Solid fill.

This function should only be called from within the user-defined *fgdrawfn* function, which is specified in *WTwindow_setfgactions* (see page 19-1).

Also see, *WTwindow_set2Dcolor* (page 19-1), *WTwindow_set2Dlinestyle* (page 19-8), and *WTwindow_set2Dlinewidth* (page 19-2).

WTwindow_draw2Dpoint

```
void WTwindow_draw2Dpoint(  
    WTwindow *window,  
    float x,  
    float y);
```

This function draws a point at the coordinates specified by the *x* and *y* values. The *x* and *y* values are in normalized coordinates (i.e., 0.0-1.0). The discussion on the function *WTwindow_setfgactions* (see page 19-1) talks about normalized window coordinates.

This function should only be called from within the user-defined *fgdrawfn* function, which is specified in *WTwindow_setfgactions* (see page 19-1). Also see *WTwindow_set2Dcolor* (see page 19-1).

WTwindow_draw2Dline

```
void WTwindow_draw2Dline(  
    WTwindow *window,  
    float x1,  
    float y1,  
    float x2,  
    float y2);
```

This function draws a line between the point specified by the *x1* and *y1* values to the point specified by the *x2* and *y2* values. The *x1*, *y1*, *x2*, and *y2* values are in normalized coordinates (i.e., 0.0-1.0). The discussion on the function *WTwindow_setfgactions* (see page 19-1) talks about normalized window coordinates.

This function should only be called from within the user-defined *fgdrawfn* function, which is specified in *WTwindow_setfgactions* (see page 19-1).

Also see, *WTwindow_set2Dcolor* (page 19-1), *WTwindow_set2Dlinestyle* (page 19-8), and *WTwindow_set2Dlinewidth* (page 19-2).

WTwindow_draw2Dtexture

```
void WTwindow_draw2Dtexture(  
    WTwindow *window,  
    char *bitmapname,  
    FLAG transparent,  
    WTp2 *xyarray,  
    WTp2 *uvarray);
```

This function drapes the specified texture bitmap specified by *bitmapname* onto the 2D polygon represented by the sequence of four coordinates defined by *xyarray* and their respective texture coordinates contained in *uvarray*. The *xyarray* should contain coordinates for four vertices, because the 2D polygon is assumed to be a quad. Moreover, the vertices *have* to be specified in the counter-clockwise order.

The *bitmapname* argument must be a name of a bitmap previously loaded in WTK, outside of the 2D callback function. The easiest way to do this is to call *WTtexture_cache(bitmapname,TRUE)* (see page 10-17) prior to starting the universe.

If a texture is transparent (*transparent=TRUE*), you will be able to see through portions of the polygon to which the texture is applied. The *transparent* flag indicates whether black pixels in the texture should be rendered; if black pixels are not rendered, then they are effectively transparent.

This function should only be called from within the user-defined *fgdrawfn* function, which is specified in *WTwindow_setfgactions* (see page 19-1).

WTwindow_set2Dfont

```
void WTwindow_set2Dfont(  
    WTwindow *window,  
    int fontindex);
```

This function sets the font index to be used when drawing 2D text to the specified window. The default font is system dependent on Windows 32-bit platforms, whereas on UNIX platforms, there is no default font. The specified *fontindex* parameter is used as an index into an ASCII file named *font.wtk*, which contains a list of font names that are loaded automatically by WTK. Valid values for the *fontindex* parameter range between 0 and the number of entries in the *font.wtk* file minus one. The user must create the *font.wtk* file and place it in the same directory as the application.

To obtain a list of available fonts on a UNIX system, run *xlsfonts*. To obtain a list of available fonts in a Windows 32-bit system, use the fonts icon in the Control Panel.

On Windows 32-bit platforms, the format of the *font.wtk* file is font name followed by font size. Following is a sample of the *font.wtk* file (2 entries) on Windows 32-bit platforms:

```
Arial Bold Italic    20  
Times New Roman     15
```

On UNIX platforms, the best way to create the *font.wtk* file is to first redirect the output of the *xlsfonts* command to a file, say *font.txt*.

```
xlsfonts > font.txt
```

Now you can open this file and copy the fonts you require to the *font.wtk* file. Following is a sample of the *font.wtk* file (2 entries) on SGI Indigo2 Impact:

```
-adobe-courier-bold-o-normal--0-0-0-0-m-0-iso8859-1  
-sgi-fixed-bold-r-normal--15-140-75-75-c-90-iso8859-3
```

This function should only be called from within the user-defined *fgdrawfn* function, which is specified in *WTwindow_setfgactions* (see page 19-1).

Also see *WTwindow_set2Dcolor* (see page 19-1).

WTwindow_draw2Dtext

```
void WTwindowdrawt2Dtext(  
    WTwindow *window,  
    float x,  
    float y,  
    char *text);
```

This function draws the specified text string (specified by the *text* parameter) starting at the *x*, *y* coordinates of the specified window. The *x* and *y* values are in normalized coordinates (i.e., 0.0-1.0). The discussion on the function *WTwindow_setfgactions* (see page 19-1) talks about normalized window coordinates.

The font used to draw the text string is set by calling *WTwindow_set2Dfont* (see page 19-6).

This function should only be called from within the user-defined *fgdrawfn* function, which is specified in *WTwindow_setfgactions* (see page 19-1).

Note: On UNIX platforms, since there is no default font, you must call *WTwindow_set2Dfont* (see page 19-6) before calling this function. On Windows 32-bit platforms, if this function is called without calling *WTwindow_set2Dfont* (see page 19-6), it uses the default system dependent font.

WTwindow_get2Dtextextents

```
void WTwindow_get2Dtextextents(  
    WTwindow *window,  
    char *string,  
    float *width,  
    float *height);
```

This function returns the width and height in normalized window coordinates of the specified text string's extents.

This function should only be called from within the user-defined *fgdrawfn* function, which is specified in *WTwindow_setfgactions*.

3D Drawing

WTwindow_setdrawfn

See *WTwindow_setdrawfn* on page 17-23 for a description.

Pre-defined 3D Drawing Functions

WTwindow_set3Dcolor

```
void WTwindow_set3Dcolor(  
    WTwindow *window,  
    unsigned char r,  
    unsigned char g,  
    unsigned char b);
```

This function specifies the color to be used by subsequent 3D drawing (*WTwindow_draw3D...*) functions. The default color is white.

This function should only be called from within the user-defined *drawfn* function, which is specified in *WTwindow_setdrawfn* (see page 19-8).

WTwindow_set3Dlinestyle

```
void WTwindow_set3Dlinestyle(  
    WTwindow *window,  
    int style);
```

This function sets the 3D line style of the specified window to the stipple pattern represented by the *style* parameter. The default line style is solid.

The *style* parameter's 16 least significant bits represent a bitmask where a 1 bit indicates that drawing occurs while a 0 bit indicates that drawing does not occur.

If the style value is:

<code>0xaaaa,</code>	The line style will be solid
<code>0x3333</code>	Results in dashed lines

This function should only be called from within the user-defined *drawfn* function, which is specified in *WTwindow_setdrawfn* (see page 19-8).

WTwindow_set3Dlinewidth

```
void WTwindow_set3Dlinewidth(  
    WTwindow *window,  
    float width);
```

This function sets the 3D line width (in pixels) for the specified window to the value specified by the *width* parameter. The default line width is 1.0.

This function should only be called from within the user-defined *drawfn* function, which is specified in *WTwindow_setdrawfn* (see page 19-8).

WTwindow_set3Dpointsize

```
void WTwindow_set3Dpointsize(  
    WTwindow *window,  
    float size);
```

This function sets the 3D point size (in pixels) of the specified window to the value specified by the *size* parameter. The default point size is 1.0.

This function should only be called from within the user-defined *drawfn* function, which is specified in *WTwindow_setdrawfn* (see page 19-8).

WTwindow_draw3Dpoints

```
void WTwindow_draw3Dpoints(  
    WTwindow *window,  
    WTp3 *pts,  
    int numpts);
```

This function draws a set of points at the coordinates specified in the *pts* array. The *numpts* parameter specifies the number of points to draw.

This function should only be called from within the user-defined *drawfn* function, which is specified in *WTwindow_setdrawfn* (see page 19-8).

See *WTwindow_set3Dcolor* on page 19-8 and *WTwindow_set3Dpointsize* above.

WTwindow_draw3Dlines

```
void WTwindow_draw3Dlines(  
    WTwindow *window,  
    WTp3 *pts,  
    int numpts,  
    FLAG style);
```

This function draws a set of lines between the points specified in the *pts* array. The *numpts* parameter specifies the number of points contained in the *pts* array.

The *style* parameter indicates the drawing style to be used. If the style is:

<i>WTLINE_SEGMENTS</i>	individual line segments
<i>WTLINE_CONNECTED</i>	a connected series of line segments
<i>WTLINE_CLOSE</i>	a closed series of line segments (i.e., single series of connected lines with an additional line segment between the last and first point)

This function should only be called from within the user-defined *drawfn* function, which is specified in *WTwindow_setdrawfn* (see page 19-8).

See *WTwindow_set3Dcolor* on page 19-8, *WTwindow_set3Dlinestyle* on page 19-8, and *WTwindow_set3Dlinewidth* on page 19-9.

Introduction

The WTK sound support library provides a common cross-platform API for playing sound files on various hardware platforms. Some platforms support spatialized sound, while others simply provide ambient sounds.

A common scenario would proceed as follows:

1. Open an audio hardware device.
2. Set up the hardware parameters (output type, rolloff, etc.).
3. Load various sound samples from disk.
4. Assign properties to sounds (volume, pitch, priority, position, etc.).
5. Cue the sounds to play on events or loop continuously during the simulation.
6. Close the audio hardware device, which removes the sound samples from memory.

Supported Devices

WINDOWS 95/NT

- Windows-compatible sound card (WINMM)

This device does not require any special software. For this device to work, a standard Windows compatible sound card should be installed and working. Using this device you can play one software-spatialized sound at a time.

- DiamondWare with Windows-compatible sound card (DWSTK)

For this device type to work, you need to have a standard Windows compatible sound card installed and working, as well as the DiamondWare STK DLL. Using this device you can play up to 16 software-spatialized sounds at a time. To install the DLL, copy the *DWSW32.DLL* file to your Windows system directory.

- Crystal River Engineering AudioReality NT Sound Server (CRE)

This device requires an AudioReality NT Sound Server from Crystal River Engineering. This device is a separate PC which contains hardware specifically designed to produce high-quality 3D audio. WTK communicates with the Sound Server through a null-modem serial cable. Using this device you can play up to four hardware-spatialized sounds at a time.

- Direct Sound (only available with Windows 95 using the Direct3D version of WTK)

This device allows up to 16 spatialized sounds to be played at once. This capability does not require any special software or hardware other than the DirectX toolkit (installed with WTK Direct) and a Windows-compatible sound card. See your Hardware Guide for more information.

SGI

- SGI Audio Library (SGI)

This device requires that you have an IRIS Audio Processor, and the Audio Library (AL) installed. The number of sounds that can be software-spatialized at a time depends on the system hardware. Multiple sounds of differing frequencies cannot be played simultaneously on this device. If you attempt to simultaneously play multiple sounds with differing frequencies on this device, only the sounds whose frequencies are identical to the frequency of the most recently loaded sound will actually play.

- VSI Synthesizer (VSI)

This device requires a synthesizer from Visual Synthesis Incorporated. Using this device you can play up to 16 hardware-spatialized sounds at a time.

- Crystal River Engineering Acoustetron (CRE)

Same as above, with the exception that the SGI requires an Acoustetron Server rather than an AudioReality NT Server.

Note: For vendor-specific information, see *Sources of Components* on page J-1.

Device-level Functionality

WTsounddevice_open

```
WTsounddevice *WTsounddevice_open(  
    int type,  
    int nplayable,  
    WTviewpoint *listener);
```

This function opens an audio device and assigns the specified viewpoint as the listener. It returns a pointer to a new sound device object, or NULL if unsuccessful.

If a NULL is passed for *listener*, then the default universe viewpoint (returned from *WTuniverse_getviewpoints()*) is set as the default listener. Use the *nplayable* argument to request a specific number of sounds you want to play simultaneously. The actual number may be adjusted by the hardware you are using. (To determine the number of sounds you can play, use *WTsounddevice_numplayable*.)

Arguments:

<i>type</i>	<i>WTSOUNDDEVICE_DS</i> <i>WTSOUNDDEVICE_WINMM</i> <i>WTSOUNDDEVICE_DWSTK</i> <i>WTSOUNDDEVICE_VSI</i> <i>WTSOUNDDEVICE_SGI</i> <i>WTSOUNDDEVICE_CRE</i>
<i>nplayable</i>	The number of sounds you would like to play at once.
<i>viewpoint</i>	The viewpoint assigned to the listener.

WTsounddevice_close

```
FLAG WTsounddevice_close(  
    WTsounddevice *device)
```

This function closes an audio device and deletes all sounds associated with the device. It returns TRUE, if successful, or FALSE otherwise. The *device* argument specifies the device you want to close.

WTsounddevice_update

```
WTsound *WTsounddevice_update (  
    WTsounddevice *device);
```

This function updates listener and sound position/orientations. For smooth motion of sounds and listeners, you should call this function from the universe's action function. For non-spatialized systems this function is used to force updates of sampling and mixing of sounds, and should still be called in the actions function.

Note: If the user does not call this function, it is called internally by WTK.

Argument:

device Device to update sounds for.

WTsounddevice_getsounds

```
WTsound *WTsounddevice_getsounds (  
    WTsounddevice *device);
```

This function gets a pointer to the list of sounds currently loaded by a device. The *device* argument specifies the device from which to get sounds.

It returns a pointer to the first sound in the device's list of sounds, NULL if no sounds are currently loaded by the device. Use *WTsound_next* (see page 20-11) to loop through the list until NULL is returned signifying the end of the list.

WTsounddevice_numplayable

```
int WTsounddevice_numplayable (  
    WTsounddevice *device);
```

This function gets the number of sources available for a device. This is the number of sounds that can play simultaneously. The *device* argument specifies the device from which to retrieve the number of sources.

It returns the number of sound sources available for the device. You can set the priority for each sound (see *WTsound_setparam* on page 20-12) so if you try to play more sounds than

the sound device is capable of playing, the highest priority sounds will play, while lower priority sounds get bumped from the list of simultaneously playing sounds.

WTsounddevice_name2sound

```
WTsound *WTsounddevice_name2sound (  
    WTsounddevice *device,  
    char *name);
```

This function gets a sound by its name. The *device* argument specifies the device from which to get the sound. The *name* argument specifies the name of the sound to get.

WTsounddevice_setparam

```
void WTsounddevice_setparam (  
    WTsounddevice *device,  
    int param,  
    float value);
```

This function sets various parameters for a sound device. The *device* argument specifies the device to modify. The *param* argument specifies the parameter to adjust. The *value* argument specifies the value to set. For example:

```
WTsounddevice_setparams(  
    myDevice,  
    WTSOUNDDEVICE_ROLLOFF,  
    10.0f);
```

Table 20-1 shows which options can be used with which hardware. Table 20-2 describes some of these options further.

Please refer to *CRE Device Parameters* on page 20-7 for descriptions of parameters for Crystal River Engineering devices.

Table 20-1: Options for WTsounddevice_setparam

	WINMM	DWSTK	CRE	DS	SGI	VSI
<i>WTSOUNDDEVICE_OUTPUT</i>			x	x		
<i>WTSOUNDDEVICE_ROLLOFF</i>	x	x	x	x	x	
<i>WTSOUNDDEVICE_ROLLOFFEXP</i>			x			
<i>WTSOUNDDEVICE_ABSORBDIST</i>			x			
<i>WTSOUNDDEVICE_SPATIALIZE</i>	x	x	x	x	x	x

Key for table 20-1:

- WINMM Windows-compatible sound card
- DWSTK DiamondWare with Windows-compatible sound card
- CRE Crystal River Engineering Audio Reality NT Sound Server
- DS Direct Sound (via DirectX) for Windows 95
- SGI SGI Audio Library
- VSI VSI Synthesizer

Table 20-2 describes some of the options for *WTSounddevice_setparams*.

Table 20-2: Description of *WTSounddevice_setparams* options

	Description	Range
<i>WTSOUNDDEVICE_OUTPUT</i>	The type of output hardware used, like headphones or speakers.	WTOUTPUT_HEADPHONE, WTOUTPUT_STEREO, WTOUTPUT_SURROUND <i>Default: HEADPHONE</i>
<i>WTSOUNDDEVICE_ROLLOFF</i>	Rolloff or clipping distance, i.e., the distance at which sound becomes silent.	0.0 to... <i>Default: 2500.0</i>
<i>WTSOUNDDEVICE_SPATIALIZE</i>	Spatialize sounds for this device.	WTSPATIALIZE_ON or WTSPATIALIZE_OFF <i>Default: WTSPATIALIZE_ON</i>

CRE Device Parameters

As indicated by table 20-1 on page 20-6, the parameters that are applicable to the CRE device are *WTSOUNDDEVICE_OUTPUT*, *WTSOUNDDEVICE_ROLLOFFEXP*, *WTSOUNDDEVICE_SPATIALIZE* and *WTSOUNDDEVICE_ABSORBDIST*. (*WTSOUNDDEVICE_ROLLOFF* is interpreted as *WTSOUNDDEVICE_ABSORBDIST*.)

WTSOUNDDEVICE_OUTPUT refers to the type of output hardware used. WTK chooses the headphones (*WTOUTPUT_HEADPHONE*) as the default for the CRE server output. The other types supported are the *WTOUTPUT_STEREO* and the *WTOUTPUT_SURROUND*, which refer to speakers built in your monitor or placed around your monitor, respectively.

WTSOUNDDEVICE_ROLLOFFEXP is an index to the roll-off due to spreading power loss. This parameter is set for the device, so it affects all sounds created. Typically, roll-off exponents in the range of 0.5 to 1.2 are of interest. The default value is 0.8. Use a float when you refer to this parameter. A zero or a negative roll-off resets it to its default value.

`WTSOUNDDEVICE_ABSORBDIST` refers to the atmospheric absorption control distance. This parameter is used to simulate atmospheric absorption and controls the amount of extra high frequency fall-off over distance. This option also is used only on the “device,” so it affects all created sounds. The argument should be a float, indicating a distance in current units. A zero or negative argument resets it to its default value.

Note: You can make use of the CRE Acoustetron environment variables to set either the port specifications or the baud rate at which to communicate with the CRE. For example, on UNIX platforms if you want to use the CRE on `/dev/ttyd2` at a baud rate of 38400, do the following:

```
setenv TRONCOM 2@384
```

Please refer to your CRE manual for more information.

WTsounddevice_getparam

```
float WTsounddevice_getparam (  
    WTsounddevice *device,  
    int param);
```

This function returns a parameter for a sound device. See table 20-4 on page 20-14. The *device* argument specifies the device to get the parameters from. The *param* argument specifies the parameters to adjust. For example:

```
float rolloff;  
rolloff=WTsounddevice_getparam (  
    myDevice,  
    WTSOUNDDEVICE_ROLLOFF);
```

WTsounddevice_setdata

```
void WTsounddevice_setdata (  
    WTsounddevice *device,  
    void *data);
```

This function attaches user-defined data to a sound device. The *device* argument specifies the device where the data will be attached. The *data* argument is a pointer to the data.

Private application data can be stored in any structure. To store a pointer to the structure within the sound, pass in a pointer, cast to a *void**, as the *data* argument.

WTsounddevice_getdata

```
void *WTsounddevice_getdata (  
    WTsounddevice *device);
```

This function retrieves the user-defined data from a sound device. The *device* argument specifies the device from which the data is retrieved.

This function returns a void pointer to the sound's data. You should cast the value returned by this function to the same type that was used to store the data in the sound with *WTsound_setdata*.

Device-level Spatializing Functions

WTsounddevice_setlistener

```
FLAG WTsounddevice_setlistener (  
    WTsounddevice *device,  
    WTviewpoint *viewpoint);
```

This function specifies a viewpoint as a listener. The *device* argument specifies the device for which the listener is set. The *viewpoint* argument specifies the location of the listener. It returns TRUE if successful, FALSE if unsuccessful.

WTsounddevice_getlistener

```
WTviewpoint *WTsounddevice_getlistener (  
    WTsounddevice *device);
```

This function gets the viewpoint that is being used as a listener by a device. The *device* argument specifies the device from which to get the listener. This function returns the viewpoint used as listener or NULL if unsuccessful.

Sound-level Functionality

WTsound_load

```
WTsound *WTsound_load (  
    WTsounddevice *device,  
    char *source);
```

This function creates a new sound from a source. The *device* argument specifies the sound device from which to load the sound. The *source* argument specifies the name of source (filename, resource name, or sample name). This function returns a pointer to the new sound or NULL if unsuccessful.

The source can be a file or, on VSI systems, a sample in a midi bank. For VSI sound systems, first call this function with the bank name to load, then make subsequent calls to load particular samples. The filename should contain the path to the sound if it is needed. Search paths are not used for loading sounds. The CRE sound servers do not require a directory path.

WTsound_delete

```
FLAG WTsound_delete (  
    WTsound *sound);
```

This function deletes a sound. The *sound* argument specifies which sound to delete. If the deletion is successful, the function returns TRUE; otherwise it returns FALSE.

WTsound_stop

```
void WTsound_stop (  
    WTsound *sound);
```

This function stops a currently playing sound. The *sound* argument specifies the name of the sound to stop playing.

WTsound_next

```
WTsound *WTsound_next (  
    WTsound *sound);
```

This function iterates through the list of sounds currently loaded by a device. The *sound* argument specifies the name of the sound to start playing. This function returns the next sound in the list of sounds, if there is one. If the sound is the last one in the list, it returns NULL.

To get a pointer to the first sound in this list, see *WTsounddevice_getsounds* on page 20-4.

WTsound_play

```
FLAG WTsound_play (  
    WTsound *sound);
```

This function cues a sound to begin playing. When a sound is finished playing, it returns to the beginning of the sample. The *sound* argument specifies the name of the sound to play.

This function returns a FALSE if the passed parameter (*sound*) is NULL (or if the device type in the sound structure is not one of the types mentioned under *WTsounddevice_open*). For all other cases, it returns TRUE.

When the sound is finished playing it calls its done function, which can be set using the *WTsound_setdonefn* function (see page 20-16).

Note that when using the SGI Audio Library, multiple sounds of differing frequencies cannot be played simultaneously. If you attempt to simultaneously play multiple sounds with differing frequencies using the SGI Audio library, only the sounds whose frequencies are identical to the frequency of the most recently loaded sound will actually play.

To check if a sound is currently playing, see *WTsound_isplaying*, below.

WTsound_isplaying

```
FLAG WTsound_isplaying (  
    WTsound *sound);
```

This function determines if a sound is currently playing. The *sound* argument specifies the name of the sound that you want to check.

This function returns a TRUE or FALSE. A playing sound is not necessarily audible due to spatialization or priority.

Note: For the Crystal River Engineering AudioReality NT Sound Server, this action requires communication over the serial port and therefore can be quite expensive from a performance perspective.

WTsound_setparam

```
void WTsound_setparam (  
    WTsound *sound,  
    int param,  
    float value);
```

This function sets various parameters for a sound. The *sound* argument specifies the name of the sound to modify. The *param* argument specifies the parameter to adjust. The *value* parameter specifies the value to set. For example:

```
WTsound_setparam (  
    mySound,  
    WTSOUND_VOLUME,  
    0.5f);
```

Sound sample rates:

WTSAMPLERATE_8KHZ

WTSAMPLERATE_11KHZ

WTSAMPLERATE_16KHZ

WTSAMPLERATE_22KHZ

WTSAMPLERATE_32KHZ

WTSAMPLERATE_44KHZ

WTSAMPLERATE_48KHZ

These sample rates can be used for setting the *WTSOUND_PLAYRATE* parameter of the sound.

Table 20-4 shows which options can be used with which hardware.

Table 20-3: Devices with which WTsound_setparam options are used

	WINMM	DWSTK	CRE	DS	SGI	VSI
WTSOUND_VOLUME		x	x	x	x	x
WTSOUND_LRPAN		x			x	
WTSOUND_FBPAN					x	
WTSOUND_PITCH		x	x			x
WTSOUND_PLAYRATE				x	x	
WTSOUND_PRIORITY	x	x		x	x	x
WTSOUND_LOOPS	x	x	x	x	x	
WTSOUND_DOPPLER			x			
WTSOUND_SPATIALIZE	x	x	x	x	x	x

Key for table 20-4:

WINMM	Windows-compatible sound card
DWSTK	DiamondWare Sound Tool Kit with Windows-compatible sound card
CRE	Crystal River Engineering Audio Reality NT Sound Server
DS	Direct Sound (via DirectX) for Windows 95
SGI	SGI Audio Library
VSI	VSI Synthesizer

Table 20-4 describes some of the *Wtsound_setparams* options.

Table 20-4: Description of *Wtsound_setparam* options

	Description	Range
<i>WTSOUND_VOLUME</i>	Volume of a sound specified in ratio, 1.0 being the normal volume of a sound.	0.0 to... <i>Default: 1.0</i>
<i>WTSOUND_LRPAN</i>	The ratio of left/right volume.	-1.0 to +1.0 <i>Default: 0.0</i>
<i>WTSOUND_FBPAN</i>	The ratio of front/back volume.	-1.0 to +1.0 <i>Default: 0.0</i>
<i>WTSOUND_PITCH</i>	The pitch to be used for playing sound.	0.5 to 2.0 <i>Default: 1.0</i>
<i>WTSOUND_PLAYRATE</i>	The frequency of the playback rate wanted (in Hz). The defined sound sample rate constants listed above can also be used.	0.0 to... <i>Default: System default</i>
<i>WTSOUND_PRIORITY</i>	The priority for this sound. If a sound has higher priority it can stop other sounds so it can play first.	0.0 to 1.0 <i>Default 1.0</i>
<i>WTSOUND_LOOPS</i>	The number of loops to play this sound.	1.0 to... (-1.0 = Infinite) <i>Default: 1.0</i>
<i>WTSOUND_DOPPLER</i>	Doppler factor for a sound.	0.0 to... <i>Default: 1.0</i>
<i>WTSOUND_SPATIALIZE</i>	Spatialize this sound.	<i>WTSPATIALIZE_ON</i> or <i>WTSPATIALIZE_OFF</i> <i>Default: WTSPATIALIZE_ON</i>

WTsound_getparam

```
float WTsound_getparam (  
    WTsound *sound,  
    int param);
```

This function retrieves the parameters for a sound. Table 20-4 on page 20-14 also applies to this function. The *sound* argument specifies the name of the sound from which to get the parameters. The *param* argument specifies the parameter to get. For example:

```
float volume;  
volume=WTsound_getparams (  
    myDevice,  
    WTSOUND_VOLUME);
```

WTsound_getname

```
char *WTsound_getname (  
    WTsound *sound);
```

This function returns a pointer to the filename of the sound. The *sound* argument specifies the sound from which to get the name.

WTsound_setdata

```
void WTsound_setdata (  
    WTsound *sound,  
    void *data);
```

This function attaches user-definable data to a sound. The *sound* argument specifies the sound where the data will be attached. The *data* argument is the pointer to the data.

This function returns a *void* pointer to the sound's data. Private application data can be stored in any structure. To store a pointer to the structure within the sound, pass in a pointer, cast to a *void**, as the *data* argument.

WTsound_getdata

```
void *WTsound_getdata (  
    WTsound *sound);
```

This function retrieves user-defined data from a sound. The *sound* argument specifies the sound from which to get the data.

This function returns a void pointer to the sound's data. You should cast the value returned by this function to the same type that was used to store the data in the sound with *WTsound_setdata*.

WTsound_setdonefn

```
void WTsound_setdonefn (  
    WTsound *sound,  
    PFVS done);
```

This function sets a function to call when a sound is finished playing. The *sound* argument specifies the sound to modify. The *done* argument is a pointer to the function that is called when the sound is done playing. *PFVS* is a type signifying a pointer to a function returning VOID and taking a pointer to a WTsound structure as a parameter. In C terminology, PFVS means the following:

```
typedef void (*PFVS)(WTsound_ptr);
```

The following is an example for *WTsound_setdonefn*:

```
void doneFn(WTsound *sound)  
{  
    WTmessage("Sound is done\n");  
}
```


WTsound_getdonefn

```
PFVS WTsound_getdonefn (  
    WTsound *sound);
```

This function retrieves the function that will be called when the sound is done playing. It returns a pointer to the sound's function, which is specified in the *done* argument. The *sound* argument specifies from which sound the function is retrieved.

Sound-level Spatializing Functions

If none of these functions are used for positioning a sound, the sound will be placed at the current viewpoint position — or the origin if no viewpoint position exists.

WTsound_setposition

```
void WTsound_setposition (  
    WTsound *sound,  
    WTp3 position);
```

This function sets a sound's position in 3D space. This setting overrides object and viewpoint attachments for placing sounds. The *sound* argument specifies the sound for which the position is specified. The *position* argument specifies the new position setting for the sound void.

WTsound_getposition

```
void WTsound_getposition (  
    WTsound *sound,  
    WTp3 position);
```

This function returns the current position setting of a sound. The *sound* argument specifies the sound from which to get the position. The *position* argument is the current position of the sound void.

WTsound_setnodepath

```
FLAG WTsound_setnodepath(  
    WTsound *sound,  
    WTnodepath *npath);
```

This function assigns the sound to a source specified by a node path. The *sound* argument specifies the sound you want to attach. The *npath* argument specifies the node path that will be attached to the sound. Setting *npath* to NULL will remove the sound from its source, or object.

WTsound_getnodepath

```
WTnodepath *WTsound_getnodepath(  
    WTsound *sound);
```

This function retrieves the current nodepath associated with the sound

Client-Server Networking

(Via the World2World Servers)

Introduction

The Object/Property/Event programming paradigm (described in Chapter 3), in conjunction with the new high level networking functionality provided in WTK Release 8, provides programmers with the ability to easily develop multi-user 3D/VR networked applications for use over LANs or the Internet. The new high level networking capabilities are designed to operate in conjunction with Sense8's World2World server product. If you have not purchased the World2World server product, you will not be able to take advantage of the high level networking capabilities described in this chapter to build multi-user simulations. See below for a brief description of World2World or contact Sense8 for detailed information about the World2World product.

Based on the Object/Property/Event paradigm, World2World-compliant simulations are composed of objects and object properties. To allow multiple users to run and participate in the same simulation, each user (client) needs to be able to receive certain updates (changes in property values) made by the other participants. For example, suppose there is a graphical object in your simulation that you want each user to be able to manipulate. If one user drags the object to a new location, you will want the other users to also see that movement.

To achieve this, the affected property must be shared by both the client that is modifying the value and the clients that want to receive the new value. Each change made to the value of a property is known as an event. When a property is shared, the events that are internally generated for each property value change are what allow the updated information to be automatically sent over the network to any other clients that have also shared that property. If desired, you can add additional event handlers to specify actions to be performed in response to an event (see page 3-23).

The mechanism by which property value changes are transmitted to all clients who are sharing the property is the World2World server product. The World2World server product consists of a Server Manager, Simulation Servers, and an optional Firewall Proxy. A multi-user client application connects to the Server Manager, which determines whether the client

has the appropriate log-in authority and directs the client to the appropriate Simulation Server, based on the simulation that the client is running. The Simulation Server stores and organizes simulation data and distributes data updates as appropriate to other users of the multi-user application connected to the same Simulation Server. The WTK Release 8 API allows programmers to specify which object properties are to be shared, to specify how that shared data will be stored and organized on a World2World Simulation Server and provides the functionality necessary to connect to the World2World servers. By limiting network data transfer to only properties that have been shared, World2World helps to reduce bandwidth usage.

WTK applications can connect to multiple World2World Simulation Servers. Each connection made by WTK to a World2World Simulation Server is represented by a WTconnection object. Each WTconnection object has one or more WTsharegroups, which are used to group together a set of shared properties. By default, each WTconnection has a single WTsharegroup, which is referred to as the *root* WTsharegroup. Additional WTsharegroups can be created in a hierarchical arrangement under the root WTsharegroup.

Note: This chapter discusses only the client-side aspects of developing a multi-user World2World-compliant simulation. For more information on the server-side components of World2World, including how to install, configure, and start the World2World servers, see the “World2World User’s Guide.”

Sharing Properties

As described in the introduction, when a client shares a property, the events that are internally generated each time that a client makes changes to the property’s value cause those updates to be sent to the World2World Simulation Server. Once the update has been made on the Simulation Server, the Simulation Server sends the property update to all the other clients who are sharing that property. These updates will happen automatically, but can be overridden by a connection callback function (see page 21-23).

In order to share a property, you must specify the *WTsharegroup* under which the property will be grouped on a World2World Simulation Server. Each Simulation Server can have a hierarchical arrangement of WTsharegroups which are used to organize the properties stored on a Simulation Server. See *Sharegroups* on page 21-11 for more information about *WTsharegroups*. A single property can be shared under multiple sharegroups, though each Simulation Server will only retain a single copy of the shared property value.

Locked Properties

Shared properties can be locked by a client, causing the Simulation Server to prohibit any other user from removing the property from its sharegroup or from modifying the property's value until the client (which holds the lock) releases the lock.

Only one client can have a lock on a particular property at any given time. Be aware that properties are also affected by locks on sharegroups in that a sharegroup lock trickles down to the sharegroup's properties (as well as its child sharegroups and their properties). See page 21-12 for information on locked sharegroups.

You can immediately lock properties upon being shared through the *shareflags* parameter of the *WTproperty_share* function (see page 21-5), or you can lock existing shared properties with the *WTproperty_lock* function (see page 21-10).

Persistent Properties

When a property is shared, it can be flagged as being persistent through the *shareflags* parameter of the *WTproperty_share* function. By making a shared property persistent, you can ensure that the property is not removed from the Simulation Server even if all of the clients who are sharing the property have disconnected from the Simulation Server. If a property is not persistent, the property will be automatically removed from the Simulation Server when there are no remaining clients who are sharing this property.

The only way a persistent property can be deleted from the Simulation Server is if a client sharing the property makes an explicit call to *WTproperty_unshare* with the *forcedelete* parameter set to TRUE. Note that a shared property will be persistent if at least one client who has asked to share the property has specified that the property is to be persistent.

Update Frequencies

Shared properties have an update frequency, specified in seconds, which determines how often property value updates are queued up to be sent over the network. It is the *connection's* update rate (see page 21-23) that controls how often the queued updates are actually sent across the network.

The default update frequency for shared properties is 0.0 (WTSHAREDATA_UPDATEONSET), which means that a property change update is queued every time the property is changed. You can control the update frequency with the *WTproperty_setupupdatefreq* function (see page 21-8). If the shared property's update frequency is set to a negative number or WTSHAREDATA_NOUPDATE, property change updates will not be queued for the property and must therefore be sent manually with calls to *WTproperty_sendupdate* (see page 21-9).

Note that it is pointless (and inefficient) to queue property updates faster than they are actually being sent across the network. In fact, if you want to reduce network traffic, and you have shared properties whose update frequency is not critical to your simulation, you can queue property updates less often than they are being sent across the network, so that property updates aren't made more often than is really necessary. Take these factors into consideration when setting your properties' update frequencies and your connections' update rates.

Be aware that reducing the number of times that an update is sent across the network may require you to employ dead reckoning techniques to smooth the data updates on receiving clients. An example of dead reckoning is provided in the Samples directory of the directory in which you installed World2World.

Time Sensitive Properties

Shared properties can be flagged as time sensitive properties (see *WTproperty_settimesensitive* on page 21-9), if it is important to accurately track the time at which its value changed. All property update events sent by a Simulation Server contain a timestamp, though the timestamp of time sensitive properties is much more accurate than the timestamp associated with non-time sensitive properties as shown below:

- *For time sensitive properties* – the timestamp used is the time at which the client first modified the property value.
- *For non-time sensitive properties* – the timestamp used is the time at which the data update was made on the Simulation Server instead of the (earlier and more accurate) time at which the client first modified the property value.

A good example of when to use time sensitive properties is when those property updates are being dead reckoned by the receiving clients. By tracing the history of a property's update style, future updates can be predicted and used to smooth the updates to a property. Because time sensitive properties require clients to send some additional data (the

timestamp) to the Simulation Server, they increase network traffic somewhat. For that reason, shared properties are, by default, non-time sensitive.

WTbase – Working with Unsupported Object Types

There are several WTK object types for which the Object/Property/Event programming paradigm does not apply (see page 3-2 for a list of the supported object types and their pre-defined properties). Consequently, these unsupported types cannot contain properties, which allow for the generation of events, and the sharing of data across a network. The most significant WTK objects that fall into this category are geometries, polygons, vertices, and materials.

To extend the Object/Property/Event paradigm, create a WTbase object and add a property to that object representing the desired object attribute, and then synchronize changes to the WTK object data with changes to the WTbase version of the data. For more information on this procedure, see page 3-7. Also, an example of this procedure is supplied in the Samples directory of the directory in which you installed World2World.

Note: Each of the functions described in this chapter are synchronous unless otherwise noted.

Property Sharing Functions

WTproperty_share

```
FLAG WTproperty_share(  
    void *object  
    const char *proprname  
    WTsharegroup *shgrp  
    int shareflags);
```

This function shares an object's property under a sharegroup of a Simulation Server. If a connection is passed in as the *shgrp* parameter, this function will share the property under the root sharegroup of that connection. The *shareflags* argument determines how the property is shared. Valid values are:

WTSHAREFLAG_LOCK

Requests a property lock from the Simulation Server for the local client. If the local client obtains a lock on this property, no other clients will be able to modify this property's value. If this property is already locked by a remote client, the lock request will fail.

WTSHAREFLAG_PERSISTENT

Ensures that this property is not removed from the Simulation Server even if all of the clients who are sharing this property have disconnected from the Simulation Server. If a property is not persistent, the property will be automatically removed from the Simulation Server when there are no remaining clients who are sharing this property.

The only way a persistent property can be deleted from the Simulation Server is if a client sharing the property makes an explicit call to *WTproperty_unshare* with the *forcedelete* parameter set to TRUE.

By setting *shareflags* to (*WTSHAREFLAG_LOCK* | *WTSHAREFLAG_PERSISTENT*), i.e. bitwise or'ing both of the above options, you can request a lock on the property and also specify that the property is to be persistent. If *shareflags* is set to 0, the local client will not obtain a lock on the property and the property will not be persistent unless another client has specified that the shared property is to be persistent.

When an object's property is shared, the object's name must be unique among all the objects whose properties are shared on a Simulation Server. If the object whose property is being shared has a non-unique name, WorldToolKit will automatically modify the object's name to make it unique. To ensure that the name you use to refer to an object whose property has been shared, use *WTbase_getname* to retrieve the potentially modified name of the object. While an object's property is being shared, you cannot use *WTbase_setname* or any of the *WT*_setname* functions to modify that object's name.

This function is, by default, asynchronous unless the connection's mode is set to synchronous. See *WTconnection_setsynchronous* on page 21-30.

WTproperty_unshare

```
FLAG WTproperty_unshare(  
    void *object  
    const char *propname  
    WTsharegroup *shgrp,  
    FLAG forcedelete);
```

This function unshares an object's property from the specified sharegroup. If *shgrp* is NULL, the object's property will be unshared from all sharegroups.

If the *forcedelete* parameter is set to TRUE, this property will be removed from the Simulation Server's data tree even if the property is persistent. See *Persistent Properties* on page 21-3.

This function is, by default, asynchronous unless the connection's mode is set to synchronous. See *WTconnection_setsynchronous* on page 21-30.

WTproperty_issshared

```
FLAG WTproperty_issshared(  
    void *object  
    const char *propname);
```

This function returns TRUE if the object's property is currently shared, otherwise it returns FALSE.

WTproperty_numshares

```
int WTproperty_numshares(  
    void *object  
    const char *propname);
```

This function returns the number of times an object's property is shared.

WTproperty_getsharegroup

```
WTsharegroup* WTproperty_getsharegroup(  
    void *object  
    const char *propname  
    int nshare);
```

This function returns the *nshare*'th sharegroup in which an object's property is shared. *nshare* can range between 0 and (*WTproperty_numshares* - 1).

WTproperty_setupdatefreq

```
void WTproperty_setupdatefreq(  
    void *object  
    const char *propname  
    double frequency);
```

This function sets the frequency with which data updates for an object's property will be queued for transmission to a Simulation Server. The *frequency* is specified in seconds. By default an update is queued every time a property is set. For more information on property update frequencies, see page 21-3.

The *frequency* argument can be any value of type double, or one of the following constants:

<i>WTSHARED</i> DATA_UPDATEONSET	Property change updates are queued every time the property changes. This is equivalent to a value of 0.0.
<i>WTSHARED</i> DATA_NOUPDATE	Property change updates will not be queued for the property and must therefore be queued manually with calls to <i>WTproperty_sendupdate</i> .

WTproperty_getupdatefreq

```
double WTproperty_getupdatefreq(  
    void *object  
    const char *propname);
```

This function returns the frequency with which data updates for an object's property occur.

WTproperty_sendupdate

```
void WTproperty_sendupdate(  
    void *object  
    const char *propname);
```

This function manually queues an update for the specified object's property. If a property's update frequency is set to WTSHAREDDATA_NOUPDATE, this function must be called in order for an update to occur. See *WTproperty_setupupdatefreq* on page 21-8.

WTproperty_settimesensitive

```
void WTproperty_settimesensitive(  
    void *object  
    const char *propname  
    FLAG timesensitive);
```

This function makes an object's property time sensitive if the *timesensitive* parameter is set to TRUE and makes the object's property non-time sensitive if *timesensitive* is set to FALSE. Time sensitive properties are received by other clients with the timestamp from the initiating client, providing more accurate timing information to receiving clients. Properties are NOT time sensitive by default. For more information on time sensitivity, see page 21-4.

WTproperty_gettimesensitive

```
FLAG WTproperty_gettimesensitive(  
    void *object  
    const char *propname);
```

This function returns TRUE if the specified object's property is time sensitive, otherwise it returns FALSE.

WTproperty_lock

```
FLAG WTproperty_lock(  
    void *object  
    const char *propname);
```

This function requests a property lock for the local client so that other clients cannot modify the specified object's *propname* property. For more information on property locks, see page 21-3.

This function is, by default, asynchronous unless the connection's mode is set to synchronous. See *WTconnection_setsynchronous* on page 21-30.

WTproperty_unlock

```
FLAG WTproperty_unlock(  
    void *object  
    const char *propname);
```

This function requests that a property that is locked by the local client be unlocked. Once the local client has unlocked an object's property, other clients can modify the object's property or can themselves request a lock on the object's property. For more information on property locks, see page 21-3.

This function is, by default, asynchronous unless the connection's mode is set to synchronous. See *WTconnection_setsynchronous* on page 21-30.

WTproperty_islocked

```
unsigned int WTproperty_islocked(  
    void *object  
    const char *propname);
```

This function returns the id of the client who has a lock on the specified object's *propname* property, or 0 if the property isn't locked.

WTproperty_islockedby

```
FLAG WTproperty_islockedby(  
    void *object  
    const char *propname);
```

This function returns TRUE if the specified object's *propname* property is locked by the local client, otherwise it returns FALSE.

WTbase_unshare

```
FLAG WTbase_unshare(  
    void *object);
```

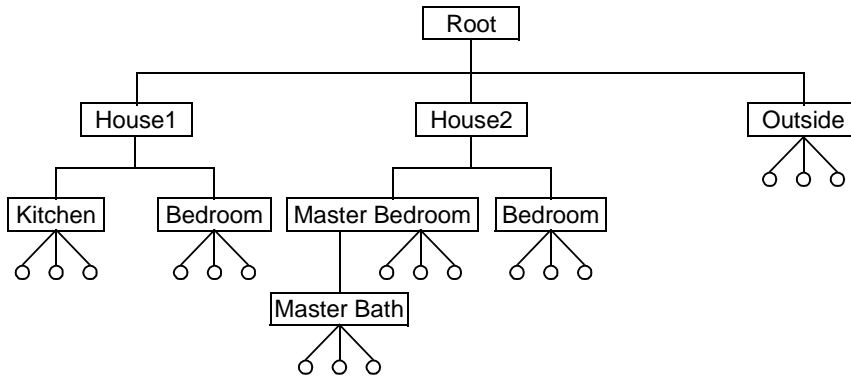
This function unshares all of an object's properties.

This function is, by default, asynchronous unless the connection's mode is set to synchronous. See *WTconnection_setsynchronous* on page 21-30.

Sharegroups

Sharegroups are container objects that are used to group one or more shared properties together on a World2World Simulation Server. Sharegroups can also contain child sharegroups. That is, they can have a parent/child relationship with other sharegroups so that a hierarchical arrangement of sharegroups can be created on a Simulation Server. Each Simulation Server (connection) has, by default, a root sharegroup. All other sharegroups created on that Simulation Server will be direct descendants (children) or indirect descendants of the root sharegroup. Sharegroups that are siblings (that is, they are children of a common parent sharegroup) must be uniquely named.

In the sample sharegroup data tree below, the Root, House1, and House2 sharegroups are placeholders. They contain no properties and exist only to add structure to the data tree.



Locked Sharegroups

Sharegroups can be locked by a client, causing the Simulation Server to prohibit all other users from adding, moving, or removing properties or child sharegroups within the locked sharegroup’s subtree, or from modifying the values of any properties contained within the locked sharegroup’s subtree until the client releases the lock. That is, the lock on a sharegroup is recursive, affecting not only the properties located directly within the sharegroup, but also any of its child sharegroups and their properties. Locks are granted on a first-come, first-served basis.

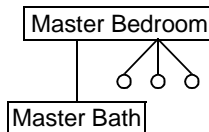
In the preceding example, if a client placed a lock on the House1 sharegroup, its Kitchen and Bedroom child sharegroups, and all the properties contained within them would also be locked.

You can immediately lock new sharegroups upon creation through the *shareflags* parameter of the *WTsharegroup_new* function (see page 21-15), or you can lock existing sharegroups with the *WTsharegroup_lock* function (see page 21-19).

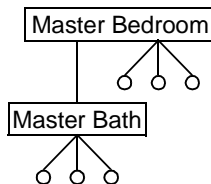
Registered Interest

While the Simulation Server keeps track of the data tree for all sharegroups and their properties, clients will only stay up-to-date with the sub-trees of sharegroups in which they have registered interest. Through connection callback events, each client receives notification of changes occurring to those sharegroups that they are interested in. (For more information on connection callbacks, see page 21-22.)

In the sample data tree on page 21-12, suppose a client registers interest in only the MasterBedroom sharegroup. Upon registering interest, the client will immediately receive notification of the current children (sub-sharegroups) and properties of MasterBedroom:



Unlike locks, registered interest is not recursive. So, the client does not receive notification of the properties of MasterBath. To receive notification of the properties of MasterBath, the client would have to also register interest in MasterBath:



If any other client participating in the multi-user simulation adds or removes any sharegroups or properties to or from MasterBedroom or MasterBath, this client will be notified of the change so that it can stay up-to-date.

Note that registering and unregistering interest does not affect the distribution of shared property updates. Suppose a client registers interest in Outside, shares the Sprinkler property, which belongs to the Outside sharegroup, and then unregisters interest in Outside. The client will still receive updates made to the Sprinkler property, but will not be notified if another client removes or adds a property or sharegroup to Outside.

You can immediately register interest in new sharegroups upon creation through the *shareflags* parameter of the *WTsharegroup_new* function (see page 21-15), or you can register interest in existing sharegroups with the *WTsharegroup_registerinterest* function (see page 21-20).

Note: By default, every client is NOT interested in the root sharegroup of a connection.

Persistent Sharegroups

Sharegroups, like properties, can be flagged as being persistent through the *shareflags* parameter of the *WTsharegroup_new* function. By making a sharegroup persistent, you can ensure that the sharegroup and its properties will not be removed from the Simulation Server, even if all of the clients who are interested in the sharegroup or who are sharing one or more of the sharegroup's properties have disconnected from the Simulation Server. Making a sharegroup persistent is equivalent to making each of the sharegroup's properties persistent. If a sharegroup is flagged as being persistent, any child sharegroups added to the persistent sharegroup will NOT also be persistent (unless the child sharegroups themselves are flagged as being persistent when they are created).

The only way a persistent sharegroup can be deleted from the Simulation Server is if the sharegroup is explicitly removed via a call to *WTsharegroup_delete* with the *forcedelete* parameter set to TRUE. A property of a persistent sharegroup for which no clients are interested in can be removed from the Simulation Server by calling *WTproperty_unshare* with the *forcedelete* parameter set to TRUE. Note that a sharegroup will be persistent if at least one client has specified that the sharegroup is to be persistent.

When a persistent property or persistent sharegroup is hierarchically below a non-persistent sharegroup in the Simulation Server's data tree, the sharegroups which are ancestors of the persistent property or sharegroup are, for all intents and purposes, also persistent. The sharegroups from the root sharegroup down to the parent sharegroup of the persistent property (or sharegroup) must be retained on the Simulation Server to maintain the structural integrity of the sharegroups and properties stored within the Simulation Server.

Sharegroup Functions

WTsharegroup_new

```
WTsharegroup* WTsharegroup_new(  
    const char *name  
    WTsharegroup *parent  
    int shareflags);
```

This function creates a new sharegroup named *name* as a child of the specified parent sharegroup. If a WTconnection is passed in as the parent parameter, then the root sharegroup of that connection will be used as the parent.

The *shareflags* parameter can be used to register interest in the sharegroup, lock the sharegroup, or make the sharegroup persistent immediately upon creation. The three options for this argument are:

- WTSHAREFLAG_INTERESTED
- WTSHAREFLAG_LOCK
- WTSHAREFLAG_PERSISTENT

The WTSHAREFLAG_INTERESTED option is used to register interest in the sharegroup so that the local client can be notified of any changes made to the sharegroup on the Simulation Server. A notification is sent whenever child sharegroups are added to or removed from the sharegroup, or if properties are added to or removed from the sharegroup. The change notifications make it possible for a client to stay up-to-date with the data stored on the Simulation Server.

The WTSHAREFLAG_LOCK option requests a lock from the Simulation Server so as to prohibit any other user from adding, moving, or removing properties or child sharegroups within the sharegroup's complete subtree, and to prohibit any other user from modifying the values of any properties contained within the sharegroup's subtree. If the lock is granted to the local client, remote clients will not be able to modify the sharegroup and its subtree until the local client releases the lock. That is, the lock on a sharegroup is recursive, affecting not only the properties located directly within the sharegroup, but also any of its child sharegroups and their properties.

The WTSHAREFLAG_PERSISTENT option causes the Simulation Server to make that sharegroup persistent. By making a sharegroup persistent, you can ensure that the

sharegroup and its properties will not be removed from the Simulation Server, even if all of the clients who are interested in the sharegroup or who are sharing one or more of the sharegroup's properties have disconnected from the Simulation Server. Making a sharegroup persistent is equivalent to making each of the sharegroup's properties persistent. If a sharegroup is flagged as being persistent, any child sharegroups added to the persistent sharegroup will NOT also be persistent (unless the child sharegroups themselves are flagged as being persistent when they are created).

The only way a persistent sharegroup can be deleted from the Simulation Server's data tree is through an explicit call to *WTsharegroup_delete* with the *forcedelete* parameter set to TRUE.

These three options can be used independently or can be combined by using the bitwise OR operator (|). For example, to register interest, request a sharegroup lock, and to make the sharegroup persistent, set the *shareflags* parameter to:

```
(WTSHAREFLAG_LOCK | WTSHAREFLAG_INTERESTED |  
WTSHAREFLAG_PERSISTENT).
```

Set *shareflags* to 0 if you don't want to register interest, lock, or make the sharegroup persistent. For more information on registered interest, see page 21-13. For more information on locked sharegroups, see page 21-12. For more information on sharegroup persistence, see page 21-14.

This function is, by default, asynchronous unless the connection's mode is set to synchronous. See *WTconnection_setsynchronous* on page 21-30.

WTsharegroup_delete

```
FLAG WTsharegroup_delete(  
    WTsharegroup *shgrp,  
    FLAG forcedelete);
```

This function deletes the specified sharegroup. Note that root sharegroups cannot be deleted. If the *forcedelete* parameter is set to TRUE, the sharegroup will be removed from the Simulation Server even if it is a persistent sharegroup. When a sharegroup is removed in this manner and there are other clients who have registered interest in this (now deleted) sharegroup, notification of the sharegroup's removal will be sent to each interested client.

This function is, by default, asynchronous unless the connection's mode is set to synchronous. See *WTconnection_setsynchronous* on page 21-30.

WTsharegroup_share

```
FLAG WTsharegroup_share(  
    WTsharegroup *group,  
    int shareflags);
```

Attempts to share an unshared WTsharegroup. If an asynchronous attempt to create a WTsharegroup fails (*WTsharegroup_new* returns NULL if a synchronous attempt fails) you can retry the share with this function. In asynchronous mode, this function will return TRUE if the request was sent successfully, but doesn't know if the request will actually succeed.

The *shareflags* parameter can be used to register interest in the sharegroup, lock the sharegroup, or make the sharegroup persistent immediately upon creation. For a description of the available options, see *WTsharegroup_new* on page 21-15.

WTsharegroup_issshared

```
FLAG WTsharegroup_issshared(  
    WTsharegroup *group);
```

This function returns the share status of the WTsharegroup *group*. Possible return values are:

0	Share failed
1	Shared
2	Share in progress

WTsharegroup_setdata

```
void WTsharegroup_setdata(  
    WTsharegroup *shgrp  
    void *data);
```

This function sets the user-defined data field on a sharegroup.

WTsharegroup_getdata

```
void* WTsharegroup_getdata(  
    WTsharegroup *shgrp
```

This function returns the user-defined data field on a sharegroup.

WTsharegroup_getconnection

```
WTconnection* WTsharegroup_getconnection(  
    WTsharegroup *shgrp);
```

This function returns the WTconnection on which the specified sharegroup exists.

WTsharegroup_print

```
void WTsharegroup_print(  
    WTsharegroup *shgrp  
    FLAG children  
    FLAG properties);
```

This function prints information about the specified sharegroup. If the *children* parameter is set to TRUE, information about the sharegroup's subtree will also be printed. If the *properties* parameter is set to TRUE, information about the properties of the sharegroup(s) will be displayed.

WTsharegroup_getname

```
char* WTsharegroup_getname(  
    WTsharegroup *shgrp);
```

This function returns the name of the specified sharegroup.

WTsharegroup_lock

```
FLAG WTsharegroup_lock(  
    WTsharegroup *shgrp);
```

This function requests a sharegroup lock for the local client so that other clients cannot modify this sharegroup's subtree. For more information on locked sharegroups, see page 21-12.

This function is, by default, asynchronous unless the connection's mode is set to synchronous. See *WTconnection_setsynchronous* on page 21-30.

WTsharegroup_unlock

```
FLAG WTsharegroup_unlock(  
    WTsharegroup *shgrp);
```

This function requests that a sharegroup that is locked by the local client be unlocked.

This function is, by default, asynchronous unless the connection's mode is set to synchronous. See *WTconnection_setsynchronous* on page 21-30.

WTsharegroup_islocked

```
unsigned int WTsharegroup_islocked(  
    WTsharegroup *shgrp);
```

This function returns the id of the client which has a lock on the specified sharegroup, or 0 if the sharegroup isn't locked.

WTsharegroup_islockedbyme

```
FLAG WTsharegroup_islockedbyme(  
    WTsharegroup *shgrp);
```

This function returns TRUE if the specified sharegroup is locked by the local client, otherwise it returns FALSE.

WTsharegroup_registerinterest

```
void WTsharegroup_registerinterest(  
    WTsharegroup *shgrp  
    FLAG interested);
```

This function registers interest in the specified sharegroup for the local client if the *interested* parameter is set to TRUE and unregisters interest in the specified sharegroup for the local client if the *interested* parameter is set to FALSE. For more information about registering interest, see page 21-13.

This function is, by default, asynchronous unless the connection's mode is set to synchronous. See *WTconnection_setsynchronous* on page 21-30.

WTsharegroup_getparent

```
WTsharegroup* WTsharegroup_getparent(  
    WTsharegroup *shgrp);
```

This function returns the parent sharegroup of the specified sharegroup. If the specified sharegroup is a root sharegroup, NULL will be returned.

WTsharegroup_numchildren

```
int WTsharegroup_numchildren(  
    WTsharegroup *shgrp);
```

This function returns the number of sharegroups that are direct children of the specified sharegroup.

WTsharegroup_getchild

```
WTsharegroup* WTsharegroup_getchild(  
    WTsharegroup *shgrp  
    int childnum);
```

This function returns the *childnum*'th child sharegroup of the specified sharegroup.

WTsharegroup_findchildbyname

```
WTsharegroup *WTsharegroup_findchildbyname(  
    WTsharegroup *group,  
    const char *name);
```

Finds an immediate child of *group* matching *name*.

WTsharegroup_numproperties

```
int WTsharegroup_numproperties(  
    WTsharegroup *shgrp);
```

This function returns the number of properties of the specified sharegroup.

WTsharegroup_getproperty

```
char* WTsharegroup_getproperty(  
    WTsharegroup *shgrp  
    int propertynum  
    void **object);
```

This function returns the name of the *propertynum*'th property of a sharegroup. Because properties are specified with an object and a property name, pass in a pointer to a void* to retrieve the object pointer for that property.

Example usage of WTsharegroup_getproperty:

```
void *object;  
char *propname;  
propname = WTsharegroup_getproperty(shgrp, 0, &object);  
if(propname) {  
    printf("The object/property is %s:%s\n", WTbase_getname(object),  
        propname);  
}
```

Network Connections

When a client application starts it will connect to a server where the application's data is to be shared. This process begins with a login call to a World2World Server Manager at a specified port, which determines what simulation this client will be entering. The Server Manager then proceeds to direct the client to the Simulation Server that has been designated to host that particular simulation. Once connected to the Simulation Server, the client can begin creating sharegroups (see page 21-11) and sharing properties (see page 21-2).

To better understand how this process works, see the example provided in the *World2World User's Guide*, Chapter 4, "Starting and Ending World2World." As the developer of the simulation, you only need to worry about the host name of the machine on which the Server Manager is located and determining a unique, unused port on that machine that you can associate with your simulation. The system administrator will take care of configuring the World2World servers to ensure that the clients of your simulation are connected to the appropriate Simulation Server.

The login process involves calling the *WTconnection_new* function (see page 21-26) followed by a call to *WTconnection_connect* (see page 21-28). Because the login process is a synchronous process, *WTconnection_connect* will not return until the client has either connected to a Simulation Server or timed out trying. Once a client has connected to a Simulation Server, it will be assigned a user name and id. The user name and id for every client in the same simulation (that is, those clients that connect to the same host name and port, specified by the *WTconnection_new* function) as the local client can be obtained through the *WTconnection* object which is returned from the *WTconnection_new* function.

Synchronous and Asynchronous Connections

A connection operates in one of two modes, *synchronous* or *asynchronous*, specified by the *WTconnection_setsynchronous* function (see page 21-30). By default, a connection operates in asynchronous mode. This mode allows for the client application to continue executing while requests are being fulfilled by the Simulation Server. For example, the client can request a property to be shared and continue interacting with the simulation, being notified at a later time that the property was shared. In synchronous mode, the connection will wait until that request is fulfilled before allowing the client simulation to continue.

Update Rates

Connections have an update rate, which determines the updates per second for the connection. This is the number of times per second that the client will send packets to the Simulation Server and the number of times per second that the Simulation Server will send packets to the client. The lower the update rate, the lower the packet traffic over the network.

For modems, or other low-bandwidth mediums, the connection's update rate should be as low as possible. The default behavior is for the connection to match the client frame-rate (*WTuniverse_getframerate*). You can control the connection's update rate with the *WTconnection_setupdaterate* function (see page 21-31). Setting the connection update frequency to 0 will set the connection's update rate back to the default client frame-rate matching mode. Any other positive value will be used to override the matching mode, and explicitly assign an update rate.

Be aware that reducing the connection's update rate may require you to employ dead reckoning techniques to smooth the data updates on receiving clients. An example of dead reckoning is provided in the Samples directory of the directory in which you installed World2World.

Connection Callbacks

Just as changes to property values generate events that you can react to, changes to Simulation Server data (such as opening new connections, locking sharegroups, etc.) also generate events that you can react to. These are known as *connection events*. The actions that result in the generation of connection events include:

- opening/closing connections
- creating/removing sharegroups
- locking/unlocking sharegroups
- sharing/unsharing properties
- locking/unlocking properties
- adding/removing users
- updating shared property values

Notice that property value changes result in the generation of both property change events (see page 3-23) and connection events.

You can react to connection events by adding *callback functions* to connection objects. These functions are used by the client to understand and react to the Simulation Server's state changes. For example, when a new user enters the Simulation Server, the client might want to create a geometry node to represent that user.

Callback functions contain a *time* parameter, which indicates the time at which the event was generated. This time of event information can be used in dead reckoning algorithms to predict the future changes to a property over a specified time frame.

A connection callback function takes the form:

```
FLAG WTconncb(  
    WTconnection * conn,    ← connection generating the event  
    WTconnevent event,    ← event generated  
    void *data1,          ← see chart below  
    void *data2,          ← see chart below  
    double time);        ← time of the event
```

The table below lists the possible connection events. With the exception of WTUSER_NEW and WTUSER_DEL, the event types fall into 2 categories: WTLOCAL and WTREMOTE. WTLOCAL events are those events that have occurred due to a request by the local client, whereas WTREMOTE events occur due to requests made by non-local clients. WTUSER_NEW and WTUSER_DEL events are fired when anyone (local or remote) enters or leaves the same entypoint on the server as this connection.

The return value of the callback function is used only for WTREMOTE_UPDATE events. For WTREMOTE_UPDATE events, the callback should return TRUE to allow the network to modify the property value, or FALSE to disallow it.

WTconnevent	data1	data2
WTLOCAL_OPENCONN	NULL	NULL
WTLOCAL_CLOSECONN	NULL	NULL

WTconnevent	data1	data2
WTLOCAL_NEWSHGRP	WTsharegroup *sharegrp	status (0=fail, 1=new, 2=existing)
WTLOCAL_DELSHGRP	WTsharegroup *sharegrp	status (0=fail, 1=success)
WTLOCAL_LOCKSHGRP	WTsharegroup *sharegrp	status (0=fail, >0=userid)
WTLOCAL_UNLOCKSHGRP	WTsharegroup *sharegrp	status (0=fail, >0=userid)
WTLOCAL_SHAREPROP	WTSHAREINFO *info	status (0=fail, 1=new, 2=existing)
WTLOCAL_UNSHAREPROP	WTSHAREINFO *info	status (0=fail, 1=success)
WTLOCAL_LOCKPROP	WTSHAREINFO *info	status (0=fail, >0=userid)
WTLOCAL_UNLOCKPROP	WTSHAREINFO *info	status (0=fail, >0=userid)
WTLOCAL_ENUMSHGRPDONE	WTsharegroup *sharegrp	unsigned int enumtreedid
WTREMOTE_UPDATE	WTSHAREINFO *info	value
WTREMOTE_UPDATEAT	WTSHAREINFO *info	value
WTREMOTE_NEWSHGRP	WTsharegroup *parent	char *childname
WTREMOTE_DELSHGRP	WTsharegroup *parent	char *childname
WTREMOTE_LOCKSHGRP	WTsharegroup *sharegrp	status (0=fail, >0=userid)
WTREMOTE_UNLOCKSHGRP	WTsharegroup *sharegrp	status (0=fail, >0=userid)
WTREMOTE_SHAREPROP	WTsharegroup *parent	WTSHAREINFO *info
WTREMOTE_UNSHAREPROP	WTsharegroup *parent	WTSHAREINFO *info
WTREMOTE_LOCKPROP	WTSHAREINFO *info	status (0=fail, >0=userid)

WTconnevent	data1	data2
WTREMOTE_UNLOCKPROP	WTSHAREINFO *info	status (0=fail, >0=userid)
WTUSER_NEW	unsigned int userid	char *username
WTUSER_DEL	unsigned int userid	char *username

```
typedef struct _WTSHAREINFO {
    WTsharegroup *sharegroup; (NULL for WT*_LOCKPROP, WT*_UNLOCKPROP,
                               WTREMOTE_UPDATE, WTREMOTE_UPDATEAT)
    int objecttype;
    char *objectname;
    char *proptype;
    int datatype;
    void *object; (NULL for WTREMOTE_SHAREPROP and WTREMOTE_UNSHAREPROP)
} WTSHAREINFO;
```

Connection Functions

All of the following WTconnection functions will accept NULL as the WTconnection pointer parameter and tells WTK to use the first WTconnection object on the universe's list of connections. This is useful if there is only one connection in the application.

WTconnection_new

```
WTconnection* WTconnection_new(
    const char *host
    unsigned short port
    const char *username
    const char *passwd);
```

This function defines a new connection to the World2World Server Manager at a specified port. To connect this connection, call *WTconnection_connect* (see page 21-28). Based on the port number, the Server Manager will direct the client to the appropriate Simulation Server as configured (see the *World2World User's Guide* for details on this process). This

call is synchronous and will not return until a connection has been made, refused, or timed out. The password argument is not currently used.

Note: When choosing a port number for the Server Manager, keep in mind that ports 0 to 1024 are generally used by your operating system. You will probably want to specify a number between 1025 and 32,000. Check with your system administrator to determine whether certain ports are available.

WTconnection_delete

```
void WTconnection_delete(  
    WTconnection *c);
```

This function disconnects and deletes a connection.

WTconnection_setdata

```
void WTconnection_setdata(  
    WTconnection *c  
    void *data);
```

This function sets the user-defined data field on a connection.

WTconnection_getdata

```
void* WTconnection_getdata(  
    WTconnection *c);
```

This function returns the user-defined data field on a connection.

WTuniverse_getconnections

```
WTconnection* WTuniverse_getconnections(  
    void);
```

This function returns a pointer to the first connection in the universe's list of connections for the local client. Use *WTconnection_next* to iterate through the list of connections.

WTconnection_next

```
WTconnection* WTconnection_next(  
    WTconnection *c);
```

This function returns the next connection in the local client's list of connections. Use *WTuniverse_getconnections* to retrieve the first connection in the list.

WTuniverse_deleteconnections

```
void WTuniverse_deleteconnections(  
    void);
```

This function deletes all connections made by the local client.

WTconnection_connect

```
FLAG WTconnection_connect(  
    WTconnection *c);
```

This function attempts to connect to the World2World Server Manager and port represented by the specified connection object. Returns TRUE if the connection was successful.

WTconnection_disconnect

```
FLAG WTconnection_disconnect(  
    WTconnection *c);
```

This function disconnects from the World2World Server Manager and port represented by the specified connection object.

WTconnection_getmyid

```
unsigned int WTconnection_getmyid(  
    WTconnection *c);
```

This function returns the local client's id for the specified connection.

WTconnection_getmyname

```
const char* WTconnection_getmyname(  
    WTconnection *c);
```

This function returns the local client's name for the specified connection.

WTconnection_getstatus

```
int WTconnection_getstatus(  
    WTconnection *c);
```

This function returns the current status of a connection. Possible return values are `WTCONNSTATUS_CONNECTED`, `WTCONNSTATUS_DISCONNECTED`.

WTconnection_print

```
void WTconnection_print(  
    WTconnection *c);
```

This function prints information about a connection such as status, latency, average latency, local user id, local user name, list of all users, and the sharegroup/property hierarchy.

WTconnection_update

```
void WTconnection_update(  
    WTconnection *c);
```

This function updates a connection (send and receive packets). This function must be called if the client does not call `WTuniverse_go` or `WTuniverse_go1`.

WTuniverse_updateconnections

```
void WTuniverse_updateconnections(  
    void);
```

This function updates all connections of the local client (send and receive packets). This function must be called if the client does not call `WTuniverse_go` or `WTuniverse_go1`.

WTconnection_synch

```
FLAG WTconnection_synch(  
    WTconnection *c);
```

This function waits for pending requests to be fulfilled (returns FALSE if time out occurs).

WTconnection_getlatency

```
double WTconnection_getlatency(  
    WTconnection *c);
```

This function returns the latency associated with a connection. That is, the amount of time it takes for packets to be transmitted to or from a World2World Simulation Server.

WTconnection_getclockdiff

```
double WTconnection_getclockdiff(  
    WTconnection *c);
```

This function returns the time, in seconds, that the local and World2World Simulation Server clocks differ.

WTconnection_setsynchronous

```
void WTconnection_setsynchronous(  
    WTconnection *c  
    FLAG synchronous);
```

If the synchronous parameter is TRUE, this function sets the operating mode of a connection to synchronous. If FALSE, the operating mode will be asynchronous. By default, the operating mode of a connection is asynchronous. For more information on the synchronous and asynchronous modes of a connection, see page 21-22.

WTconnection_issynchronous

```
FLAG WTconnection_issynchronous(  
    WTconnection *c);
```

This function returns TRUE if the operating mode of a connection is synchronous, returns FALSE otherwise.

WTconnection_setupdaterate

```
void WTconnection_setupdaterate(  
    WTconnection *c  
    unsigned short fps);
```

This function sets the rate at which data packets are sent over the connection. The *fps* parameter indicates how many times per second that data packets should be sent. Data packets include such data as property value changes, lock/unlock requests, share/unshare requests, etc. By default, a connection's update rate is set to match the client's frame rate. If a *fps* of 0 is specified, the connection's update rate will be set to match the client's frame rate. For more information about a connection's update rate, see page 21-23.

WTconnection_getupdaterate

```
unsigned short WTconnection_getupdaterate(  
    WTconnection *c);
```

This function returns the the number of times per second that data packets are sent for a connection. Data packets include such data as property value changes, lock/unlock requests, share/unshare requests, etc.

WTconnection_addcallback

```
void WTconnection_addcallback(  
    WTconnection *c  
    WTconncb cb);
```

This function adds a callback to a connection. For more information on connection callbacks, see page 21-23.

WTconnection_removecallback

```
void WTconnection_removecallback(  
    WTconnection *c  
    WTconncb cb);
```

This function removes a callback from a connection.

WTconnection_numcallbacks

```
int WTconnection_numcallbacks(  
    WTconnection *c);
```

This function returns the number of callbacks on a connection.

WTconnection_getcallback

```
WTconncb WTconnection_getcallback(  
    WTconnection *c  
    int conncbnum);
```

This function returns a numbered callback on a connection.

WTconnection_getroot

```
WTsharegroup* WTconnection_getroot(  
    WTconnection *c);
```

This function returns the root sharegroup for a connection.

WTconnection_numusers

```
unsigned int WTconnection_numusers(  
    WTconnection *c);
```

This function returns the number of users on a connection.

WTconnection_getuserid

```
unsigned int WTconnection_getuserid(  
    WTconnection *c  
    unsigned int usernum);
```

This function returns a numbered user's id.

WTconnection_getusername

```
const char* WTconnection_getusername(  
    WTconnection *c  
    unsigned int usernum);
```

This function returns a numbered user's name.

WTconnection_getuseridbyname

```
unsigned int WTconnection_getuseridbyname(  
    WTconnection *c  
    const char*username);
```

Given a user name, this function returns a user's id.

WTconnection_getusernamebyid

```
const char* WTconnection_getusernamebyid(  
    WTconnection *c  
    unsigned int userid);
```

Given a user id, this function returns a user's name.

Enumeration

Enumeration is the process of a client requesting and receiving a copy, or snapshot, of the Simulation Server’s data tree or sub-tree, i.e. its sharegroups and properties. An enumeration can be requested for any WTsharegroup object on the client. The result of an enumeration is a data tree consisting of a hierarchical arrangement of WTbase objects which represent the data on the Simulation Server. See *Example of an Enumeration Tree* below for more details.

Each enumeration tree is created by calling *WTsharegroup_enumerate*. Each enumeration tree is stored on the local client machine and is accessed via the WTconnection object which corresponds with the Simulation Server whose data has been enumerated. Since each connection can store an unlimited number of enumeration trees simultaneously, each enumeration tree is assigned an id number. The enumeration tree id along with the WTconnection pointer can be passed into the *WTconnection_getenumtreebyid* function to return a pointer to the *root* WTbase object of the corresponding enumeration tree. The WTbase_* functions such as *WTbase_numchildren* and *WTbase_getchild* can be used to access each element of the enumeration tree.

Example of an Enumeration Tree

WTbase objects	ID property	Value property
W2WEnumTree	enumtreeid=22	
ROOT	shgrpид=0	
users	shgrpид=92	
USR_15	shgrpид=93	
Darts	shgrpид=94	
Dart15_0:Translation	dataid=107	value=-54.01,-38.45,195.00
Dart15_0:Rotation	dataid=108	value=0.08,0.09,0.52,0.23
Dart15_1:Translation	dataid=109	value=-53.79,-38.3,195.00
Dart15_1:Rotation	dataid=110	value=0.08,0.09,0.09,0.93

WTbase objects	ID property	Value property
USR_15:msg	dataid=95	value=Hi, want to play?
USR_15:url	dataid=96	value=http://www.s8.com/bill.wrl
USR_15:Rotation	dataid=97	value=0.00,0.00,0.00,1.00
USR_15:Translation	dataid=98	value=0.00,-75.00,0.00
USR_16	shgrpId=99	
Darts	shgrpId=100	
Dart16_0:Translation	dataid=105	value=0.00,0.00,195.00
Dart16_0:Rotation	dataid=106	value=0.22,0.00,0.00,0.97
USR_16:msg	dataid=101	value=Sure, hold on...
USR_16:url	dataid=102	value=http://www.s8.com/rog.wrl
USR_16:Rotation	dataid=103	value=0.00,0.00,0.00,1.00
USR_16:Translation	dataid=104	value=0.00,-50.00,0.00

The W2WEnumTree WTbase object is the WTbase returned by *WTconnection_getenumtree* and *WTconnection_getenumtreebyid*. This object has one property, *enumtreeid*, of type WTUINT which contains the enumtreeid for the WTbase tree it contains. The only child of the W2WEnumTree object is a WTbase representing the WTsharegroup that was enumerated with a call to *WTsharegroup_enumerate*; we refer to it as the root of the enumeration. Under the root of the enumeration is a hierarchy of WTbase objects representing the sharegroups and properties (if requested) contained on the server.

Each WTbase object has an ID property, either *shgrpId* or *dataid* of type WTUINT. If the object has a *shgrpId* property, it represents a WTsharegroup object, and the name of the WTbase object corresponds to the name of the WTsharegroup. If the object has a *dataid* property, it represents a shared property, and the name of the WTbase object corresponds to the name of the object, followed by the name of the property (separated by a colon). The *value* property on these WTbase objects is the value of the property on the server when it was enumerated. The datatype of this *value* property corresponds to the datatype of the property on the server.

WTsharegroup_enumerate(

```
unsigned int WTsharegroup_enumerate(  
    WTsharegroup *parent,  
    FLAG recursive,  
    FLAG properties);
```

This function requests an enumeration of the specified *parent* sharegroup so that the local client has a copy or snapshot of the current state of the data tree on the Simulation Server. If the *recursive* parameter is set to TRUE, the specified sharegroup's sub-tree will also be enumerated. If the *properties* parameter is set to TRUE, the enumeration tree will also contain information about the properties contained within the Simulation Server's data tree. The return value of this function is the enumeration tree's id. Use *WTconnection_getenumtreebyid* to obtain a pointer to the *root* WTbase object of the enumeration tree.

If the WTconnection that the *parent* sharegroup belongs to is in synchronous mode, this function will not return until the enumeration tree has been created or until the connection times out.

This function is, by default, asynchronous unless the connection's mode is set to synchronous. See *WTconnection_setsynchronous* on page 21-30.

WTconnection_deleteallenumtrees

```
void WTconnection_deleteallenumtrees(  
    WTconnection *connection);
```

This function deletes all of the enumeration trees that are currently stored with the specified WTconnection object.

WTconnection_deleteenumtreebyid

```
void WTconnection_deleteenumtreebyid(  
    WTconnection *connection,  
    unsigned int enumid);
```

This function deletes the specified enumeration tree from a WTconnection object.

WTconnection_getenumtreebyid

```
WTbase *WTconnection_getenumtreebyid(  
    WTconnection *connection,  
    unsigned int enumid);
```

This function returns the *root* WTbase object of the enumeration tree whose enumeration tree id is *enumid*. The *root* WTbase object represents the Simulation Server's sharegroup which was enumerated via a call to *WTsharegroup_enumerate*. Children of this WTbase object are other WTbase objects. The entire WTbase hierarchy represents a snapshot of the Simulation Server's data sub-tree.

WTconnection_numenumtrees

```
unsigned int WTconnection_numenumtrees(  
    WTconnection *connection);
```

This function returns the number of enumeration trees currently stored with a WTconnection object.

WTconnection_getenumtree

```
WTbase *WTconnection_getenumtree(  
    WTconnection *connection,  
    unsigned int nenumtree);
```

This function returns the *root* WTbase object for an enumeration tree of the specified connection by index. The *root* WTbase object represents the Simulation Server's sharegroup which was enumerated via a call to *WTsharegroup_enumerate*. Children of this WTbase object are other WTbase objects. The entire WTbase hierarchy represents a snapshot of the Simulation Server's data sub-tree.

WTconnection_getenumtreeid

```
unsigned int WTconnection_getenumtreeid(  
    WTconnection *connection,  
    unsigned int nenumtree);
```

This function returns the enumeration tree id for an enumeration tree of the specified connection by index.

WorldToolKit and World Up Compatible Properties

If you are developing multi-user applications with both WorldToolKit and World Up, you may be interested to know that some of the pre-defined properties of WorldToolKit objects are directly compatible with properties of World Up objects. The table below lists the pre-defined WorldToolKit property in the left column and the corresponding World Up property in the right column.

WorldToolKit	World Up
WTNODE_BOUNDINGBOX	Bounding Box
WTNODE_CHILDREN	Children
WTNODE_ENABLED	Enabled
WTNODE_ROTATION	Rotation
WTNODE_TRANSLATION	Translation
WTANCHOR_LOCATION	URL
WTFOG_COLOR	Color
WTFOG_LINEARSTART	Linear Start
WTFOG_MODE	Is Exponential
WTFOG_RANGE	Range

WorldToolKit	World Up
WTINLINE_LOCATION	Location
WTLIGHT_ANGLE	Angle
WTLIGHT_ATTENUATION	Attenuation
WTLIGHT_EXPONENT	Exponent
WTLIGHT_INTENSITY	Intensity
WTLOD_RANGE	Ranges
WTSWITCH_WHICHCHILD	Active Child
WTVIEWPOINT_ORIENTATION	Orientation
WTVIEWPOINT_PARALLAX	Parallax
WTVIEWPOINT_POSITION	Position
WTWINDOW_BGRGB	Background Color
WTWINDOW_HITHER	Hither Clipping
WTWINDOW_ROOTNODE	Root Node
WTWINDOW_VIEWPOINT	Viewpoint
WTWINDOW_VIEWANGLE	View Angle
WTWINDOW_YON	Yon Clipping
WTSENSOR_ANGULARRATE	Angular Rate
WTSENSOR_MISCDATA	Misc Data
WTSENSOR_ROTATION	Rotation
WTSENSOR_SENSITIVITY	Sensitivity
WTSENSOR_TRANSLATION	Translation
WTPATH_DIRECTION	Forward
WTPATH_PLAYING	Playing
WTPATH_PLAYSPEED	Speed

WorldToolKit	World Up
WTPATH_RECORDING	Recording
WTPATH_RECORDLINK	Record From

Multicast Networking

Introduction to Networking in WTK

WTK's networking capability enables you to build applications that can asynchronously communicate over an Ethernet between several PCs and UNIX workstations. This allows distributed simulations to be created where a mixture of PCs and UNIX workstations support a single simulation. Note that additional licenses are required to use this feature.

This functionality is provided in the form of calls such as *WTnet_open*, *WTnet_close*, *WTnet_additem* and *WTnet_removeitem*. To help you develop an application using these calls, WTK provides a demo program called "netdemo.c" which allows multiple users to share the same virtual world. In this demo, each user can see the other people in the world. Graphical objects (called "avatars") represent the other people in the simulation and are located at the other users' viewpoints. The demo can also measure the amount of "network lag" in the simulation at any point in time. This demo is just an example of the many possibilities enabled by these functions. The following terminology will be used when discussing distributed simulations:

Local	The objects residing on the local computer.
Remote	The objects residing on simulation hardware other than that of the local computer.
Private	Objects that only exist on the local computer. They are not part of the distributed simulation.
Public	Objects that are part of the distributed simulation. They may be controlled from a single computer, but their state is updated on all computers participating in the simulation.

WTK network applications share a common API so that a single application can be run on both PC and UNIX platforms without modification. However, the procedure for configuring your Windows system to use the network capability varies greatly depending on the platform. Please consult your hardware guide for more information on this subject.

Figure 22-1 illustrates the WTK networking layers that are discussed in this chapter.

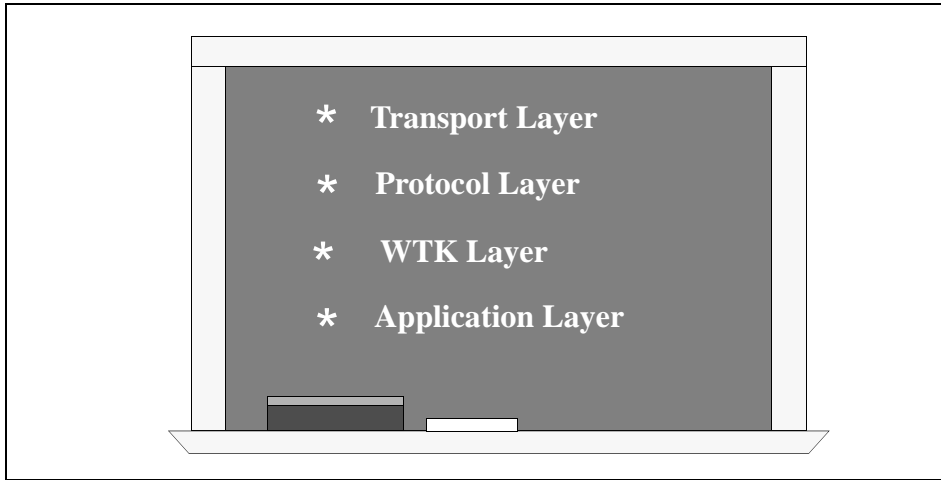


Figure 22-1: WTK networking layers.

How the Transport Layer Works

At the transport layer, multiple PCs or UNIX workstations have to be connected with standard ethernet hardware and cabling. WTK supports only the DEC/Intel/Xerox (DIX or referred to as the Bluebook Ethernet) standard. This is what almost all workstations use. WTK does not support 802.3/802.5, or Token Ring.

PCs can either have their own physical and independent network, or they can be strung off the same Ethernet line used for the UNIX workstations. Byte-ordering problems between PCs and workstations are handled invisibly by WTK (though there are some user considerations). You must make sure that your PC network is configured in the proper Ethernet fashion as there are many different methods of fashioning networks for PC systems. Consult your system administrator if you have any questions or prior to connecting any new machines to a pre-existing network.

How the Protocol Layer Works

WTK's networking capability is built upon IP and UDP guidelines. This means that you can use this capability on top of pre-existing networks without causing problems for the entire network. This also means that it is possible to multicast messages onto the Internet for geographically disbursed simulations.

Packets from a particular machine are multicast to the network (UDP) instead of communicated point-to-point (TCP). Multicasting allows for communication to other sub-nets based upon proper routing information. Multicasting also imposes less of an impact on the network than broadcasting.

How the WorldToolKit Layer Works

Networked communications are initialized by a call to *WTnet_open*. This call does not check to see if the network is up and running; it assumes this is the case. Communications with other machines on the network is established by using a valid multicast group address (Internet class D address). These range from 224.0.0.0 to 239.255.255.255. In addition to the group address, simulation machines will only communicate with other machines that share identical port addresses. This information is passed to WTK through the *WTnet_open* call.

Once connectivity has been achieved, application-specific information is passed between machines using discrete message items that are assembled into valid UDP packets. These message items are assembled by the application using the *WTnet_additem* function, and sent out onto the network automatically by WTK. (See *WTnet_flush* on page 22-13 for information on sending messages outside of the simulation loop.)

The receiving application processes these message items by stripping them out of the packet using the *WTnet_removeitem* function.

How the Application Layer Works

Communication is limited to the transmittal and receipt of specific message items that are multicast to the network. While WTK provides the substrate for communications to occur, your application (and not WTK) defines what information these items contain. In other words, it is the application's responsibility to make sense of these pieces of information and to do something with them.

In the *netdemo.c* program, each machine sends individual message items that, for example, describe the position and orientation of the local viewer (an example of a public object), or describe an application specific event. Each machine also checks whether it has received any message items from other machines. If so, it extracts them using the *WTnet_removeitem* function (see page 22-11). This information is then used to update the state of the local simulation. Because each machine is running an identical simulation, only changes in the state of a particular machine need to be passed to other machines. This minimizes the information passed across the network.

Sample Transaction

Local Machine

In this example, the local user changes their viewpoint, which moves their virtual body to a different position and orientation. This avatar (a graphical object representing the local user) is an example of a local, public object.

When the local machine enters its user-defined action function (see *WTuniverse_setactions* on page 2-12), an application function (*net_actions*) executes. This function uses *WTnet_additem* to send the current position and orientation of the local viewer.

In the same *net_actions* routine on the local machine, the *WTnet_next* function is called to see whether any valid message items have arrived from a remote machine. If a message item has been received, it is extracted and decoded using the *WTnet_removeitem* function. For example, these message items might describe the position and orientation of the remote public objects.

Remote Machines

When each of the remote machines reach their *net_actions* function, they also send messages that describe the position and orientation of their local public objects (their viewpoints in this case). Each remote machine then processes the message items received from other machines.

When a message item containing the new position and orientation information for a remote user's viewpoint is received, this information is used to move the avatar representing the remote user to the updated location.

This cycle of sending and receiving application-specific messages is repeated, resulting in a distributed simulation.

Message Latency

The time it takes for a change in the distributed simulation (such as changing the appearance of a remote object) to propagate through the entire simulation depends on these factors:

- The number of machines in the simulation
- The current packet traffic
- The frame-rate of the slowest machine in the network
- The network's communication bandwidth and latency

In a worst case scenario, this latency could be upwards of several seconds. In a best case scenario, you will experience at least a two-frame propagation delay (one frame to send, one frame to receive, and process/display). Based on this, you can calculate the actual latency or propagation delay in your particular application.

Byte Ordering

Data is transmitted over the network in “net” order (big-endian byte order). This is not of concern to the user unless data is being transferred to different types of machines. The byte swapping is handled internally by WTK as long as the data type is known. However, structures containing members that are less than four bytes in size must be declared differently on different platforms. This is because different platforms handle byte-alignment and padding differently. (See the discussion about message types in *Network Functions* on page 22-7.) Here is an example of declaring such a structure:

```
#if WIN32 /* Intel and DEC order */
typedef struct xxyyzz {
    char cc;
    char bb;
    short aa;
    float dd;
} xxyyzz;
#else /* SGI or SUN order */
typedef struct xxyyzz {
    short aa;
    char bb;
    char cc;
    float dd;
} xxyyzz;
#endif /* WIN32 */
```

WTK’s *WTpq* structure, which consists of arrays of floating point values, is an example of a structure where the storage order is transparently handled between machines, because the structure members are all 4-byte aligned. The examples under the *WTnet_additem* and *WTnet_removeitem* functions illustrate how to send and receive *WTpq*’s over the net. You may need to consult your compiler documentation for more information about data storage order and alignment on your machine.

Network Functions

Network communications are established using the concept of individual message items that are assembled into IP/UDP packets, which are then sent to a specific port number on the network. Only simulations that are “listening” to the particular port will be able to receive the packet.

Each message item has a *type* and *tag* field associated with it. The type field describes the contents of the buffer being passed. The value of the type field must be between 1 and 255, so that the user can define up to 255 different types of messages. The value 1 is reserved for char data, and value 2 is reserved for shorts.

The tag field is used as an identifier, for example to identify the WTK object with which the message item is associated. The WTK application defines constants to use with the tag field. Tags may range between 0 and 65535. To distinguish between message items having the same type value, different tag values must be used.

Message items must be at least one byte in length but no more than 256 bytes.

WTnet_open

```
FLAG WTnet_open(  
    char *group,  
    unsigned short port,  
    unsigned char range);
```

This function opens up the network if it hasn't already been opened. It must be called before other network functions are called. It returns 0 (zero) if *WTnet_open* has already been called or if the network could not be opened. If a NULL or zero value is supplied for any of the parameters, then the default values for those parameters are used.

The *group* parameter specifies the multicast IP address. The default group is “224.0.0.88.” This parameter must be a valid multicast address that has not been previously assigned. All machines in the distributed simulation must have the same group parameter in order to communicate with each other. A valid multicast address must be an IP address in the range from 224.0.0.0 to 239.255.255.255. Within this range the following addresses are already assigned (via RFC 1340), and so cannot be used:

224.0.0.0 - 224.0.0.9
224.0.1.0 - 224.0.1.9
224.0.2.1 - 224.0.2.2
224.0.3.0 - 224.0.4.255
224.1.0.0 - 224.2.255.255
232.0.0.0 - 232.255.255.255

The *port* number specifies which port to talk or listen to on the network. The default port number is 1233. The range of valid port numbers is (1024-5000). All machines in the distributed simulation must have the same port parameter in order to communicate with each other.

The *range* parameter specifies the Internet Protocol Time to Live value which has a default value of 1 (one). This parameter determines the life of the packet. Every time the packet passes through a network gateway, the packet’s life is decremented. Once the life value reaches zero, the packet “dies” and is no longer propagated through the net. If communication is to take place only over a subnet, the range value should be 1 (one). The range value must be between 1 and 255.

In the following example, *WTnet_open* is called, and the default values for group, port, and range are assigned.

```
/* open the network using all default parameter values */  
WTnet_open(NULL, 0, 0);
```

The following example calls *WTnet_open* with user-defined parameters:

```
/* open the network with user defined parameter values */  
WTnet_open(“224.3.2.1”, 4567, 2);
```

WTnet_close

```
void WTnet_close(
    void);
```

This function closes the network, if it is currently open, and deletes private data structures.

WTnet_additem

```
FLAG WTnet_additem(
    void *item,
    int len,
    int type,
    int tag);
```

This function adds an item to the packet sent over the network. If the net is not open or the item length is greater than 256 or less than 1 (one), then the call is rejected and FALSE is returned.

Each message item has a type and tag field associated with it along with the user-definable item buffer. The *type* field describes the contents of the buffer being passed. The value of the type field must be between 1 and 255, which means you can define up to 255 different types of messages. The value 1 is reserved for char data, and the value 2 is reserved for shorts. If the *type* value passed in to this function is less than 1 or greater than 255, FALSE is returned.

The *tag* field is an identifier. For example, you can use it to identify the WTK object with which the message item is associated. Your WTK application should define constants to be used for the tag field. Tags may range between 0 and 65535. To distinguish between message items having the same type value, different tag values must be used.

In the following example, a *WTpq* structure containing viewpoint information is sent out on the net. An example of retrieving this message is provided under the function *WTnet_removeitem*.

```
/* define message type (can't use 1 or 2, which are reserved)
   so that receiving end knows what type of msg being sent */
#define NETVIEWWPOS    5

/* the parameter myid is a unique identifier, which lets the
```

```
receiving applications know the msg came from me. */
void net_sendview(int myid)
{
    WTPq myview;
    WTviewpoint_getposition(WTuniverse_getviewpoints(), myview.p);
    WTviewpoint_getorientation(WTuniverse_getviewpoints(), myview.q);
    WTnet_additem(&myview, sizeof(WTPq), NETVIEWPOS, myid);
}
```

WTnet_addstring

```
FLAG WTnet_addstring(
    void *string,
    int len,
    int type,
    int tag);
```

This function adds an item to be sent over the network, whose data is a string. This function allows you to either use the reserve type number 1 for char data, or to use a defined type value other than 1 for sending char data. Otherwise, the function is the same as *WTnet_additem*. This is an example that sends char data as type 8:

```
/* we could use the predefined message type 1
for char data, but we define our own */
#define MYCHARTYPE 8

void send_hello(void)
{
    char *msg = "Hello";

    /* note that 1 is added to strlen */
    WTnet_addstring(msg, strlen(msg)+1, MYCHARTYPE, 0);
}
```

WTnet_next

```
int WTnet_next(  
    int *tag,  
    int *retlen);
```

This function determines what the next available item is. A return value of zero means no data is available while a return value greater than zero indicates the type of the next available item. The *tag* and *retlen* parameters are set to the corresponding values of the next available item.

An example for this function is provided under *WTnet_removeitem*, below.

WTnet_removeitem

```
int WTnet_removeitem(  
    void *item,  
    int len,  
    int *tag,  
    int *retlen);
```

This function copies the next available item into the buffer specified by the *item* parameter. If the length of the item is greater than the *len* parameter, a return value of -1 is returned. The *retlen* parameter is set to the length of the item. The *tag* parameter is set to the tag associated with the item.

If no item was available, a value of 0 (zero) is returned. If successful, the return value is the type of the item.

In the following code fragment, a message item corresponding to the viewpoint location of a remote machine is obtained from the network and used to update the location of the remote user's avatar. The array of graphical objects representing the users, *netobjects*, is assumed to have been set elsewhere in the application.

```
/* define maximum number of nodes */  
#define NUMNODES    16  
/* define viewpoint message type */  
#define NETVIEWPOS    5
```

```
int type, tag, retlen;
```

```
WTpq remoteview;
WTm4 mview;

WTnode *netobjects[NUMNODES]; /* These are assumed to be movable geometry
                                nodes */

/* process incoming packets */
while ( (type = WTnet_next(&tag, &retlen) ) > 0 ) {
    switch ( type ) {
        case NETVIEWPOS:
            WTnet_removeitem(&remoteview, sizeof(WTpq), &tag, &retlen);
            WTpq_2m4(&remoteview,mview);

            /* the tag identifies which remote user packet came from */
            if ( tag>=0 && tag<NUMNODES ) {
                WTnode_settransform(netobjects[tag], mview);
            }
            break;

            /* .... handle other cases as required by application... */

        default:
            WTnet_skip();
    }
}
```

WTnet_removestring

```
int WTnet_removestring(
    void *item,
    int len,
    int *tag,
    int *retlen);
```

This function is the same as *WTnet_removeitem* but is used to get data that is known to be a string.

WTnet_skip

```
void WTnet_skip(  
    void);
```

This function removes the next available item from the input queue. Use this function after a call to *WTnet_next* if the next item is not needed. An example of using this function is provided under *WTnet_removeitem*.

WTnet_flush

```
void WTnet_flush(  
    void);
```

This function ensures that all message items added by previous calls to *WTnet_additem* and *WTnet_addstring* are sent out on the network. There is no need to call this function for message items added while the simulation is running (i.e., after calling *WTuniverse_go*), since this is handled internally. However, to send a message outside of the simulation loop, for example, during initialization of a simulation, you must call this function.

WTnet_getrange

```
FLAG WTnet_getrange(  
    void);
```

This function returns the current range (Time to Live) value.

WTnet_getport

```
unsigned short WTnet_getport(  
    void);
```

This function returns the current port value.

Introduction to the Serial Port Class

The functions described in this chapter simplify the task of communicating over serial ports. They can be used to communicate with an input device following the WTK sensor driver specification (see Appendix E) or for any other serial communication. Consult your Hardware Guide for system-specific considerations concerning serial ports.

Serial Port Construction and Destruction

WTserial_new

For Windows 32-bit platforms:

```
WTserial *WTserial_new(  
    char *port,  
    int baud,  
    char parity,  
    int databits,  
    int stopbits,  
    int buffersize);
```

For UNIX platforms:

```
WTserial *WTserial_new(  
    char *port,  
    int baud);
```

Arguments:

<i>port</i>	A character string that identifies the serial port on your machine. You may use the platform independent defines SERIAL1, SERIAL2.... as the “port.”
<i>baud</i>	Specifies the baud rate at which communication will take place over the serial port. Valid values are 1200, 2400, 4800, 9600, 19200, 38400, 57600.
<i>parity</i>	Specifies parity. “N” for no parity.
<i>databits</i>	Specifies number of data bits. (8 for 8 data bits).
<i>stopbits</i>	Specifies number of stop bits. (1 for 1 stop bit).
<i>buffersize</i>	Specifies the number of bytes in a circular buffer. This argument is ignored for now, so a dummy value of 256 should do.

This function creates a serial port object. Once created, the serial port object can be passed in to the generic sensor construction function *WTsensor_new* (see page 13-7), when you wish to create a sensor object whose records are obtained by communication over the serial port. However, if you are using the device-specific macro to create a sensor object, it is unnecessary to call *WTserial_new* because the macro creates the serial port object by calling *WTserial_new*.

Examples for this function are provided in the device-specific macro definitions in the *sensor.h* file in the *include* directory.

WTserial_delete

```
void WTserial_delete(  
    WTserial *serial);
```

This function frees a serial port object. If you are using the serial port functions as part of a WTK sensor driver, it is unnecessary to call *WTserial_delete* when you are done with the sensor object because *WTsensor_delete* (see page 13-10) calls *WTserial_delete*.

Reading and Writing to a Serial Port Object

WTserial_read

```
short WTserial_read(  
    WTserial *serial,  
    char *data,  
    int length,  
    FLAG retry);
```

This function reads a string of a specified length (number of bytes) from a serial port into the buffer called *data*. It returns the number of characters that were actually read. The length requested must be no larger than the buffer size for the serial port. Your Hardware Guide describes how this buffer size is set. The *serial* argument refers to the serial port object from which you want to read data.

The *retry* flag is used to specify what should happen if fewer bytes than the requested number are actually available. This can happen if *WTserial_read* calls are made in such rapid succession that the requested number of characters hasn't had time to arrive. The retry flag is used as follows.

On Windows 32-bit platforms:

If the *retry* flag is set, WTK polls the serial port until either the desired number of characters have been read or the clock times out. (The default value for the time-out is three times *CLOCKS_PER_SEC*.) If time-out occurs before the desired number of characters are read, the "data" field contains the characters read so far (if any), but the function returns -1 to indicate an incomplete read. If however, WTK is able to read the number of characters desired (in the allowed time), this function returns the number that were read. If the *retry* flag is not set, WTK polls the serial port just once, reads any characters that are waiting at the port, and returns the number read. If there are no characters to be read, it returns -1.

On UNIX platforms:

If the *retry* flag is set, WTK polls the serial port to see if the requested number of characters are there to be read. The port is read *only* if the number of characters available is greater than or equal to the number desired. Otherwise, *WTserial_read* just returns -1 without reading even the ones that are waiting at the port. If the *retry* flag is not set, *WTserial_read*

reads the characters available at the port but NOT greater than the number requested. It returns the number of characters read (-1 if none are read).

WTserial_ntoread

```
int WTserial_ntoread(  
    WTserial *serial);
```

This function determines how many characters are waiting to be read at the serial port. The *serial* argument refers to the serial port object being read.

WTserial_write

```
short WTserial_write(  
    WTserial *serial,  
    char *buffer,  
    int length);
```

This function writes a string of a given length to a serial port and returns the number of characters that were successfully written. The *serial* argument refers to the serial port object to which you want to write. The *buffer* argument contains the string that you want to write to the serial port (pointed to by the serial port object, *serial*). The *length* argument is the number of characters to write.

User-specifiable Serial Data

WTserial_setdata

```
void WTserial_setdata (  
    WTserial *serial,  
    void *data);
```

This function sets a user-defined data field in a serial port object. Private application data can be stored in any structure. To store a pointer to the structure within a serial port object, pass in a pointer to the structure, cast to a void*, as the data argument.

WTserial_getdata

```
void *WTserial_getdata (  
    WTserial *serial);
```

This function retrieves user-defined data stored within a serial port object. You should cast the value returned by this function to the same type used to store the data with the *WTserial_setdata* function.

Platform Specific Functions

The following serial port functions are platform-specific.

Windows 32-bit platforms:

```
WTserial_setbaud  
WTserial_getbaud  
WTserial_setbytesize  
WTserial_getbytesize  
WTserial_setRTS
```

UNIX platforms:

```
WTserial_readx  
WTserial_setsize  
WTserial_setbaud (only on SUN platforms)
```

Refer to your platform-specific Hardware Guide for more platform specific serial port functions.

Providing for Portability

WTK runs on a variety of platforms, including both Windows 32-bit and UNIX systems. Most of WTK is portable across every platform, including both source code and data, making cross-system development easy. However, if the rest of your application (code other than WTK calls) makes assumptions about the operating system on which it runs, you lose portability. For this reason, WTK provides facilities to handle common cross-platform needs in a portable way.

Reading the Keyboard

You can use typical system-specific calls to read the keyboard, but if you want your code to be portable, use the following functions provided to handle the keyboard.

WTkeyboard_open

```
void WTkeyboard_open(  
    void);
```

This function initializes the process of reading the keyboard. Once initialized, you can use the functions *WTkeyboard_getkey* and *WTkeyboard_getlastkey* to obtain keyboard input. *WTkeyboard_open* must be called before any calls are made to these functions. See *Can WTK Detect Keyboard Events?* on page A-8 for an example of how to use this function.

On some systems, it may be useful to close the keyboard (using *WTkeyboard_close*) when input from the keyboard is no longer needed.

WTkeyboard_getkey

```
short WTkeyboard_getkey(  
    void);
```

This function removes the next key from the keyboard input buffer and returns it. Since this function removes only a single key press event from the input buffer each time it is called (compared to *WTkeyboard_getlastkey*, which empties the buffer), it is your responsibility to call it frequently enough to clear the events from the input buffer.

This function is usually used within a “while” statement in an application’s action function (see *WTuniverse_setactions* on page 2-12), as shown in the following code fragment. This example assumes that *WTkeyboard_open* has been called in the main program before *WTuniverse_go*, and that *handle_key* is a function defined in the application, possibly containing a “switch” statement as in the example under *WTkeyboard_getlastkey* (see below).

```
short key;  
  
/* use 'while' statement to process all keyboard events */  
while ( key = WTkeyboard_getkey() ) {  
    handle_key(key);  
}
```

See *Can WTK Detect Keyboard Events?* on page A-8 for an example of how to use this function. See *Keyboard Constants* on page C-4 for defined constants pertaining to the extended keyboard.

WTkeyboard_getlastkey

```
short WTkeyboard_getlastkey(  
    void);
```

This function returns the most recently pressed key and discards any others in the input buffer. For example, if the keys “a”, “b”, and “c” were pressed, it ignores “a” and “b” (removing them from the input buffer and discarding them) and returns “c.”

In typical use, *WTkeyboard_getlastkey* is called in the application's action function (see *WTuniverse_setactions* on page 2-12) and might contain code like the following:

```
short key;
key = WTkeyboard_getlastkey();
switch ( key ) {
    case 'a':
        WTmessage("got an 'a'\n");
        break;
    case 'q':
        WTmessage("Quitting application\n");
        WTuniverse_stop();
        break;
    default:
        break;
}
```

This example assumes that *WTkeyboard_open* has previously been called in the main program before *WTuniverse_go*. See *Keyboard Constants* on page C-4 for defined constants pertaining to the extended keyboard.

WTkeyboard_close

```
void WTkeyboard_close(
    void);
```

This function closes a keyboard. Use this function if you have previously called *WTkeyboard_open* (see page 24-1), but no longer need to read input from the keyboard. On some systems, this can be important for the efficiency of event processing.

Reading File Directories

WTK provides a portable facility for reading directories. Its use is demonstrated with the following example, in which a directory is opened and its contents are listed.

```
WTdirectory *dir;
char *fname;
/* obtain a pointer to a WTdirectory structure for the specified path */
dir = WTdirectory_open("/wtk/models");

/* if successfully opened directory, print a list of its contents and close it */
if ( dir ) {
    while ( (fname = WTdirectory_getentry(dir)) != NULL )
        WTmessage("%s\n", fname);
    WTdirectory_close(dir);
}
```

WTdirectory_open

```
WTdirectory * WTdirectory_open(
    char *path);
```

This function opens a directory for the purpose of reading its contents. If successful, a *WTdirectory* object is returned, otherwise `NULL` is returned. When you are done examining the contents of the opened directory, close the directory by passing the *WTdirectory* object to *WTdirectory_close* (see below).

WTdirectory_getentry

```
char * WTdirectory_getentry(
    WTdirectory *dir);
```

This function returns the next entry from an opened directory. A filename is returned each time the function is called, until there are no more filenames and `NULL` is returned.

WTdirectory_close

```
void WTdirectory_close(  
    WTdirectory *dir);
```

This function closes an open directory.

Messages and Errors

Warnings and error messages in WTK provide status information about your application. A description of possible warnings and errors is provided in Appendix D, *Error Messages and Warnings*. WTK also provides functions for the developer to specify application-specific messages and error conditions.

WTmessage

```
void WTmessage(  
    char *format,  
    ....);
```

This function is similar to the C library function *printf*; it prints messages from your application to output. Use this function exactly like *printf* for printing messages from your WTK application. The advantage of using *WTmessage* is that your messages are easily redirected to the console or to a file using the function *WTmessage_sendto* (see below). Also, if directed to the console, your message can print even on platforms that do not easily support standard C I/O, such as Windows applications. Using *WTmessage* instead of *printf* therefore increases the portability of your application.

To make the change from *printf* to *WTmessage* transparent, you could use the following definition:

```
#define printf WTmessage
```

WTwarning

```
void WTwarning(  
    char *format,  
    ....);
```

This function is similar to the C library function *printf*; it prints messages from your application to output. Like *WTmessage* (see above), it takes arguments similar to *printf*. However, when you call *WTwarning*, the arguments are printed preceded by the string *WTK Warning*:. The advantage of using *WTwarning* is that your warning messages are easily redirected to the console or to a file using the function *WTmessage_sendto*. Also, if directed to the console, your warning message can print even on platforms that do not easily support standard C I/O, like Windows applications. Using *WTwarning* instead of *printf* therefore increases the portability of your application.

WTmessage_sendto

```
void WTmessage_sendto(  
    int type,  
    int where,  
    FILE *output_file);
```

This function redirects messages, warnings and errors from your application to a file, console, or callback function. By default, all messages go to the console.

The *type* argument indicates the type of message to redirect. It should be one of the following:

<i>WTMESSAGE_ERROR</i>	Error messages.
<i>WTMESSAGE_WARNING</i>	Warning message.
<i>WTMESSAGE_USER</i>	Messages from your application (using the <i>WTmessage</i> function).

The *where* argument indicates where to direct the messages. It should be some combination of the following:

<i>WTMESSAGE_TONOWHERE</i>	Do not send the messages anywhere.
<i>WTMESSAGE_TOCONSOLE</i>	Send the messages to the standard console.

<code>WTMESSAGE_TOFILE</code>	Send the messages to a file.
<code>WTMESSAGE_TOCALLBACK</code>	Send the messages to a user-specified callback function.

If `WTMESSAGE_TOFILE` is given, the `output_file` argument should point to a valid file (opened with `fopen()`). Otherwise, the `output_file` argument is ignored and the message goes to the console.

The following is an example of sending a warning to nowhere:

```
/* Do NOT display any WTK warnings */
WTmessage_sendto(WTMESSAGE_WARNING,
                 WTMESSAGE_TONOWHERE, NULL);
```

If `WTMESSAGE_TOCALLBACK` is given, you should call the `WTmessage_setcallback` function to specify the function where you want the messages sent.

You can also direct messages to both the console and a file, using the logical OR operator (`|`), as illustrated here:

```
/* Send user message to both the console and a file */
FILE *fp = fopen("output", "w");
WTmessage_sendto(WTMESSAGE_USER,
                 WTMESSAGE_TOCONSOLE | WTMESSAGE_TOFILE, fp);
```

WTmessage_setcallback

```
void WTmessage_setcallback (
    void (*callback function) (char *str));
```

This function sets the destination for message callbacks. The function passed as the argument should be in this form:

```
void my_function (char *string)
```

WTErrror

```
void WTErrror(  
    char *format,  
    ....);
```

This function displays an error message and immediately exits the application. Like *WTmessage* (see page 24-5), it takes arguments similar to the C function *printf*. However, when you call *WTErrror*, the arguments are printed preceded by the string *WTK error:* and the application terminates immediately. Use this function to abort your application if an unresolvable problem occurs, such as not being able to load an essential model.

For example:

```
char *filename;  
  
/* try to load a file, and terminate with error msg if failure */  
node = WTnode_load (root, filename, 1.0f);  
if (!node)  
    WTErrror("Couldn't load '%s'", filename);
```

Waiting

WTmsleep

```
void WTmsleep(  
    int msec);
```

This function causes the application to “sleep” for a given amount of time in milliseconds. The *msec* argument is the number of milliseconds to sleep. Note that due to differences between platforms, this function is not always precisely accurate. It is usually accurate to the nearest 1/100 of a second.

A typical use for this function is to pause while initializing a sensor. For example:

```
WTserial *serial;
WTserial_write(serial, "Reset", 5);
WTmsleep(500);    /* Wait for half a second */
```

Memory Allocation

WTmalloc

```
void *WTmalloc(
    size_t size);
```

This function is identical in functionality to the C library function *malloc*, except that in the multi-pipe/multi-processing version of WTK, shared memory is allocated instead of private (non-shared) memory.

Shared memory must be used in these cases:

- To store image data that is passed to *WTtexture_replace*
- For data that may be used by a task or universe action function
- For window drawing callback functions

Shared memory can be allocated through a call to *WTmalloc*.

WTcalloc

```
void *WTcalloc(
    size_t num,
    size_t size);
```

This function is identical in functionality to the C library function *calloc*, except that in the multi-pipe/multi-processing version of WTK, shared memory is allocated instead of private (non-shared) memory.

Shared memory must be used in these cases:

- To store image data that is passed to *WTtexture_replace*
- For data that may be used by a task or universe action function
- For window drawing callback functions

Shared memory can be allocated through a call to *WTcalloc*.

WTrealloc

```
void *WTrealloc(  
    void *old,  
    size_t newsize);
```

This function is identical in functionality to the C library function *realloc*, except that in the multi-pipe/multi-processing version of WTK, shared memory is allocated instead of private (non-shared) memory.

Shared memory must be used in these cases:

- To store image data that is passed to *WTtexture_replace*
- For data that may be used by a task or universe action function
- For window drawing callback functions

Shared memory can be allocated through a call to *WTrealloc*.

WTfree

```
void WTfree(  
    void *pointer);
```

This function is identical in functionality to the C library function *free*, except that it is used to free memory that was allocated by calls to *WTmalloc*, *WTcalloc*, or *WTrealloc*.

Introduction

The WTK math library contains functions for managing position and orientation data. These are the data types used in the math library:

- *WTp2* 2D Vector – array of 2 floating point values. (see page 25-4)
- *WTp3* 3D Vector – array of 3 floating point values. (see page 25-5)
- *WTq* Quaternion – array of 4 floating point values. (see page 25-12)
- *WTpq* Coordinate Frame – structure containing *WTp3* and *WTq*. (see page 25-19)
- *WTm3* 3D Matrix – 3x3 array of floating point values. (see page 25-21)
- *WTm4* 4D Matrix – 4x4 array of floating point values. (see page 25-22)

In WTK, orientation records are stored in quaternion form. If you prefer to work with matrices or euler angles, or if you are writing a sensor driver for a device that returns orientation records in matrices or euler angles, then you will need to convert the orientation records into quaternion form. Conversion functions are provided for going between matrix, euler angle, and quaternion representations of an orientation.

It may be convenient, when indexing mathematical quantities in WTK, to use the constants X, Y, Z, and W, which have been defined as 0, 1, 2, and 3 respectively.

The functions *WTp3_print*, *WTpq_print*, and *WTq_print* are provided to allow you to easily print out the value of position and orientation variables, for debugging an application or for other purposes.

Many of the functions in this chapter are actually macros defined in the *mathlib.h* header file. These are indicated by “[MACRO]” in the corresponding descriptions in this chapter. Be careful *not* to nest function calls within a macro call as in the following example.

This example uses the macro *WTP3_mults* and a function *f*, which could be any function returning a floating point value:

```
WTP3 pos;  
WTP3_mults(pos, f(pos)); /* gives incorrect result */
```

The reason that this gives an incorrect result is that *f(pos)* is evaluated three times by *WTP3_mults*, at the same time that *WTP3_mults* is changing the components of *pos*! To obtain the intended result, you would use the following instead:

```
WTP3 pos;  
float val;  
val = f(pos);  
WTP3_mults(pos, val); /* gives correct result */
```

This approach is preferable not just because it gives the correct value, but because when implemented this way, *f(pos)* is evaluated only once rather than three times.

WTK Math Conventions

The WTK coordinate system has the X axis pointing to the right, the Y axis pointing down, and the Z axis pointing straight ahead. This is a right-handed coordinate system because it obeys the right-hand rule. Align the fingers of your right hand with the first axis, facing the palm of your hand towards the second axis. Your thumb will point in the third axis direction (see Figure 25-1). Rotations also obey the right-hand rule. With the thumb of your right hand pointing along an axis, positive rotation is in the direction that your fingers bend.

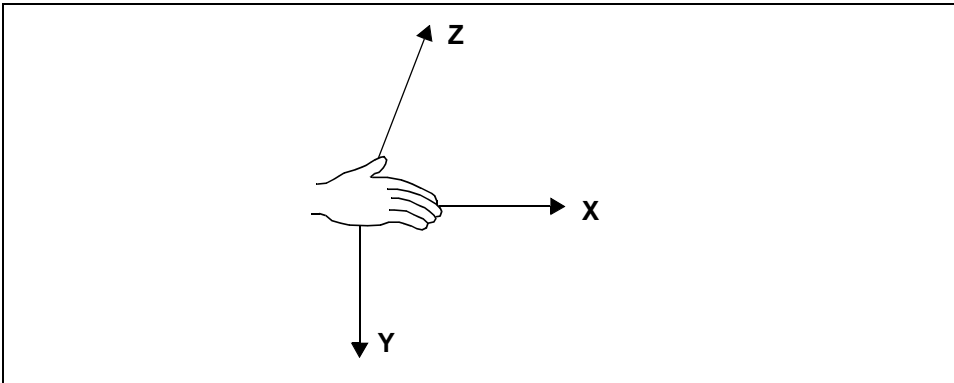


Figure 25-1: An Illustration of the “right-hand rule”.

Rotation operators in WTK operate from right to left. What does this mean? Let’s say you are working with an orientation that is represented by a 3×3 matrix. When this matrix multiplies a 3D vector, the 3D vector is rotated (hence the matrix is a “rotation operator”). There are two ways to multiply a vector and a matrix to create another vector:

- Either the vector is a *column vector* and the matrix is to its left (the matrix operates from left to right)
- Or the vector is a *row vector* and the matrix is to its right (the matrix operates from right to left)

Different results are obtained in these two cases. In WTK, matrices and quaternions follow the convention of operating from right to left. If you have a matrix that does not obey this convention (and it is a unitary matrix), call `WTm3_transpose` or `WTm4_transpose` to generate an acceptable matrix. Similarly, if you have a quaternion that does not obey this convention, call `WTq_invert` to generate an acceptable quaternion.

WTP2: 2D Vectors

A *WTP2* is type defined as an array of two floats.

WTP2_init

```
[MACRO] WTP2_init(  
    WTP2 p);
```

This function initializes a 2D vector so that $p[X] = p[Y] = 0.0$.

WTP2_copy

```
[MACRO] WTP2_copy(  
    WTP2 pin,  
    WTP2 pout);
```

This function copies *pin* into *pout*.

WTP2_mag

```
[MACRO] WTP2_mag(  
    WTP2 p);
```

This function returns the magnitude of *p*.

WTP2_norm

```
[MACRO] WTP2_norm(  
    WTP2 p);
```

This function normalizes *p* (scales it to unit length).

WTP2_dot

```
[MACRO] WTP2_dot(  
    WTP2 a,  
    WTP2 b);
```

This function returns the dot product of *a* and *b*, which is defined to be:

$$\text{dot} = a[X]*b[X] + a[Y]*b[Y]$$

WTP2_subtract

```
[MACRO] WTP2_subtract(  
    WTP2 p1,  
    WTP2 p2,  
    WTP2 pout);
```

This function subtracts vector *p2* from *p1* and puts the result in *pout*.

WTP3: 3D Vectors

A *WTP3* is type defined as an array of three floats.

WTP3_init

```
[MACRO] WTP3_init(  
    WTP3 p);
```

This function initializes a 3D vector so that $p[X] = p[Y] = p[Z] = 0.0$.

Wtp3_copy

```
[MACRO] Wtp3_copy(  
    Wtp3 pin,  
    Wtp3 pout);
```

This function copies *pin* into *pout*.

Wtp3_invert

```
[MACRO] Wtp3_invert(  
    Wtp3 pin,  
    Wtp3 pout);
```

This function negates a 3D vector. For example, it sets:

```
pout[X] = - pin[X]; pout[Y] = - pin[Y]; pout[Z] = -pin[Z];
```

Wtp3_mag

```
[MACRO] Wtp3_mag(  
    Wtp3 p);
```

This function returns the length of a 3D vector.

Wtp3_norm

```
[MACRO] Wtp3_norm(  
    Wtp3 p);
```

This function normalizes a 3D vector, dividing each of its components by the vector's length.

WTP3_add

```
[MACRO] WTP3_add(  
    WTP3 p1,  
    WTP3 p2,  
    WTP3 pout);
```

This function adds vectors $p1$ and $p2$ and puts the result in $pout$.

WTP3_subtract

```
[MACRO] WTP3_subtract(  
    WTP3 p1,  
    WTP3 p2,  
    WTP3 pout);
```

This function subtracts vector $p2$ from $p1$ and puts the result in $pout$.

WTP3_dot

```
[MACRO] WTP3_dot(  
    WTP3 p1,  
    WTP3 p2);
```

This function returns the dot product of $p1$ and $p2$, which is defined to be:

$$\text{dot} = p1[X]*p2[X] + p1[Y]*p2[Y] + p1[Z]*p2[Z]$$

WTP3_cross

```
void WTP3_cross(  
    WTP3 p1,  
    WTP3 p2,  
    WTP3 pout);
```

This function finds the cross product of vectors $p1$ and $p2$ and puts the result in $pout$ (as illustrated in Figure 25-2). If you imagine a plane passing through $p1$ and $p2$, $pout$ is a vector normal to, or sticking straight out of, that plane. Note that using the right-hand rule, $p1$ and $p2$ determine $pout$.

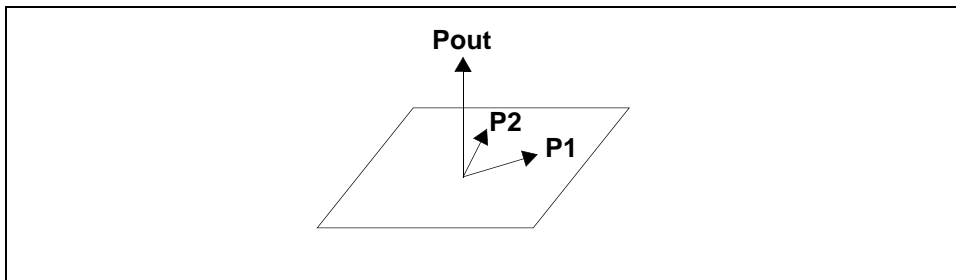


Figure 25-2: An illustration of the cross product: $P1 \times P2 = Pout$.

WTP3_equal

```
[MACRO] WTP3_equal(  
    WTP3 p1,  
    WTP3 p2);
```

This function tests two 3D vectors for equality. The 3D vectors $p1$ and $p2$ are considered equal if each of their three components is equal within the defined constant $WTFUZZ$ (0.004). If $p1$ and $p2$ are equal then TRUE is returned. Otherwise FALSE is returned.

WTP3_exact

```
[MACRO] FLAG WTP3_exact(  
    WTP3 p1,  
    WTP3 p2);
```

This function tests two 3D vectors for equality. All components of the vectors $p1$ and $p2$ must be exactly equal. If $p1$ and $p2$ are equal then TRUE is returned. Otherwise, FALSE is returned.

WTP3_rotate

```
void WTP3_rotate(  
    WTP3 pin,  
    WTq q,  
    WTP3 pout);
```

This function rotates a 3D vector (specified in pin) through the rotation represented by q , and puts the result into $pout$.

WTP3_rotatept

```
void WTP3_rotatept(  
    WTP3 pin,  
    WTq q,  
    WTP3 pout,  
    WTP3 point);
```

This function rotates a 3D vector (specified in pin) around a 3D $point$ by the rotation represented by a quaternion q , and puts the result into $pout$.

WTP3_xform

```
void WTP3_xform(  
    WTP3 pin,  
    WTPq *pq,  
    WTP3 pout);
```

This function transforms a point (specified in *pin*) from world coordinates to a local reference frame (specified in *pq*). The resulting point is placed into *pout*.

WTP3_multm3

```
void WTP3_multm3(  
    WTP3 v1  
    WTM3 m,  
    WTP3 v2);
```

This function multiplies a direction vector by a *WTM3* matrix. The result is placed into *v2*.

WTP3_multm4

```
void WTP3_multm4 (  
    WTP3 v1,  
    WTM4 m,  
    WTP3 v2);
```

This function multiplies a direction vector by a *WTM4* matrix. The result is placed into *v2*.

WTP3_mults

```
[MACRO] WTP3_mults(  
    WTP3 p,  
    float s);
```

This function multiplies the vector *p* by the scalar (floating point value) *s* and places the result into *p*.

WTP3_distance

```
float WTP3_distance(  
    WTP3 p1,  
    WTP3 p2);
```

This function returns the distance between points *p1* and *p2*.

WTP3_disttovector

```
float WTP3_disttovector(  
    WTP3 pt,  
    WTP3 ptondir,  
    WTP3 dir);
```

This function returns the perpendicular distance from a point to a vector in 3D space. The point is passed in as *pt* and the vector is passed in via two components: *ptondir* is a point on the vector and *dir* is the direction of the vector. The direction vector *dir* must be a unit vector (i.e. it must be normalized).

WTP3_coplanar

```
short WTP3_coplanar(  
    WTP3 *verts,  
    int nverts,  
    WTP3 normal);
```

This function tests whether a set of 3D points are coplanar to within the WTK fuzz value *WTFUZZ* (0.004). This is the same function that is used internally to check for coplanarity when new polygons are created.

The *verts* argument should be an array of *WTP3*'s and the *nverts* argument indicates the number of elements in the array. The return value is as follows:

- 0 If the points are not coplanar.
- 1 If the points are coplanar and the polygon defined by the points is convex.

- 2 If the points are coplanar and the polygon defined by the points is non-convex.

This function also computes the normal vector for this set of points and places it in the *normal* argument. If the set of points is not coplanar (i.e., if FALSE is returned) then the normal that is computed may not be meaningful.

WTp3_print

```
void WTp3_print(  
    WTp3 pos,  
    char *string);
```

This function prints to *WTMESSAGE_USER* the value of the specified string, followed by the value of each element of the *WTp3* structure. For example:

```
WTp3 pos;  
WTp3_print(pos, "This is my current view position");
```

would generate the following output:

```
This is my current view position 10.001 56.256 -34.567
```

This function uses the *WTmessage* call, so that the output of *WTp3_print* can be redirected using *WTmessage_sendto* (see page 24-6).

WTq: Quaternions

Quaternions in WTK are stored in a *WTq*, which is type defined as an array of four floats and is assumed to be normalized to a length equal to 1.0. In order to understand what quaternions are and how they are used, a discussion of rotations is in order.

There are 24 conventions for describing an orientation in 3D-space involving rotations about the cardinal axes. Two examples, YXZ Euler (or body-fixed) and ZXY fixed (or world-fixed), are in the description of *WTeuler_2q*. Others include permutations of three and also only two axes: ZYZ, YXY, etc. Any euler convention is equivalent to the fixed convention with reversed axis order (EulerYXZ = FixedZXY).

These conventions are easy to visualize, but have limitations. For each convention, any orientation can still be described by two sets of angles. This is why `WTq_2euler` returns two sets of euler angles; both are correct. For example, in WTK (which uses EulerXYZ, or "pitch, yaw, then roll"), the default orientation corresponds to looking down the Z axis, and your rotation is [0,0,0]. Looking down the Z axis could also be described as [180,180,180].

Also, for a three-axes convention, when the second rotation is $\pm \pi/2$, the first and third rotation are synonymous. For example, in WTK, let us pitch up $\pi/4$, then yaw right $\pi/2$. Now, forgetting how we got there, let's describe the orientation. We are facing right, tilting our head diagonally towards our right shoulder. Did we not pitch, yaw $\pi/2$ right, then roll $\pi/4$ right? Did we pitch up $\pi/2$, yaw $\pi/2$ right, then roll $\pi/4$ left? There are infinite solutions at this point, a situation known as Gimble lock, which is why `WTq_2euler` assumes zero pitch when yaw is $\pm \pi/2$. The same Gimble lock happens for two-axes conventions: when the second rotation is zero, obviously the first and third rotations are indistinguishable.

Instead of describing an orientation by rotations about cardinal axes, there exists a "single-axis" vector around which a rotation will achieve the goal orientation. This vector and angle together are what compose a quaternion.

However, do not confuse this vector and angle pair with the popular term "dir and twist". The direction vector and single-axis vector, and twist angle and single-axis angle are completely different. The direction vector points out of the nose of the viewpoint whichever way it is facing, with twist being the roll applied around it. The quaternion comprises the single-axis vector around which you rotated, and the amount of that rotation to get to that orientation. For example, Yaw left by $\pi/2$ and roll left by $\pi/2$. The direction vector is [-1,0,0] and the twist is $-\pi/2$. The single-axis vector (before normalization) to achieve the same goal is [1,-1,-1] and the angle is $\pi/2$.

Now, quaternions are more useful than just a compact description of a rotation. In their raw form they can be quickly multiplied together (thereby adding rotations as described in `WTeuler_2q`), and converted to a rotation matrix in one step: important qualities when creating a real-time 3DVR toolkit. However, to gain this functionality, the raw form is not intelligible without knowing how the values are obtained.

The single-axis vector must be normalized, then multiplied by $\sin(\text{single-axis angle}/2)$, yielding the first three values. The fourth value is $\cos(\text{single-axis angle}/2)$.

$$q = [x*\sin(a/2), y*\sin(a/2), z*\sin(a/2), \cos(a/2)]$$

The quaternion is therefore a four-dimensional unit vector which is why it is easily manipulated with four-dimensional math. Because the single-axis vector is combined with a transcendental function, having all values in the quaternion near zero is not an error.

For many applications it is enough to simply think of a quaternion as a representation of orientation or as a rotation operator. For example, viewpoints and graphical objects are two WTK object types that have an associated orientation. These orientations can be obtained with *WTviewpoint_getorientation* and *WTnode_getorientation*. The quaternion that is obtained by calling these functions can be thought of as an operator which, if applied to axes initially aligned with the world coordinates axes, would make them align with the local coordinate frame axes of the viewpoint or object.

For those who would like to learn more about quaternions, we recommend the following references:

- Craig, J. *Introduction to Robotics*, Addison-Wesley, 2nd edition, 1989. Chapter 2 is an excellent description of spatial transformations including a number of representations for orientation.
- Shoemake, K., “Animating Rotation with Quaternion Curves”, ACM SIGGRAPH 1985, San Francisco, pp. 245-254. The appendix lists a number of conversions including quaternion-to-rotation matrix, rotation matrix-to-quaternion, euler angles to quaternion, matrix to euler angles. Also contains good pointers to other references.
- Funda, J., and Paul, R.P., “A Comparison of Transforms and Quaternions in Robotics”, 1988 IEEE Robotics and Automation, pp. 886-891. More mathematical treatment of quaternions as spatial transforms. Describes efficient computational techniques for performing spatial operations.
- Maillot, P.-G., “Using Quaternions for Coding 3D Transformations”, pp. 498-515 in *Graphics Gems*, A.S. Glassner, ed., Academic Press, 1990.

WTq_init

```
[MACRO] WTq_init(  
    WTq q);
```

This function initializes a quaternion, so that:

$$q[X] = q[Y] = q[Z] = 0.0; q[W] = 1.0;$$

A quaternion q , initialized as above, represents no rotation.

WTq_copy

```
[MACRO] WTq_copy(  
    WTq qin,  
    WTq qout);
```

This function copies qin into $qout$.

WTq_invert

```
[MACRO] WTq_invert(  
    WTq qin,  
    WTq qout);
```

This function inverts a quaternion, setting:

```
qout[X] = - qin[X]; qout[Y] = -qin[Y]; qout[Z] = -qin[Z];  
qout[W] = qin[W];
```

The inverse of a quaternion corresponds to an inverse rotation.

WTq_mag

```
[MACRO] WTq_mag(  
    WTq q);
```

This function returns the length of a quaternion vector. Note that all quaternions used in WTK are expected to be normalized, that is, have length = 1.0. See the next macro, *WTq_norm*.

WTq_norm

```
[MACRO] WTq_norm(  
    WTq q);
```

This function normalizes a quaternion, that is, it scales each component of the quaternion so that the sum of the squares of the components equals one.

WTq_exact

```
[MACRO] FLAG WTq_exact(  
    WTq q1,  
    WTq q2);
```

This function tests two quaternions for equality. All components of the quaternions $q1$ and $q2$ must be exactly equal. If $q1$ and $q2$ are equal then TRUE is returned. Otherwise, FALSE is returned.

WTq_equal

```
[MACRO] FLAG WTq_equal(  
    WTq q1,  
    WTq q2);
```

This function tests two quaternions for equality. The quaternions $q1$ and $q2$ are considered equal if each of the four components is equal to within the defined constant *WTFUZZ* (0.004). If $q1$ and $q2$ are equal, then TRUE is returned. Otherwise, FALSE is returned.

WTq_getvector

```
void WTq_getvector(  
    WTq q,  
    WTP3 vec);
```

This function returns in *vec* the unit vector which is rotated around by the quaternion.

WTq_getangle

```
float WTq_getangle(  
    WTq q);
```

This function returns the angle sweep of the quaternion. This is the amount of rotation in the quaternion. This angle is returned in radians.

WTq_scale

```
void WTq_scale(  
    WTq qin,  
    WTq qout,  
    float scale);
```

This function controls the angle sweep (the amount of rotation) in a quaternion. The input quaternion is specified in *qin*, the amount by which to scale is specified in the *scale* parameter. The resultant quaternion is placed in *qout*.

WTq_construct

```
void WTq_construct(  
    WTp3 vec,  
    float ang,  
    WTq qout);
```

This function composes a quaternion *qout* from the vector *vec* and angle *ang* specified in radians.

WTq_mult

```
void WTq_mult(  
    WTq q1,  
    WTq q2,  
    WTq qout);
```

This function multiplies quaternion *q2* into *q1* and puts the result into *qout*. Multiplying quaternions corresponds to composing rotations. An example of using this function is

provided under the function *WTeuler_2q* on page 25-27.

WTq_multinv

```
void WTq_multinv(  
    WTq q1,  
    WTq q2,  
    WTq qout);
```

This function multiplies quaternion *q2* into the inverse of *q1* and puts the result into *qout*. To understand the purpose of this function, you can think of it as first “undoing” rotation *q1* and then applying rotation *q2* to obtain *qout*. The *qout* parameter is therefore the rotation needed to go from orientation *q1* to orientation *q2*.

WTq_interpolate

```
void WTq_multinv(  
    WTq q1,  
    WTq q2,  
    float u,  
    WTq qout);
```

This function finds the spherical linear interpolation using the shortest path between quaternion *q1* and *q2*, and places the result into *qout*. The *u* argument is a factor between 0 (zero) and 1 (one), which specifies the point of interpolation. For example, *u = 0.5* would find a quaternion halfway between *q1* and *q2*.

WTq_dot

```
[MACRO] WTq_dot(  
    WTq q1,  
    WTq q2);
```

This function returns the dot product of *q1* and *q2*.

WTq_print

```
void WTq_print(  
    WTq orientation,  
    char *string);
```

This function prints to *WTMESSAGE_USER* the value of the string followed by the value of each element of the *WTq* structure. For example:

```
WTq q;  
WTviewpoint_getorientation(WTuniverse_getviewpoint(), q);  
WTq_print(q, "This is my current view orientation");
```

would generate the following output:

```
This is my current view orientation -0.025 0.453 0.346 -0.556
```

This uses a *WTmessage* call, so *WTq_print* output is redirected using *WTmessage_sendto* (see page 24-6).

WTpq: Coordinate Frame Structure

WTK defines the following type definition for 6D coordinate frame records:

```
typedef struct _WTpq {  
    WTp3 p;  
    WTq q;  
} WTpq;
```

To pass a *WTpq* to a routine in C, you must take the address of the structure using the C ampersand operator (&) to pass it by address. The following functions and macros illustrate the passing in of a *WTpq* by address.

WTPq_init

```
[MACRO] WTPq_init(  
    WTPq *pq);
```

This function initializes the *WTPq* structure as follows:

```
p[X] = p[Y] = p[Z] = 0.0f;  
q[X] = q[Y] = q[Z] = 0.0f;  
q[W] = 1.0f;
```

For example, you might have:

```
/* declare and initialize a WTPq struct */  
WTPq pq;  
WTPq_init(&pq);
```

WTPq_copy

```
[MACRO] WTPq_copy(  
    WTPq *pqin,  
    WTPq *pqout);
```

This function copies the contents of *pqin* into *pqout*.

WTPq_print

```
void WTPq_print(  
    WTPq *pq,  
    char *string);
```

This function prints to *WTMESSAGE_USER* the value of the string followed by the value of each element of the *WTPq* structure. For example:

```
WTPq viewpq;  
WTPq_print(&viewpq, "This is my current viewpoint");
```

would generate the following output:

This is my current viewpoint p: 10.003 5.004 10.123, q: -0.25 0.453 0.346 -0.556

This function uses the *WTmessage* call, so that the output of *WTPq_print* can be redirected using *WTmessage_sendto* (see page 24-6).

WTm3: 3D Matrices

Besides quaternions, WTK supports 3D matrices which are commonly used for rotation operations. A *WTm3* is type defined as a 3x3 array of floats. If the *WTm3* is a rotation matrix (created by *WTeuler_2m3* or *WTq_2m3*) then it is orthonormal and its inverse is its transpose.

WTm3_init

```
[MACRO] WTm3_init(  
    WTm3 m);
```

This function sets a matrix equal to the identity matrix. This is a matrix with 1s along its diagonal and 0s everywhere else, corresponding to no rotation (see Figure 25-3).

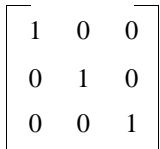

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 25-3: The 3x3 identity matrix.

WTm3_copy

```
[MACRO] WTm3_copy(  
    WTm3 min,  
    WTm3 mout);
```

This function copies *min* into *mout*.

WTm3_transpose

```
void WTm3_transpose(  
    WTm3 min,  
    WTm3 mout);
```

This function puts the transpose of *min* into *mout*, swapping the rows with the columns.
 $mout[x,y] = min[y,x]$

WTm3_multm3

```
void WTm3_multm3(  
    WTm3 m1,  
    WTm3 m2,  
    WTm3 mout);
```

This function performs the matrix multiplication $mout = m1 * m2$. Matrix *mout* is the composite rotation of *m1* and *m2*.

WTm4: 4D Matrices

In addition to *WTpq* structures, WTK supports 4D transformation matrices. A *WTm4* is a 4x4 array of floats. It represents a translation followed by a rotation, and effectively contains a *WTm3* rotation in the upper left 3x3, and a *Wtp3* translation in the lower left 3x1, with the remaining far right column containing special values that should not be manipulated.

WTm4_init

```
[MACRO] WTm4_init(
    WTm4 m);
```

This function sets a matrix equal to the identity matrix. This is a matrix with 1s along its diagonal and 0s everywhere else, corresponding to no translation or rotation (see Figure 25-4).

$$\begin{bmatrix} 1 & 0 & 0 & \vdots & 0 \\ 0 & 1 & 0 & \vdots & 0 \\ 0 & 0 & 1 & \vdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 0 & 0 & \vdots & 1 \end{bmatrix}$$

Figure 25-4: The 4x4 identity matrix.

WTm4_copy

```
[MACRO] WTm4_copy(
    WTm4 min,
    WTm4 mout);
```

This function copies *min* into *mout*.

WTm4_transpose

```
void WTm4_transpose
    WTm4 min,
    WTm4 mout);
```

This function puts the transpose of *min* into *mout*, swapping the rows with the columns.
 $mout[x,y] = min[y,x]$

WTm4_multm4

```
void WTm4_multm4
    WTm4 m1,
    WTm4 m2,
    WTm4 mout);
```

This function performs the matrix multiplication $mout = m1 * m2$. Matrix *mout* is the composite transformation of *m1* and *m2*.

WTm4_invert

```
int WTm4_invert(
    WTm4 src,
    WTm4 dst);
```

This function inverts a *WTm4* matrix. The matrix to invert is specified in the *src* argument, and its inverse is returned in the *dst* parameter. Matrix *src* represents a translation then rotation. The inverse of *src*, *dst*, is a transformation that unrotates then untranslates by the same amounts, such that $dst * src = identity$.

If the matrix has a zero determinant value, it is uninvertable and 0 (FALSE) is returned, with a warning to that effect. Otherwise, the function returns 1 (TRUE).

WTm4_xformp3

```
void WTm4_xformp3(
    WTm4 m4,
    WTp3 pin,
    WTp3 pout);
```

This function transforms a point *pin* by the transform matrix in *m4*. The result is returned in *pout*.

WTm4_xformp3 is useful when you want to know the position of a point after it is affected by a transformation. For example, to determine where a geometry is rendered, you can transform the geometry's midpoint by the cumulative transformation matrix along the node path from the root node to the geometry node.

Mathematically, *pin* is taken as a row vector and post-multiplied by the transform matrix *m4*.

WTm4_rotatep3

```
void WTm4_rotatep3(  
    WTm4 m4,  
    WTp3 pin,  
    WTp3 pout);
```

This function rotates a vector *pin* by a matrix, *m4*. The result is returned in *pout*. This function ignores the translational component of *m4*, and applies only the rotational part to the *WTp3* vector.

WTm4_rotatep3 is useful when you need to rotate a "ray" by an accumulated transformation. A ray is a vector representing a direction. If a ray is affected by a transformation matrix, its new direction is obtained by simply applying the rotational element to it. That is why *WTm4_rotatep3* ignores the translational component and in this way differs from *WTm4_xformp3*.

Conversion Functions

This section contains a variety of conversion functions that you can use to convert different representations of orientation.

Functions for converting a matrix or quaternion to euler angles are not always well-defined. That is, for a given orientation it is possible to find more than one set of euler angles which would generate that orientation. Therefore, the euler angles retrieved may not be the ones used in *WTeuler_2q* or *WTeuler_2m3*. However, the eulers will be valid for that orientation.

WTm3_2q

```
void WTm3_2q(  
    WTm3 m,  
    WTq q);
```

This function converts a 3D rotation matrix to a quaternion.

This conversion routine makes sense only if the matrix passed in is a legitimate rotation operator. If the rows of the matrix are not orthonormal, the function first generates the closest orthonormal matrix to the one passed in, and then converts that matrix to a quaternion.

If the quaternion that results is the inverse of what you would expect, it may be because the matrix passed in is one intended to multiply vectors from the left, whereas the WTK convention is for matrices to multiply vectors from the right. The functions *WTq_invert* (see page 25-15) or *WTm3_transpose* (see page 25-22) are useful in this case.

If the matrix passed in cannot be interpreted as a rotation operator, it returns the unit quaternion:

```
q[X] = q[Y] = q[Z] = 0.0; q[W] = 1.0;
```

For example, a matrix where the magnitude of one of the row vectors was 0.0 would yield the result shown above.

WTq_2m3

```
void WTq_2m3(  
    WTq q,  
    WTm3 m);
```

This function converts a quaternion to a 3D matrix.

WTeuler_2m3

```
void WTeuler_2m3(  
    float wx,  
    float wy,  
    float wz,  
    WTm3);
```

This function constructs a 3D rotation matrix from the given euler angles (specified in radians).

WTm3_2euler

```
void WTm3_2euler(  
    WTm3 m,  
    WTp3 first,  
    WTp3 second);
```

This function extracts euler angles (specified in radians) from the specified rotation matrix. The *first* and *second* are the two possible sets of euler angles that can be derived from the matrix. The *first* is the solution with yaw (Y-rotation) between $\pm \pi/2$. The *second* always has the magnitude of yaw between $\pi/2$ and π .

When the Y-rotation is found to be exactly $\pm \pi/2$, the X-rotation (pitch) is assumed to be zero.

WTeuler_2q

```
void WTeuler_2q(  
    float wx,  
    float wy,  
    float wz,  
    WTq q);
```

This function converts euler angles (specified in radians) to a unit quaternion. The euler angle rotation is defined by right-hand rotations in a *body-fixed coordinate frame* in the following order:

- first rotating about the X axis (of the body frame) by the angle w_x ,
- then about the (rotated body) Y axis by w_y ,
- and finally about the (rotated body) Z axis by w_z .

The *body-fixed coordinate frame* is similar to the local coordinate system (LCS), with the following difference. The *body-fixed coordinate frame* changes orientation when the first rotation (to the X axis) is applied to it. The body has changed orientation in 3D space, so the X, Y, and Z axes are now different in relation to the world coordinate system. When the Y rotation is applied (to the now rotated body), it is not rotating on the same axis that was the Y axis originally.

If, instead, the angles are defined with respect to a *fixed coordinate frame* (e.g., the world reference frame), then *WTeuler_2q* applies the rotations in the reverse order, that is:

- first rolling by w_z about Z (of the fixed frame),
- then yawing by w_y about the (fixed) Y axis,
- and finally pitching by w_x about the (fixed) X axis.

If for some reason you have euler angles that must be combined in a different order than described above to produce the correct final orientation, then by making multiple calls to *WTeuler_2q*, you can produce the appropriate quaternion q .

For example, let's say you are writing a sensor driver for a device that returns euler angles. Suppose these angles either:

- are defined with respect to a body-fixed reference frame and determine the correct orientation when rotation about Y is applied first, then rotation about X, and finally rotation about Z (i.e., in Y, X, Z order)
- or they are defined with respect to a fixed reference frame (like the world reference frame), and determine the correct orientation when rotation about Z is applied first, then rotation about X, and finally rotation about Y. (i.e., in Z, X, Y order)

To obtain the corresponding quaternion q , you could use the following function:

```
/*
 * This function builds up the quaternion q by generating
 * the quaternion for rotation about each axis and then
 * multiplying them together in the correct order.
 */
void my_euler_2q(wx, wy, wz, q)
    float wx, wy, wz;
    WTq q;
{
    WTq qx, qy, qz;

    /* generate the quaternions for rotation about each axis */
    WTeuler_2q(wx, 0.0, 0.0, qx);
    WTeuler_2q(0.0, wy, 0.0, qy);
    WTeuler_2q(0.0, 0.0, wz, qz);

    /* Note the order of quaternion multiplication. */
    WTq_mult(qz, qx, q);
    WTq_mult(q, qy, q);
}
```

Another example of using this function is provided under *WTsensor_rotate* on page 13-20.

WTq_2euler

```
void WTq_2euler(
    WTq q,
    WTp3 first,
    WTp3 second);
```

This function extracts euler angles, specified in radians, from the specified unit quaternion. The *first* and *second* are the two possible sets of euler angles that can be derived from the quaternion. The *first* is the solution with yaw (Y-rotation) between $\pm \text{PI}/2$. The *second* always has the magnitude of yaw between $\text{PI}/2$ and PI .

When the Y-rotation is found to be exactly $\pm \text{PI}/2$, the X-rotation is assumed to be zero.

WTq_2dir

```
void WTq_2dir(  
    WTq q,  
    WTp3 dir);
```

This function converts a quaternion into a direction vector. This direction vector is the vector that results from taking a unit vector along the Z axis (i.e. [0,0,1]) and rotating it by q . Given an orientation (like from *WTnode_getorientation* or *WTviewpoint_getorientation*) this function finds the vector pointing out the front of the node, or nose of the viewpoint.

WTdir_2q

```
void WTdir_2q(  
    WTp3 dir,  
    WTq q);
```

This function converts a unit vector to a quaternion. In generating q , it is assumed there is no twist (roll) about the direction vector (otherwise, see *WTq_2dirandtwist*).

If the specified direction vector is not normalized (i.e., does not have length equal to 1), the resulting quaternion will not be valid.

WTq_2dirandtwist

```
void WTq_2dirandtwist(  
    WTq q,  
    WTp3 dir,  
    float *twist);
```

This function is similar to *WTq_2dir*, which converts a quaternion q into a direction vector *dir*. *WTq_2dirandtwist* gets the twist factor (roll) from the quaternion, apart from the direction. The direction vector is the vector that results from taking a unit vector along the Z axis (i.e. [0,0,1]) and rotating it by q . Given an orientation (like from *WTnode_getorientation* or *WTviewpoint_getorientation*) this function finds the vector pointing out the front of the node, or nose of the viewpoint.

WTdirandtwist_2q

```
void WTdirandtwist_2q(  
    WTp3 dir,  
    float twist  
    WTq q);
```

This function converts a vector *dir* to a quaternion having a twist as specified by the *twist* argument. This function is different from *WTdir_2q* in that it considers the twist (roll) about the vector direction. It is required for the vector direction to be non-zero to obtain a valid quaternion. This result is placed in *q*.

WTm4_2pq

```
void WTm4_2pq(  
    WTm4 m,  
    WTpq *pq);
```

This function converts a *WTm4* matrix into a *WTpq* structure that holds position (*p*) and orientation (*q*) values. The function accepts a pointer to a *WTpq* structure whose *p* and *q* fields are respectively updated by the calculated position and orientation values.

If the specified matrix is bad, an initialized *q* (i.e. [0,0,0,1]) is returned as the orientation part of the *WTpq* structure. An illegal matrix is one where the first three elements of any of the first three rows are all zero.

WTpq_2m4

```
void WTpq_2m4(  
    WTpq *pq,  
    WTm4 m);
```

This function converts a *WTpq* structure with position and orientation fields into a *WTm4* matrix. The resultant matrix is returned as *m*.

WTq_2m4

```
void WTq_2m4(  
    WTq q,  
    WTM4 m);
```

This function converts a quaternion to a 4x4 matrix, and places the result into *m*. The transitional component of *m* is set to [0,0,0].

WTq_2eulernear

```
void WTq_2eulernear(  
    WTq q,  
    WTP3 nearp,  
    WTP3 euler);
```

This function returns an euler that is closest to the specified euler, corresponding to the specified *WTq*. This *WTq*, specified in the *q* argument, can be converted into two eulers, one of which is closer to the specified reference euler, *nearp*. The closer euler is stored in the *euler* parameter.

WTm3_2eulernear

```
void WTM3_2eulernear(  
    WTM3 m,  
    WTP3 nearp,  
    WTP3 euler);
```

This function returns an euler that is closest to a specified euler, corresponding to the specified *WTm3* (a 3x3 matrix). This *WTm3*, specified in the *m* argument, can be converted into two eulers, one of which is closer to the specified reference euler, *nearp*. The closer euler is stored in the *euler* parameter.

WTnormal_2slope

```
float WTnormal_2slope(  
    WTp3 normal);
```

This function takes a normal, like the normal for a polygon or ground surface, and returns the corresponding slope in radians. The returned value is between 0.0 and $\text{PI}/2$, with 0.0 returned for a polygon parallel to the X-Z plane, and $\text{PI}/2$ returned for a vertically-oriented polygon. The *normal* argument must have unit magnitude for the function to work.

For example, to obtain the slope of a polygon you would call:

```
Wtpoly *poly;  
Wtp3 normal;  
float slope;  
Wtpoly_getnormal(poly, normal);  
slope = WTnormal_2slope(normal);
```

Floating-point Comparisons

WTzero

```
[MACRO] WTzero(  
    float value);
```

This function returns TRUE if the magnitude of the specified value is less than the defined constant *WTFUZZ* (0.004). Otherwise, *WTzero* returns FALSE.

Please refer to the discussion under *Roundoff and Scaling* on page 6-13 for information about floating-point tolerances within WTK.

Reference-frame Math Utilities

This section contains functions for converting positions and orientations from one reference frame to another. More information about reference frames is provided in *Geometry Motion Reference Frames* on page 13-19.

WTP3_frame2frame

```
void WTP3_frame2frame(  
    WTP3 pin,  
    WTPq *frame1,  
    WTPq *frame2,  
    WTP3 pout);
```

This function takes *frame1*'s position (specified in *pin*) and determines its position relative to *frame2*, then places the result into *pout*.

WTP3_local2worldframe

```
void WTP3_local2worldframe(  
    WTP3 pin,  
    WTPq *frame,  
    WTP3 pout);
```

This function takes the *frame*'s position (specified in *pin*) and determines its position relative to the world coordinate frame, then places the result into *pout*.

WTP3_world2localframe

```
void WTP3_world2localframe(  
    WTP3 pin,  
    WTPq *frame,  
    WTP3 pout);
```

This function takes the position of the world coordinate frame (specified in *pin*) and determines its position relative to the specified local *frame*, then places the result into *pout*.

WTq_frame2frame

```
void WTq_frame2frame(  
    WTq qin,  
    WTpq *frame1,  
    WTpq *frame2,  
    WTq qout);
```

This function takes the specified orientation *qin* relative to *frame1* and determines its orientation relative to *frame2*, then places the result into *qout*.

WTq_local2worldframe

```
void WTq_local2worldframe(  
    WTq qin,  
    WTpq *frame,  
    WTq qout);
```

This function takes the specified orientation *qin* for the local *frame* and determines its orientation relative to the world coordinate frame, then places the result into *qout*.

WTq_world2localframe

```
void WTq_world2localframe(  
    WTq qin,  
    WTpq *frame,  
    WTq qout);
```

This function takes a specified orientation *qin* for the world coordinate frame and determines its orientation relative to the local *frame*, then places the result into *qout*.

WTPq_frame2frame

```
void WTPq_frame2frame(  
    WTPq *pqin,  
    WTPq *frame1,  
    WTPq *frame2,  
    WTPq *pqout);
```

This function takes a specified position and orientation *pqin* for *frame1* and determines the corresponding position and orientation relative to *frame2*, then places the result into *pqout*.

WTPq_local2worldframe

```
void WTPq_local2worldframe(  
    WTPq *pqin,  
    WTPq *frame,  
    WTPq *pqout);
```

This function takes the specified local *frame*'s position and orientation *pqin*, determines its position and orientation relative to the world coordinate frame, then places the result into *pqout*.

WTPq_world2localframe

```
void WTPq_world2localframe(  
    WTPq *pqin,  
    WTPq *frame,  
    WTPq *pqout);
```

This function takes a position and orientation structure *pqin*, specified in relation to the world coordinate frame, determines the corresponding position and orientation relative to the local *frame*, and places the result into *pqout*.

C++ Programming

Introduction

This chapter defines the C++ classes and methods that correspond to the WTK objects and functions described in this programmer's guide. The C++ class descriptions are provided to allow you to build your applications in a C++ programming environment, which allows you to use object-oriented methodologies.

Each class description lists the methods which are applicable to the class. Because the class method names generally correspond to WTK's C function names, you will need to refer to descriptions of each function elsewhere in this Reference Guide.

To use this chapter, you need to understand how the naming convention in C++ differs from the naming convention in C. In WTK, the C function names are made up of three parts, which helps to identify the purpose of the function. All WTK functions begin with WT, then include a word that identifies the general purpose of the function, followed by an underscore and a word that identifies the specific purpose. For example, `WTviewpoint_setdirection` is a viewpoint function that sets the direction of a viewpoint. In C++, the corresponding class is `WtViewPoint` and the corresponding method is `SetDirection`.

For the most part the function names can be understood using this approach. Occasionally the C++ method name differs from the function name in C, because the method's name is more descriptive. For these methods, there are cross references to help you find the appropriate C function. The cross references are shown to the left of the method in bold print. For example, if you want to use a method get the first motion link in a scene graph, this cross reference would tell you which function to look up.

WTuniverse_getmotionlinks static `WtMotionLink *GetFirstMotionLink(void);`

Use the Index in the Reference manual to look up more information for each function.

Class Diagrams

The relationship of the C++ classes are shown in figure 26-1 and figure 26-2

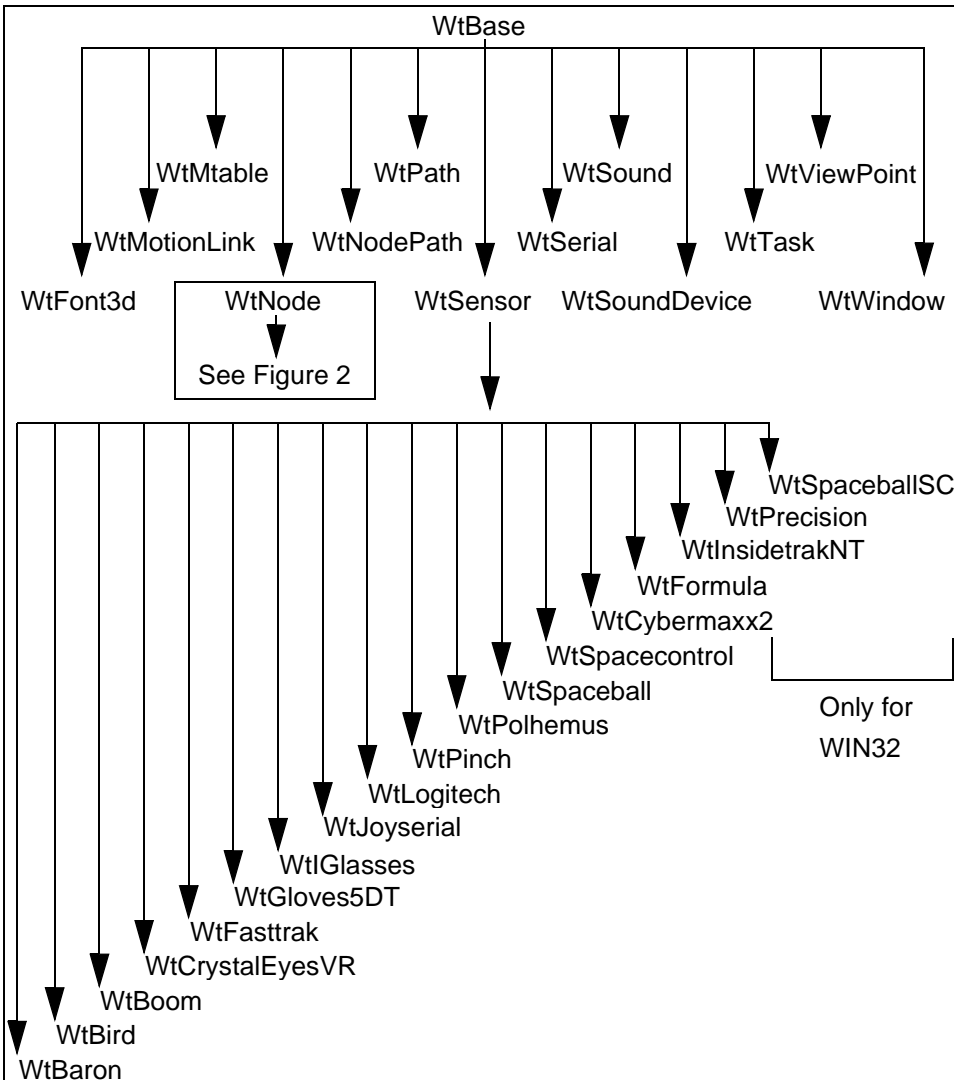


Figure 26-1: C++ Base and Sensor classes

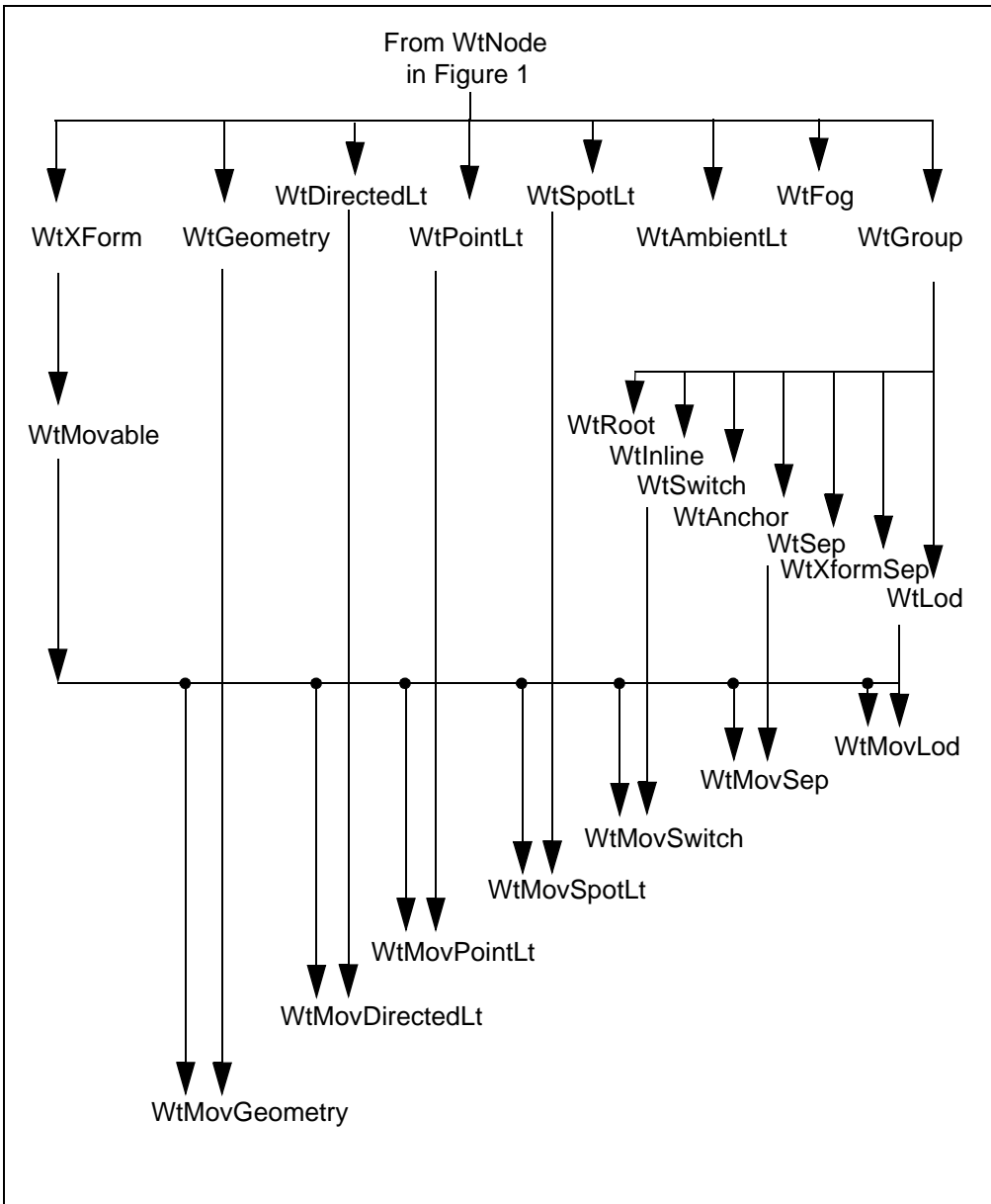


Figure 26-2: WtNode classes

<u>Stand-alone Classes</u>	<u>Math Classes</u>
WtDir	WtM3
WtKeyBoard	WtM4
WtNet	WtP2
WtScreen	WtP3
WtTexture	WtQ
WtUniverse	WtPQ
WTConnection	
WTSharegroup	

Figure 26-3: Stand-alone and Math classes

Classes and their Methods

The classes and their methods are described in the following order: First, the prototypes for global functions are described, beginning on page 26-5. Then the classes shown on the diagram in figure 26-1 are described from left to right (i.e., in alphabetical order), starting with `WtFont3d` on page 26-6. Each of the classes is described along with all of the classes below it on the diagram. For example, `WtNode` also includes all the classes shown in figure 26-2.

After all the classes in figure 26-1 are described, the classes shown in figure 26-3 are described, starting with the stand-alone classes on page 26-33 and then the math classes on page 26-39.

Finally, the defines are described, beginning on page 26-45.

Prototypes for Global functions

```
extern WtGeometry *GeometryNodeLoad(  
    char *filename,  
    float scale);  
extern WtNode *NodeLoad(  
    char *filename,  
    float scale);  
extern WtMovable *MovNodeLoad(  
    char *filename,  
    float scale);
```

Note: The above functions are the same as the methods in the `WtGroup` class on page 26-12, except that the corresponding node which is created has no parent. These nodes can be added to the scene graph. The following function checks to see whether the constructor function for the corresponding class was successful or not.

```
extern FLAG IsValid(  
    WtBase *base);  
  
char *ValueToString(  
    WTdatatype dtype,  
    void *value);
```

World2World Client C++ Applications

Define `W2WCPP` in your makefile or project settings.

WtBase Classes

WTBASE

Event handler callback function prototype:

```
typedef void(*WtEventHandler)(
    WtBase *object,
    const char *propname,
    void *value,
    double time,
    WTEventsource src);
```

```
class WtBase
{
    WtBase();
    WtBase(WtBase *parent);
    virtual ~WtBase();
    virtual void *GetWTKStructure() {return baseobject;}
    /* the following only work on actual WtBase objects created with "new WtBase" */
    WtBase *Next();
    void AddParent(WtBase *parent);
    void RemoveParent(WtBase *parent);
    int NumParents();
    WtBase *GetParent(int parentnum);
    int NumChildren();
    WtBase *GetChild(int childnum);
    FLAG IsChild(WtBase *child);
    WtBase *FindChild(const char *name);
    /* The following methods work on WTBASE, WTNODE, WTWINDOW,
       WTVIEWPOINT, WTSENSOR, and WTPATH */
    void SetData(void *data) {userdata = data;}
    void *GetData() {return userdata;}
    int GetType();
    void Print();
    void SetName(const char *name);
    const char *GetName();
```

```
int NumProperties();
const char *GetProperty(int propnum);
const char *nFindProperty(const char *propname, int ntcamp);
FLAG DeleteProperties();
static WtBase *Find(int type, const char *name);
static WtBase *nFind(int type, const char *name, int ntcamp);
FLAG PropertyNew(const char *propname, WTdatatype dtype);
FLAG PropertyDelete(const char *propname);
FLAG PropertyExists(const char *propname);
void PropertySetData(const char *propname, void *data);
void *PropertyGetData(const char *propname);
WTdatatype PropertyGetDatatype(const char *propname);
unsigned int PropertyGetSizeOfData(const char *propname);
FLAG PropertySet(const char *propname, void *value);
FLAG PropertySeti(const char *propname, int value);
FLAG PropertySetui(const char *propname, unsigned int value);
FLAG PropertySetf(const char *propname, float value);
FLAG PropertySetd(const char *propname, double value);
FLAG PropertySetp2(const char *propname, WTp2 value);
FLAG PropertySetp3(const char *propname, WTp3 value);
FLAG PropertySetq(const char *propname, WTq value);
FLAG PropertySets(const char *propname, const char *value);
FLAG PropertySetp(const char *propname, void *value);
FLAG PropertySetAt(const char *propname, void *value, double time);
FLAG PropertyGet(const char *propname, void *value);
int PropertyGeti(const char *propname);
unsigned int PropertyGetui(const char *propname);
float PropertyGetf(const char *propname);
double PropertyGetd(const char *propname);
FLAG PropertyGetp2(const char *propname, WTp2 value);
FLAG PropertyGetp3(const char *propname, WTp3 value);
FLAG PropertyGetq(const char *propname, WTq value);
char *PropertyGets(const char *propname);
void *PropertyGetp(const char *propname);
char *PropertyGetAsString(const char *propname);
FLAG PropertyAddHandler(const char *propname, WtEventHandler eh);
FLAG PropertyRemoveHandler(const char *propname, WtEventHandler eh);
int PropertyNumHandlers(const char *propname);
```

```
WtEventHandler PropertyGetHandler(const char *propname, int handlernum);
void PropertyRemoveAllHandlers(const char *propname);
FLAG RemoveAllHandlers();
#if W2WCPP /* The following methods are only available if linking with
           World2World wrapper library */
/* The following methods work on WTBASE, WTNODE, WTWINDOW,
   WTVIEWPOINT, WTSENSOR, and WTPATH */
FLAG PropertyShare(const char *propname, WtSharegroup *shgrp = NULL,
                  FLAG shareflags = 0);
FLAG PropertyShare(const char *propname, WtConnection *conn = NULL,
                  FLAG shareflags = 0);
FLAG PropertyUnshare(const char *propname, WtSharegroup *shgrp = NULL,
                    FLAG forcedelete = 0);
FLAG PropertyIsShared(const char *propname);
int PropertyNumShares(const char *propname);
WtSharegroup *PropertyGetSharegroup(const char *propname, int sharenum);
void PropertySetUpdateFreq(const char *propname, double freq);
double PropertyGetUpdateFreq(const char *propname);
void PropertySendUpdate(const char *propname);
void PropertySetTimeSensitive(const char *propname, FLAG timesensitive);
FLAG PropertyGetTimeSensitive(const char *propname);
FLAG PropertyLock(const char *propname);
FLAG PropertyUnlock(const char *propname);
unsigned int PropertyIsLocked(const char *propname);
FLAG PropertyIsLockedByMe(const char *propname);
FLAG Unshare();
#endif
};
```

WTFONT3D

```
class WtFont3d : public WtBase
{
Wtfont3d_load      WtFont3d(char *filename);
Wtfont3d_delete  ~WtFont3d(void);
                  FLAG CharExists(char character);
                  void GetExtents(WtP3 extents[2]);
                  float GetSpacing(void);
                  void SetSpacing(float spacing);
};
```

WTMOTIONLINK

```
class WtMotionLink : public WtBase
{
WTmotionlink_new  WtMotionLink(void *source, void *target, int from_type, int to_type);
WTmotionlink_delete ~WtMotionLink(void);
                  FLAG AddConstraint(int dof, float min, float max);
                  void Enable(FLAG flag);
                  int GetConstraintFrame(void);
                  int GetReferenceFrame(void);
                  FLAG GetSource(void **source, int *type);
                  FLAG GetTarget(void **target, int *type);
                  FLAG IsEnabled(void);
                  WtMotionLink *Next(void);
                  FLAG RemoveConstraint(int dof);
                  FLAG SetConstraintFrame(int constraintframe);
                  FLAG SetReferenceFrame(int frame, WtViewPoint *vpoint);
};
```

WTMTABLE

```
class WtMtable : public WtBase
{
WTmtable_load      WtMtable(char *filename);
WTmtable_new      WtMtable(int definedprops, int estimatedentries, char *name);
WTmtable_delete  ~WtMtable(void);
                   int CopyEntry(int matid, WtMtable *to);
                   static WtMtable *GetByName(char *name);
                   int GetEntryByName(char *name);
                   char *GetEntryName(int matid);
                   char *GetName(void);
                   int GetNumentries(void);
                   int GetProperties(void);
                   FLAG GetValue(int matid, float *value, int propertybit);
                   static WtMtable *Merge(WtMtable *table1, WtMtable *table2);
                   int NewEntry(void);
                   FLAG Save(void);
                   FLAG SetEntryName(int matid, char *name);
                   FLAG SetName(char *name);
                   FLAG SetProperties(int definedprops);
                   FLAG SetValue(int matid, float *value, int propertybit);
};
```

WTNODE

```
class WtNode : public WtBase
{
    WtMovGeometry *ToMovGeometry(void)
    {
        return (WtMovGeometry *) (WtMovable *) this;
    }
    WtMovDirectedLt *ToMovDirectedLt(void)
    {
        return (WtMovDirectedLt *) (WtMovable *) this;
    }
    WtMovPointLt *ToMovPointLt(void)
    {
```

```

        return (WtMovPointLt *) (WtMovable *) this;
    }
    WtMovSpotLt *ToMovSpotLt(void)
    {
        return (WtMovSpotLt *) (WtMovable *) this;
    }
    WtMovLod *ToMovLod(void)
    {
        return (WtMovLod *) (WtMovable *) this;
    }
    WtMovSep *ToMovSep(void)
    {
        return (WtMovSep *) (WtMovable *) this;
    }
    WtMovSwitch *ToMovSwitch(void)
    {
        return (WtMovSwitch *) (WtMovable *) this;
    }

```

Note: The above methods typecast a *WtNode* to the corresponding movable node.

```

    }
WtNode_delete    ~WtNode(void);
                  char *GetName(void);
                  WtNode *GetParent(int parentum);
                  int GetType(void);
WtNode_hasboundingbox  FLAG HasBBox(void);
                  FLAG IsEnabled(void);
                  FLAG IsMovable(void);
                  int NumParents(void);
                  void Print(void);
                  WtPoly *RayIntersect(WtP3 ray, WtP3 origin,
                                      float *distance, WtNodePath **npath);
                  FLAG Remove(void);
                  FLAG Save(char *filename, WtViewPoint *vpoint, int filetype, int options);
                  FLAG SetName(char *name);
};

```

WtGroup

```
class WtGroup : public WtNode
{
    WTgroupnode_new    WtGroup(WtGroup *parent);
                    FLAG AddChild(WtNode *child);
WTnode_boundingbox  FLAG BBox(FLAG enable);
                    FLAG CanAddChild(WtNode *child);
                    FLAG DeleteChild(int childnum);
                    WtNode *GetChild(int childnum);
                    FLAG GetExtents(WtP3 &extents);
                    FLAG GetMidpoint(WtP3 &midpoint);
                    float GetRadius(void);
                    WtGeometry *GeometryNodeLoad(char *filename, float scale = 1.f);
                    FLAG InsertChild(WtNode *child, int childnum);
                    FLAG LightNodeLoad(char *filename);
                    WtMovable *MovNodeLoad(char *filename, float scale = 1.f);
                    WtNode *NodeLoad(char *filename, float scale = 1.f);
                    int NumChildren(void);
                    FLAG RemoveChild(int childnum);
};
```

WtAnchor

```
class WtAnchor : public WtGroup
{
    WTanchornode_new  WtAnchor(WtGroup *parent);
                    char *Getlocation(void);
                    FLAG SetLocation(char *link);
};
```

WtInline

```
class WtInline : public WtGroup
{
    WTinlinenode_new  WtInline(WtGroup *parent);
                    char *Getlocation(void);
                    FLAG SetLocation(char *link);
};
```


WtLod

```
class WtLod : public WtGroup
{
WtLodnode_new    WtLod(WtGroup *parent);
                 FLAG GetCenter(WtP3 &center);
                 FLAG GetRange(float *range, int num);
                 int NumRanges(void);
                 FLAG SetCenter(WtP3 center);
                 FLAG SetRange(float *range, int num);
};
```

WtRoot

```
class WtRoot : public WtGroup
{
WtRootnode_new  WtRoot(void);
                 WtRoot *Next(void);
};
```

WtSep

```
class WtSep : public WtGroup
{
WtSepnode_new  WtSep(WtGroup *parent);
                 FLAG Enable(FLAG enable);
                 int GetCullmode(void);
                 FLAG SetCullmode(int mode);
};
```

WtSwitch

```
class WtSwitch : public WtGroup
{
WtSwitchnode_new WtSwitch(WtGroup *parent);
                 int GetWhichChild(void);
                 FLAG SetWhichChild(int which);
};
```

WtXformSep

```
class WtXformSep : public WtGroup
{
WtXformsepnod_new    WtXformSep(WtGroup *parent);
                    FLAG Enable(FLAG enable);
};
```

WtFog

```
class WtFog : public WtNode
{
WtFognode_new    WtFog(WtGroup *parent);
                    FLAG Enable(FLAG enable);
                    FLAG GetColor(float &red, float &grn, float &blue, float &alpha);
                    float GetLinearStart(void);
                    int GetMode(void);
                    float GetRange(void);
                    FLAG SetColor(float red, float grn, float blue, float alpha);
                    FLAG SetLinearStart(float start);
                    FLAG SetMode(int mode);
                    FLAG SetRange(float range);
};
```

Note: In the WTK C API, there is the concept of a geometry and a geometry node. In the WTK C++ API, however, these are combined into one entity — the *WtGeometry* class.

WtGeometry

```
class WtGeometry : public WtNode
{
WTgeometry_newcylinder    WtGeometry(int type, float height, float radius,
                                int tess, FLAG bothsides, FLAG gouraud);
WTgeometry_newblock    WtGeometry(int type, float lx, float ly, float lz,
                                FLAG bothsides);
WTgeometry_newcone    WtGeometry(int type, float height, float radius, int tess,
                                FLAG bothsides);
                    WtGeometry(int type, float radius, int nlat, int nlong,
```

WTgeometry_newsphere FLAG bothsides,FLAG gouraud);

WTgeometry_newrectangle WtGeometry(int type, float height, float width,
FLAG bothsides);

WTgeometry_newtrunccone WtGeometry(int type, float height, float toprad, float baserad,
int tess, FLAG bothsides, FLAG gouraud);

WTgeometry_newextrusion WtGeometry(int type, WtP2 points[],int numpts,float height,
FLAG bothsides,FLAG gouraud);

WTgeometry_begin WtGeometry(int type);

Note: Check the defines on page 26-45 for the type argument.

WTgeometry_newtext3d WtGeometry(WtFont3d *font3d, char *string);

WTgeometry_copy WtGeometry(WtGeometry *copy);
FLAG BeginEdit(void);
WtPoly *BeginPoly(void);

WTnode_boundingbox FLAG BBox(FLAG enable);
FLAG ChangeTexture(char *bitmap, FLAG shaded, FLAG transparent);
FLAG Close(void);
FLAG ComputeVertexNormal(WTvertex *vertex);
FLAG DeletePrebuild(void);
void DeleteTexture(void);
FLAG Enable(FLAG enable);
FLAG EndEdit(void);
FLAG GetExtents(WtP3 & extents);
FLAG GetMidpoint(WtP3 & midpoint);
WtMtable *GetMtable(void);
WtPoly *GetFirstPoly(void);
float GetRadius(void);
int GetRenderingStyle(void);
int GetVertexMatId(WTvertex *vertex);
FLAG GetVertexNormal(WTvertex *vertex, WtP3 & norm);
void GetVertexPosition(WTvertex *vertex, WtP3 & pos);
FLAG GetVertexRGB(WTvertex *vertex, unsigned char *r, unsigned char *g,
unsigned char *b);
WTvertex *GetFirstVertex(void);
WtPoly *Id2Poly(short id);
WtPoly *Merge(WtGeometry *mergewith);
WTvertex *NewVertex(WtP3 p);

WTgeometry_scale

```
int NumPolys(void);
FLAG Prebuild(void);
void RecomputeStats(FLAG clearverts);
void ScaleVerts(float factor, WtP3 center);
FLAG SetMatId(int id);
void SetMtable(WtMtable *table);
FLAG SetRenderingStyle(int modes, int style);
void SetRGB(unsigned char r, unsigned char g, unsigned char b);
FLAG SetTexture(char *bitmap, FLAG shaded, FLAG transparent);
FLAG SetTextureUV(char *bitmap, float (*fu)(WtP3),
                  float (*fv)(WtP3), FLAG shaded, FLAG transparent);
FLAG SetUV(float (*fu)(WtP3), float (*fv)(WtP3));
FLAG SetVertexMatId(WTvertex *vertex, int id);
FLAG SetVertexNormal(WTvertex *vertex, WtP3 norm);
FLAG SetVertexPosition(WTvertex *vertex, WtP3 pos);
FLAG SetVertexRGB(WTvertex *vertex, unsigned char r, unsigned char g,
                  unsigned char b);
void StretchVerts(WtP3 factors, WtP3 center);
void TransformVerts(WtPQ *xform);
void TransformVerts(WtM4 m4);
void TranslateVerts(WtP3 offSet);
};
```

WTgeometry_stretch**WTgeometry_transform****WTgeometry_transform****WTgeometry_translate****WtAmbientLt****WTlightnode_newambient**

```
class WtAmbientLt : public WtNode
{
    WtAmbientLt(WtGroup *parent);
    FLAG Enable(FLAG enable);
    FLAG GetAmbient(float &r, float &g, float &b);
    float GetIntensity(void);
    int GetType(void);
    FLAG Save(char *filename);
    FLAG SetAmbient(float r, float g, float b);
    FLAG SetIntensity(float intensity);
};
```

WtDirectedLt

```

class WtDirectedLt : public WtNode
{
WTlightnode_newdirected    WtDirectedLt(WtGroup *parent);
                            FLAG Enable(FLAG enable);
                            FLAG GetAmbient(float &r, float &g,float &b);
                            FLAG GetDiffuse(float &r, float &g,float &b);
                            FLAG GetDirection(WtP3 &dir);
                            float GetIntensity(void);
                            FLAG GetSpecular(float &r, float &g, float &b);
                            int GetType(void);
                            FLAG Save(char *filename);
                            FLAG SetAmbient(float r, float g, float b);
                            FLAG SetDiffuse(float r, float g, float b);
                            FLAG SetDirection(WtP3 dir);
                            FLAG SetIntensity(float intensity);
                            FLAG SetSpecular(float r, float g, float b);
};

```

WtPointLt

```

class WtPointLt : public WtNode
{
WTlightnode_newpoint    WtPointLt(WtGroup *parent);
                            FLAG Enable(FLAG enable);
                            FLAG GetAmbient(float &r, float &g, float &b);
                            FLAG GetAttenuation(float &atten0,float &atten1,float &atten2);
                            FLAG GetDiffuse(float &r, float &g,float &b);
                            float GetIntensity(void);
                            FLAG GetPosition(WtP3 &pos);
                            FLAG GetSpecular(float &r, float &g, float &b);
                            int GetType(void);
                            FLAG Save(char *filename);
                            FLAG SetAmbient(float r, float g, float b);
                            FLAG SetAttenuation(float atten0,float atten1,float atten2);
                            FLAG SetDiffuse(float r, float g, float blue);
                            FLAG SetIntensity(float intensity);
};

```

```
        FLAG SetPosition(WtP3 pos);
        FLAG SetSpecular(float r, float g, float b);
};
```

WtSpotLt

WtLightnode_newspot

```
class WtSpotLt : public WtNode
{
    WtSpotLt(WtGroup *parent);
    FLAG Enable(FLAG enable);
    FLAG GetAmbient(float &red,float &grn,float &blue);
    float GetAngle(void);
    void GetAttenuation(float &atten0,float &atten1,float &atten2);
    FLAG GetDiffuse(float &red,float &grn,float &blue);
    FLAG GetDirection(WtP3 &dir);
    float GetExponent(void);
    float GetIntensity(void);
    FLAG GetPosition(WtP3 &pos);
    FLAG GetSpecular(float &red, float &grn, float &blue);
    int GetType(void);
    FLAG Save(char *filename);
    FLAG SetAmbient(float red, float grn, float blue);
    FLAG SetAngle(float angle);
    FLAG SetAttenuation(float atten0,float atten1,float atten2);
    FLAG SetDiffuse(float red, float grn, float blue);
    FLAG SetDirection(WtP3 dir);
    FLAG SetExponent(float exponent);
    FLAG SetIntensity(float intensity);
    FLAG SetPosition(WtP3 pos);
    FLAG SetSpecular(float red, float grn, float blue);
};
```

WtXform

```

class WtXform : public WtNode
{
WTxformnode_new    WtXform(WtGroup *parent);
                   WtMotionLink *AddSensor(WtSensor *sensor);
                   FLAG AxisRotation(int axis, float angle, int frame);
                   FLAG GetOrientation(WtQ &ori);
                   FLAG GetRotation(WtM3 &mat);
                   FLAG GetTransform(WtM4 &mat);
                   FLAG GetTranslation(WtP3 &pos);
                   void RemoveSensor(WtSensor *sensor);
                   FLAG RotateM3(WtM3 m3, int frame);
                   FLAG RotateM4(WtM4 m4, int frame);
                   FLAG RotateQ(WtQ rot, int frame);
                   FLAG Rotation(float angley, float anglex, float anglez, int frame);
                   FLAG SetOrientation(WtQ ori);
                   FLAG SetRotation(WtM3 mat);
                   FLAG SetTransform(WtM4 mat);
                   FLAG SetTranslation(WtP3 pos);
                   FLAG Translate(WtP3 pos, int frame);
};

```

WtMovable

```

class WtMovable : public WtXform
{
WTmovnode_axisrotation    FLAG AlignAxis(int axis, WtP3 dir);
                           FLAG Attach(WtNode *child, int childnum);
                           FLAG DeleteAttachment(int childnum);
                           FLAG Detach(int childnum);
                           WtNode *GetAttachment(int attachnum);
                           WtMovable *Instance(WtMovable *parent);
                           int NumAttachments(void);
                           FLAG MovAxisRotation(int axis, float angle);
};

```

WtMovGeometry

```
class WtMovGeometry : public WtMovable, public WtGeometry
{
    WtMovGeometry(int type, float height, float radius,
                  int tess, FLAG bothsides, FLAG gouraud);
    WtMovGeometry(int type, float lx, float ly, float lz,
                  FLAG bothsides);
    WtMovGeometry(int type, float height, float radius, int tess,
                  FLAG bothsides);
    WtMovGeometry(int type, float radius, int nlat, int nlong,
                  FLAG bothsides,FLAG gouraud);
    WtMovGeometry(int type, float height, float width,
                  FLAG bothsides);
    WtMovGeometry(int type, float height, float toprad, float baserad,
                  int tess, FLAG bothsides, FLAG gouraud);
    WtMovGeometry(int type, WtP2 points[],int numpts,float height,
                  FLAG bothsides,FLAG gouraud);
    WtMovGeometry(int type);
    WtMovGeometry(WtFont3d *font3d, char *string);
};
```

WtMovLod

```
class WtMovLod : public WtMovable, public WtLod
{
WTmovelodnode_new    WtMovLod(WtGroup *parent);
};
```

WtMovSep

```
class WtMovSep : public WtMovable, public WtSep
{
WTmovsepnod_new    WtMovSep(WtGroup *parent);
};
```


WtMovSwitch

```

class WtMovSwitch : public WtMovable, public WtSwitch
{
    WtMovSwitch(WtGroup *parent);
};

```

WTmovswitchnode_new

WtMovDirectedLt

```

class WtMovDirectedLt : public WtMovable, public WtDirectedLt
{
    WtMovDirectedLt(WtGroup *parent);
};

```

WTmovlightnode newdirected

WtMovPointLt

```

class WtMovPointLt : public WtMovable, public WtPointLt
{
    WtMovPointLt(WtGroup *parent);
};

```

WTmovlightnode_newpoint

WtMovSpotLt

```

class WtMovSpotLt : public WtMovable, public WtSpotLt
{
    WtMovSpotLt(WtGroup *parent);
};

```

WTmovlightnode_newspot

WTNODEPATH

```

class WtNodePath : public WtBase
{
    WtNodePath(WtNode *leaf, WtNode *root, int instnum);
    ~WtNodePath(void);
    WtMotionLink *AddSensor(WtSensor *sensor,int frame);
    FLAG GetExtents(float ext[2][3]);
    WtNode *GetNode(int num);
    FLAG GetOrientation(WtQ &ori);
};

```

WTnodepath_new
WTnodepath_delete

```
FLAG GetTransform(WtM4 &m4);
FLAG GetTranslation(WtP3 &pos);
int GetTraversal(int *nodes, int max);
FLAG IntersectBBox(WtNodePath *nodepath2);
WtNodePath *IntersectNode(WtNode *node, int occurrence);
FLAG IntersectPoly(WtNodePath *nodepath2);
int NumNodes(void);
void RemoveSensor(WtSensor *sensor);
};
```

WTPATH

```
class WtPath : public WtBase
{
    WtPath_load      WtPath(char *filename);
    WtPath_interpolate WtPath(WtPath *interpolate, int npoints,int method);
    WtPath_copy      WtPath(WtPath *copy);
    WtPath_new       WtPath(void);
    WtPath_delete   ~WtPath(void);
    void AppendElement(WTpathelement *Element);
    short GetConstraints(void);
    WTpathelement *GetCurrentElement(void);
    short GetDirection(void);
    WTpathelement *GetFirstElement(void);
    short GetMode(void);
    int GetPlaySpeed(void);
    int GetSamples(void);
    FLAG GetVisibility(void);
    void InsertElement(WTpathelement *Element);
    FLAG IsPlaying(void);
    FLAG IsRecording(void);
    WtPath *Next(void);
    int NumElements(void);
    void Play(void);
    void Play1(void);
    FLAG Record(void);
    FLAG Record1(void);
    void Rewind(void);
};
```

```

FLAG Save(char *filename);
FLAG Seek(int offSet,int where);
void SetConstraints(short constraints);
FLAG SetCurrentElement(WTpathelement *SetElement);
void SetDirection(short dir);
void SetMode(short mode);
void SetPlaySpeed(int speed);
FLAG SetRecordLink(WtMotionLink *link);
void SetSamples(int frames_per_Element);
void SetVisibility(FLAG flag);
void ShowCurrentElement(void);
void Stop(void);
};

```

WtSENSOR

```

class WtSensor : public WtBase
{
WtSensor_new      WtSensor(int(* openfn)(WTsensor *), void(* closefn)(WTsensor *),
                    void(* updatefn)(WTsensor *), WtSerial *serial,
                    short unit, short location);
WtSensor_new      WtSensor(int(* openfn)(WtSensor *), void(* closefn)(WtSensor *),
                    void(* updatefn)(WtSensor *), WtSerial *serial,
                    short unit, short location);
WtSensor_delete  ~WtSensor(void);
                    float GetAngularRate(void);
                    void GetLastRecord(WtP3 & absolute_p, WtQ & absolute_q);
                    int GetMiscData(void);
                    void *GetRawData(void);
                    void GetRotation(WtQ & rot);
                    float GetSensitivity(void);
                    WtSerial *GetSerial(void);
                    void GetTranslation(WtP3 & pos);
                    short GetUnit(void);
                    WtSensor *Next(void);
                    void RelativizeRecord(WtP3 absolute_p, WtQ absolute_q,
                                             WtP3 & relative_p, WtQ & relative_q);
                    void Rotate(WtQ rot);

```

```
        void SetAngularRate(float v);
        void SetLastRecord(WtP3 absolute_p, WtQ aAbsolute_q);
        void SetMiscData(int x);
        void SetRawData(void *dataptr);
        void SetRecord(WtP3 relative_p, WtQ relative_q);
        void SetSensitivity(float v);
        void SetUpdateFn(void(* updatefn)(WtSensor *));
        void SetUpdateFn(void(* updatefn)(WtSensor *));
};
class WtBaron : public WtSensor
{
WTbaron_new    WtBaron(char *port);
};
class WtBird : public WtSensor
{
WTbird_new    WtBird(char *port, int unit);
                void AutoHemisphere(void);
                void GetAbsoluteRecord(WtP3 absp, WtQ absq);
                int GetHemisphere(void);
                static FLAG InitERC(char *port);
                FLAG SetERCSerial(WtSerial *ert_serial);
                void SetHemisphere(int hemi);
                void SetSync(short type);
};
class WtBoom : public WtSensor
{
WTboom_new    WtBoom(char *port);
};
WTcrystaleyesVR_new class WtCrystalEyesVR : public WtSensor
{
                WtCrystalEyesVR(char *port);
};
class WtFastrak : public WtSensor
{
WTfasttrak_new WtFastrak(char *port, int unit);
                FLAG AFilter(float F, float FLOW, float FHIGH, float FACTOR);
                void AFilterOff(void);
                FLAG PFilter(float F, float FLOW, float FHIGH, float FACTOR);
};
```

```

        void PFilterOff(void);
    };
    class WtGeoball : public WtSensor
    {
WTgeoball_new        WtGeoball(char *port);
                        FLAG Present(WtSerial *serial);
    };
    class WtGlove5DT : public WtSensor
    {
WTglove5DT_new    WtGlove5DT(char *port);
                        void CalibrateClosed(void);
                        void CalibrateOpen(void);
                        int RawUpdate(void);
                        void UpdateFingers(void);
    };
    class WtIGlasses : public WtSensor
    {
WTiglasses_new    WtIGlasses(char *port);
                        int RawUpdate(void);
    };
    class WtIsotrak2 : public WtSensor
    {
WTisotrak2_new    WtIsotrak2(char *port, int unit);
    };
    class WtJoyserial : public WtSensor
    {
WTjoyserial_new    WtJoyserial(char *port);
                        float GetDrift(void);
                        void GetRange(WtP2 range);
                        int RawUpdate(void);
                        void ReadCalibrationFile(void);
                        void SetDrift(float drift);
    };
    class WtLogitech : public WtSensor
    {
WTlogitech_new    WtLogitech(char *port);
    };
    class WtMouse : public WtSensor

```

```
WTmouse_new      {
                  WtMouse(void);
                  void RawUpdate(void);
                  FLAG InWindow(WtWindow *window);
                  void SetTrackballDrift(float drift);
                  void SetTrackballSnap(float snap);
                  void SetTrackballSnapAngle(float snapangle);
                  void TrackballReset (void);
                  void TrackballVPoint(void);
                  WtWindow *WhichWindow(void);
```

Note: The following functions should be used only if the update function being used is WTmouse_trackball

```
                  float GetTrackballDrift(void);
                  float GetTrackballSnap(void);
                  float GetTrackballSnapAngle(void);
```

```
                };
                class WtPinch : public WtSensor
```

```
WTPinch_new      {
                  WtPinch(char *port, int baud);
                  void RawUpdate(void);
```

```
                };
                class WtPolhemus : public WtSensor
```

```
WtPolhemus_new  {
                  WtPolhemus(char *port);
```

```
                };
                class WtSpaceball : public WtSensor
```

```
WTspaceball_new {
                  WtSpaceball(char *port);
                  void Rezero(void);
```

```
                };
                class WtSpacecontrol : public WtSensor
```

```
WTspacecontrol_new {
                  WtSpacecontrol(char *port);
                  void RawUpdate(void);
```

```
                };
```

Note: The following sensors are for WIN32 systems only.

```
class WtCybermaxx2 : public WtSensor
{
WTcybermaxx2_new    WtCybermaxx2(char *port);
                    FLAG RawUpdate(void);
};

class WtFormula : public WtSensor
{
WTformula_new     WtFormula(int unit);
                    void RawUpdate(void);
};
class WtInsidetrakNT : public WtSensor
{
WTinsidetrak_new  WtInsidetrakNT(short unit);
};
class WtPrecision : WtSensor
{
WTprecision_new  WtPrecision(char *port);
                    int RawUpdate(void);
                    int RawUpdate2(void);
};
class WtSpaceballISC : public WtSensor
{
WTspaceballISC_new WtSpaceballISC(char *port);
                    FLAG Rezero(void);
                    FLAG SetWindow(WtWindow *window);
};
```

WTSERIAL

```
class WtSerial : public WtBase
{
    #if WIN32
WTserial_new      WtSerial(char *port, int baud, char parity,
                        int databits, int stopbits, int buffersize);
    #else // Unix
WTserial_new      WtSerial(char *port, int baud);
    #endif // WIN32/Unix
WTserial_delete  ~WtSerial(void);
    short Read(char *data, int length, FLAG retry);
    FLAG SetBaud(int baud);
    short Write(unsigned char *buffer, int length);
    #if WIN32
        int GetBaud(void);
        int GetByteSize(void);
        int NToRead(void);
        FLAG SetByteSize(int size);
        FLAG SetRTS(FLAG Set);
    #else // Unix
        short NToRead(void);
        short Readx(char *data, int length);
        void SetRTS(FLAG Set);
        void SetSize(short size);
    #endif // WIN32/Unix
};
```

WTSOUND

```
class WtSound : public WtBase
{
WTsound_load      WtSound(WtSoundDevice *device, char *source);
WTsound_delete  ~WtSound(void);
    PFVS GetDoneFn(void);
    char *GetName(void);
    WtNodePath *GetNodePath(void);
    float GetParam(int param);
};
```



```

    void GetPosition(WtP3 & pos);
    FLAG IsPlaying(void);
    WtSound *Next(void);
    FLAG Play(void);
    void SetDoneFn(PFVS donefn);
    FLAG SetNodePath(WtNodePath *npath);
    void SetParam(int param, float value);
    void SetPosition(WtP3 pos);
    FLAG Stop(void);
};

```

WTSOUNDDEVICE

Wtsounddevice_open
Wtsounddevice_close

```

class WtSoundDevice: public WtBase
{
    WtSoundDevice(int type, int nplayable, WtViewPoint *vpoint);
    ~WtSoundDevice(void);
    WtViewPoint *GetListener(void);
    float GetParam(int param);
    WtSound *GetFirstSound(void);
    WtSound *Name2Sound(char *name);
    int NumPlayable(void);
    FLAG SetListener(WtViewPoint *viewpoint);
    void SetParam(int param, float value);
    void Update(void);
};

```

WTTASK

WTtask_new
WTtask_delete

```

class WtTask : public WtBase
{
    WtTask(void *ptr, WTtask_function fptr, float priority = 1.f);
    ~WtTask(void);
    FLAG Add(void);
    WTtask_function GetFunction(void);
    float GetPriority(void);
    FLAG Remove(void);
};

```

```
        FLAG SetPriority(float priority);  
    };
```

WtVIEWPOINT

```
class WtViewPoint : public WtBase  
{  
Wtviewpoint_copy    WtViewPoint(WtViewPoint *copy);  
Wtviewpoint_new    WtViewPoint(void);  
Wtviewpoint_delete ~WtViewPoint(void);  
    WtMotionLink *AddSensor(WtSensor *sensor);  
    void AlignAxis(short axis, WtP3 dir);  
    void GetAxis(short axis, WtP3 & vector);  
    float GetConvDistance(void);  
    short GetConvergence(void);  
    void GetDirection(WtP3 & dir);  
    void GetDirectionFrame(WtP3 & dir, WtPQ frame);  
    void GetFrame(WtPQ & frame);  
    void GetLastOrientation(WtQ & ori);  
    void GetLastPosition(WtP3 & pos);  
    void GetOrientation(WtQ & ori);  
    void GetOrientationFrame(WtQ & ori, WtPQ frame);  
    float GetParallax(void);  
    void GetPosition(WtP3 & pos);  
    void GetPositionFrame(WtP3 & pos, WtPQ frame);  
    void IntersectPoly(WTpoly *poly, WtNodePath *npath, float distance);  
    void Local2World(WtP3 pin, WtP3 & pout);  
    void Move(WtPQ move, short frame);  
    void MoveFrame(WtPQ move, WtPQ frame);  
    void MoveTo(WtPQ moveto);  
    void MoveToFrame(WtPQ moveto, WtPQ frame);  
    WtViewPoint *Next(void);  
    void RemoveSensor(WtSensor *sensor);  
    void Rotate(short axis, float rad, short frame);  
    void RotateFrame(short axis, float rad, WtPQ frame);  
    void SetConvDistance(float x);  
    void SetConvergence(short x);  
    void SetDirection(WtP3 dir);
```

```

void SetDirectionFrame(WtP3 dir, WtPQ frame);
void SetOrientation(WtQ ori);
void SetOrientationFrame(WtQ ori, WtPQ frame);
void SetParallax(float x);
void SetPosition(WtP3 pos);
void SetPositionFrame(WtP3 pos, WtPQ frame);
void Translate(WtP3 trans, short frame);
void TranslateFrame(WtP3 trans, WtPQ frame);
void World2Local(WtP3 pin, WtP3 & pout);
};

```

WTWINDOW

```

class WtWindow : public WtBase
{
    WTwindow_new      WtWindow(int x0, int y0, int xsize, int ysize, int flags);
    WTwindow_delete ~WtWindow(void);
    FLAG ContainsPoint(int x, int y);
    void Draw2DCircle(float xc, float yc, float radius, int mode);
    void Draw2DLine(float x1, float y1, float x2, float y2);
    void Draw2DPoint(float x, float y);
    void Draw2DRectangle(float x1, float y1, float x2, float y2, int mode);
    void Draw2DText(float x, float y, char *text);
    void Draw2DTexture(char *name, FLAG transp, WtP2 *xyarray,
                      int xysize, WtP2 *uvarray, int uvsize);
    void Draw3DLines(WtP3 *pts, int npts, int style);
    void Draw3DPoints(WtP3 *pts, int npts);
    void Enable(FLAG enable);
    FLAG Exists(void);
    void Get2DTextExtents(char *string, float *width, float *height);
    void GetBgRGB(unsigned char *r, unsigned char *g, unsigned char *b);
    short GetEye(void);
    float GetHitherValue(void);
    WTwinidtype GetIdx(void);
    FLAG GetImage(int x, int y, int pixels, int scanlines, unsigned char *image);
    void GetParams(FLAG eye, float *left, float *right, float *bottom,
                  float *top, float *nearp, float *farp);
    void GetPosition(int *x0, int *y0, int *width, int *height);

```

```
int GetProjection(void);
FLAG GetRay(WtP2 point, WtP3 & rayorigin, WtP3 & ray);
WtRoot *GetRootNode(void);
int GetScreen(void);
float GetViewAngle(void);
WtViewPoint *GetViewPoint(void);
WtViewPoint *GetViewPoint2(void);
float GetYonValue(void);
FLAG IsEnabled(void);
FLAG LoadImage(char *filename, float zval, FLAG swapbuf, FLAG bitmapdel);
WtWindow *Next(void);
int NumPolys(void);
WtPoly *PickPoly(WtP2 pt2D, WtNodePath **npath, WtP3 & pt3D);
FLAG ProjectPoint(int eye, WtP3 pos, WtP2 point);
void Set2DColor(unsigned char r, unsigned char g, unsigned char b);
void Set2DFont(int fontindex);
void Set2DLineStyle(int linestyle);
void Set2DLineWidth(float width);
void Set3DColor(unsigned char r, unsigned char g, unsigned char b);
void Set3DLineStyle(int linestyle);
void Set3DLineWidth(float lwidth);
void Set3DPointSize(float size);
void SetBgRGB(unsigned char r, unsigned char g, unsigned char b);
void SetDrawFn(void (*drawfn)(WtWindow *, FLAG));
void SetEye(short eye);
void SetFgActions(void (*fgactions)(WtWindow *, FLAG));
void SetHitherValue(float val);
void SetParams(FLAG eye, float left, float right, float bottom,
               float top, float nearf, float farf);
void SetPosition(int x0, int y0, int width, int height);
void SetProjection(int type);
FLAG SetRootNode(WtRoot *rootnode);
void SetViewAngle(float angle);
void SetViewPoint(WtViewPoint *viewpoint);
void SetViewPoint2(WtViewPoint *viewpoint);
void SetYonValue(float val);
void ZoomViewPoint(void);
void ZoomViewToNode(WtNode *zoomnode, int which);
```

```

                                #if WIN32
WTwindow_newuser           WtWindow(HWND parent, int window_config);
                                static void UseWindow(HWND parent);
                                #else /* UNIX */
WTwindow_newuser           WtWindow(Widget parent, int window_config);
                                Widget GetWidget(void);
                                static void UseWindow(Widget parent);
                                #endif /* WIN32 */
                                };
```

Stand-alone Classes

WtDIR

```

                                class WtDirectory
                                {
Wtdirectory_open           WtDirectory(char *path);
Wtdirectory_close         ~WtDirectory(void);
                                char *GetEntry(void);
                                };
```

WtKEYBOARD

```

                                class WtKeyboard
                                {
                                static void Close(void);
                                static void Open(void);
                                static short GetKey(void);
                                static short GetLastKey(void);
                                };
```

WTNET

```
class WtNetwork
{
    static FLAG AddItem(void *buf, int buflen, int type, int tag);
    static FLAG AddString(void *buf, int buflen, int type, int tag);
    static void Close(void);
    static void Flush(void);
    static unsigned short GetPort(void);
    static FLAG GetRange(void);
    static int Next(int *tag, int *retlen);
    static FLAG Open(char *group, unsigned short port, unsigned char range);
    static int RemoveItem(void *buf, int buflen, int *tag, int *retlen);
    static int RemoveString(void *buf, int buflen, int *tag, int *retlen);
    static void Skip(void);
};
```

WTSCREEN

```
class WtScreen
{
    static int GetYBlank(void);
    static FLAG Load(char *filename);
    static WtPoly *PickPoly(int screen, WtP2 pt2D,
        WtNodePath **npath, WtP3 & pt3D);
    static void SetYBlank(int val);
};
```

WTTEXTURE

```
class WtTexture
{
    static FLAG Cache(char *bitmap, FLAG enable);
    static FLAG GetFilter(char *bitmap, int *magfilter, int *minfilter);
    static int GetMemory(void);
    static FLAG IsCached(char *bitmap);
    static unsigned char *Load(char *bitmap, int *width, int *height);
    static FLAG Replace(char *bitmap, int format, int width,
```

```

        int height, unsigned char *image);
    static FLAG SetFilter(char *bitmap, int magfilter, int minfilter);
};

```

WTUNIVERSE

```

class WtUniverse
{
    static float AvgFrameRate(int samples);
WTuniverse_delete    static void Delete(void);
    static void DeleteLink(void *source, void *tarGet);
    static WtNode *FindNodeByName(char *name, int occurence);
    static int FrameCount(void);
    static float FrameRate(void);
    static void GetBgRGB(unsigned char *r, unsigned char *g, unsigned char *b);
    static int GetCurrScrIdx(void);
    static WtWindow *GetCurrWindow(void);
    static WTwinidtype GetCurrWinIdx(void);
    static short*GetEventOrder(void);
WTuniverse_getmotionlinks    static WtMotionLink *GetFirstMotionLink(void);
WTuniverse_getpaths        static WtPath *GetFirstPath(void);
WTuniverse_getrootnodes    static WtRoot *GetFirstRootNode(void);
WTuniverse_getsensors     static WtSensor *GetFirstSensor(void);
WTuniverse_getviewpoints   static WtViewPoint *GetFirstViewPoint(void);
WTuniverse_getwindows     static WtWindow *GetFirstWindow(void);
WTuniverse_getbases       static WtBase *GetFirstBase(void);
    static void GetInitialView(WtPQ & pos);
    static int GetOption(int option);
    static FLAG GetRendering(void);
    static float GetSubfaceOffset(void);
    static WtTask *GetTaskByPointer(void *ptr, int whichtask);
    static void Go(void);
    static void Go1(void);
WTuniverse_new           static void New(int display_config, int window_config);
    static void Ready(void);
    static void ResetFrameCount(void);
    static void ResetTime(void);
    static void SetActions(void(* actions)(void));

```

```
static void SetBBoxRGB(float r, float g, float b);
static void SetBgRGB(unsigned char r, unsigned char g, unsigned char b);
WTinit_setdefaults static FLAG SetDefaults(int *argc, char **argv);
static FLAG SetEventOrder(short nevents, short * events);
WTinit_setimages static void SetImages(const char *paths);
WTinit_setmodels static void SetModels(const char *paths);
static void SetOption(int option, int value);
static void SetRendering(FLAG style);
static void SetSubfaceOffset(float val);
static void SetViewPoint(WtViewPoint *vpoint);
static void Stop(void);
static float Time(void);
WTnode_vacuum static void Vacuum(void);
static void ProcessEvents(void);
};
```

Note: The following classes are only available if linking with the World2World wrapper library.

WTCONNECTION

WtConnection Callback function prototype:

```
typedef FLAG(*WtConnCb)(
    WtConnection *conn,
    Wtconnevent event,
    void *data1,
    void *data2,
    double time);

class WtConnection {
    WtConnection(const char *host, unsigned short port, const char *username =
        NULL, const char *password = NULL);
    ~WtConnection(void);
    virtual void *GetWTKStructure(void) {return (void*)connection;}
    void SetData(void *data) {userdata = data;}
    void *GetData(void) {return userdata;}
    static WtConnection *GetFirst(void);
};
```



```
WtConnection *Next(void);
static void DeleteAll(void);
FLAG Connect(void);
FLAG Disconnect(void);
unsigned int GetMyId(void);
const char *GetMyName(void);
int GetStatus(void);
void Print(void);
void Update(void);
static void UpdateAll(void);
void Synch(void);
double GetLatency(void);
double GetClockDiff(void);
void SetSynchronous(FLAG synchronous);
FLAG IsSynchronous(void);
void SetUpdateRate(unsigned short freq);
unsigned short GetUpdateRate(void);
void AddCallback(WtConnCb cb);
void RemoveCallback(WtConnCb cb);
int NumCallbacks(void);
WtConnCb GetCallback(int callbacknum);
WtSharegroup *GetRoot(void);
unsigned int NumUsers(void);
unsigned int GetUserId(unsigned int usernum);
const char *GetUserName(unsigned int usernum);
unsigned int GetUserIdByName(const char *username);
const char *GetUserNameById(unsigned int userid);
void DeleteAllEnumTrees(void);
void DeleteEnumTreeById(unsigned int enumid);
WtBase *GetEnumTreeById(unsigned int enumid);
unsigned int NumEnumTrees(void);
WtBase *GetEnumTree(unsigned int nenumtree);
unsigned int GetEnumTreeId(unsigned int nenumtree);
};
```

WTSHAREGROUP

```
class WtSharegroup {
    WtSharegroup(const char *name, WtSharegroup *parent = NULL,
                 int shareflags = 0);
    WtSharegroup(const char *name, WtConnection *conn = NULL,
                 int shareflags = 0);
    ~WtSharegroup(FLAG forcedelete);
    virtual void *GetWTKStructure(void) {return (void*)shgrp;}
    void SetData(void *data) {userdata = data;}
    void *GetData(void) {return userdata;}
    void Print(FLAG children, FLAG properties);
    WtConnection *GetConnection(void);
    const char *GetName(void);
    FLAG Lock(void);
    FLAG Unlock(void);
    unsigned int IsLocked(void);
    FLAG IsLockedByMe(void);
    void RegisterInterest(FLAG interested);
    WtSharegroup *GetParent(void);
    int NumChildren(void);
    WtSharegroup *GetChild(int childnum);
    int NumProperties(void);
    const char *GetProperty(int propertynum, void **object);
    unsigned int Enumerate(FLAG recursive, FLAG properties);
};
```

Math Classes

WtM3

```
class WtM3
{
```

Note: The following constructor creates a *WtM3* initialized to the 3x3 array called *array*.

```

WtM3(float array[3][3]);
WtM3(void);
~WtM3(void);
void Euler2M3(float rx, float ry, float rz);
void M32Euler(WtP3 & first, WtP3 & second);
void M32EulerNear(WtP3 nearp, WtP3 & euler);
void M32Q(WtQ & q);

```

WTm3_int

Note: The following method sets the *WtM3* to the identity matrix.

```

void SetToIdentity(void);
WtM3 Transpose(void);
WtM3 operator*(WtM3 & second);

```

WTm3_multm3

Note: The following method allows access to matrix elements (e.g., *m3[2][2]*)

```
Row3 & operator[](int i);
```

Note: The following method prints a *WtM3* (e.g., *cout<<M3*).

```

friend ostream & operator<<(ostream & os, WtM3 & print);
};

```

WTM4

```
class WtM4
{
```

Note: The following constructor creates a *WtM4* initialized to the 6x6 array called *array*.

```
WtM4 (float array[4][4]);
WTm4_init WtM4(void);
~WtM4(void);
int Invert(WtM4 & dst);
void M42PQ(WtPQ & pq);
void RotateP3(WtP3 vecin, WtP3 & vecout);
```

Note: The following method sets the *WtM4* to the identify matrix.

```
void SetToIdentify(void);
void TransformP3(WtP3 pin, WtP3 & pout);
WtM4 Transpose(void);
WTm4_multm4 WtM4 operator*(WtM4 & second);
```

Note: The following method allows access to matrix elements (e.g., *m4[2][2]*)

```
Row4 & operator[](int i);
```

Note: The following method prints a *WtM4* (e.g., *cout<<M4*).

```
friend ostream & operator<<(ostream & os, WtM4 & print);
};
```

WTP2

```
class WtWindow;
class WtP2
{
```

Note: The following constructor creates a *WtP2* initialized to *x* and *y*.

```
WtP2(float x, float y);
```

Note: The following constructor creates a *WtP2* initialized to 0 (zero).

```
WtP2(void);
```

Note: The following constructor creates a *WtP2* initialized to array [0] and array [1].

```
WtP2 (float array [2]);
~WtP2(void);
void Initialize(void)
double Mag(void);
void Norm(void);
```

Note: The following method sets a *WtP2* to *x* and *y*.

```
void Set(float x, float y);
class WtWindow *WhichWindow(void);
WtP2 operator-(WtP2 & second);
```

Wtp2_subtract

Note: The following method allows access to array elements (e.g., *p2[1]*)

```
float & operator[](int i);
friend float Dot(WtP2 & first, WtP2 & second);
```

Note: The following method prints a *WtP2* (e.g., *cout<<P2*).

```
friend ostream & operator<<(ostream & os, const WtP2 & print);
};
```

WtP3

```
class WtP3
{
```

Note: The following constructor creates a *WtP3* initialized to *x*, *y*, and *z*.

```
WtP3(float x, float y, float z);
```

Note: The following constructor creates a *WtP3* initialized to array[0], array[1], array[2];

WtP3_init WtP3(float array[3]);
WtP3(void);
~WtP3(void);
static short Coplanar(WtP3 *verts, int nverts, WtP3 & normal);
float DistToVector(WtP3 ptondir, WtP3 dir);
void Frame2Frame(WtPQ frame1, WtPQ frame2, WtP3 & pout);

WtP3_init void Initailize(void);
void Local2WorldFrame(WtPQ frame, WtP3 & pout);
double Mag(void);
void Norm(void);
float Normal2Slope(void);
WtP3 Rotate(WtQ rotate);
WtP3 RotatePt(WtQ rotate, WtP3 point);

Note: The following method sets a WtP3 to x, y, and z.

void Set(float x, float y, float z);
void World2LocalFrame(WtPQ frame, WtP3 & pout);
WtP3 Transform(WtPQ xform);
WtP3_equal FLAG operator==(WtP3 & second);
WtP3_add WtP3 operator+(WtP3 & second);
WtP3_subtract WtP3 operator-(WtP3 & second);
void operator+=(const WtP3 & second);
void operator-=(const WtP3 & second);

WtP3_invert WtP3 operator~(void);

Note: The following method allows access to array elements (e.g., P3[2]).

WtP3_multm3 float & operator[](int i);
WtP3 operator*(WtM3 & m);
WtP3_multm4 WtP3 operator*(WtM4 & m);

*Note: The following method allows multiplication of a WtP3 by a scalar (e.g., P3 *scalar).*

WtP3_mults WtP3 operator*(float scalar);
WtP3 operator/(float div);

Note: The following method allows multiplication of a scalar by a *WtP3* (e.g., scalar **P3*).

```
WtP3_mults      friend WtP3 operator*(float scalar, WtP3 & vector);
                  friend WtP3 Cross(WtP3 & first, WtP3 & second);
                  friend float Dot(WtP3 & first, WtP3 & second);
                  friend float Distance(WtP3 & first, WtP3 & second);
                  friend FLAG Equal(WtP3 & first, WtP3 & second);
```

Note: The following method prints a *WtP3* (e.g., cout<<*P3*).

```
friend ostream & operator<<(ostream & os, const WtP3 & print);
};
```

WTQ

```
class WtQ
{
```

Note: The following constructor creates a *WtQ* initialized to *x*, *y*, *z*, and *w*.

```
WtQ(float x, float y, float z, float w);
```

Note: The following constructor creates a *WtQ* to array[0], array[1], array[2], and array[3].

```
WtQ(float array[4]);
WTq_init      WtQ(void);
                  ~WtQ(void);
                  void Dir2Q(WtP3 dir);
                  void DirAndTwist2Q(WtP3 dir, float twist);
                  void Euler2Q(float wx, float wy, float wz);
                  void Frame2Frame(WtPQ frame1, WtPQ frame2, WtQ & qout);
                  float GetAngle(void);
WTq_init      void Initialize(void);
                  void Local2WorldFrame(WtPQ frame, WtQ & qout);
                  double Mag(void);
                  void Norm(void);
                  void Q2Dir(WtP3 & dir);
```

```
void Q2DirAndTwist(WtP3 & dir, float *twist);  
void Q2Euler(WtP3 & first, WtP3 & second);  
void Q2EulerNear(WtP3 nearp, WtP3 & euler);  
void Q2M3(WtM3 & m);  
void Q2M4(WtM4 & m);  
void Scale(float scale, WtQ & qout);
```

Note: The following method sets a *WtQ* to *x*, *y*, *x*, and *w*.

```
void Set(float x, float y, float z, float w);  
void World2LocalFrame(WtPQ frame, WtQ & qout);  
WtQ_invert WtQ operator~(void);  
WtQ_mult WtQ operator*(WtQ & second);
```

Note: The following method allows access to array elements (e.g., *Q[2]*).

```
WtQ_equal float & operator[](int i);  
FLAG operator==(WtQ & second);  
friend float Dot(WtQ & first, WtQ & second);  
friend FLAG Equal(WtQ & first, WtQ & second);  
friend WtQ Interpolate(WtQ & first, float u, WtQ & second);  
friend WtQ Multinv(WtQ & first, WtQ & second);
```

Note: The following method prints a *WtQ* (e.g., *cout<<Q*).

```
friend ostream & operator<<(ostream & os, const WtQ & print);  
};
```

WTPQ

```
class WtPQ  
{  
    WtP3 p;  
    WtQ q;
```

Note: The following constructor creates a *WtPQ* initialized to *pthree* and *que*.

```
WtPQ(WtP3 pthree, WtQ que);
```

Note: The following constructor creates a *WtPQ* with *p* initialized to 0,0,0, and *q* initialized to 0,0,0,1.

```
WtPQ(void);
```

Note: The following constructor creates a *WtPQ* initialized to arrays *parray* and *qarray*.

```
WtPQ(float parray[3]; float qarray[4]);
~WtPQ(void);
void Frame2Frame(WtPQ frame1, WtPQ frame2, WtPQ & pqout);
```

Note: The following method sets the *p* component of *WtPQ* to 0,0,0, and the *q* component of *WtPQ* to 0,0,0,1.

```
void Initialize(void);
void Local2WorldFrame(WtPQ frame, WtPQ & pqout);
void PQ2M4(WtM4 & m);
```

Note: The following method sets the *WtPQ* to *pthree* and *que*.

```
void Set(WtP3 pthree, WtQ que);
void World2LocalFrame(WtPQ frame, WtPQ & pqout);
```

Note: The following method prints a *WtPQ* (e.g., `cout<<PQ`).

```
friend ostream & operator<<(ostream & os, const WtPQ & print);
};
```

Defines

Note: The following defines are used as the “type” argument in the *WtGeometry* and *WtMovGeometry* constructors.

```
#define WTBLOCK          1
#define WTCONE          2
#define WTCUSTOM        3
#define WTCYLINDER     4
#define WTEXTRUSION    5
```

```
#define WTHEMISPHERE 6
#define WTRECTANGLE 7
#define WTSPHERE 8
#define WTRUNCONE 9
```

Frequently Asked Questions

Introduction

WTK Release 6/7 has many advanced features for developing high-performance, real-time 3D graphics applications. This appendix provides answers to some common questions on how to use many of these powerful features.

Specifically, it focuses on the following questions:

<i>What Is The Difference Between <code>WTnode_load</code> And <code>WTgeometrynode_load</code>?</i>	<i>Page A-3</i>
<i>What Is The Difference Between <code>WTmovnode_load</code> and <code>WTnode_load</code>?</i>	<i>Page A-4</i>
<i>How Do I Display Multiple Instances Of An Object?</i>	<i>Page A-5</i>
<i>How Do I Pick The Frontmost Polygon At A Specific Point In A Specific Window?</i>	<i>Page A-6</i>
<i>Can WTK Detect Keyboard Events?</i>	<i>Page A-8</i>
<i>How Can I Detect Button Events Using The “Misc Data” Functions?</i>	<i>Page A-10</i>
<i>How Do I Use Material Tables for Colors?</i>	<i>Page A-11</i>
<i>How Do I Get Transparencies In A Texture?</i>	<i>Page A-12</i>
<i>How Do I Dynamically Change The Appearance Of A Geometry?</i>	<i>Page A-13</i>
<i>How Do I Create Special Effects:Clouds, Missile Trails, Explosions, Etc...</i>	<i>Page A-13</i>
<i>How Do I Load Lights As Movable?</i>	<i>Page A-15</i>
<i>How Do I Make An Object Follow A Light?</i>	<i>Page A-16</i>
<i>How Do I Make An Object Follow The Viewpoint?</i>	<i>Page A-16</i>
<i>How Do I Recursively “Walk” Down The Scene Graph?</i>	<i>Page A-19</i>

How Do I Get A Pointer To A Node Using Its Name?..... Page A-20

How Do I Associate A Task With a Particular Object?..... Page A-21

How Do I Handle Portals In This Release?..... Page A-22

*How Do I Test For Intersections Between The Viewpoint
And The Universe?..... Page A-25*

*How Do I Test For Objects Intersecting With
Other Objects In The Universe?..... Page A-25*

How Do I Get The Rendered Position Of An Object?..... Page A-25

How Do I Create A Simple Animation Using Switch Nodes?..... Page A-26

How Can I Optimize Performance Using LOD Nodes?..... Page A-29

What Is Terrain Following?..... Page A-31

*How Do I Keep An Object Perpendicular To
The Viewpoint Direction At All Times?..... Page A-33*

How Do I Change The Event Order?..... Page A-34

*How Do I Integrate A WTK Rendering Window
With A Host-Specific Window?..... Page A-35*

*How Do I Use Orientation-Tracking Sensors (On A Head-Mount-Display)
That Are Not Positioned Along The Central Axis Of The HMD?..... Page A-36*

How Do I Measure Performance On My Machine?..... Page A-38

*On UNIX Platforms, How Do I Get A Pointer To The Display
That WTK Is Using?..... page A-38*

*How do I use Boston Dynamic's DiGuy with WTK (or any other BDI character
set)?..... Page A-39*

What Is The Difference Between *WTnode_load* And *WTgeometrynode_load*?

Both *WTgeometrynode_load* (see page 4-46) and *WTnode_load* (see page 4-46) read in geometry data from a file. If the file has data concerning a single geometry only, these functions are equivalent, in that, a single geometry node is created. It is when the file has definitions for multiple geometries that these functions produce different results.

WTgeometrynode_load merges all the geometries into one and creates a single geometry node. *WTnode_load* treats each individual geometry separately and creates a geometry node for each one. This way you have as many geometry nodes as there were geometries in the file.

For example, consider a data file that has geometry information for an office model. Suppose the file contains a geometry that defines the office layout, a geometry for a chair, a geometry for a desk, and a table lamp. When you use *WTgeometrynode_load* to read this file, you create a single node that contains the polygonal information for all the mentioned objects. On the other hand, when you use *WTnode_load* you are creating a node for the office, a node for the chair, one for the desk, and one for the table lamp. Knowing this difference between these two functions, you can decide when to choose one function over the other; the choice is purely application dependent.

You would usually use *WTgeometrynode_load* to load static geometry. For instance, in the previous example, after reading in the geometry data, if you wanted the chair to revolve about its axis, it would not have been possible if you had used *WTgeometrynode_load*. However, *WTnode_load* would have created a separate node for the chair and you could then set a behavior to it, independent of the rest of the objects in the office.

WTgeometrynode_load is unavailable for filetypes that contain hierarchical information. (e.g., MultiGen's FLT files, and VRML files.) This is because *WTgeometrynode_load* creates *only* geometry nodes; all other node types are ignored and the hierarchical information is lost.

Use *WTnode_load* (or *WTmovnode_load* see page 5-5) for hierarchical filetypes. Also, an added feature while using *WTnode_load* is that you can specify the source of data to be an http URL pointing to a VRML file, instead of having to read data off of your local disk only.

What Is The Difference Between *WTmovnode_load* and *WTnode_load*?

The basic difference between the *WTmovnode_load* (see page 5-5) and *WTnode_load* (see page 4-46) functions is that *WTmovnode_load* creates a movable node out of the data it reads from a file. (See page 5-1, for the structure of a movable node.) Using a movable node is sometimes more convenient because you can add behaviors to the node directly – without having to introduce additional transform nodes.

To illustrate this, consider the following example. Suppose you have a model of an airplane in a data file. When you use *WTnode_load* to read this file, you are – in effect – creating a geometry node. Since the position and orientation of geometry nodes is effected by transform nodes you would have to insert a transform node before the geometry if you wanted the airplane model to move in your virtual world. You could then apply translation and rotation functions on the transform node to move the airplane. The use of a transform node in your scene graph means that you must insure that other geometry nodes are not also affected by the addition of a transform node; i.e. you may need to use separator or transform separator nodes to prevent the transform node from affecting other geometry nodes.

The above task is simplified if you use a movable geometry node in place of a regular geometry node. If you use *WTmovnode_load* you are creating a “movable-airplane.” The advantage of this, is that the movable geometry node has a built-in transform component, so the node translation and rotation functions are directly applicable to the movable. You do not have to insert a transform node. The movable is a self-contained entity so you do not have to be concerned about transform information “leaking” to other parts of the scene graph.

If certain geometry nodes do not move during a simulation, you should load those geometry nodes using *WTnode_load*. This optimizes memory usage, since the movable node’s separator component and transform component are no longer needed and hence additional memory is not allocated for those components.

Remember, however, if the data file has multiple geometries in it, *WTmovnode_load* creates *one* movable node representing all the geometries. You still have individual nodes for each geometry, but since they are all part of a single movable node, they move as one rigid entity. (See the *Movables* chapter, starting on page 5-1 for more information.)

How Do I Display Multiple Instances Of An Object?

WTK Release 6/7 introduces *instancing* (see page 4-37 of the *Scene Graphs* chapter for more information on this concept). Basically, instancing allows you to use multiple representations of the same object, without having multiple copies of the object (as was required in WTK V2.1).

For example, suppose you want to create a fence consisting of 50 planks. Assuming you have the model of the plank, you can use the *WTmovnode_load* (see page 5-5) function to load the plank once, and then use the *WTmovnode_instance* (see page 5-13) function to create 49 instances. (Compare this to WTK V2.1 where you would have to load the plank 50 times, resulting in 50 copies of the identical plank being stored.)

Note the following when using *WTmovnode_instance*:

- The node that is to be instanced must be a movable node.
- Each instance has its own separator and transform component but shares the content of the base node, i.e., the node that was instanced, (figure 5-1 on page 5-1).
- Since each instance has its own transform component, you can apply unique transformations to each instance, so that each instance appears at the desired location on the WTK rendering window. If you do not apply transformations to the individual instances, you would see just the base node, because the instances would overlap one another. The following function, creates a fence consisting of 50 planks and displays them with a spacing of 10 units. It assumes that you have already created the base movable node corresponding to the object that you want to instance (in this case a plank).

EXAMPLE CODE

```
void Create_Instances(WTnode *base_movable, int num_of_instances)
{
    int i;
    WTP3 p3;
    WTP3_init(p3);
    for (i = 0; i < num_of_instances; i++)
    {
```

```
        WTnode *instance;
        instance = WTmovnode_instance(root, base_movable);
        p3[X] +=10;
        WTnode_settranslation(instance, p3);
    }
}
```

Be careful when instancing. Because you are loading the object multiple times, you may get different results than you expect.

Suppose, you want the fence to be multicolored (i.e., ten red planks, ten blue, ten green, ten yellow, and ten white). In the above example code, the color you set for the one plank applies to all the instances of that plank. Thus, you would not be able to achieve a multicolored fence with this code. To achieve a multicolored fence, you would load the plank five times to get five different planks, set the colors of these planks, and then instance each one nine times to give you ten planks of each color.

How Do I Pick The Frontmost Polygon At A Specific Point In A Specific Window?

WTK provides two functions, *WTscreen_pickpoly* (see page 4-91) and *WTwindow_pickpoly* (see page 17-20), to let you pick the frontmost polygon rendered at a specified 2D point in a specified screen or window. The difference between these two functions is in the way you specify the 2D point.

WTscreen_pickpoly takes the 2D point in screen coordinates whereas *WTwindow_pickpoly* takes the 2D point in window coordinates. Screen coordinates are specified as 2D floating point values, with (0.0, 0.0) representing the top-left corner of the screen, and (screen width-1, screen height-1) representing the bottom-right corner of the screen. Window coordinates on the other hand are 2D floating point values, with (0.0, 0.0) representing the top-left corner of the *WTwindow*, and (window width-1, window height-1) representing the bottom-right corner of the window.

These functions provide the following information:

- the intersected polygon
- the 3D point (in world coordinates) at which the polygon is intersected
- a node path indicating which instance of the node the picked polygon is associated with

If you are not interested in the node path information you can pass in NULL for the node path argument.

It is often the case where you want to obtain the node corresponding to the polygon picked. The following example code retrieves this node using *WTscreen_pickpoly* and *WTwindow_pickpoly* respectively.

Note: If you want to obtain the node corresponding to the polygon picked, you cannot pass in NULL for the node path argument.

EXAMPLE CODE

```
WTnode *Screen_Pickpoly(WTsensor *mouse)
{
    WTmouse_rawdata *raw;
    WTnodepath *np;
    WTpoly *poly;
    WTnode *node = NULL;
    WTp3 p;
    /* gets the mouse raw data structure (note typecasting)*/
    raw = (WTmouse_rawdata *)WTsensor_getrawdata(mouse);

    /* returns the polygon under the mouse cursor */
    poly = WTscreen_pickpoly(WTuniverse_getcurrscridx(),
        raw->pos, &np, p);

    /* if a polygon was picked, return the node corresponding to the polygon
    picked. This is the last node on the node path, because the node path
    starts at the root node and ends at the node that was picked */
    if ( poly )
```

```
        node = WTnodepath_getnode( np, WTnodepath_numnodes(np)-1);
    return node;
}

WTnode *Window_Pickpoly(WTwindow *w)
{
    int x0, y0, width, height;
    WTnodepath *np;
    WTpoly *poly;
    WTnode *node = NULL;
    WTp3 p;
    WTp2 point;
    WTwindow_getposition(w, &x0, &y0, &width, &height);
    point[X] = width/2.f;
    point[Y] = height/2.f;
    /* picks the frontmost polygon in the center of the window */
    poly = WTwindow_pickpolygon(w, point, &np, p);
    /* if a polygon was picked, it returns the node corresponding to the polygon
    picked. Again, this is the last node on the node path because the
    node path starts at the root node and ends at the node that was picked */
    if ( poly )
        node = WTnodepath_getnode( np, WTnodepath_numnodes(np)-1);
    return node;
}
```

Can WTK Detect Keyboard Events?

Yes, WTK lets you detect keyboard events. This is helpful when you need to trigger certain events depending on the key pressed. Most of the demonstration programs in the demo directory on your WTK distribution make use of keyboard input to trigger events.

The basic steps are as follows:

1. Call *WTkeyboard_open* (see page 24-1) in your main function before calling *WTuniverse_go* (see page 2-7).
2. Call *WTkeyboard_getkey* (see page 24-2) or *WTkeyboard_getlastkey* (see page 24-2) in the universe action function.
3. Use *WTkeyboard_close* (see page 24-3), if you have previously called *WTkeyboard_open*, but no longer need to read input from the keyboard.

The following example code illustrates the above concept.

EXAMPLE CODE

```
int main(int argc, char *argv[])
{
    ...
    ...
    WTuniverse_setactions(action);
    WTkeyboard_open();
    WTuniverse_go();
    return 0;
}
void action(void)
{
    short key;
    key= WTkeyboard_getlastkey();
    switch(tolower(key))
    {
        case 'q':
            WTuniverse_stop();
            break;
    }
}
```

How Can I Detect Button Events Using the “Misc Data” Functions?

For those times when you want to perform certain actions based on button events pertaining to various sensors, you can use WTK’s *WTsensor_getmiscdata* function (see page 13-15).

For example you can detect a left-button press on the mouse or a “button transitioned down” event (generated each time a button moves from up to down position) on a Spaceball.

The following example detects whether the left mouse button was pressed, using the *WTsensor_getmiscdata* function.

EXAMPLE CODE

```
void Read_Mouse_Record(WTsensor *mouse)
{
    int buttons;
    FLAG leftbutton = FALSE;
    /* get button press data */
    buttons = WTsensor_getmiscdata(mouse);
    /* checks whether left mouse button was pressed */
    leftbutton = buttons & WTMOUSE_LEFTBUTTON;
    if (leftbutton)
        /* add the relevant code here */
}
```

The WTK defined constants used with *WTsensor_getmiscdata* for each device are described in the corresponding sections of the Sensors chapter (starting on page 13-1), and are also listed in Appendix B, *Defined Constants*.

How Do I Use Material Tables for Colors?

The concept of a material table is new to Release 6/7. In previous releases, there was only the concept of an RGB color for each polygon or vertex. In Release 6/7, however, every geometry is associated with a material table. The material table for a geometry has entries that correspond to all the colors that the geometry uses. Each polygon (or vertex) has a material ID that references an entry in the material table. This is the way polygons and vertices get their colors.

Each ID can have one or more of the following properties: ambient, diffuse, specular, shininess, emissive, and opacity. There is also an ambient–diffuse property (a combination of the ambient and diffuse properties) which is equivalent to setting the RGB color in previous versions of WTK (see the *Materials* chapter, starting on page 8-1, for a detailed description of all the material properties). You can use the convenient functions *WTPoly_setmatid* (see page 7-3) and *WTPoly_getmatid* (see page 7-3) to set or get the material ID of a particular polygon. For example, to set the properties of polygon *B* to that of polygon *A* you would do the following:

```
WTPoly_setmatid(B, WTPoly_getmatid(A));
```

Note that this yields the desired effects only if both polygons are contained within the same geometry (that is, if they reference the same material table), or in the case where the polygons are contained in different geometries, but the geometries refer to the same material table.

To set the property of the entire geometry, *geom*, to that of polygon, *poly*, you would do the following:

```
WTgeometry_setmatid(geom, WTPoly_getmatid(poly)); (see page 6-31)
```

Now consider the following code segment that uses *WTPoly_setrgb* and *WTPoly_getrgb* (see page 7-2),

```
unsigned char r, g,b;  
WTPoly_getrgb(A, &r, &g, &b);  
WTPoly_setrgb(B, r, g, b);
```

The above code segment only copies the ambient–diffuse property of polygon *A* to that of polygon *B*. None of the other polygon properties change. If you want a polygon to have all the material properties of another polygon, you need to use material IDs.

You can edit a material table to make appropriate changes. For example, suppose you have a geometry that is red in color and you want some of the geometry’s polygons to be yellow. You should use *WTmtable_newentry* to add a new entry (which corresponds to the “yellow” color) into the material table. Then, set the material IDs of the relevant polygons to point to this new entry. See the *Materials* chapter, starting on page 8-1.

How Do I Get Transparencies In A Texture?

WTK lets you enable transparency within portions of a texture. If a texture’s transparency is enabled, the texture image is not rendered where the portions of the image have pixels set to black. So, to make a texture transparent, first color the pixels black, then enable the transparent flag in the appropriate texture application function, such as *WTPoly_settexture* (see page 10-11), *WTgeometry_settexture* (see page 10-12), *WTPoly_settextureuv* (see page 10-13), or *WTgeometry_settextureuv* (see page 10-15).

You can also assign textures to polygons from within 3D file formats, and designate those textures to be transparent (see *Assigning Textures in 3D File Formats* on page 10-22).

Further you can set the environment variable *WTKALPHATEST* to get transparencies in a texture (see the section on the *WTKALPHATEST* environment variable in your WTK Hardware Guide for more information). Note also that textures can be shaded, transparent, or both.

How Do I Dynamically Change The Appearance Of A Geometry?

WTK provides functions to edit geometries at the vertex level, thus allowing you to dynamically change the appearance of a geometry, or to “morph” a geometry into another. See *Vertex-level Geometry Editing* on page 6-42.

Keep in mind, that you need to make calls to the vertex editing functions in between your *WTgeometry_beginedit* and *WTgeometry_endedit* calls. A call to *WTgeometry_beginedit* informs WTK that you are going to edit a particular geometry. A call to *WTgeometry_endedit* marks the end of the editing of the geometry and WTK updates the internal state of the geometry, re-calculates polygon normals if necessary, and so on. The morph demonstration program (*morph.c*) in the WTK demo directory contains an example of how to use these functions.

How Do I Create Special Effects: Clouds, Missile Trails, Exhaust and Explosions

The Missiles demo program (in the demo directory of the CD) provides a few possible solutions to the problem of creating models of gaseous or particulate effects such as smoke, explosions and clouds. The demonstration is presented as a simple space combat game. The game shows the use of a 3D star-field, spaceship exhaust plumes, missile exhaust trails, explosions and clouds of mysterious red gas. The following is a short explanation of the effects and their implementation.

Gas Clouds

The red clumps of gas floating about the scene consist of an array of separate elements. Each element is an OpenGL callback node that draws a textured, translucent rectangle. To properly composite the numerous translucent polygons that make up a single cloud, OpenGL calls are required to manipulate the drawing state. When the universe is created, an initialization function loads the texture and creates an OpenGL display list. The display list is a series of OpenGL commands that can be executed multiple times. It is more

efficient than executing the commands explicitly every time they are needed. Instead, the callback node executes the display list when it is traversed by the scene graph manager. Within the display list, the z-buffer is set to a read-only state. This means that the geometry will be obscured by objects already drawn to the screen but it will not obscure objects that are drawn after it is added to the scene. This allows each element to combine its color values with all other values drawn before and after it. The disadvantage of this technique is that the elements are not depth sorted. If all other objects are drawn before the clouds and the elements are very transparent, the lack of depth sorting is not obvious though. The blending method is set and the textured rectangle is drawn with OpenGL calls in the display list as well. The texture is a 32-bit targa image of a cloud with an alpha channel. The alpha channel is a rough circle with smoothly graduated edges. To finish the effect, a billboard task is attached to the node so that it is always facing the viewpoint.

See `cloud.c` and `cloud.h`

Missile plumes

The missile plumes are identical to the clouds except for their positioning. A task is added to the missile node that simply creates a billboarded callback node at the current position of the missile each frame. The elements also have a task that tracks the age of the node and removes it after a set time. The texture is the same texture used for the cloud elements.

See `trail.c` and `trail.h`

Spaceship exhaust

The spaceship exhaust is handled a little differently from that of the missiles. The user defined drawing function is used to draw the exhaust. Translucent 3D points are drawn along the path of the emitting object. The points are scaled according to their distance from the viewpoint. Since the points are not members of the scene graph, a linked list of structs tracking the position of clumps of points is maintained. The list also allows the tracking of the structs, or 'puffs' according to their age. A simple algorithm is used to set the opacity and color of the puffs in relation to their age in frames since creation. There is an option to toggle point smoothing on and off. Point smoothing makes points appear as solid circles instead of squares. Depending upon the hardware and the platform's implementation of OpenGL, point smoothing may be faster or slower than unsmoothed points. Many systems, however, have an upper limit to the size of smoothed points, which may cause degradation

in the appearance of the effect when the viewpoint is very close to the emitting object. The 'S' key toggles point smoothing during the demo. The use of many 3D points can be applied effectively to simulate most particle systems such as bubbles and water spray.

See `puff.h` and `puff.c`

Explosions

The explosions consist of 3D points and use much of the same code as the spaceship exhaust. The only real difference is that the emitting nodes are `WTmovsepnodes` that behave like shrapnel generated by an explosion. Once the explosion is created, the nodes move away from the center with a randomly determined speed and direction, generating a stream of puffs behind them. The age of the particle puff determines the color.

See `puff.h`, `puff.c`, `explosion.h`, and `explosion.c`

You can modify and interchange these techniques to suit your application. For instance, the billboarded polygon technique demonstrated with the missile trails could be used in the explosion effect instead of using 3D points. With further modification, the polygons could change in scale, transparency or color over time.

How Do I Load Lights As Movable?

You can use the `WTmovnode_load` (see page 5-5) function to load a light file as a movable. However, if you have more than one light in a light file and you use `WTmovnode_load` to read in the light file, a single movable containing all the lights is created (and not one movable per light). So, if you need to create individual movables for each light in a light file, you should break the file down into “single-light” files. These files contain only one light each, and `WTmovnode_load` can then be called once for each light file to create individual movable light nodes.

You can also create movable spot, point, and directed lights by calling `WTmovlightnode_newspot` (see page 5-4), `WTmovlightnode_newpoint` (see page 5-3), and `WTmovlightnode_newdirected` (see page 5-4), respectively.

How Do I Make An Object Follow A Light?

It is sometimes necessary to have an object follow a light. For example, you might want a cone object to follow a spot-light. The simplest way to do this is to load the light as a movable and attach the object as an attachment to the movable light node.

The following example code assumes you have the light in a light file called “light” and you have the object in a file called “object.”

EXAMPLE CODE

```
void Object_Follow_Light(void)
{
    WTnode *spotlight;
    WTnode *cone;
    spotlight = WTmovnode_load(root, "light", 1.f);
    cone = WTnode_load(NULL, "object", 1.f);
    WTmovnode_attach( spotlight, cone, 0 );
}
```

Note: If you have more than one light in a light file and you use WTmovnode_load (see page 5-5) to read in the light file, then a single movable containing all of the lights is created (not a movable per light). So, if you need to create movables for say three lights (specified in a light file), you need to separate the light file into three different files each containing one light and call WTmovnode_load for each light file.

How Do I Make An Object Follow The Viewpoint?

In some applications, you want an object to move along with the viewpoint. In other words, you want the object to move such that it is stationary with respect to the viewpoint. This means that it should translate and rotate along with the viewpoint, so that it is not only always in view, but also at the same position and orientation in the viewpoint frame.

For example, suppose you are simulating a football game in which the user views the world through the face mask of a helmet. You would want to have the helmet “follow” the viewpoint such that the helmet remains stationary with respect to the viewpoint, and at all times, it appears that the user is viewing the world through the face mask. You could, of course, set the viewpoint’s position and orientation to correspond to a moving helmet, however, in most applications you want to have the *viewpoint* controlled by a sensor, and have the helmet follow it.

The following example code shows how to accomplish this task using the *Follow_Viewpoint* function. In this code segment, the object is moved such that it is always facing the viewpoint and is positioned at 50 units down the z-axis in the viewpoint frame.

EXAMPLE CODE

```
void Follow_Viewpoint( WTnode *node )
{
    WTp3 vpoint_pos, object_offset;
    WTq vpoint_q;

    /* Step1 : Create the WTp3 offset, which is the object's position in the
    viewpoint frame. In this example, the object is positioned at 50 units
    down the z-axis. It is assumed that the object is modelled at the origin.
    If not, you need to tune object_offset to get the object to the desired
    location */
    WTp3_init( object_offset );
    object_offset[Z] = 50.0f;

    /* Step2 : Get the viewpoint's position and orientation */
    WTviewpoint_getposition( WTuniverse_getviewpoints(),
        vpoint_pos );
    WTviewpoint_getorientation( WTuniverse_getviewpoints(),
        vpoint_q );

    /* Step3 : Transform the object_offset into the viewpoint frame */
    WTp3_rotate( object_offset, vpoint_q, object_offset );
    WTp3_add( object_offset, vpoint_pos, object_offset);

    /* Step4 : Apply the translation and the orientation to the object.
```

```
    The orientation is the same as the viewpoint's orientation
    because you want the object to face the viewpoint. */
    WTnode_settranslation( node, object_offset );
    WTnode_setorientation( node, vpoint_q );
}
```

You should call the *Follow_Viewpoint* function from the universe action function so that it is executed each frame. Keep in mind, however, that you have to change the default event order in WTK, such that the sensor updates are completed before this function is called. Every frame the viewpoint should be updated by the sensor's motion before you set an object to follow it. Use the function *WTuniverse_seteventorder* to change the event order. (See the *How Do I Change The Event Order?* on page A-34).

Another way to call the *Follow_Viewpoint* function every frame is through tasks. This way you don't need to change the event order because by default tasks are executed after sensors are updated. The example code below illustrates this.

EXAMPLE CODE

```
WTnode *node;
...
myfunction()
{
    ...
    /* after the node creation code assign the task to it. */
    WTtask_new( node, Follow_Viewpoint, 1.0f);
    ...
}
```

See the example demonstration program *button.c* (located in the demo directory on your WTK distribution) which makes use of task functions for this purpose.

How Do I Recursively “Walk” Down The Scene Graph?

Often you will want to “walk” down the scene graph tree and manipulate certain nodes as you encounter them. Having a recursive function that parses your scene graph to identify the relevant nodes helps you in this task. Assume, for example, you want to improve the performance of your application by making use of the geometry optimization feature, *WTgeometry_prebuild* (see page 6-40). To use this function, you have to identify all the geometry nodes in your scene and rebuild the geometry in each one of them. The following code example shows you how to accomplish this. (Also see *Scene Graph Traversal* on page 4-78.)

EXAMPLE CODE

```
void traverse_node(WTnode *node)
{
    int nChildren, nAttachments;

    /* prebuild if it is a geometry node or a movable geometry node */
    if( (WTnode_gettype(node)==WTNODE_GEOM) ||
        (WTnode_gettype(node)==WTNODE_MGEOM) )
    {
        WTgeometry_prebuild( WTnode_getgeometry(node) );
    }

    /* if not a geometry node, we have to recurse */
    nChildren = WTnode_numchildren(node);
    if (nChildren > 0)
    {
        for(i=0; i<nChildren; i++)
        {
            traverse_node(WTnode_getchild(node,i));
        }
    }

    /* handle movable node attachments also */
    nAttachments = WTmovnode_numattachments(node);
```

```
if(nAttachments > 0)
{
    for(i=0; i<nAttachments; i++)
    {
        traverse_node(WTmovnode_getattachment(node,i));
    }
}
```

How Do I Get A Pointer To A Node Using Its Name?

If you know the name of a node in your scene graph and you want to get a pointer to that node, use the function `WTuniverse_findnodebyname` (see page 4-49). If you have multiple nodes that have the same name, you should inform WTK exactly which occurrence you are trying to retrieve. For this purpose, `WTuniverse_findnodebyname` takes an integer, *num*, as an argument apart from the string that contains the name. Use *num* to indicate the occurrence of the node you want.

For example, if you have a node in your scene graph called “mynode,” you can retrieve it by calling:

```
WTuniverse_findnodebyname( "mynode", 0);
```

Since there is only one occurrence of “mynode,” you should pass in “0” for the occurrence number *num*.

Now suppose you create three nodes *A*, *B* and *C* in that order. And, suppose you name all three nodes “myname.” To retrieve *C*, you should use:

```
WTuniverse_findnodebyname( "myname", 0 );
```

To retrieve *B*, you should use:

```
WTuniverse_findnodebyname( "myname", 1 );
```

And to retrieve A, you should call:

```
WTuniverse_findnodebyname( "myname", 2 );
```

Notice that the nodes are returned in the reverse order in which they are created.
(Occurrence 0 is C not A.)

In developing applications, you will run into several situations where it would help to use *WTuniverse_findnodebyname*. For example, suppose you have a model of a school campus, and you want the viewpoint to zoom-in on a geometry node called “laboratory” on start up. The following code allows you to do that:

```
node = WTuniverse_findnodebyname( "laboratory", 0 );  
WTwindow_zoomviewtonode( window, node, 0 );
```

How Do I Associate A Task With a Particular Object?

Tasks are useful when you want to assign behaviors to individual objects. You can use the *WTtask_new* (see page 11-2) function to associate a task to a particular object. For example suppose you wanted to rotate an object about its Y axis by five degrees each frame. A simple way to do this is to associate a task to the object.

The following example code illustrates how to associate a task to an object.

EXAMPLE CODE

```
int main(int argc, char *argv[])  
{  
    ...  
    /* Create a movable “node” */  
    ..  
    WTtask_new(node, Rotate_Y, 1.0f);  
    ...  
}
```

```
void Rotate_Y(WTnode *node)
{
    WTMovnode_axisrotation(node, Y, 5.f*PI/180.f);
}
```

Execution of task functions (associated with objects) is one of the four events that occur in each frame (see page 2-9 for more information on the events that occur each frame). When you associate an object with a task, the task function is called once every frame.

You could have implemented the previous example by calling *WTmovnode_axisrotation* in the universe action function. However, the usage of tasks is cleaner, particularly when the task function is not as simple as in the example shown here. It is always better to associate a task to an object rather than clutter up the universe action function.

An object can have multiple tasks associated with it. You can use the functions *WTtask_remove* to remove a task (deactivate it, but not delete it), *WTtask_add* to add back a task (activate it), and *WTtask_delete* to delete a task (see the *Tasks* chapter, starting on page 11-1).

Note: Associating a new task to an object does not delete an already existing task associated with that same object. If you don't want a previously created task active when you create a new task for that object, i.e. you want the new task to replace the previous task, you must explicitly delete the previous task.

How Do I Handle Portals In This Release?

You can think of “Portals” as the 3D equivalent of links in a hypertext system. They are doorways in space that connect the currently displayed world with an alternate world. Earlier versions of WTK provided portalling as a facility— there were functions by which you could specify which polygon in the current world was a “doorway,” and which world to portal into if the viewpoint crossed that polygon. Once these were specified, WTK handled the intersection testing and automatically loaded the new world if the viewpoint intersected with the portal polygon.

The current release of WTK handles portals differently. Any portal information in NFF files is ignored. WTK now provides a function, *WTviewpoint_intersectpoly* (see page 16-26), that tests whether the viewpoint's motion in a particular frame resulted in an intersection with a specified polygon. So instead of WTK automatically checking whether a portal was crossed, *you* now have explicit control over this by calling *WTviewpoint_intersectpoly* every frame (in your action function). If the function returns TRUE, you should have appropriate code in your application to stop processing the current world and move into a new world. This is usually done by assigning a different root node to the WTK window. (This should be the root node of the scene graph that corresponds to the new world.) See the demonstration program *portal.c* (located in the demo directory of your WTK distribution) for more information on the implementation of portals.

Your application needs to identify the portal polygon and pass this as an argument to the function *WTviewpoint_intersectpoly*. The demo *portal.c* loads the model *oplan.nff* and uses the polygon with the texture "picture.tga" as the portal polygon. The NFF file is edited to add a polygon id to this polygon, so that the application can identify it using the function *WTgeometry_id2poly*. The following example code shows (in brief) an implementation of portals.

EXAMPLE CODE

```
{
    ...
    node = WTnode_load( root_node1, "oplan.nff", 1.0f);
    geom = WTnode_getgeometry( node );
    /* gets the polygon with id=1 */
    portal_poly = WTgeometry_id2poly( geom, 1);
    /* creates a node path to the geometry node */
    portal_npath = WTnodepath_new( node, root_node, 0);
    /* creates a second root node and pre-loads a second world.
    This root node is initially unused. */
    root_node2 = WTrootnode_new();
    WTnode_load( root_node2, "lobby.nff", 1.0f);
}

action_function()
{
    if( WTviewpoint_intersectpoly(
        vpoint, portal_poly, portal_npath, 0.0f )
```

```
{
/* portal has been crossed. load second root node */
WTwindow_setrootnode( window, root_node2);
/* add code to update portal_poly and portal_npath. These
should now be a part of the new scene graph, so that you can portal
back into the original world */
}
}
```

How Do I Test For Intersections Between The Viewpoint And The Universe?

When you fly your viewpoint through the universe (or the world) that you have created, you do not want the viewpoint to fall through objects. Thus, you should have collision detection algorithms that suit your application. WTK does not test for intersections between the viewpoint and objects in your world. It does, however, provide functions like *WTnode_rayintersect*, *WTPoly_rayintersect*, *WTviewpoint_intersectpoly* and a host of other intersection testing functions, that should help you implement your algorithms. See descriptions for each of these functions (starting on page 4-85) to understand which one would best suit your application. WTK retains the viewpoint's position in the last frame. If your algorithm detects an intersection between the viewpoint and an object in one frame, you can always reset the viewpoint to its position in the previous frame. (See the *WTviewpoint_getlastposition* function on page 16-9. To use this function, however, you have to change the WTK event order appropriately. See the section *How Do I Change The Event Order?* on page A-34.)

How Do I Test For Objects Intersecting With Other Objects In The Universe?

Applications often require you to test whether an object intersected with other objects in the world. WTK offers a number of functions to perform collision detection, each suited for a specific case. (Refer to the section *Intersection Testing* on page 4-85 in the *Scene Graphs* chapter.)

To optimize performance, you should organize your scene graph such that you test for intersections only with objects that are relevant. For example, if you are building a simulation that consists of a car moving along a race track, it would be enough to test for collisions between the car and the road, the railings along the road, and other cars on the road. It would not be necessary to test for collisions with buildings and other scenic details alongside the road. Therefore, you should place such irrelevant geometries under a different sub-tree. For most of the intersection functions that WTK provides, you have to specify a node path indicating a sub-tree below which you want collision detection to be done. You have to make sure that only the geometries that you want the intersection test to be performed on are contained in the sub-tree.

How Do I Get The Rendered Position Of An Object?

In the current release of WTK, getting the rendered position of an object is a two step process. (In this discussion and in most other parts of this book, an object refers to a geometry node.) The function *WTgeometry_getmidpoint* (see *WTgeometry_getmidpoint* on page 6-28) returns the midpoint of the geometry contained in a geometry node. If you do not call functions such as *WTgeometry_translate* and *WTgeometry_transform*, the vertex positions of a geometry never change, and *WTgeometry_getmidpoint* will always return the same coordinates for the midpoint.

To move an object around during a simulation you should use transform nodes. WTK “parses” the scene graph noting all the transform nodes that affect an object. The cumulative transformation matrix of these transform nodes is then used to determine where the object is rendered. (OpenGL multiplies each vertex position by the cumulative transform matrix and renders the object accordingly.) The geometry information contained

in the object is therefore not altered. That is why the position returned by `WTgeometry_getmidpoint` never changes, though the object may not actually be rendered there, as it is influenced by transform nodes. To get the rendered midpoint of the object you have to multiply the geometry midpoint by the cumulative transformation matrix. The following example code demonstrates this.

EXAMPLE CODE

```
/* This example assumes that "mynode" is either a geometry node or a movable
geometry node.*/
WTgeometry_getmidpoint( mynode, midpoint );
/* create a node path from the root node to mynode. This is required to obtain
the cumulative transform matrix that affects mynode. */
npath = WTnodepath_new( mynode, root_node, 0);
/* get the transform matrix along this path */
WTnodepath_gettransform( npath, matrix );
/* now transform the midpoint with this matrix */
WTm4_xformp3( matrix, midpoint, rendered_midpoint );
```

Note: If you model your geometry such that its midpoint is at the origin, it simplifies the above task. The translational part of the transform matrix tells you where the object is rendered. Moreover, if your geometry is modeled at the origin, and the object is a movable node not affected by any external transform nodes, simply call `WTnode_gettranslation`. This gives you the rendered position.

How Do I Create A Simple Animation Using Switch Nodes?

When you animate (that is, move, resize, or reshape) an object, it is not always practical to do it programmatically. For simple animations, as shown in Figure A-1 below, it is much easier to use a flipbook method. You can accomplish this using switch nodes.

Switch nodes allow you to determine which of several children are processed. This is particularly useful if you are creating an animation sequence, in which case you can cycle through the switch node's children – switching once for each frame. You should break your

animation sequence down into a finite set of distinguishable frames. Next, you should create a geometry node that corresponds to each frame, and add it as a child to the switch node. Then, every frame you can use the function `WTswitchnode_setwhichchild` (see page 4-57) to “select” one of the switch node’s children to be processed. This way, sequentially switching between the switch node’s children, you can build an animation.

Figure A-1 shows a three frame animation. The first step is to create a new switch node. Then add the three geometries that you want the switch node to “flip” through. (Keep in mind, that by default, the switch node does not process any of its children.) Then, it’s as simple as telling the node which child it should process.

Note: If your geometry is large (e.g., 1 MB+), you should not use this method for animations. It wouldn't be practical to have 30 frames for 1 MB geometry.

In certain cases, you can even optimize memory usage by only including transform nodes below the switch node. For example, assume you are animating a person walking. When you are building the animation sequence for the motion of the person’s legs, you do not have to create a geometry node for every distinct position of a leg. You can instead have one geometry node that represents the leg, and have multiple transform nodes below the switch node. Each of these transform nodes transforms the leg into a different, distinct position. So, cycling through the transform nodes you have an animation with an optimized usage of memory.

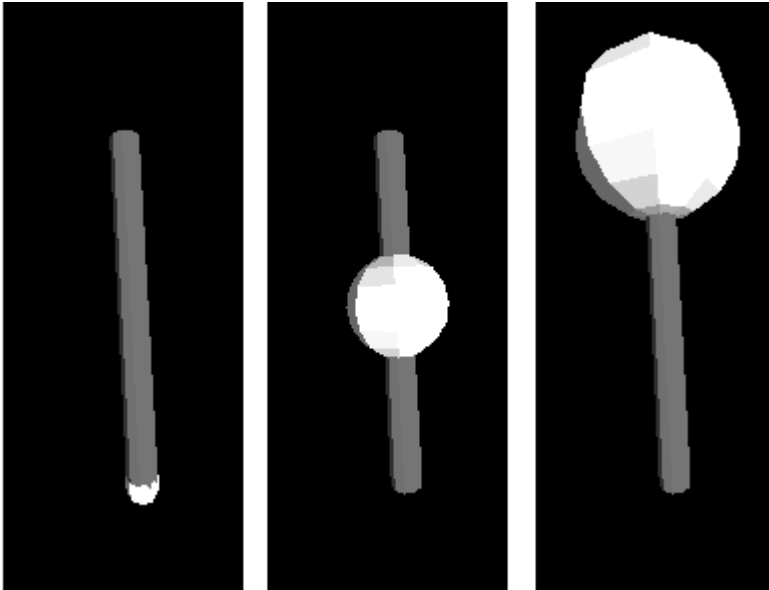


Figure A-1: An Illustration of Simple Animation

EXAMPLE CODE

```
switch_node= WTswitchnode_new(root);
/* Child 0 */
geom = WTgeometry_newsphere(0.1f,8,8,FALSE,FALSE);
WTmovgeometrynodel_new(switch_node, geom);
new_pos[X] = new_pos[Z] = 0.0f;
new_pos[Y] = 1.0f;
WTgeometry_translate(geom, new_pos);

/* Child 1 */
geom = WTgeometry_newsphere(0.3f,8,8,FALSE,FALSE));
WTmovgeometrynodel_new(switch_node, geom);

/* Child 2 */
geom= WTgeometry_newsphere(0.5f,8,8,FALSE,FALSE));
WTmovgeometrynodel_new(switch_node, geom);
```

```
new_pos[Y]= -1.0f;
WTgeometry_translate(geom,new_pos);
...
int current_child = 0;
WTswitchnode_setwhichchild(switch_node, 0);
while (1)
{
    /*time_to_move is an arbitrary timing function*/
    if (time_to_move())
    {
        current_child++;
        /*Loop through the children from first to last*/
        if (current_child > WTnode_numchildren(switch_node))
            current_child = 0;
        WTswitchnode_setwhichchild(switch_node, current_child);
    }
}
```

How Can I Optimize Performance Using LOD Nodes?

To improve performance (i.e., to obtain a higher frame rate), it is often necessary to trade-off the visual quality of objects in your scene. As your viewpoint moves farther away from your objects, it is no longer necessary to render those objects in as much detail as if they were placed directly in front of you. You can use LOD nodes to switch to less detailed versions (i.e., lower quality) of an object as your viewpoint moves further away from the object.

Figure A-2 and the example code below show three different levels of detail for a sphere object. The term “range” is often used in the context of LOD nodes. Range indicates the distance between the viewpoint and the center of the LOD node. (Remember that if the LOD node is affected by transform nodes, its center is correspondingly transformed by the cumulative matrix. The default value for the LOD center is [0.0,0.0,0.0].)

A range therefore represents a “switch-out” distance. You specify a set of ranges to an LOD node by passing an array of floats to the function *WTlodnode_setrange* (see page

4-55). The first range value indicates the distance from the LOD center at which the first child switches out (i.e., is not processed anymore) and the second child switches in (is processed). Similarly, the second range value indicates the distance at which the second child switches out and the third child switches in.

In the example below, the LOD node has three children. That is why you need to set only *two* ranges, because, for *all* distances beyond the second range, you want the third child to be processed. In other words, you don't set an upper limit on the distance, so the third child can't switch out.

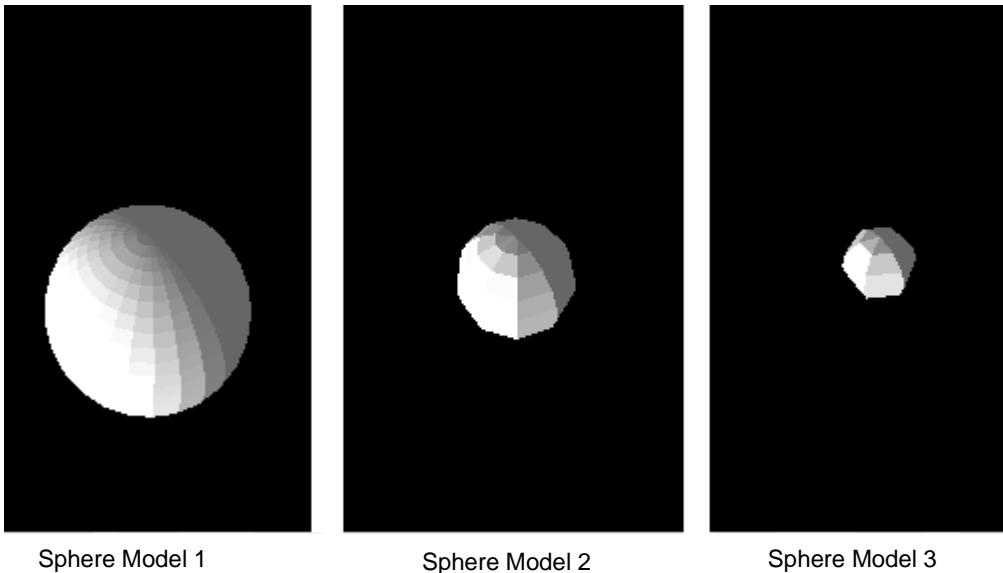


Figure A-2: An Illustration of Three Different Levels of Detail

Figure A-2 shows three spheres with varying detail. Model 1 has the highest detail and Model 3 has the least detail. An LOD node is used to switch between these models. Model 1 is processed when the distance between the LOD center and the viewpoint is less than three units. Model 2 is processed when this distance is greater than three but less than six units. Model 3 is processed for all distances greater than six units.

EXAMPLE CODE

```
WTnode *lod_node;
WTgeometry *geom;
float range[2]= {3.0f,6.0f};
lod_node= WTlodnode_new(root);

/* Child 0. This is displayed for ranges less than 3.0
since it has the highest detail.*/
geom = WTgeometry_newsphere(1.0f,30,30,FALSE,FALSE));
WTmovgeometrynode_new(lod_node,geom);

/* Child 1. This geometry has medium detail. Displayed for ranges between
3.0 and 6.0. */
geom = WTgeometry_newsphere(1.0f,20,20,FALSE,FALSE));
WTmovgeometrynode_new(lod_node,geom);

/* Child 2. This geometry has the least detail.
Displayed for ranges beyond 6.0.*/
geom = WTgeometry_newsphere(1.0f,10,10,FALSE,FALSE));
WTmovgeometrynode_new(lod_node,geom);
WTlodnode_setrange(lod_node,range,2);
```

What Is Terrain Following?

Terrain following means flying over a terrain at a constant height. If your viewpoint, controlled by a sensor, is moved around over a terrain, it will at times fly too high over the terrain and might also sometimes fall through the terrain. This kind of motion may become very distracting to the user. If your application incorporates some sort of a terrain following algorithm, the viewpoint would glide over the surface at a constant height. This makes flying easier and also more pleasant.

WTK does not provide any automatic means for terrain following. You should use functions such as *WTnode_rayintersect* (see page 4-88) and *Wtpoly_rayintersect* (see page 4-88) and implement an algorithm of your own. A brief example is shown below that uses

the function `WTnode_rayintersect` to maintain the viewpoint at a height of ten units above a terrain.

EXAMPLE CODE

```
/* This example assumes that no other geometry apart from the terrain, is
   below the root node. We intend shooting rays straight down from the viewpoint
   position to get the distance to the terrain. */
Wtp3 ray, origin;
float distance;

/* initialize the ray to point straight down */
ray[X] = 0.0; ray[Y] = 1.0; ray[Z] = 0.0;

/* set the origin of the ray to be the viewpoint's current position */
WTviewpoint_getposition( vpoint, origin );

if( WTnode_rayintersect( root_node, ray, origin, &distance, NULL ) ) {

    /* intersected the terrain below. 'distance' has the actual distance
       between the viewpoint and the terrain. If this is not 10 you should
       reset the viewpoint's position appropriately */

    if( distance < 10.0 ) {
        origin[Y] += (10 - distance);
        WTviewpoint_setposition( vpoint, origin );
    } else {
        origin[Y] -= (distance - 10);
        WTviewpoint_setposition( vpoint, origin );
    }
} else {
    /* the ray did not intersect the terrain. Either the viewpoint has gone
       off the edge of the terrain or has fallen through it. In either case
       set it back to previous position */
    WTviewpoint_getlastposition( vpoint, origin );
    WTviewpoint_setposition( vpoint, origin );
}
```

Note that the above code segment expects that the sensor updates for this frame have been completed. Because the function *WTviewpoint_getlastposition* is used, you should change the default event order such that the universe action function is called after the viewpoint has been updated by the sensor. (See the section *How Do I Change The Event Order?* on page A-34).

How Do I Keep An Object Perpendicular To The Viewpoint Direction At All Times?

It is sometimes desirable that an object, whenever in view, always faces the viewpoint (a billboard effect, for example). You can achieve this by keeping the object perpendicular to the viewpoint at all times. For example, suppose in your application you are simulating an explosion by displaying a sequence of textures on a rectangular polygon. Assume that when you are navigating, you want the explosion to always face you (in other words, the rectangular polygon should face you).

The following example code ensures that a particular object is always perpendicular to the viewpoint, and then assigns it as a task to the object.

EXAMPLE CODE

```
void Perpendicular_Viewpoint(WTnode *node)
{
    WTq vq;
    WTviewpoint_getorientation(WTuniverse_getviewpoints(), vq);
    WTnode_setorientation(node, vq);
}

WTnode *node;
int main(int argc, char *argv[])
{
    ...
    node = WTmovgeometrynodeload(root, WTgeometry_newrectangle(1,1,TRUE));
    WTgeometry_settexture(WTnode_getgeometry(node),
        "explosion", TRUE, FALSE);
}
```

```
WTtask_new(node, Perpendicular_Viewpoint, 1.f);
WTwindow_zoomviewpoint(WTuniverse_getwindows());
...
}
```

How Do I Change The Event Order?

Every frame, before rendering into the windows, WTK first executes the user's universe action function. Then, the graphical objects and viewpoints are updated by sensors attached to them, the task functions are executed, and lastly the paths that are in record or play mode are stepped. (See *Simulation Management* on page 2-5.) These are the four events that are processed: actions, sensors, tasks and paths, and the default event-processing order is the order in which they have been mentioned above.

Certain applications may need to have these events processed in a different sequence. WTK provides for this through the function *WTuniverse_seteventorder* (see page 2-9). For example, if you are using functions like *WTviewpoint_intersectpoly*, you need to have the sensor updates done before the actions function is called. The following example code segment shows how you can do this.

EXAMPLE CODE

```
{
    short event_array[4] = { WTEVENT_OBJECTSENSOR,
        WTEVENT_ACTIONS,
        WTEVENT_TASKS,
        WTEVENT_PATHS };

    WTuniverse_new( WTDISPLAY_DEFAULT, WTWINDOW_DEFAULT;
    WTuniverse_seteventorder( 4, event_array );
}
```

If the event order is set using this code, objects and viewpoints are updated by the sensors that control them before the universe action function is called. Note that though *WTuniverse_seteventorder* takes an argument for the number of events, WTK requires this to be 4.

How Do I Integrate A WTK Rendering Window With A Host-Specific Window?

You can use the *WTinit_usewindow* (see page 17-6) function and the *WTwindow_newuser* (see page 17-7) function to integrate a WTK rendering window with a host-specific window. All you need to do is specify the ID of the host-specific window (*HWND* for a Windows application and *Widget* for UNIX applications).

If you do not specify *WTDISPLAY_NOWINDOW* in your call to *WTuniverse_new* (see page 2-2), WTK creates a system specific drawing area and renders into it. The functions *WTinit_usewindow* and *WTwindow_newuser* (see page 17-7) inform WTK to use the user-specified *HWND* (or *Widget*) as the rendering area. WTK then, does not create a new window on Windows platforms, and does not create a new drawing area on UNIX platforms. WTK renders into the window you specified.

There is, however, a minor difference between these two functions. *WTwindow_newuser* creates an internal structure, a *WTwindow*, and updates this structure to recognize the user-specified window as the drawing area.

WTinit_usewindow does not do this. It does *not* create a *WTwindow*, so WTK is not yet aware that a drawing area has been specified. It is only when you call *WTuniverse_new* (with an argument other than *WTDISPLAY_NOWINDOW*) that WTK creates the internal *WTwindow* and recognizes the drawing area you provided earlier.

The following example code illustrates how you use *WTinit_usewindow*.

EXAMPLE CODE

```
/* On Windows platforms 'mywindow' will be an HWND and on UNIX
platforms, 'mywindow' will be a Widget */

WTinit_usewindow( mywindow );
WTuniverse_new( WTDISPLAY_DEFAULT, WTWINDOW_DEFAULT );
```

Or, if you want to customize your window using *WTwindow_new*,

```
WTuniverse_new( WTDISPLAY_NOWINDOW, WTWINDOW_DEFAULT );  
WTinit_usewindow( mywindow );  
WTwindow_new( x_pos, y_pos, x_size, y_size, custom_flags );
```

The following example code illustrates how you use *WTwindow_newuser*.

EXAMPLE CODE

```
/* On Windows platforms 'mywindow' will be an HWND and on UNIX  
platforms, 'mywindow' will be a Widget */
```

```
WTuniverse_new( WTDISPLAY_NOWINDOW, WTWINDOW_DEFAULT );  
WTwindow_newuser( parent, custom_flags );
```

WTwindow_newuser should always be called after *WTuniverse_new*.

Orienting Sensors Differently

How Do I Use Orientation-Tracking Sensors (On A Head-Mount-Display) That Are Not Positioned Along The Central Axis Of The HMD?

If you are using a head-mount display, such as the Vissette from Virtuality, the sensor that tracks orientation may not be positioned along the central axis of the HMD. In the case of the Vissette, this sensor is an InsideTRAK, with the receiver built-in somewhere near the user's left ear (rather than directly on top of the user's head). Further, the sensor may be oriented in such a way that its reference frame does not align with the reference frame of the user's head. For example, if the receiver is incorporated in such a way that it is positioned sideways (that is rotated around its local z-axis by 90 degrees), it will not return correct rotations. When the user moves his/her head from left to right, the sensor will detect a pitch rather than a yaw. To correct this, use the function *WTsensor_rotate* (see page 13-20). The following example code shows how this is done.

EXAMPLE CODE

```
/* You have to know the exact orientation of the receiver. This can be found in the
   instruction manual that accompanies the HMD device that you are using. If the
   user's head is positioned upright, x-axis is to the user's right, y-axis goes
   straight down, and the z-axis is straight ahead. You should obtain the rotations
   (in radians) that should be applied to the user's reference frame to transform it
   into the sensor's reference frame. In this example these are taken to be
   X_ROT, Y_ROT and Z_ROT. */
```

```
#define X_ROT xx /* fill in the rotations in radians */
```

```
#define Y_ROT yy
```

```
#define Z_ROT zz
```

```
WTq qx,qy,qz,q;
```

```
WTsensor *inside_trak;
```

```
{
```

```
/* initialize the insidettrak sensor that is incorporated into the head mount display */
inside_trak = WTinsidettrak_new( 1 );
```

```
/* constrain sensor translations */
```

```
WTsensor_setconstraints( inside_trak, WTCONSTRAIN_X | WTCONSTRAIN_Y |
                        WTCONSTRAIN_Z );
```

```
/* convert the eulers to quaternions */
```

```
WTeuler_2q( X_ROT, 0.0, 0.0, qx );
```

```
WTeuler_2q( 0.0, Y_ROT, 0.0, qy );
```

```
WTeuler_2q( 0.0, 0.0, Z_ROT, qz );
```

```
/* next multiply the quaternions together to get the cumulative rotation */
```

```
WTq_mult( qx, qy, q );
```

```
WTq_mult( q, qz, q );
```

```
/* now call WTsensor_rotate with this resultant quaternion */
```

```
WTsensor_rotate( inside_trak, q );
```

```
}
```

Note: WTK supports these kinds of HMD devices (i.e., ones that have built-in sensors) to track orientation only. That is, you have to constrain translational output from the sensor. In this example, since we are considering the Vissette, the sensor is an InsideTRAK receiver. You should include translations in your application by using a Joystick or some other device.

How Do I Measure Performance On My Machine?

There are two aspects to measuring performance on your machine. First, you might want to gauge the general performance of your video card (and its driver), and your CPU performance with the system memory you have. You can accomplish this by using the benchmark program called Indy3D which can be found at <http://www.sense8.com/indy3d>. Compare the results on your machine with those that EAI/SENSE8 provides for each platform.

Secondly, you might want to quantify the performance of one of your applications. Use *WTuniverse_framerate* (see page 2-23) to get the average number of frames per second at which your application is running. You can use the function *WTwindow_numpolys* (see page 17-25) to determine how many polygons are being rendered in a window. This function's return value does not include polygons whose geometries have been culled (because they are out of view).

On UNIX Platforms, How Do I Get A Pointer To The Display That WTK Is Using?

Most Xt calls require a pointer to the display that WTK is using. WTK provides a function *WTwindow_getwidget* (see page 17-29) that can help you in obtaining the display. By passing in a pointer to a WTK window into this function you can retrieve the window's X id (that is, the corresponding widget). Once you have the widget, you can get a pointer to the display by calling *XtDisplay*.

```
Display = XtDisplay(Widget);
```


How do I use Boston Dynamic's DiGuy with WTK (or any other BDI character set)?

The introduction of the OpenGL callback node in WTK release 9 has allowed for a much simpler integration of 3rd party OpenGL based products into a WTK application. Boston Dynamic's DiGuy human animation system is a good example of how WTK and 3rd party vendors can combine to create a great application. In the /demo/diguy subdirectory of your WTK installation you will find an example program on how to develop an application that uses both WTK and DiGuy. The example file, `wtkdiguy.c`, can be used either as a demonstration program for users with DiGuy, or as an application template for interactive simulations. The code for this application is written in a very approachable manner and is commented heavily so that users that are not so familiar with WTK or DiGuy can have their application up and running within a manner of minutes. The example code is also very easy to extend and has been written in a very flexible manner. Note: To use this demonstration application you must either (1) own a licensed copy of both WTK and DiGuy or (2) be evaluating either or both products with the appropriate evaluation licenses.

When attempting to use the demonstration code there are a few places where the developer must make changes to the demonstration source code, these places have been marked with appropriate comments and require changes that would take at most a minute or two. These changes are related only to setting the proper environment variables to allow DiGuy to function properly on the user's machine.

B

Environment Variables

WorldToolKit uses environment variables to customize its operation on your computer. For example, you can set new paths for image and model files, set the Z-buffer size for your graphics card, etc. To add environment variables in Windows NT 3.51, choose Control Panel from the Main Program group, then select System. To add environment variables in Windows NT 4.0 choose Settings, Control Panel from the Start menu then select System and click the Environment tab. On UNIX platforms, you can add environment variables to your unix shell using the `setenv` or `set` command.

The examples below all assume you have installed WorldToolKit into the `C:\Program Files\wtk` directory on the Windows platform. If you have installed the program to another directory or drive, or if you are on the UNIX platform, modify the examples accordingly.

WTKCODES

The *WTKCODES* environment variable is used to specify the file path to the *WTKCODES* file which contains your system-specific software license code. By setting the *WTKCODES* environment variable to the file path where your *WTKCODES* file exists, you can ensure that WorldToolKit will always be able to find the software license code and run properly. For example:

Variable	WTKCODES
Value	c:\Program Files\wtk

To specify multiple file paths for the *WTKCODES* environment variables, separate the path names with a semicolon (;) [on UNIX platforms separate path names with a colon (:)] as in the following example, which sets the *WTKCODES* environment variable to `patha` and `pathb`:

Variable	WTKCODES
Value	patha;pathb

WTK first searches for the file in the current working directory, and then along the paths specified in the corresponding environment variable, in the order that they occur. In the example above, the current directory will be searched ahead of `patha` and `patha` will be searched before `pathb`.

WTIMAGES

By default, WorldToolKit looks in the current directory to search for texture image files. To set additional file paths for image files, use the environment variable `WTIMAGES` to specify a path to your texture images subdirectory. For example:

Variable	<code>WTIMAGES</code>
Value	<code>c:\Program Files\wtk\images</code>

To specify multiple file paths for the `WTIMAGES` environment variables, separate the path names with a semicolon (;) [on UNIX platforms separate path names with a colon (:)] as in the following example, which sets the `WTIMAGES` environment variable to `patha` and `pathb`:

Variable	<code>WTIMAGES</code>
Value	<code>patha;pathb</code>

When an image is loaded, WTK first searches for the file in the current working directory, and then along the paths specified in the corresponding environment variable, in the order that they occur. In the example above, the current directory will be searched ahead of `patha` and `patha` will be searched before `pathb`.

WTMODELS

By default, WorldToolKit looks in the current directory to search for model files. To set additional file paths for model files, use the environment variable `WTMODELS` to specify a path to your models subdirectory. For example:

Variable	<code>WTMODELS</code>
Value	<code>c:\Program Files\wtk\models</code>

To specify multiple file paths for the *WTMODELS* environment variables, separate the path names with a semicolon (;) [on UNIX platforms separate path names with a colon (:)] as in the following example, which sets the *WTMODELS* environment variable to *patha* and *pathb*:

Variable	<i>WTMODELS</i>
Value	<i>patha;pathb</i>

When a model is loaded, WTK first searches for the file in the current working directory, and then along the paths specified in the corresponding environment variable, in the order that they occur. In the example above, the current directory will be searched ahead of *patha* and *patha* will be searched before *pathb*.

WTKZBUFFERSIZE

WTK performs its calculations assuming that a Z-buffer of depth 24 exists. Some graphics cards only support 16-bit (or less) Z-buffers. If you don't set this value correctly, the hardware graphics card may not function properly or may be disabled entirely causing the default software OpenGL implementation to be used instead and can result in a significant drop in performance. You can avoid this by setting *WTKZBUFFERSIZE* to the depth of the actual hardware Z-buffer. For example:

Variable	<i>WTKZBUFFERSIZE</i>
Value	16

WTKALPHATEST

This environment variable is used to set the transparency threshold of pixels. Pixels, whose final computed transparency value (after factoring in the polygon's material opacity and texture alpha values) is below this threshold value (0-255) will not be written to the framebuffer. This will ensure that all pixels whose transparency value is below a specified threshold value to be treated as completely transparent. This can be useful when you want to have a "cookie-cutter" effect with your textures. For example:

Variable	<i>WTKALPHATEST</i>
Value	24

The default is 78 on UNIX platforms while the default is 0 on Windows platforms.

WTKMAXTEXSIZE

Texture images will be scaled down, if necessary, so that the image width and height in pixels will not exceed this value. By setting this environment variable to an appropriate value you can help ensure that your application does not exceed your hardware texture memory limits. For example:

Variable	WTKMAXTEXSIZE
Value	256

The default is 1024 (this is also the maximum).

WTKSQRTEX

Texture images will be scaled down, if necessary, so that the texture's width and height are equal. The possible values are 0 (zero) and 1 (one), where 0 = off. For example:

Variable	WTKSQRTEX
Value	1

The default is off (zero).

WTKPROXY

http proxy server (hostname:port). Used when reading VRML files, i.e., URLs contained in anchor and/or inline nodes are relative to the proxy server specified here. For example:

Variable	WTKPROXY
Value	BATMOBILE:8080

WTKALPHAENABLE

Depth of alpha buffer to allocate. Valid values range from 0 to 8. This environment variable can only be used on systems which have alpha buffer hardware. For example:

Variable	WTKALPHAENABLE
Value	8

The default is 0.

WTBIRDDELAY

This environment variable is used with Ascension's Bird, Motionstar, 6DOF Mouse or Flock of Birds sensor device. Occasionally, WTK is unable to communicate with the device during the initialization process. This happens when the Flock of Birds takes a little longer to respond than usual. In such cases you may use this environment variable to specify to WTK, an additional amount of time to wait before polling the serial port for a response from the bird.

You must specify the value for *WTBIRDDELAY* in milliseconds. In most cases a value of 1500 is sufficient for the bird to respond. This variable is used only during the initialization process. You do not need to set this variable unless you get a WTK Warning saying "Bird not responding".

Variable	WTKBIRDDELAY
Value	500

WTKLS

Used to specify the location of the WTK license server daemon when WTK's floating license option is used. (Note that the license server daemon can only be run on SGI platforms.) The license server location is specified by the IP address of the server, followed by a colon, followed by the port number.

Variable	WTKLS
Value	130.20.152.24:2000

WTKNOSTEREO

By default, WTK requests a stereo capable visual on UNIX and a stereo capable PFD (Pixel Format Description) on Windows platforms. Because of this, on some graphics systems such as ELSA, applications which do not need or use stereo windows can experience a performance degradation. If your application does not use stereo windows, you can insure maximum graphics performance by setting this environment variable to 1. Doing so will force WTK to use a non-stereo capable visual/PFD.

Variable	WTKNOSTEREO
Value	1

WTKMULTISAMPLE

Available on SGI platforms only and specifies the anti-aliasing sampling rate (must be a power of 2). A higher sampling rate will result in a better quality image but will take more processing time. For example:

Variable	WTKMULTISAMPLE
Value	16

The default is 0.

WTKCPU

Available only on the SGI Multi-pipe/Multi-processor version. Though WTK MP/MP runs multi-process, by default it does not lock processes to particular processors. You may be able to improve performance by locking the WTK processes down to specific processors using the *WTKCPU* environment variable.

This is a numeric variable. For example:

```
setenv WTKCPU 0123
```

locks down the first four processes that WTK spawns to the processor numbered 0, 1, 2, and 3, respectively. This is an appropriate setting for WTK running on a three graphics

pipeline computer with four processors, since WTK has a single application process and one process for each screen (or virtual screen).

The syntax for the value field of the environment variable is:

```
setenv WTKCPU [application processor] [screen0 processor] [screen1 processor] ...
```

with up to nine digits, one digit for the application processor, and up to eight digits for screen processors. The digit value must be one of the following:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f

If this environment variable is not set, it is possible the application will bounce from one CPU to another each frame. Each pipe may also bounce from CPU to CPU.

WTKDISPLAY

Available only on the SGI Multi-pipe/Multi-processor version and on the PRO-1000. (For usage on the PRO-1000 refer to the PRO-1000 Installation and Hardware Guide.) Tells WTK MP/MP which X display to associate with each screen. For example:

```
setenv WTKDISPLAY 'earth:0,wind:0.1,earth:1.0,fire.sense8.com:0.0'
```

results in the following association:

```
WTWINDOW_SCREEN0 the default screen of the machine "earth"  
WTWINDOW_SCREEN1 the :0.1 screen of the machine "wind"  
WTWINDOW_SCREEN2 the :1.0 screen of the machine "earth"  
WTWINDOW_SCREEN3 the default screen of the machine "fire.sense8.com"
```

For more information on this environment variable, see *Using Multiple Screens on UNIX/X Windows in the SGI Installation and Hardware Guide*.

WTKSHMEM

Available only on the SGI Multi-pipe/Multi-processor version. Amount of shared memory to allocate in megabytes. Your swap space must be at least as large as the amount of shared memory allocated. This is a numeric variable. For example:

```
setenv WTKSHMEM 128
```

The default is 64 MB. For more information on this environment variable, see Application Memory Allocation and Shared Memory in the SGI Installation and Hardware Guide.

Defined Constants

You can use WTK's constant definitions to make your applications easier to read and more portable. These are declared in the respective header files in the *include* directory.

They are listed here alphabetically.

Constraint Constants

Used with the function *WTpath_setconstraints* (see page 14-20) and as returned by the function *WTpath_getconstraints* (see page 14-21) for the path class. Also used with the functions *WTmotionlink_addconstraint* (see page 15-11) and *WTmotionlink_removeconstraint* (see page 15-12) for the motion link class. Declared in *sensor.h*.

```
WTCONSTRAIN_X  
WTCONSTRAIN_XROT  
WTCONSTRAIN_Y  
WTCONSTRAIN_YROT  
WTCONSTRAIN_Z  
WTCONSTRAIN_ZROT
```

Display Constants

Declared in *other.h*.

WTDISPLAY_CYRSTALEYES
WTDISPLAY_DEFAULT
WTDISPLAY_DIRECT3DFX (only for WTK Direct – full screen mode)
WTDISPLAY_DIRECTFULLSCREEN (only for WTK Direct – 3Dfx device)
WTDISPLAY_MONO
WTDISPLAY_NOWINDOW
WTDISPLAY_NEEDSTENCIL
WTDISPLAY_RBSTEREO
WTDISPLAY_STEREO
WTDISPLAY_STEREOWINDOW

Drawing Constants

Used with the functions *WTwindow_draw2Dcircle* (see page 19-3) and *WTwindow_draw2Drectangle* (see page 19-3). Declared in *other.h*.

WTDRAW2D_HOLLOW
WTDRAW2D_SOLID

Used with the function *WTwindow_draw3Dlines* (see page 19-10). Declared in *other.h*.

WTLINE_CLOSE
WTLINE_CONNECTED
WTLINE_SEGMENTS

Event Order Constants

Used with the function *WTuniverse_seteventorder* (see page 2-9) and as returned by the function *WTuniverse_geteventorder* (see page 2-10). Declared in *other.h*.

WTEVENT_ACTIONS
WTEVENT_OBJECTSENSOR
WTEVENT_PATHS
WTEVENT_TASKS

Eye Constants

Used with the function *WTwindow_seteye* (see page 17-12) and as returned by the function *WTwindow_geteye* (see page 17-12). Declared in *other.h*.

WTEYE_LEFT
WTEYE_RIGHT

Filetype Constants

Used with the functions *WTnode_save* (see page 4-48) and *WTgeometry_save* (see page 6-26). Declared in *other.h*.

WTFILETYPE_BFF	(SENSE8 Binary NFF)
WTFILETYPE_DXF	(AutoCAD DXF format)
WTFILETYPE_NFF	(SENSE8 Neutral File Format)
WTFILETYPE_WRL	(VRML 1.0)

Frames of Reference Constants

Declared in *vr.h*.

WTFRAME_LOCAL
WTFRAME_PARENT
WTFRAME_VPOINT
WTFRAME_WORLD

Keyboard Constants

As returned by the functions *WTkeyboard_getkey* (see page 24-2), and *WTkeyboard_getlastkey* (see page 24-2). Declared in *sensors.h*.

WTKEY_BACKSPACE
WTKEY_DOWNARROW
WTKEY_END
WTKEY_ENTER
WTKEY_ESC
WTKEY_F1
WTKEY_F2
WTKEY_F3
WTKEY_F4
WTKEY_F5
WTKEY_F6
WTKEY_F7
WTKEY_F8
WTKEY_F9
WTKEY_F10
WTKEY_F11
WTKEY_F12
WTKEY_HOME
WTKEY_LEFTARROW
WTKEY_MIDDLE
WTKEY_PAGEDOWN
WTKEY_PAGEUP
WTKEY_RIGHTARROW

WTKEY_TAB
WTKEY_UPARROW

Light Type Constants

As returned by the function *WTlightnode_gettype* (see page 12-18). Declared in *other.h*.

WTLIGHTTYPE_AMBIENT
WTLIGHTTYPE_DIRECTED
WTLIGHTTYPE_POINT
WTLIGHTTYPE_SPOT

Material Table Property Constants

Used with the functions *WTmtable_new* (see page 8-7), *WTmtable_setvalue* (see page 8-15), *WTmtable_getvalue* (see page 8-15), and *WTmtable_setproperties* (see page 8-9), and as returned by the function *WTmtable_getproperties* (see page 8-11). Declared in *other.h*.

WTMAT_AMBIENT
WTMAT_AMBIENTDIFFUSE
WTMAT_DIFFUSE
WTMAT_EMISSION
WTMAT_OPACITY
WTMAT_SHININESS
WTMAT_SPECULAR

Mathematical Constants

Declared in *vr.h*.

PI, PID2, PID4, PID6, PIT2(PI = 3.14..., PI divided by 2, PI divided by 4, PI divided by 6, and PI times 2)

TRUE=1, FALSE=0

U=0, V=1

WTFUZZ=0.004

WTzero(x) (ABS(x) < WTFUZZ)

X=0, Y=1, Z=2, W=3

Message Constants

Used with the function *WTmessage_sendto* (see page 24-6). Declared in *other.h*.

Message types:

WTMESSAGE_ERROR

WTMESSAGE_USER

WTMESSAGE_WARNING

Message destinations:

WTMESSAGE_TOCALLBACK

WTMESSAGE_TOCONSOLE

WTMESSAGE_TOFILE

WTMESSAGE_TONOWHERE

Motion Link Source and Target Constants

Used with the functions *WTmotionlink_new* (see page 15-3), and as returned by the functions *WTmotionlink_getsource* (see page 15-6), and *WTmotionlink_gettarget* (see page 15-6). Declared in *other.h*.

Motion link sources:

WTSOURCE_PATH
WTSOURCE_SENSOR

Motion link targets:

WTTARGET_MOVABLE
WTTARGET_NODEPATH
WTTARGET_TRANSFORM
WTTARGET_VIEWPOINT

Node Constants

As returned by the function *WTnode_gettype* (see page 4-50). Declared in *other.h*.

WTNODE_ANCHOR
WTNODE_FOG
WTNODE_GEOMETRY
WTNODE_GLNODE
WTNODE_GROUP
WTNODE_ILLEGAL
WTNODE_INLINE
WTNODE_LOD
WTNODE_LIGHT
WTNODE_MGEOMETRY
WTNODE_MLIGHT
WTNODE_MLOD
WTNODE_MSEP
WTNODE_MSWT
WTNODE_ROOT

WTNODE_SEP
WTNODE_SWT
WTNODE_WTOBJECT
WTNODE_XFORM
WTNODE_XFORMSEP

Used with the function *WTfognode_setmode* (see page 4-66) and as returned by the function *WTfognode_getmode* (see page 4-66).

WTFOG_EXP
WTFOG_EXPSQUARED
WTFOG_LINEAR
WTFOG_NONE

Used with the function *WTsepnode_setcullmode* (see page 4-57) and as returned by *WTsepnode_getcullmode* (see page 4-57).

WTNODE_CULLAUTO
WTNODE_CULLOFF
WTNODE_CULLON

Used with the function *WTswitchnode_setwhichchild* (see page 4-57) and as returned by *WTswitchnode_getwhichchild* (see page 4-58).

WTSWITCH_ALL
WTSWITCH_NONE

Used with the function *WTmovnode_attach* (see page 5-11).

WTNODE_APPEND

Option Constants

Used with *WTuniverse_setoption* (see page 2-24), and as returned by the function *WTuniverse_getoption* (see page 2-27). Declared in *other.h*.

WTOPTION_3DSCHGTEXEXT
WTOPTION_MGENREADVCOLOR
WTOPTION_NEWMGENREAD
WTOPTION_NFFWRITE12
WTOPTION_NFFWRITEUV
WTOPTION_NFFWRITEV21
WTOPTION_NOAUTOALPHA
WTOPTION_NOPOSTQUIT
WTOPTION_OLD3DS
WTOPTION_OLDTEXTROT
WTOPTION_OLDWFRONT
WTOPTION_USEWTPUMP
WTOPTION_VERTWARN
WTOPTION_XFORMSCALE

Path Constants

Used with the function *WTpath_interpolate* (see page 14-6). Declared in *other.h*.

WTPATH_BEZIER
WTPATH_BSPLINE
WTPATH_LINEAR

Used with the function *WTpath_seek* (see page 14-18). Declared in *other.h*.

WTPATH_CURRENT
WTPATH_FIRST
WTPATH_LAST

Used with the function *WTpath_setdirection* (see page 14-19) and as returned by the function *WTpath_getdirection* (see page 14-20). Declared in *vr.h*.

WTDIRECTION_BACKWARD
WTDIRECTION_FORWARD

Used with the function *WTpath_setmode* (see page 14-21) and as returned by the function *WTpath_getmode* (see page 14-22). Declared in *vr.h*.

WTPLAY_CONTINUOUS
WTPLAY_OSCILLATE
WTPLAY_TOEND

Projection Type Constants

Used with the function *WTwindow_setprojection* (see page 17-14). Declared in *other.h*.

WTPROJECTION_ASYMMETRIC
WTPROJECTION_GENERAL
WTPROJECTION_ORTHOGRAPHIC
WTPROJECTION_SYMMETRIC

Rendering Constants

Used with the function *WTgeometry_setrenderingstyle* (see page 6-33).

WTRENDER_ALLMODES
WTRENDER_DEFAULT

Used with the functions *WTuniverse_setrendering* (see page 2-18) and *WTgeometry_setrenderingstyle* (see page 6-33), and as returned by the functions *WTuniverse_getrendering* (see page 2-20) and *WTgeometry_getrenderingstyle* (see page 6-35). Declared in *other.h*.

WTRENDER_ANTIALIAS
WTRENDER_BEST
WTRENDER_GOURAND
WTRENDER_LIGHTING
WTRENDER_NOSHADER
WTRENDER_PERSPECTIVE

WTRENDER_SMOOTH
WTRENDER_TEXTURED
WTRENDER_WIREFRAME

Sensor Constants

Used with the function *WTsensor_new* (see page 13-7). Declared in *sensor.h*.

WTSENSOR_DEFAULT

All the constants for a particular sensor are declared in *sensor.h*.

ANY STANDARD MOUSE

As returned by the function *WTsensor_getmiscdata* (see page 13-15).

Mouse button is currently down.

WTMOUSE_LEFTDOWN
WTMOUSE_MIDDLEDOWN
WTMOUSE_RIGHTDOWN

Mouse button has just been pressed.

WTMOUSE_LEFTBUTTON
WTMOUSE_MIDDLEBUTTON
WTMOUSE_RIGHTBUTTON

Mouse button has just been released.

WTMOUSE_LEFTUP
WTMOUSE_MIDDLEUP
WTMOUSE_RIGHTUP

Mouse button has been double clicked.

WTMOUSE_LEFTDBLCLK

WTMOUSE_MIDDLEDBLCLK
WTMOUSE_RIGHTDBLCLK

ASCENSION BIRD, FLOCK OF BIRDS, AND EXTENDED RANGE BIRD

Used with the function *WTbird_sethemisphere* (see page 13-43) and as returned by the function *WTbird_gethemisphere* (see page 13-43).

WTBIRD_AFT
WTBIRD_FORWARD
WTBIRD_LEFT
WTBIRD_LOWER
WTBIRD_RIGHT
WTBIRD_UPPER

As returned by the function *WTsensor_getmiscdata* (see page 13-15). These constants are for Ascension's 6DOF mouse.

WTBIRD_LEFTBUTTON
WTBIRD_MIDDLEBUTTON
WTBIRD_RIGHTBUTTON

CIS GRAPHICS GEOMETRY BALL, JR.

As returned by the function *WTsensor_getmiscdata* (see page 13-15).

WTGEOBALL_LEFTBUTTON
WTGEOBALL_RIGHTBUTTON

FAKESPACE MONOCHROME BOOM, TWO-COLOR BOOM2C, AND FULL-COLOR BOOM3C

For BOOMs with joystick feature.

WTBOOM_DOWN
WTBOOM_LEFT
WTBOOM_RESET

WTBOOM_RIGHT
WTBOOM_UP

For BOOMs with left and right buttons, as returned by the function *WTsensor_getmiscdata* (see page 13-15).

WTBOOM_LEFTBUTTON
WTBOOM_RIGHTBUTTON

FAKESPACE PINCH GLOVE SYSTEM

As returned by the function *WTsensor_getrawdata* (see page 13-15).

Pinch Glove fingers (common constant for left and right fingers).

WTPINCH_FINGERS
WTPINCH_INDEX
WTPINCH_MIDDLE
WTPINCH_PINKIE
WTPINCH_RING
WTPINCH_THUMB

Pinch Glove individual fingers (different constants for left and right fingers).

WTPINCH_LINDEX
WTPINCH_LMIDDLE
WTPINCH_LPINKIE
WTPINCH_LRING
WTPINCH_LTHUMB
WTPINCH_NOTOUCH
WTPINCH_RINDEX
WTPINCH_RMIDDLE
WTPINCH_RPINKIE
WTPINCH_RRING
WTPINCH_RTHUMB

FIFTH DIMENSION TECHNOLOGIES' 5DT GLOVE

As returned by the function *WTsensor_getmiscdata* (see page 13-15).

WTGLOVE5DT_ALL (WTGLOVE5DT_THUMB & WTGLOVE5DT_INDEX &
WTGLOVE5DT_MIDDLE & WTGLOVE5DT_RING &
WTGLOVE5DT_PINKY)
WTGLOVE5DT_CLOSED
WTGLOVE5DT_INDEX
WTGLOVE5DT_MIDDLE
WTGLOVE5DT_OPEN
WTGLOVE5DT_PINKY
WTGLOVE5DT_RING
WTGLOVE5DT_THUMB

LOGITECH 3D MOUSE (RED BARON)

As returned by the function *WTsensor_getmiscdata* (see page 13-15).

WTLOGITECH_FLAGBIT
WTLOGITECH_FLYING
WTLOGITECH_FRIBIT
WTLOGITECH_LEFTBUTTON
WTLOGITECH_MIDDLEBUTTON
WTLOGITECH_OUTBIT
WTLOGITECH_PEDESTALBUTTON
WTLOGITECH_RIGHTBUTTON
WTLOGITECH_SUSPENDBUTTON

LOGITECH SPACE CONTROL MOUSE (MAGELLAN)

As returned by the function *WTsensor_getmiscdata* (see page 13-15).

WTSPACECONTROL_BUTTON1
WTSPACECONTROL_BUTTON2
WTSPACECONTROL_BUTTON3
WTSPACECONTROL_BUTTON4
WTSPACECONTROL_BUTTON5
WTSPACECONTROL_BUTTON6
WTSPACECONTROL_BUTTON7
WTSPACECONTROL_BUTTON8
WTSPACECONTROL_BUTTONA

POLHEMUS STYLUS

As returned by the function *WTsensor_getmiscdata* (see page 13-15).

WTFASTRAK_STYLUSBUTTON_DOWN

SPACETEC IMC SPACEBALL

As returned by the function *WTsensor_getmiscdata* (see page 13-15).

Spaceball button is being held down.

WTSPACEBALL_BUTTON1
WTSPACEBALL_BUTTON2
WTSPACEBALL_BUTTON3
WTSPACEBALL_BUTTON4
WTSPACEBALL_BUTTON5
WTSPACEBALL_BUTTON6
WTSPACEBALL_BUTTON7
WTSPACEBALL_BUTTON8
WTSPACEBALL_PICKBUTTON
WTSPACEBALL_BUTTONS

Spaceball button has just been pressed.

WTSPACEBALL_BUTTON1_DOWN
WTSPACEBALL_BUTTON2_DOWN
WTSPACEBALL_BUTTON3_DOWN
WTSPACEBALL_BUTTON4_DOWN
WTSPACEBALL_BUTTON5_DOWN
WTSPACEBALL_BUTTON6_DOWN
WTSPACEBALL_BUTTON7_DOWN
WTSPACEBALL_BUTTON8_DOWN
WTSPACEBALL_PICKBUTTON_DOWN
WTSPACEBALL_BUTTONS_DOWN

Spaceball button has just been released.

WTSPACEBALL_BUTTON1_UP
WTSPACEBALL_BUTTON2_UP
WTSPACEBALL_BUTTON3_UP
WTSPACEBALL_BUTTON4_UP
WTSPACEBALL_BUTTON5_UP
WTSPACEBALL_BUTTON6_UP
WTSPACEBALL_BUTTON7_UP
WTSPACEBALL_BUTTON8_UP
WTSPACEBALL_PICKBUTTON_UP
WTSPACEBALL_BUTTONS_UP

SPACETEC IMC SPACEBALL SPACECONTROLLER

As returned by the function *WTsensor_getmiscdata* (see page 13-15).

WTSPACEBALLSC_BUTTON1
WTSPACEBALLSC_BUTTON2
WTSPACEBALLSC_BUTTONS

THRUSTMASTER SERIAL JOYSTICK

As returned by the function *WTsensor_getmiscdata* (see page 13-15).

WTJOYSERIAL_BOTTOMDOWN
WTJOYSERIAL_HATDOWN
WTJOYSERIAL_HATLEFT
WTJOYSERIAL_HATRIGHT
WTJOYSERIAL_HATUP
WTJOYSERIAL_SIDEDOWN
WTJOYSERIAL_TOPDOWN
WTJOYSERIAL_TRIGGERDOWN
WTJOYSERIAL_WCS1
WTJOYSERIAL_WCS2
WTJOYSERIAL_WCS3
WTJOYSERIAL_WCS4
WTJOYSERIAL_WCS5
WTJOYSERIAL_WCS6
WTJOYSERIAL_WCS7
WTJOYSERIAL_WCSTOGGLEA
WTJOYSERIAL_WCSTOGGLEB

Serial Port Constants

Used for the *port* argument to the sensor macros, (e.g., *WTspaceball_new* – see page 13-102). Declared in *serial.h*.

SERIAL1
SERIAL2

Sound Constants

Used with the function *Wtsound_setparam* (see page 20-12). Declared in *other.h*.

WTSOUND_DOPPLER
WTSOUND_FBPAN
WTSOUND_LOOPS
WTSOUND_LRPAN
WTSOUND_PITCH
WTSOUND_PLAYRATE
WTSOUND_PRIORITY
WTSOUND_SPATIALIZE
WTSOUND_VOLUME

Used with the function *Wtsound_setparam* (see page 20-12) for setting the *WTSOUND_PLAYRATE* parameter. Declared in *other.h*.

WTSAMPLERATE_8KHZ
WTSAMPLERATE_11KHZ
WTSAMPLERATE_16KHZ
WTSAMPLERATE_22KHZ
WTSAMPLERATE_32KHZ
WTSAMPLERATE_44KHZ
WTSAMPLERATE_48KHZ

Used with the function *Wtsound_setparam* (see page 20-12) for setting the *WTSOUND_SPATIALIZE* parameter. Declared in *other.h*.

WTSPATIALIZE_OFF
WTSPATIALIZE_ON

Sound Device Constants

Used with the function *WtSounddevice_open* (see page 20-3). Declared in *other.h*.

WTSOUNDDEVICE_CRE
WTSOUNDDEVICE_DS
WTSOUNDDEVICE_DWSTK
WTSOUNDDEVICE_SGI
WTSOUNDDEVICE_VSI
WTSOUNDDEVICE_WINMM

Used with the function *WtSounddevice_setparam* (see page 20-5). Declared in *other.h*.

WTSOUNDDEVICE_ABSORBDIST
WTSOUNDDEVICE_OUTPUT
WTSOUNDDEVICE_ROLLOFF
WTSOUNDDEVICE_ROLLOFFEXP
WTSOUNDDEVICE_SPATIALIZE

Used with the function *WtSounddevice_setparam* (see page 20-5) for setting the *WTSOUNDDEVICE_OUTPUT* parameter. Declared in *other.h*.

WTOUTPUT_HEADPHONE
WTOUTPUT_STEREO
WTOUTPUT_SURROUND

Used with the function *WtSounddevice_setparam* (see page 20-5) for setting the *WTSOUNDDEVICE_SPATIALIZE* parameter. Declared in *other.h*.

WTSPATIALIZE_OFF
WTSPATIALIZE_ON

Texture Constants

Used with the function *WTtexture_setfilter* (see page 10-25), and as returned by the function *WTtexture_getfilter* (see page 10-27). Declared in *other.h*.

```
WTFILTER_LINEAR  
WTFILTER_LINEARMAPLINEAR  
WTFILTER_LINEARMAPNEAREST  
WTFILTER_NEAREST  
WTFILTER_NEARESTMAPLINEAR  
WTFILTER_NEARESTMAPNEAREST
```

Used with the function *WTtexture_replace* (see page 10-16). Declared in *other.h*.

```
WTIMAGE_RGBA
```

User Interface Constants

Declared in *wtkuio.h*.

Size and position (some UI objects only).

```
WTUIATT_HEIGHT  
WTUIATT_LEFT  
WTUIATT_TOP  
WTUIATT_WIDTH
```

Label type (for UI objects created with *WTuiabel_new* only).

```
WTUI_FILE  
WTUI_TEXT
```

Scrolled list type (for UI objects created with *WTuisrolledtext_new* only).

```
WTUI_EDITABLE  
WTUI_NOTEDITABLE
```

Menu-item state (for UI objects created with *WTuimenuitem_new* only).

WTUI_DIM
WTUI_UNDIM

Window Constants

Constants used to set window characteristics. Declared in *other.h*.

WTWINDOW_DEFAULT
WTWINDOW_INTERLACEEVENODD
WTWINDOW_INTERLACEODDEVEN
WTWINDOW_NOBORDER
WTWINDOW_SCREEN1
WTWINDOW_SCREEN2
WTWINDOW_SCREEN3
WTWINDOW_SCREEN4
WTWINDOW_SCREEN5
WTWINDOW_SCREEN6
WTWINDOW_SCREEN7
WTWINDOW_SCREEN8
WTWINDOW_RBSTEREO
WTWINDOW_STEREO
WTWINDOW_STEREOVSPLIT

Other Constants

A forward slash (/) on UNIX, a backward slash (\) on Windows 32-bit systems.

WTFIELD_DELIM

A colon (:) on UNIX, a semicolon (;) on Windows 32-bit systems.

WTFIELD_PATHDELIM

The maximum length of a filename (which includes the full pathname).

WTPATHLEN

See your Hardware Guide for constants that are specific to your computer platform.

Error Messages and Warnings

In some circumstances, a WTK application may terminate with a diagnostic error message. These fatal errors signify a condition that should not occur in application code. In non-fatal situations, when WTK encounters an unusual condition, a warning message is provided. You can suppress or redirect error messages and warnings with the *WTmessage_sendto* function (see page 24-6).

The following sections list the WTK error and warning messages, and offer suggestions for what corrective actions you can take. If you obtain an error message that's not listed here, please contact SENSE8 Technical Support. For support information, see Appendix L, *Technical Support*.

Error Messages

Correct code not found in the WTKCODES file

To run a WTK application on some platforms, a system-specific code is required in a file called WTKCODES. If you also get one of the warnings “Couldn't find file of security codes, WTKCODES” or “Couldn't open file of security codes, WTKCODES”, then the problem is that the file could not be found or opened. Otherwise, the problem is that you are missing the correct code, and you should contact your distributor.

Couldn't get mode for <device>

On a UNIX platform, *WTserial_new* had problems with the device name that you attempted to use as a serial port.

Couldn't find file WTKLS. Is environment variable WTKLS properly set?

On UNIX platforms, with certain license types, a file called WTKLS is required for interactions with the license server. This file must be located in the same directory as the WTK executable.

Error <n> initiating contact with license server

This error indicates a problem obtaining a license from the WTK license server. If you get this error, it will usually have been preceded by a warning which provides information about the specific problem.

Incorrect license type? This library is for a class <x> license.

The library you are using is inappropriate for your system. Contact your distributor.

Internal fault <num>

WorldToolKit has encountered a prohibited condition internally. Please file a bug report, including the error number <num> and as detailed a description of the conditions which produced the problem as you are able to provide.

Invalid texture name: <texture name>

Texture names, when supplied from as a DXF layer name or in an NFF file, must begin with one of the strings: “_S_”, “_T_”, “_V_” to signify that the texture is to be shaded, transparent, or plain vanilla. When textures are supplied by a call to *WTPoly_settexture* or *WTGeometry_settexture*, the texture name should not contain the prefix, as texture type is given by other arguments to the texture function.

Make_arc - too many verts in arc

There is currently a restriction of 512 vertices for any arc specified in a DXF file being loaded into WorldToolKit. Change your DXF file geometry to limit the number of vertices in any arc.

No Z-buffer available

To run WTK on some platforms (for example, SGI), a Z-buffer (or software Z-buffering capability) is required.

OUT OF MEMORY!

An attempt to allocate more dynamic memory from the heap has failed, because your computer has exhausted its free memory. Either you are loading models too big for the amount of physical memory in your computer, or are running other programs that are occupying too much memory.

Polyline has more than <num> vertices

There is a restriction on the number of vertices which can be present in any polyline entity in a DXF file being loaded into WorldToolKit. This is currently set to 512. Change your DXF file geometry to limit the number of vertices in any polyline.

Scan.I - group code problem

The DXF parser has encountered a syntactic anomaly with the DXF file being read. Ensure the DXF file can be read by AutoCAD. It is possible that the contents of the file are corrupt or that it was generated by a third party DXF output program that does not generate standard DXF.

Serial port being deleted is invalid

You will only see this error if you are passing an uninitialized pointer to *WTserial_delete*.

Serial port <port>: data overflow

When a serial port is created, a buffer is created to hold characters as they are received. You should make the buffer large enough to handle the maximum number of characters that you expect will ever be waiting for processing. If the buffer is full and more characters come in, this message is printed and the application immediately terminates.

Spaceball not responding

The Spaceball device is not returning data as expected. Check that the unit is powered on, and check your serial cable and baud rate.

Unable to open serial port <devicename>

On a UNIX platform, *WTserial_new* had problems with the device which you attempted to use as a serial port.

Unrecognized baud rate <n>

The supported baud rates on UNIX platforms are 1200, 2400, 4800, 9600, 19200, and 38400, ... baud.

Write to device timed out

A problem was encountered during a *WTserial_write*. A hardware fault or memory corruption is indicated.

WTserial read - requested more than buffer size

When serial ports are created with a call to *WTserial_new*, a buffer size is supplied to indicate the size of the input buffer to create. An attempt to read more than this number of bytes with a call to *WTserial_read* is not meaningful, and results in this error message.

Warnings

<n> open POLYLINE entities ignored

or

<n> POLYLINE3D entities ignored

The DXF reader ignores open POLYLINE entities and non-coplanar POLYLINE3D entities in a DXF file. This warning lets you know that some POLYLINEs were ignored.

<n> polygons were ignored.

The NFF and Wavefront file readers ignore invalid polygons, for example, polygons with fewer than three vertices and non-coplanar polygons, and report how many were ignored.

<n> unused vertices were removed

When creating objects, WTK removes vertices that aren't referenced by any polygons.

All serial ports in use

The Windows 32-bit versions of WTK only support a maximum of four open serial ports.

Already opened serial port

This message is produced by the function *WTserial_new* when a serial port is being opened that has already been opened by your application and has not yet been deleted with a call to *WTserial_delete*. You cannot open the same serial port twice.

Bad Group Address

or

Bad Port

or

Bad Range

Please check your arguments to *WTnet_open*.

Baron not responding

The Logitech 3D Mouse (Red Baron) sensor is not returning data as expected. Check that the unit is powered on, and check your serial cable and baud rate.

Bird unit 1 must be opened first.

The Bird sensors in a Flock of Birds must be opened in order of their Bird addresses, starting with Bird 1.

Boom not responding

The Boom sensor is not returning data as expected. Check that the unit is powered on, and check your serial cable and baud rate.

Can't initialize the spaceball.

The SGI version of WTK uses the SGI-supplied Spaceball support. If this Spaceball support fails to find and initialize the Spaceball, this warning results. Check that the serial port to which the Spaceball is attached is configured for a Spaceball.

Can't make geometry - too few vertices

An object must have at least one polygon, which means it must have at least three vertices. Attempting to create an object with less than three vertices results in this warning.

Can't open image file <image>

The SGI version of WTK is having trouble opening an image file (.rgb file). Please check that the file is present either in the current directory or on the WTIMAGES path. Also check that it is in the correct file format, e.g., verify that the image can be viewed using the Irix image utility "ipaste."

Could not bind socket

When initializing networking, the port you attempted to use was possibly in use by another application. You could use the netstat command to find ports in use.

Could not create receive socket

See your network administrator for an explanation of why WTK was unable to create a socket for networking.

Could not find host name

Make sure that your WTHOSTS points to a valid hosts file.

Could not join group

Another application may be using the group address (see *WTnet_open* on page 22-7).

Couldn't configure serial port for Bird

The SGI version of WTK is having trouble with the serial port when initializing the Bird.

Couldn't find 3D font file <filename>

The *WFont3d_load* function was unable to find the specified file. Please check that the file is in the current directory or on the WTMODELS path.

Couldn't find file of security codes, WTKCODES

or

Couldn't open file of security codes, WTKCODES

To run a WTK application on some platforms, a system-specific code is required in a file called WTKCODES. If you also get one of the warnings “Couldn't find file of security codes, WTKCODES” or “Couldn't open file of security codes, WTKCODES”, then the problem is that the file could not be found or opened. Otherwise, the problem is that you are missing the correct code, and you should contact your distributor.

Couldn't get first Boom record.

Couldn't get first record for Bird unit <n>.

Couldn't get first record from CrystalEyesVR device.

Couldn't get first record from Logitech.

Couldn't get first record from Red Baron.

Couldn't open CrystalEyesVR device.

Couldn't open Logitech.

Couldn't open Red Baron.

For any of the messages above, please check the connections and baud rate for the specific device.

Couldn't get license

The WTK license server was unable to obtain a license. It may be that all licenses are already in use, or that your license isn't properly installed, or that your license has expired.

Current node is invalid

You are trying to play a path which does not have any current element. This should normally not occur.

Do not have a valid group address

See *WTnet_open* on page 22-7 for a description of valid group addresses.

DXF file line <n>, _yytext <text>

The DXF parser has encountered a syntactic anomaly with the DXF file being read. Ensure the DXF file can be read by AutoCAD. It is possible that the contents of the file are corrupt or that it was generated by a third party DXF output program that does not generate standard DXF.

Fastrak does not have a unit <n>

FASTRAKs only have up to four units, and as few as one may be available.

Fastrak does not have receiver <n>

The application attempted to open a receiver (sensor unit number) which the FASTRAK doesn't have. The sensor unit number for a FASTRAK must be a number from 1 to 4.

Fastrak not responding

The FASTRAK sensor is not returning data as expected. Check that the unit is powered on and check your serial cable.

Fastrak unit 1 must be opened first.

The receiver units of a FASTRAK must be opened in order, starting with unit 1 (one).

File <filename> doesn't contain usable 3D letters

The function *WTfont3d_load* didn't find any 3D character objects in the file. See *NFF 3D Font Files* on page 9-5 for a description of the format of 3D font files.

File <filename> has no readable objects

The NFF or 3DS reader encountered a file with no objects, or at least no objects containing geometry from which a WTK object could be constructed. An NFF or 3DS file should have at least one object.

File <filename> is not a path file

WTpath_load was called for a file that is not a path file (i.e., saved by *WTpath_save*).

File does not have correct image format: <filename>

Check that the specified file is the correct rgb file format. Verify, for example, that the file can be viewed using the Irix image utility "ipaste."

Font file <filename> contains more than one object for character <n>

WTfont3d_load encountered a duplicate object for some character (for instance, two objects named "char81"). In this case, the first object for this character is used.

Font file <filename> contains <n> non-character objects

WTfont3d_load encountered object with names not beginning with "char."

Function *WTuniverse_go* is not reentrant

or

Function *WTuniverse_go1* is not reentrant

These functions can not be called recursively, i.e., from inside the simulation loop. Make sure you aren't calling them from the universe action function or an object task function.

Geoball not responding

The Geometry Ball, Jr. sensor is not returning data as expected. Check that the unit is powered on, and check your serial cable.

Illegal host address <address>

A host address in your hosts file was not a legal internet address.

Incorrect Bird configuration format line <n>

The file bird.dat should be 37 lines, each with an ascii decimal number representing a code to send to the Bird.

Incorrect license type for your machine?

You may be trying to run a version of WTK which is intended for a different class machine. Contact your distributor.

Invalid interrupt <n> for serial port creation

Please check the value you are passing to the *WTserial_new* call. It must be a valid serial port interrupt.

Invalid quaternion passed to <function>

A common mistake is to initialize a quaternion with $q[X] = q[Y] = q[Z] = q[W] = 0$ which is not a valid quaternion. Use of an invalid quaternion like this produces unstable, undesirable results, so a warning is reported. To initialize a quaternion, set $q[W]$ to 1, not 0. See the function *WTq_init* on page 25-14.

License server initialization error

If this is a floating license, check whether the wtklsd daemon is running. If this is a nodelocked license, check that your license is properly installed.

Logitech device: diagnostics failed

The CrystalEyesVR or Logitech Head Tracker device is having trouble initializing. Check all connections on the device, and power cycle if necessary.

Logitech not responding...

The Logitech Head Tracker device is not returning data as expected. Check all connections on the device.

Mouse not found

WorldToolKit could not communicate with the mouse. Be sure your mouse driver is loaded, and that the mouse is properly cabled to your computer.

Net is already open

You should only open the network once.

No comprehensible geometry in DXF file

A DXF file must contain some 3D surfaces in order for WTK to load it; a DXF file containing only 2D and line entities can not be loaded.

No serial port found

An attempt was made to open a serial port using the *WTserial_new* call, but the requested serial port hardware was not found. Be sure that the serial port you are trying to open exists on your computer.

Polhemus not responding

The Polhemus ISOTRAK or ISOTRAK II sensor is not returning data as expected. Check that the unit is powered on, and check your serial cable and baud rate.

Polygon <n> has only <n> verts!

The NFF writer (*WTgeometry_save* or *WTnode_save*) found a polygon with fewer than three vertices, which is not valid and indicates degenerate geometry. If you see this warning, please file a bug report.

Polygon <x> has <n> vertices -- outside [3,256]

In WorldToolKit, polygons may not be degenerate; all polygons must have at least three vertices. The maximum number of vertices allowed is 256 (although for DXF files it is 512). A polygon with a number of vertices outside of this range is rejected.

Polygon vertices are not coplanar

Polygon vertices are expected to lie in a plane. A small amount of inaccuracy or non-planarity is tolerated (within the *WTFUZZ* value), but a polygon is rejected if it is too non-planar.

Red Baron Flag bit bad in 1st record

The Logitech 3D Mouse (Red Baron) is having trouble initializing. Check all connections on the device, and power cycle if necessary.

Requested resolution not supported.

In order to use the RGB 800x600 and PAL hi-resolution modes, you must set the resolution with the *860mode* utility so that the high resolution is the default resolution at application start-up.

Serial read timed out

Upon call to *WTserial_read*, a specified time can elapse for arrival of the requested number of bytes before the read fails with this error message, indicating the transmitter is no longer alive. The time-out value is currently set to three seconds. The time-out can only occur if the retry flag argument for *WTserial_read* is set to TRUE.

Spaceball has funny packet

or

Spaceball not responding

The Spaceball sensor is not returning data as expected. Check that the unit is powered on, and check your serial cable.

Texture <texturename> not found

Make sure that the specified texture file is in your current directory or on the WTIMAGES path.

Unable to parse 3D Studio file <filename>

A problem was encountered attempting to parse a 3D Studio file. You could check the file for validity by attempting to load it with 3D Studio itself. If the file loads with 3D Studio but not with WTK, please report the file to technical support.

Window functions are not supported on this platform

The application attempted to call *WTwindow* functions on a platform for which WTK does not support window management.

WTK 1.01 binary format is no longer supported.

The file you are trying to load was probably created by the old *universe_save* or *object_save* functions, which creates a binary format that cannot be read by current versions of WTK.

Writing a Sensor Driver

If you do not want to use WTK's prepackaged driver functions (the *openfn*, *closefn*, and *updatefn* functions provided for each sensor supported in WTK), or if you have a device that is not yet supported in WTK, then you will need to provide your own driver functions as described in this chapter.

Writing a sensor driver in WTK consists of providing arguments to the function *WTsensor_new*. This call, which creates a sensor object, has the format:

```
WTsensor *WTsensor_new(  
    int (*openfn)(WTsensor*),  
    void (*closefn)(WTsensor*),  
    void (*updatefn)(WTsensor*),  
    WTserial *serial,  
    short unit,  
    short location);
```

where

- *openfn* is a function that initializes the device
- *closefn* is a function that closes the device and cleans up
- *updatefn* is a function that gets records from the device
- *serial* is a serial port object as returned by the function *WTserial_new*
- *unit* is the Nth unit for multi-unit devices
- *location* is a value that should simply be set to *WTSENSOR_DEFAULT*.

Overview

WTK Math Conventions

The data read from your device must be made consistent with WTK's math conventions (see the *Math Library* chapter, starting on page 25-1). In particular, keep in mind the following:

- You may have to transform the position and orientation records from your device to be consistent with WTK's coordinate convention.
- In WTK, orientation records, including those stored with sensor objects, are stored in quaternion form. If you prefer to work with matrices or euler angles, or if your device returns orientation records in one of these representations, then you will need to convert these records into quaternion form as part of generating a new sensor record. Conversion functions *WTm3_2q* (see page 25-26) and *WTeuler_2q* (see page 25-27) are provided as part of the WTK math library.
- Orientation records must be stored in such a way that they operate from right to left. If you have a matrix that does not obey this convention (and it is a unitary matrix), call either *WTm3_transpose* (see page 25-22) or *WTm4_transpose* (see page 25-23) to generate an acceptable matrix. Similarly, if you have a quaternion which does not obey this convention, call *WTq_invert* (see page 25-15) to generate an acceptable quaternion.

Sensor Records Must Be Relative

All devices in WTK are expected to generate relative position and orientation records. If your device returns absolute records, then the driver function *updatefn* will have to compute the change in position and orientation since the last time the sensor was read. WTK provides the function *WTSensor_relativizerecord* (see page 13-24) that simplifies this task.

Constraining Sensor Records

If you want the ability to apply constraints to your sensor input – see the WTK functions *WTsensor_setconstraints* (see page 13-21) and *WTmotionlink_addconstraint* (see page 15-11) – then your sensor driver, namely the function *updatefn* should generate a sensor record that is consistent with the constraint flags set for the sensor.

There are 6 constraint flags, 3 for constraining translations (*WTCONSTRAIN_X*, *WTCONSTRAIN_Y*, and *WTCONSTRAIN_Z*) and 3 for constraining rotations (*WTCONSTRAIN_XROT*, *WTCONSTRAIN_YROT*, and *WTCONSTRAIN_ZROT*).

Scaling Sensor Records

Two scale factors, one for translations and one for rotations, are stored in the *WTsensor* structure. The WTK calls *WTsensor_setsensitivity* (see page 13-11) and *WTsensor_setangularrate* (see page 13-12) are provided so that these scale factors can be modified. For example, in many WTK applications, sensitivity values are scaled with the size of graphical entities in the universe.

If you wish to take advantage of this feature when writing your sensor driver, then multiply the translational values returned by your device by the value returned by *WTsensor_getsensitivity* (see page 13-12), and the angular values returned by your device by the value returned by *WTsensor_getangularrate* (see page 13-13).

If your device returns absolute (rather than relative) records, then it may not be desirable to scale rotation records, although scaling translation records may still be useful. For example, suppose your device is an absolute sensor worn on the head, used to track the viewpoint. In a realistic simulation, a 360 degree turn of the head should correspond to a 360 degree turn in the virtual world. If this is what is desired, then rotational input from the device should not be scaled by the value returned by *WTsensor_getangularrate*. Sensor input *will* still have to be relativized, however, as described in the section below on *updatefn*.

It may also be useful to scale input from the sensor by the largest value returned by the device. For translation records, then, the resulting scaled values would be in the range [*sensitivity*, *sensitivity*], where *sensitivity* is the value returned by the function *WTsensor_setsensitivity*, which is in the same units as distances in the graphical world. The advantage of this is that *sensitivity* can then be interpreted as a maximum speed along any axis, as described under the function *WTsensor_setsensitivity*. The same applies for rotation records and the value returned by *WTsensor_getangularrate*.

When a new sensor object is created (with *WTsensor_new*), the following default values are set:

```
WTsensor_setsensitivity(sensor, 1.0);  
WTsensor_setangularrate(sensor, PI/36.0); /* 5 degrees, in radians */
```

If sensor values are scaled in the manner described above, and the device is attached to the viewpoint object, then each time through the simulation loop, the viewpoint will translate at most 1 (one) distance unit along any axis and will rotate at most five degrees about any axis. These rates can be changed with calls to *WTsensor_setsensitivity* and *WTsensor_setangularrate*. The example shown in *Example 3: Update Function for Absolute Device (Pseudocode)* on page E-15 illustrates how to incorporate these scale factors into a sensor driver.

Talking to the Serial Port

Many sensors are serial peripheral devices. WTK contains routines for reading and writing to serial ports. See the *Serial Ports* chapter (starting on page 23-1) for more on this subject.

Include Files

Your sensor driver should have the following include statement:

```
#include "wt.h"
```

Assuming your compiler can find the path to this include file, which is supplied with WTK, all required type defines (such as that for *WTsensor*) should be found.

Driver Functions

You only need to refer to the driver functions *openfn*, *closefn*, and *updatefn* when you pass them in to the sensor object constructor function (*WTsensor_new*). Otherwise, you should not refer directly to the driver functions in your program.

openfn

```
int openfn(  
    WTsensor *sensor);
```

The purpose of *openfn* is to initialize the device.

If the device you are using returns absolute position and orientation records, obtain the first sensor record and store it with the sensor object, using this call:

```
WTsensor_setlastrecord(sensor, p, q);
```

where *p* is of type *WTp3*, and *q* is of type *WTq* (quaternion). Of course, *p* and *q* must be consistent with WTK's math conventions as described above. If your device returns orientation records in either matrix form or as euler angles, then the functions *WTm3_2q* (see page 25-26) and *WTeuler_2q* (see page 25-27) can be used to obtain the corresponding quaternion *q*.

(The absolute position/orientation record stored with the sensor object with the call *WTsensor_setlastrecord* is used in *updatefn* to generate a relative position/orientation record.)

Finally, if the device is to be polled each time through the simulation loop rather than streaming data continuously, then you should request the next record before exiting *openfn*.

On some platforms, certain serial devices require that the serial port's RTS signal be kept in either a high or low state or the device will not communicate with the serial port. See the function *WTserial_setRTS* on page 23-5 for more information. Also consult your hardware guide for other system-specific considerations concerning serial ports.

If you successfully managed to open the device, you should return a non-zero integer value. If you had problems in opening the device, you should return NULL or zero.

closefn

```
void closefn(  
    WTsensor *sensor);
```

The function *closefn* is called by WTK when you call *WTsensor_delete* (see page 13-10) or *WTuniverse_delete* (see page 2-5) (which in turn calls *WTsensor_delete*).

WTsensor_delete calls the *closefn* before calling *WTserial_delete* (see page 23-2) (which frees the sensor's serial port object) and before freeing the sensor's raw data structure, so that if necessary the serial port object and raw data can still be accessed from the *closefn*.

If your device is a serial port device, then with this function you can retrieve any remaining data that was sent from the device to the serial port. It is unnecessary to call *WTserial_delete* from the *closefn* because *WTsensor_delete* calls *WTserial_delete*.

updatefn

```
void updatefn(  
    WTsensor *sensor);
```

This function obtains a new sensor record. It is called automatically each time through the simulation loop by the WTK simulation manager for all WTK sensor objects.

The function *updatefn* has four parts:

1. A new data record is obtained from the device.
2. The new record is used to generate relative position (p) and orientation (q) values, where p is a *WTP3* and q is a *WTq* (quaternion). These values may be scaled, as described in *Scaling Sensor Records* on page E-3.
3. p and q are stored with the sensor object by calling

```
WTsensor_setrecord(sensor, p, q);
```

p and q make up the new sensor record. If the sensor is attached to a graphical object, for example, then p and q are used to change the object's position and orientation.

4. If the sensor is a serial port device and is being polled, the next record is requested.

Steps 1 and 4 require that you know how to “talk to” the sensor device. WTK provides utility functions for reading and writing to a serial port (see the *Serial Ports* chapter for a description of these functions). Or, you may provide your own routines for communicating with the device.

Step 3 simply involves calling *WTSensor_setrecord* exactly as shown above.

That leaves Step 2, that is, how to generate *p* and *q* from the data read from your device. How *p* and *q* are generated from the data read from the device is really up to you. For example, your input device might generate only X and Y coordinate information and button presses. (This is what the typical mouse device returns.) *Example 1: Update Function for the Mouse* on page E-8 is an example of a typical update function for such a device. This update function generates yaws from button presses, forward and back motion from Y screen values, and left and right motion from X screen values. (For more information specific to use of the mouse, see the *Sensors* chapter, starting on page 13-1.)

If your device returns position or orientation records that are three-dimensional, you may need to convert the data so that it is consistent with the WTK math conventions described above.

Then, if the resulting record is relative (that is, it corresponds to a change in position and orientation rather than an absolute position and orientation), you need only store this information in *p* and *q* and you are done with Step 2. If your orientation record is in matrix or euler angle form, then *q* may be obtained by calling *WTm3_2q* or *WTEuler_2q*.

If, on the other hand, the sensor record is absolute, you will need to turn it into a relative record. To do so, first convert the orientation record into a quaternion if not already stored that way. Then call:

```
WTSensor_relativizerecord(sensor, absolute_p, absolute_q, p, q);
```

where *absolute_p* and *absolute_q* are the absolute records passed in, and *p* and *q* are the relative records returned, which can then be passed in to *WTSensor_setrecord*.

Since the *WTSensor_relativizerecord* function uses the absolute sensor record from the last time through the simulation loop (which was stored with the call to *WTSensor_setlastrecord*), you need to call these functions with the new absolute record, for use next time through the loop. In other words, after the call to *WTSensor_relativizerecord*, you should call

```
WTSensor_setlastrecord(sensor, absolute_p, absolute_q);
```

if your device returns absolute records.

An alternative approach to writing an update function for a device returning absolute records is outlined in *Example 3: Update Function for Absolute Device (Pseudocode)* on page E-15.

Finally, you may wish to store other kinds of data such as button presses with the sensor object. This information might be used, for example, in the universe's action function as a trigger of activity (see the function *WTuniverse_setactions* on page 2-12). To store this data with the sensor, use the call

```
WTsensor_setmiscdata(sensor, x);
```

where *x* is a int. This data can then be retrieved with the call

```
x = WTsensor_getmiscdata(sensor);
```

If your driver is for a device supported in WTK, then you may wish to use the defined constants for button press and other data given in Appendix C.

Example 1: Update Function for the Mouse

This example provides an illustration of taking the input from a device that does not generate 3D position or orientation data and converting it into such a record. On some platforms, to have a mouse cursor appear as part of your mouse sensor driver, a function *WTmouse_drawcursor* must be called. Please consult your Hardware Guide.

```
/*  
 * Example of a mouse update function.  
 * This update function first obtains the raw screen coordinates  
 * and button presses from the mouse device by calling  
 * WTmouse_rawupdate.  
 *  
 * The sensor translation record p is then computed, with X screen  
 * values used to generate left/right motion, and Y screen values  
 * used to generate forward/back motion.  
 *  
 * The sensor rotation record is obtained from left and right button
```

```
* presses. Left button presses generate yaw left; right button presses
* generate yaw right.
*/
void mouse_myupdate(WTsensor *sensor)
{
    float wy;           /* to store yaw value, in radians */
    int buttons;       /* stores button press data */
    FLAG lbutton, rbutton; /* left and right button presses */
    WTmouse_rawdata *raw; /* raw mouse x,y record */
    WTp3 p;           /* for call to WTsensor_setrecord */
    WTq q;           /* for call to WTsensor_setrecord */
    float speed;      /* sensor sensitivity value */
    WTwindow *w;
    int x, y, width, height;
    /* get new raw data record from device */

    WTmouse_rawupdate(sensor);

    /* determine window mouse is in */
    w = WTmouse_whichwindow(sensor);
    if (!w) {
        WTp3_init(p);
        WTq_init(q);
        WTsensor_setrecord(sensor, p, q);
        return;
    }

    /* get new window height and width */
    WTwindow_getposition(w, &x, &y, &width, &height);

    /* get raw x and y mouse values in screen coordinates */
    raw = (WTmouse_rawdata *)WTsensor_getrawdata(sensor);

    /* transform raw screen values to translation record.
    Scale the values to lie between -speed and +speed */
    speed = WTsensor_getsensitivity(sensor);
    WTp3_init(p);
```

```
/* are any buttons currently down? */
buttons = WTsensord_getmiscdata(sensor);
lbutton = buttons & WTMOUSE_LEFTDOWN;
rbutton = buttons & WTMOUSE_RIGHTDOWN;

/* generate yaw value using button presses, and scaled
by the sensor's angular rate */
/* left button press */
if ( lbutton && !rbutton ) {
    wy = -WTsensord_getangularrate(sensor);
    WTeuler_2q(0.0, wy, 0.0, q);
}
/* right button press */
else if ( rbutton && !lbutton ) {
    wy = WTsensord_getangularrate(sensor);
    WTeuler_2q(0.0, wy, 0.0, q);
}
else {
    WTq_init(q);
}

/* store the record with the sensor */
WTsensord_setrecord(sensor, p, q);
}
```

Example 2: Driver for the Geometry Ball Jr.

This is an example of a complete sensor driver for a device returning relative records. The Geometry Ball Jr. is a desktop device produced by CIS Graphics, Inc. It senses forces and torques and returns relative x, y, z, roll, pitch, yaw records.

```
/*
* geoball.c: interface to the Geometry Ball jr.
* initialization; termination; calibration
* This is an example of writing a WTK device driver.
* This is not a standalone application.
*

```



```
* Copyright (c) 1990-1997 SENSE8 Corporation
*/

#include "wt.h"

#define ESC 0x1b          /* hex value of ESC character */
#define WTGEOBALL_NBYTES 12 /* 12 bytes for binary pos/orientation record */
#define WTGEOBALL_MAXVAL 128 /* maximum value returned by geoball */

/* command buffer - sent to ball */
static char cmd[3] = {ESC};

/*
 * Called to initialize the geoball
 */
int WTgeoball_open(WTsensor *sensor)
{
    WTserial *serial;
    char geoball[WTGEOBALL_NBYTES];

    /* get pointer to serial object */
    serial = WTsensor_getserial(sensor);

    /* allocate raw data struct */
    WTsensor_setrawdata(sensor, (void *)malloc(sizeof(WTgeoball_rawdata)));

    /* set Geoball polling mode to request */
    cmd[1] = 'R';
    WTserial_write(serial, cmd, 2);
    WTmsleep(300);

    /* set Geoball output mode to binary */
    cmd[1] = 'P';
    cmd[2] = 'C';
    WTserial_write(serial, cmd, 3);
    WTmsleep(300);

    /* request Geoball test record */
    cmd[1] = 0x05;
```

```
    WTserial_write(serial, cmd, 2);
    WTmsleep(150);

    /* read Geoball test record */
    if ( WTserial_read(serial, geoball, WTGEOBALL_NBYTES, TRUE) == -1 ) {
        WTwarning("Geoball not responding.\n");
        return FALSE;
    }

    /* request 1st Geoball record */
    WTserial_write(serial, cmd, 2);

    return TRUE;
}

/*
 * All done; free the geoball
 */
void WTgeoball_close(WTsensor *sensor)
{
    /* no special shutdown required */
}

/*
 * acquire a new record
 */
void WTgeoball_update(WTsensor *sensor)
{
    WTserial *serial;
    static char geoball[WTGEOBALL_NBYTES];

    /* rotation angles about x,y,z */
    float wx,wy,wz;
    float trans_factor,ang_factor;
    WTgeoball_rawdata *raw;
    WTpq l6d;
    static short got_bytes = 0;
```

```
/* allow 5 frames to get a complete record */
static short count = 0;
short need, got;

/* get pointers to serial object and raw data */
serial = WTsensor_getserial(sensor);
raw = (WTgeoball_rawdata *)WTsensor_getrawdata(sensor);

/* read up to the needed # of bytes at the serial port */
need = WTGEOBALL_NBYTES - got_bytes;
got = WTserial_read(serial, geoball+got_bytes, need, FALSE);
got_bytes += got;
if ( got!=need ) {
    /* if incomplete record at serial port, initialize sensor record */
    WTpq_init(&l6d);
    WTsensor_setrecord(sensor, l6d.p, l6d.q);
    WTsensor_setmiscdata(sensor, 0);
    WTp3_init(raw->p);
    WTp3_init(raw->w);
    count++;
    if ( count==5 ) {
        count = 0;
        WTwarning("Geoball not responding...\n");
        /* request a new record */
        WTserial_write(serial, cmd,2);
        got_bytes = 0; /* reset byte counter to start over */
    }
    return;
}
count = 0;
got_bytes = 0; /* reset byte counter now that have complete record */

/* put raw position data into raw data struct */
raw->p[X] = geoball[4];
raw->p[Y] = geoball[5];
raw->p[Z] = geoball[6];
raw->w[X] = geoball[7];
raw->w[Y] = geoball[8];
```

```
raw->w[Z] = geoball[9];

/* button presses */
WTsensor_setmiscdata(sensor, geoball[3]);

/* scale geoball inputs to get velocities in range */
trans_factor = WTsensor_getsensitivity(sensor)/WTGEOBALL_MAXVAL;
ang_factor = WTsensor_getangularrate(sensor)/WTGEOBALL_MAXVAL;

/* position */
l6d.p[X] = (float) geoball[4] * trans_factor;
l6d.p[Y] = (float) -geoball[5] * trans_factor;
l6d.p[Z] = (float) -geoball[6] * trans_factor;

/* orientation */
wx = (float) geoball[7] * ang_factor;
wy = (float) -geoball[8] * ang_factor;
wz = (float) -geoball[9] * ang_factor;

WTeuler_2q(wx,wy,wz,l6d.q);

/* store translational and rotational information */
WTsensor_setrecord(sensor, l6d.p, l6d.q);

/* request next record */
WTserial_write(serial, cmd, 2);
}
```

Note: Constraints apply to Sensors as well as Motion Links. If you need to constrain the sensor as a whole, use `WTsensor_setconstraints` (see page 13-21); whereas, if you need to constrain the motion of an entity that is attached to a sensor, use `WTmotionlink_addconstraint` (see page 15-11).

Example 3: Update Function for Absolute Device (Pseudocode)

This example illustrates an alternative approach to writing an update function for a device returning absolute position and euler angle records. The italicized function names indicate functions which would have to be written before this function could be used.

```
void pseudocode_update(WTsensor *sensor)
{
    WTp3 abs_p, abs_w; /* current absolute record */
    WTp3 last_abs_p, last_abs_w; /* last absolute record */
    WTp3 p, w; /* current relative record */
    WTq q; /* for WTsensor_setrecord */
    float trans_factor, ang_factor;

    /* read current absolute p, w */
    read_sensor(absolute_p, absolute_w);

    /* function to obtain prior absolute p, w, perhaps from the
       raw data storage (see WTsensor_getrawdata on page 13-15) */
    get_last(last_abs_p, last_abs_w);

    /* calculate relative p, w */
    WTp3_subtract(abs_p, last_abs_p, p);
    WTp3_subtract(abs_w, last_abs_w, w);

    /* get scale factors */
    trans_factor = WTsensor_getsensitivity(sensor)/MAXSENSORVAL;
    ang_factor = WTsensor_getangularrate(sensor)/MAXSENSORVAL;

    /* scale factor to relative record - note that you might
       not want to scale the rotation records */
    p[X] = p[X] * trans_factor;
    w[X] = w[X] * ang_factor;

    /* convert from XYZ relative euler angles to relative quat
       (see the documentation under WTeuler_2q on page 25-27) */
    WTeuler_2q(w[X], w[Y], w[Z], q);
}
```

```
/* store relative translation and rotation record */
WTsensor_setrecord(sensor, p, q);

/* store current absolute translation and rotation record,
perhaps to raw storage (see WTsensor_setrawdata on page 13-26) */
save_last(abs_p, abs_w);
}
```

WTK Neutral File Format

The NFF Format

The SENSE8 neutral file format (NFF) is a generic representation for polygonal geometry. NFF files are written in ASCII format.

In the NFF file, objects are represented as sets of polygons, and polygons are ordered collections of vertices. The format specifies polygon color and texture application, back face rejection, vertex normals, object names, and polygon IDs.

To save out a *geometry node* in the NFF format, use the defined constant `WTFILETYPE_NFF` as the filetype argument to the `WTnode_save` function (see page 4-48).

To save out a *geometry* in the NFF format, use the defined constant `WTFILETYPE_NFF` as the filetype argument to the `WTgeometry_save` function (see page 6-26).

To read in a geometry from a file in NFF format, simply call `WTnode_load` or `WTgeometrynode_load` with the filename of the file you wish to load (and other arguments as appropriate), and WTK will automatically detect whether the file is indeed in NFF format.

The BFF Format (Binary NFF)

WTK's NFF format is now supported in binary (BFF). The BFF format is identical to the NFF version, with the exception that 12-bit colors are not supported. (All colors are 24-bit.)

To save out a *geometry node* in the BFF format, use the defined constant `WTFILETYPE_BFF` as the filetype argument to the `WTnode_save` function (see page 4-48).

To save out a *geometry* in the BFF format, use the defined constant `WTFILETYPE_BFF` as the filetype argument to the `WTgeometry_save` function (see page 6-26).

To read in a geometry from a file in BFF format, simply call *WTnode_load* or *WTgeometrynode_load* with the filename of the file you wish to load (and other arguments as appropriate), and WTK will automatically detect whether the file is indeed in BFF format.

NFF Syntax

The following describes the current version of the NFF format. For changes from earlier versions, see *NFF Version History, Backward Compatibility* on page F-9.

NFF files contain a header and a set of one or more object specifications.

NFF files may have comments placed on any line. The characters “//” introduce a comment. All characters on the line following the “//” are ignored. The comments are not retained by WTK and are therefore not written out to an NFF file. The NFF reader is also very flexible with white space; any number of tabs or spaces are allowed before, between and after words in the file.

All lines must be terminated by a line feed character, but the PC end-of-line convention CR-LF (carriage return - line feed) is also supported. (Note that the CR-LF is read in, but isn’t written out. Only LFs are written out to an NFF file, since this maintains compatibility across platforms).

NFF Header

The file must begin with a line containing the string token *nff*. This is used by WTK to determine the type of the file.

The second line in the file should be the NFF version number. The current version is 3.0. Although the version number is optional, providing it ensures that the file will be read correctly even if the NFF format changes in the future. The optional viewpoint is specified as two lines with the tokens *viewpos* and *viewdir*. These specify the viewpoint’s location and view direction respectively.

Here is the entire header syntax:

```
nff
[version n.nn]
[viewpos x y z]
[viewdir x y z]
```

Here is an example of an NFF header:

```
nff
version 3.0
viewpos 0.000 0.000 0.000
viewdir 0.000 0.000 1.000
```

NFF Objects

Each object specification starts with a line of text giving the object's symbolic name, followed by the description of the geometry of the object. The syntax is as follows:

```
<objectname>
<material table reference>
<number of vertices>
<first vertex>
...
<last vertex>
<number of polygons>
<first polygon>
...
<last polygon>
```

An NFF file can contain any number of objects, each described by its own name and geometry. The file structure is:

```
<NFF header>
<first NFF object>
...
<last NFF object>
```

NFF Materials

After the name of the object, there is a reference to (i.e., the name of) a material table. WTK parses the specified material table and associates it with the NFF object.

The values for each of the materials used with a geometry are contained in its *material table*. A material table is a collection of “robust” colors. These colors are termed robust because they include more reflectance information than the ambient-diffuse color reflectance available in previous versions of WTK.

Material tables are indexed from 0 (zero) to the number of materials in the table. Each polygon or vertex contains an index into the material table. This means that each polygon or vertex has a number — not a color — attached to it. This number references an entry in the material table.

More than one geometry may point to the same material table, and a geometry may point to different tables depending on the effect you need. For more information, see the *Materials* chapter, starting on page 8-1.

NFF Vertices

After the NFF object name and the material table reference, the next line should be a single integer value defining the total number of vertices in the object. Vertex X, Y, Z coordinates, as real numbers, follow one per line. The vertex coordinate lines should contain three real numbers (as could be read in C with a “%f %f %f” format string). One or more spaces or tabs must separate the numbers.

If your hardware supports Gouraud-shaded polygons, you can optionally specify a normal vector for each vertex (this is used to calculate the shading intensity for each vertex).

The vertex normal is introduced by the keyword “norm” and is defined as three real numbers. Here is the vertex definition syntax:

```
<number of vertices>  
x y z [<norm x y z>]  
...  
x y z [<norm x y z>]
```

Here is an example of defining three vertices with vertex normals for Gouraud-shading:

```
3 // number of vertices to be defined
0.00 0.00 0.00 norm 0.707 0.707 0.00
-100.00 0.00 0.00 norm 1.00 0.00 0.00
0.00 100.00 0.00 norm -0.707 -0.707 0.00
```

If your hardware supports vertex colors, you can optionally specify a vertex color. If vertex colors are provided for all the vertices of a polygon, the polygon will be rendered with those colors instead of the polygon's color. The vertex color is introduced by the keyword “rgb” and takes the form 0xrrgb, a hexadecimal number in the range 0x000000 to 0xffffffff, with 8 bits each for red, green, and blue. For example, red is 0xff0000, black is 0x000000, white 0xffffffff and yellow 0xffff00.

This is an example of a vertex color:

```
0.00 2.00 0.00 rgb 0x0000ff // a blue vertex
```

Note that specifying the vertex colors in this format is not accepted in NFF files version 3.0 and later. In version 3.0 files, you should use material indices for each vertex that has color. The vertex is colored according to the material table entry that is referenced by the vertex's index.

This is an example of a vertex color in a version 3.0 file:

```
1.00 0.00 0.00 matid 2
```

It is assumed that the material table referenced by this geometry has an entry that corresponds to index=2.

A vertex may also specify an explicit texture coordinate (u,v). If present for all the vertices of a polygon, these (u,v) coordinates will precisely define the texture mapping. The vertex (u,v) values are introduced by the keyword “uv” and take the form of two floating-point numbers.

This is an example of a vertex uv value:

```
1.00 0.00 0.00 uv 0.5 0.5 // a uv coordinate
```

NFF Polygons

After defining the vertices, the next line in the NFF file contains the number of polygons in the object. Polygon specification lines follow, one for each polygon.

Each polygon specification line is in this form:

```
<#vertices><verticies><matid>[both] [<texture>[<attributes>]] [id=n] [<portal>]
```

The polygon specification line starts with an integer giving the number of vertices in the polygon. Following that is a list of vertex indices for the current polygon, with zero referring to the first vertex in the object’s vertex list. For back face rejection purposes it is important to note that the front face of a polygon is defined as the side of the polygon for which the vertices go around counter-clockwise. Back face rejection is further discussed below where the “both” flag is described.

After the list of vertex indices is a color designator that is specified by the keyword *matid* (for material index). *matid* is followed by an interger that associates a polygon with an entry in the geometry’s material table. Earlier versions of NFF files (prior to 3.0) used hexadecimal numbers to specify colors, ranging from 0x0 to 0xfffff.

The optional string “both” indicates that both sides of the polygon are to be visible. If “both” is not specified, then only front facing polygons, as defined above, are rendered.

Optionally, a texture name and attributes can be specified for the polygon. When texturing is on, color is ignored for the textured polygons since the surface properties come from the texture. The texture name specifies the file containing the bitmap to be used as a texture and also specifies whether the texture is to be plain, shaded, or transparent. Shaded textures have their brightness affected by the lights present in the model. Transparent textures are rendered so that all black pixels in the source bitmap are transparent when the texture is applied to a polygon. Texture names begin with the character “_”; the character following the “_” indicates the type of texture, according to the following:

v	plain vanilla texture (no shading)
s	shaded texture
t	transparent texture
u	shaded and transparent texture

For example, a texture named `_v_rug` causes a texture from a file named `rug` to be used. A texture named `_s_rug` would apply the same texture, but is shaded based on lighting.

You can specify texture attributes in two ways:

- By providing uv mapping coordinates for the vertices using the `uv` keyword as described in *NFF Vertices* on page F-4
- By using the following keywords in the polygon specification:
`[rot <value>] [scale <value>] [trans <value> <value>] [mirror]`

for texture rotation, scaling, translation, and mirroring respectively.

Rotations are specified in radians, and all operations are performed in u,v texture coordinate space. Any or none of these attributes may appear, but they must be placed after the texture name.

If every vertex in the polygon has uv coordinates specified, then these uv values are used to determine the mapping of the texture onto the polygon, and any keywords which may be present (`rot`, `scale`, `trans`) are ignored.

Regardless of the order of the attributes, when the polygon is loaded they are applied in the following order:

- mirroring
- rotation
- scaling
- translation

Since the NFF file's description of these texture attributes does not uniquely specify every possible transformation, if you require that files saved by WTK to retain their exact transformation when loaded back in, either apply your attributes in the same order (mirroring, rotation, scaling, and translation) before saving, or save your file using the vertex uv option (see *WTuniverse_setoption* on page 2-24).

The polygon's material color will be blended with the texture if you use the keyword `'tint'`. By default, blending does not take place, which means the polygon's color does not contribute to the texture. The keyword `'tint'` may be used anywhere on the polygon specification line after the polygon's `'matid'` has been specified. Using `'tint'` for a polygon that is not textured will not have any effect, even if the polygon is textured programmatically by the application.

Using the optional polygon ID token *id=n*, you can assign an integer value *n* to any polygon in your NFF file (for example: *id=567*). Then, from within your WTK application, you can use the *Wtpoly_getid* function (see page 7-7) to retrieve the ID number for the polygon in question. You can use this feature to “link” polygons in your NFF file with polygons in your application using the function *WTgeometry_id2poly* (see page 6-33).

Earlier versions of NFF files had portal information. This is ignored in version 3.0 and later. See *How Do I Handle Portals In This Release?* on page A-22.

This is a sample polygon specification, illustrating all possible options:

```
5 0 1 2 3 4 0xffff00 both _s_rug rot 1.0 scale 0.5 trans 1.0 1.0 id=5
```

This polygon has five vertices and is colored yellow, although the yellow will not appear unless you are rendering without textures. Both sides of the polygon are visible, and a shaded rug texture is applied. The rug texture is rotated one radian, scaled to half-size, and translated by (1.0,1.0) in (u,v) space. The polygon’s ID number is set to five and if the viewpoint crosses this polygon, the universe “rugworld” will be loaded (if using WTK version 2.1 functions).

NFF Format Extensions

Automatic Normal Generation

Since adding vertex normals by hand can be difficult, WorldToolKit supports an automatic normal generation procedure for NFF files (this doesn’t work for DXF or other file formats). To use this feature, you would add an “N” at the end of any vertex line for which you wanted WorldToolKit to calculate the normals. When the file is read into WorldToolKit, the “N” is replaced with an approximate vertex normal, based on the average of the polygons surrounding that vertex. This approximation may lead to an incorrect normals if the polygons are defined haphazardly. You may also encounter problems if some vertices are shared by polygons that are not Gouraud-shaded. In this case, you will have to make duplicate vertices - one with a vertex normal and one without. After reading in an object with automatic normals, you may want to write the object back out so that the next time it is read in, the normals are already calculated.

NFF Version History, Backward Compatibility

- 3.0 added reference to a material table used by the geometries; colors are no longer accepted as hexadecimal numbers. *matids* are used to associate vertices and polygons with entries in a material table; portal information is ignored.
- 2.1 no changes
- 2.09 binary nff file format introduced
 - added “rgb” keyword for vertex colors
 - added “uv” keyword for vertex texture mapping coordinates
 - added *_u_* designation for textures which are both shaded and transparent
- 2.0 added “norm” keyword to introduce vertex normals; allow 24-bit color for polygons
- 1.9 no changes
- 1.7 changed object shading to =on and =off instead of =flat and =none
- 1.6 first numbered version

To produce more compact files, texture application information is by default written out using the *rot*, *scale*, *trans*, and *mirror* parameters described under *NFF Polygons* on page F-6. (The same was true under WTK Version 2.0.) To have texture application information written out as vertex uv values instead, you must call:

```
WTuniverse_setoption(WTOPTION_NFFWRITEUV,TRUE);
```

before saving out your objects, or set the *writeuv* parameter to TRUE using the resource facility described in *Resource Files* on page 2-28. This option also pertains to BFF files.

The *rot*, *scale*, *trans*, and *mirror* values are only set by calls to *Wtpoly_rotatetexture*, *Wtpoly_scaletexture*, *Wtpoly_translatetexture*, and *Wtpoly_mirrortexture*.

If you make any calls to *Wtpoly_settextureuv*, *Wtpoly_setuv* or *Wtpoly_stretchtexture*, or if your textured objects were originally created using u,v information, write out your objects with the *writeuv* option set to TRUE to preserve the texture application information.

A Sample NFF File

The following is an example of a NFF file containing simple cube structures. Some polygons of the first cube are textured and some are not.

```
nff                // This is the first word of any NFF file.
version 3.0

// The following two lines are optional.
viewpos 0.0 0.0 0.0 // Viewpoint is at the origin
viewdir 0.0 0.0 1.0 // and looking straight forward.

SimpleCube        // Name of the object.
mtable mt1       //The material table referenced.
8                // Number of vertices.
3.0 3.0 -3.0     // Vertex info.
3.0 -3.0 -3.0
-3.0 -3.0 -3.0
-3.0 3.0 -3.0
3.0 3.0 3.0
3.0 -3.0 3.0
-3.0 -3.0 3.0
-3.0 3.0 3.0
6                // Number of polygons.
4 0 1 2 3 matid 0 both // id 0 of material table mt1 referenced
4 7 6 5 4 matid 0 both // "both" sides of the cube's faces are visible,
4 0 4 5 1 matid 0 both // so it is visible even from inside the cube.
4 1 5 6 2 matid 0 both _S_wings// A shaded texture called "wings"
4 2 6 7 3 matid 0 both _T_fish rot 1.0// A rotated, transparent fish texture
4 3 7 4 0 matid 0 both _V_kproom -kproom
// a portal to universe "kproom"
// with a texture called "kproom"

SecondObject     // Name of the object.
8                // Number of vertices.
9.0 9.0 -9.0 uv 0.0 0.0 // Vertex info, including texture uv.
9.0 -9.0 -9.0 uv 1.0 0.0
```



```
-9.0 -9.0 -9.0    uv 1.0 0.5
-9.0 9.0 -9.0    uv 0.0 0.5
9.0 9.0 9.0    rgb 0xff0000
9.0 -9.0 9.0    rgb 0xff8800 // orange
-9.0 -9.0 9.0    rgb 0x0000ff
-9.0 9.0 9.0    rgb 0xff0000
6                // Number of polygons.
4 0 1 2 3 0xff0000 both _T_fish// Texture is applied using vertex uv's
                        // to apply bottom half of texture to polygon
4 7 6 5 4 0x00ff00 both // Note that this polygon will use vertex colors
                        // instead of the polygon color!
4 0 4 5 1 0x0000ff both
4 1 5 6 2 0xffff00 both
4 2 6 7 3 0xffffff both
4 3 7 4 0 0x000000 both
```


Transitioning From Version 2.1 To Release 6/7/8/9

Introduction

While WTK Release 6/7/8/9 introduces many significant improvements and features, its new architecture and paradigms require a shift in the way you think about developing your real-time, 3D applications. This release offers backward compatibility support for your simulations created in WTK V2.1; simply recompile them and they will run correctly. However, if you want to modify the simulation or create a new simulation, it's important to first understand how the paradigm has changed.

This chapter provides key information to smooth your transition from WTK V2.1 to this release. The main sections of this chapter are as follows:

- *Paradigms of this Release* – briefly describes the major new features of this release. (see page G-2)
- *Mapping 2.1 Functions to this Release* – lists WTK V2.1 functions and their corresponding current release function. (see page G-9)
- *Details on Mapping WTK V2.1 Functions to this Release* – where there is not a direct mapping, this section discusses and guides you through how to accomplish the task in the current release. (see page G-22)
- *New Functions to Facilitate Incorporation of WTK V2.1 Applications into the R6/R7/R8/R9 Paradigm* – lists the new functions that have been added to facilitate the incorporation of WTK V2.1 applications into the Release 6/7/8/9 scene graph paradigm. (see page G-34)

Note: Throughout this chapter, the words “this release” and “the current release” are used to denote Release 6, Release 7, Release 8 and Release 9.

Paradigms of this Release

There are many new concepts and features in WTK. This section briefly introduces you to some of the major concepts, specifically the new scene graph and its content elements, instancing, materials, lights, special effects, 3D sound, user-interface elements, motion links, switch nodes, and level-of-detail nodes.

The Scene Graph

The scene graph is the most important addition to this release and affects many of the other new features. It provides a very powerful scene structure for real-time 3D simulation. Specifically, it provides the hierarchical framework for easily grouping objects together spatially. This is essential for maintaining performance in scenes which contain many individual objects. Because you can group objects together in a positional hierarchy, you are able to use the scene graph to easily construct and maintain efficient simulations that contain individual moving parts. See Chapter 3, *Scene Graphs*, for a detailed discussion on scene graph concepts and how to use WTK's functions to build your hierarchical scenes.

WTK lets you build a scene (simulation) by assembling geometries, lights, and positional information into a hierarchical structure (known as a scene graph). The scene graph is the structure that holds all of the current scene data, and dictates how the scene is rendered. You can think of the scene graph as an upside down tree, where the root is on the top and the branches and leaves are on the bottom.

In WTK V2.1 objects existed in the universe in a flat hierarchy as shown in figure G-1. If you wanted to attach objects together, you specifically called each object and told WTK how and where to attach it.

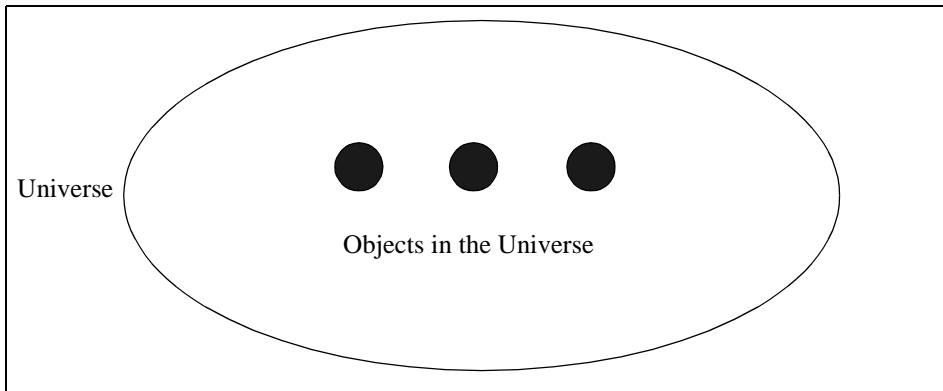


Figure G-1: Objects in WTK V2.1

This release provides a much broader range of nodes which are hierarchically arranged in the scene graph and each of which represent part of the simulation. Now, objects are replaced by geometries, which are part of the scene graph as shown in figure G-2.

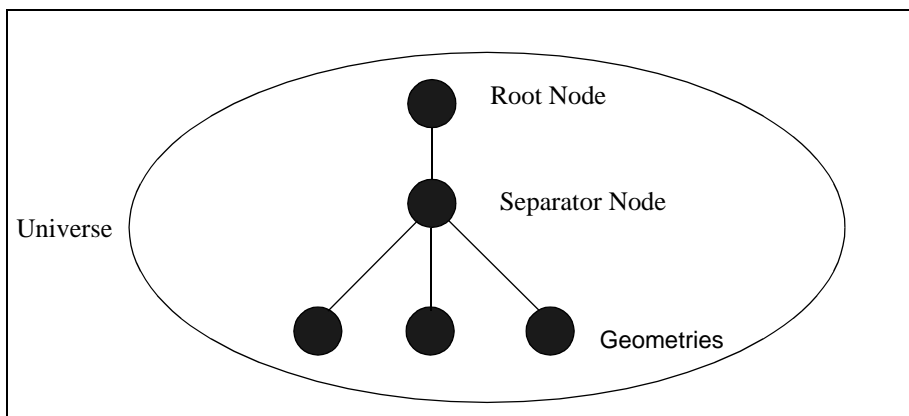


Figure G-2: Geometries in Current Release

In this release, a geometry must be attached to a scene graph to become visible in the simulation. This is a two-step process; first you create a geometry, then you create a geometry node so that you can attach it to the scene graph. The scene graph contains other parts of the simulation, like lights, LOD nodes, switches, etc. Only geometry nodes require

the two-step process of creation and attachment. All other scene graph components can be created in a single step by creating a node of the appropriate type.

Multiple scene graphs are also possible. This allows you to create different environments in a simulation and only display the environment that is currently being used. For example, if your simulation includes both an indoor and an outdoor environment, the universe would contain two scene graphs, each with different lighting and different geometries. As the user moves from the indoor to the outdoor environment, the outdoor scene graph becomes active. Previously, this transition between environments was accomplished using Portals. Portals are no longer needed. See *Changes in Reading/Writing NFF Files* on page G-24. Also see *How Do I Handle Portals In This Release?* on page A-22.

THE NODE

The *node* is the fundamental element of the scene graph, it is the basic building block that you use to construct scene graphs. A node is simply an element of content, or a grouping/organizational element used to maintain scene hierarchy. A node can serve several purposes. It can:

- represent an element of the simulation, like a geometry or light
- represent controlling elements in a simulation, like a switch node or level-of-detail node
- function as an unseen element in the simulation, like a transform node that gives position and orientation information
- prevent information about one part of the scene graph from influencing other parts of the scene graph, like separator node

In WTK V2.1, many of the functions were object (*WObject_xx*) functions. These functions are now node (*WNode_xx*) functions.

See in the *The Node* on page 4-5 of the *Scene Graphs* chapter for more information on node types.

Instancing

One of the advantages of a scene graph is the ability to *instance* a node. An instance is a reference to the original node. Instancing means you have only one object loaded into memory, but you can make as many references to it as you need. Each of these instances usually has a unique position in the scene graph. For example, you can have a model of one car that is instanced several places in the simulation, resulting in several cars on the road at the same time, all of which look the same. In a scene graph which has multiple instances of the same car model, a *node path* can be used to identify a specific *instance* of the car model. See the *Motion Links* chapter (starting on page 15-1), the *Scene Graphs* chapter (starting on page 4-1), and *Node Paths* on page 4-79 of the *Scene Graphs* chapter for more information.

Materials

While WTK V2.1 supported RGB color for objects, this release expands the concept to materials for geometries. A material is a combination of light and color attributes which you use to define the appearance of a geometry or collection of geometries. WTK functions let you create, edit, and save material information. Materials have the following properties:

- **Ambient:** The color reflected from the material without regard to light direction.
- **Diffuse:** The color reflected from the material as a function of light direction. This “diffuse” color corresponds directly with the WTK V2.1 concept of color.
- **Specular:** The color reflected from the highlights of the geometry. The specular material property is what makes a geometry appear to be “shiny” with highlights appearing on its surface.
- **Shininess:** The narrowness of focus of specular highlights. The higher the value, the shinier the appearance of the material. Shininess can range from 0 to 128 (floating point). The lower the shininess value, the more “spread out” the highlight is; the higher the shininess value, the sharper the highlight is.
- **Emissiveness:** The color of light produced (not reflected) by the material even when there is no light.
- **Opacity (Translucency):** The extent to which the color value of a pixel is combined with the color value behind. For example, a window has very little opacity, so it will pass the color of the objects behind it more than a wall, which is entirely opaque.

These material properties offer a substantial improvement over simple color for objects in WTK V2.1. Materials are contained in material tables. Each geometry that uses a material has its own material table, which stores its ambient, diffuse, and specular RGB values, as well as the other material properties. See the *Materials* chapter, starting on page 8-1, *Changes in Reading/Writing NFF Files* on page G-24, and *How Do I Use Material Tables for Colors?* on page A-11.

Note that you do not need to specify all of the material properties for a geometry. Using fewer fields can generate moderate improvements in performance. Also note that materials are *not* nodes.

Lights

In WTK R6/7/8/9, lights, along with geometries, positional information, and fog, make up the content elements (nodes) you use to build your scene. This release handles lights differently than WTK V2.1. One conceptual difference between the way WTK V2.1 handled lights and the way they are handled now, is that lights are now treated as nodes. This means when you create lights in your scene you have to specify a parent below which to add the light node.

In this release, a light node affects the elements that are to the right and below it in the scene graph. Thus, when you build your scene as a hierarchy of nodes, it matters where you place your light nodes in the scene graph. In WTK V2.1, however, lights illuminated every object in the universe. See *Handling of Lights in This Release* on page G-26, the *Lights* chapter (starting on page 12-1), the *Scene Graphs* chapter (starting on page 4-1), and *How Do I Associate A Task With a Particular Object?* on page A-21 for detailed information on lights.

Special Effects (Fog)

As mentioned above, fog nodes are content elements of a scene graph. You can use fog nodes to simulate special effects like fog, haze, smog, mist, smoke, and clouds in the atmosphere or general cloudiness for underwater simulations. You set the attributes of fog nodes to obtain these special effects.

Fog obscures distant objects in the scene more than closer objects. You can control the amount that objects are obscured, the distance at which objects begin to be obscured, and the distance at which objects are totally obscured by the fog. You can also specify whether the fog increases in a linear or exponential manner. See the *Scene Graphs* chapter (starting on page 4-1) for detailed information on fog nodes.

3D Sound

You can now enhance the realism of your scenes using 3D sound. 3D sound refers to the spatial characteristics of sound. See the *Sound* chapter, starting on page 20-1.

Multiple Windows

Multiple windows are not new to this release, but they are easier to implement. A viewpoint must be associated with each window in the universe. See the *Windows* chapter, starting on page 17-1.

User-Interface (UI) Objects

You can now add user interface features, like pushbuttons, toolbars, or menu bars to your simulations. Since WTK's UI functionality is cross-platform, you can design these UI elements to work both in Microsoft Windows and X/Windows. UI elements can pass information back, call child windows, and perform other functions associated with a graphical user interface (GUI) environment. See the *Adding User-Interface (UI) Objects* chapter, starting on page 18-1.

Motion Links

A motion link connects a *source* of position and orientation information with a *target* that moves to correspond with that changing set of information. The source can be a path or a sensor, and there are four types of targets: a viewpoint, a movable node, a node path, and a transform node. See *Moving from WTxx_addsensor to Motion Links* on page G-27, the *Motion Links* chapter (starting on page 15-1), the *Scene Graphs* chapter (starting on page 4-1), and the *Paths* chapter (starting on page 14-1) for more information.

Switches and Level of Detail Nodes

Switch nodes and level-of-detail (LOD) nodes are procedural elements that control how portions of the scene graph are processed. See *Scene Graph Concepts in Detail* on page 4-5.

Replaced Features

Several features of WTK V2.1 have been replaced with new features. These include:

portals	Portals existed to transition between one universe and another. For example, to move from indoors to outdoors. This transition is now done using multiple scene graphs. (See <i>Changes in Reading/Writing NFF Files</i> on page G-24, or the <i>portal.c</i> demonstration program in the demo directory of your WTK distribution. Also see <i>How Do I Handle Portals In This Release?</i> on page A-22.)
animation	Animation consisted of calling several objects in succession (that is, an animation sequence). You can now use a switch node now to accomplish the same thing. (See <i>Animation</i> on page G-31, and <i>How Do I Create A Simple Animation Using Switch Nodes?</i> on page A-26.)
terrain	Terrains are still supported in old universe files that were created in WTK V2.1. In new universe files, use a geometry and scale it to the size needed for a terrain. See <i>What Is Terrain Following?</i> on page A-31.

Mapping WTK V2.1 Functions To This Release

Where there is a fairly direct one-to-one mapping between a function in this current release and a WTK V2.1 function, the corresponding current release function is listed in the right column of table G-1. For those WTK V2.1 functions that do not have a direct one-to-one mapping to a function in this release, the right column refers you to a function that is similar in functionality to the WTK V2.1 function. Also see the notes following table G-1.

Table G-1: Version 2.1 functions mapped to current release

Version 2.1 Function	Current Release Function
WTanimation_*	See WTswitchnode_new. See <i>Animation</i> on page G-31, and see <i>How Do I Create A Simple Animation Using Switch Nodes?</i> on page A-26.
WTfont3d_textobject	WTgeometry_newtext3d
WTgeometry_load	WTgeometrynode_load See <i>Loading In Objects</i> on page G-22 and <i>How Do I Get A Pointer To A Node Using Its Name?</i> on page A-20.
WTgroup_*	See WTgroupnode_new. See <i>The Lack of WTgroup_* Functions</i> on page G-32.
WTlight_add	WTlightnode_new
WTlight_addsensor	See Wtmotionlink_new. See <i>Moving from WTxx_addsensor to Motion Links</i> on page G-27.
WTlight_delete	WTnode_delete
WTlight_deleteall	See <i>How Do I Associate A Task With a Particular Object?</i> on page A-21.
WTlight_getambient	WTlightnode_getintensity

Table G-1: Version 2.1 functions mapped to current release (continued)

Version 2.1 Function	Current Release Function
WTlight_getambientrgb	WTlightnode_getambient
WTlight_getangle	WTlightnode_getangle
WTlight_getattenuation	WTlightnode_getattenuation
WTlight_getdata	WTnode_getdata
WTlight_getdirection	WTlightnode_getdirection
WTlight_getexponent	WTlightnode_getexponent
WTlight_getintensity	WTlightnode_getintensity
WTlight_getposition	WTlightnode_getposition
WTlight_getrgb	See WTlightnode_getambient, WTlightnode_getdiffuse, and WTlightnode_getspecular.
WTlight_gettype	WTlightnode_gettype
WTlight_load	WTlightnode_load
WTlight_newdirected	WTlightnode_newdirected
WTlight_newpoint	WTlightnode_newpoint
WTlight_newspot	WTlightnode_newspot
WTlight_next	<i>See How Do I Associate A Task With a Particular Object?</i> on page A-21.
WTlight_remove	WTnode_remove
WTlight_remoovesensor	WTmotionlink_delete
WTlight_save	WTlightnode_save
WTlight_setambient	WTlightnode_setintensity
WTlight_setambientrgb	WTlightnode_setambient
WTlight_setangle	WTlightnode_setangle
WTlight_setattenuation	WTlightnode_setattenuation

Table G-1: Version 2.1 functions mapped to current release (continued)

Version 2.1 Function	Current Release Function
WTLight_setdata	WTnode_setdata
WTLight_setdirection	WTLightnode_setdirection
WTLight_setexponent	WTLightnode_setexponent
WTLight_setintensity	WTLightnode_setintensity
WTLight_setposition	WTLightnode_setposition
WTLight_setrgb	See WTLightnode_setdiffuse, WTLightnode_setambient, and WTLightnode_setspecular.
WLObject_add	See WTnode_addchild, Note #1.
WLObject_addperformer	(not needed)
WLObject_addsensor	See WTMotionlink_new. See <i>Moving from WTxx_addsensor to Motion Links</i> on page G-27.
WLObject_addvertex	WTgeometry_newvertex
WLObject_alignaxis	WTmovnode_alignaxis
WLObject_alignwithworldaxes	WTmovnode_alignaxis
WLObject_attach	WTmovnode_attach See <i>Attaching Objects To One Another</i> on page G-25.
WLObject_begin	WTgeometry_begin
WLObject_beginedit	WTgeometry_beginedit
WLObject_boundingbox	See WTnode_boundingbox.
WLObject_changecolor	See WTmtable_setvalue
WLObject_changergb	See WTmtable_setvalue.
WLObject_close	WTgeometry_close

Table G-1: Version 2.1 functions mapped to current release (continued)

Version 2.1 Function	Current Release Function
WObject_copy	WTgeometry_copy
WObject_delete	WTnode_delete
WObject_deleteperformer	(not needed)
WObject_deleteprebuild	WTgeometry_deleteprebuild
WObject_deletetask	WTtask_delete <i>See Differences in Applying Tasks on page G-29.</i>
WObject_deletetexture	WTgeometry_deletetexture
WObject_detach	WTmovnode_detach
WObject_endedit	WTgeometry_endedit
WObject_getanimation	See WTswitchnode_new. <i>See Animation on page G-31, and How Do I Create A Simple Animation Using Switch Nodes? on page A-26.</i>
WObject_getaxis	See WTnode_gettransform, Note #2.
WObject_getchildren	WTnode_getchild
WObject_getcolor	See Wtpoly_getrgb.
WObject_getdata	WTnode_getdata
WObject_gettextents	WTnode_gettextents
WObject_getgeometry	WTnode_getgeometry
WObject_gethandle	See WTgeometry_getmidpoint. <i>See Pivot Points And Handles on page G-32.</i>
WObject_getmidpoint	WTnode_getmidpoint
WObject_getname	WTnode_getname

Table G-1: Version 2.1 functions mapped to current release (continued)

Version 2.1 Function	Current Release Function
WObject_getpathnode	See WTPath_getmarker
WObject_getorientation	WTnode_getorientation
WObject_getpivot	See <i>Pivot Points And Handles</i> on page G-32.
WObject_getpolys	WTgeometry_getpolys
WObject_getposition	WTnode_gettranslation
WObject_getsensorframe	WTmotionlink_getreferenceframe
WObject_getvertexposition	WTgeometry_getvertexposition
WObject_getvertexnormal	WTgeometry_getvertexnormal
WObject_getradius	WTnode_getradius
WObject_getrendering	WTgeometry_getrenderingstyle
WObject_getrgb	See WTPoly_getrgb.
WObject_getshading	WTgeometry_getrenderingstyle
WObject_gettask	WTuniverse_gettaskbypointer
WObject_getvertexrgb	WTgeometry_getvertexrgb
WObject_getvertices	WTgeometry_getvertices
WObject_getvisibility	See WTnode_isenabled.
WObject_id2poly	WTgeometry_id2poly
WObject_insimulation	See WTnode_getparent, Note #1.
WObject_intersect	WTnodepath_intersectbbox
WObject_levelofdetail	See WTlodnode_new.
WObject_local2world	See WTnodepath_gettransform.
WObject_move	See WTnode_translate and WTnode_rotate.

Table G-1: Version 2.1 functions mapped to current release (continued)

Version 2.1 Function	Current Release Function
WObject_moveto	WTnode_settransform
WObject_ncolors	See WTmtable_getnumentries.
WObject_new	WTgeometrynode_new
WObject_newblock	WTgeometry_newblock
WObject_newcone	WTgeometry_newcone
WObject_newcylinder	WTgeometry_newcylinder
WObject_newextrusion	WTgeometry_newextrusion
WObject_newhemisphere	WTgeometry_newhemisphere
WObject_newrectangle	WTgeometry_newrectangle
WObject_newsphere	WTgeometry_newsphere
WObject_newtruncone	WTgeometry_newtruncone
WObject_next	See WTroutnode_next. See <i>Traversing the Scene Graph Tree</i> on page 4-9.
WObject_nextremoved	See WTnode_getparent, Note #1.
WObject_npolygons	WTgeometry_numpolys
WObject_prebuild	WTgeometry_prebuild
WObject_recomputestats	WTgeometry_recomputestats
WObject_remove	WTnode_remove, Note #1.
WObject_removesensor	See WTmotionlink_delete.
WObject_rotate	WTnode_rotateaxis

Table G-1: Version 2.1 functions mapped to current release (continued)

Version 2.1 Function	Current Release Function
WObject_rotatepoint	See WTgeometry_transform. <i>See Notes on Using WTgeometry_translate and WTgeometry_transform to Change the Vertex Postions on page G-28.</i>
WObject_save	WTnode_save
WObject_scale	WTgeometry_scale
WObject_setalpha	See WTgeometry_setmatid.
WObject_setaxes	See WTnode_settransform, Note #2.
WObject_setcolor	WTgeometry_setrgb
WObject_setdata	WTnode_setdata
WObject_sethandle	See WTgeometry_translate, Note #3. <i>See Notes on Using WTgeometry_translate and WTgeometry_transform to Change the Vertex Postions on page G-28.</i>
WObject_setname	WTnode_setname
WObject_setorientation	WTnode_setorientation
WObject_setpivot	See WTgeometry_translate. <i>See Pivot Points and Handles on page G-32.</i>
WObject_setpostion	WTnode_setposition
WObject_setrendering	WTgeometry_setrenderingstyle
WObject_setrgb	WTgeometry_setrgb
WObject_setsensorframe	WTmotionlink_setreferenceframe
WObject_setshading	WTgeometry_setrenderingstyle

Table G-1: Version 2.1 functions mapped to current release (continued)

Version 2.1 Function	Current Release Function
Wtobject_setsensorframe	WTmotionlink_setreferenceframe
Wtobject_settask	WTtask_new <i>See Differences in Applying Tasks on page G-29.</i>
Wtobject_settextureuv	WTgeometry_settextureuv
Wtobject_setuv	WTgeometry_setuv
Wtobject_setvertexnormal	WTgeometry_setvertexnormal
Wtobject_setvertexposition	WTgeometry_setvertexposition
Wtobject_setvertexrgb	WTgeometry_setvertexrgb
Wtobject_setvisibility	See WTnode_enable.
Wtobject_stretch	WTgeometry_stretch
Wtobject_translate	WTnode_translate
Wtobject_world2local	See WTnodepath_gettransform, Note #2.
WTpath_getobject	WTpath_getrecordlink
WTpath_setnodeobject	WTpath_setmarker
WTpath_setobject	WTpath_setrecordlink
WTpathnode_getobject	See WTpath_getmarker
Wtpoly_begin	WTgeometry_beginpoly
Wtpoly_getcolor	Wtpoly_getrgb
Wtpoly_getobject	Wtpoly_getgeometry
Wtpoly_getportal	See Note #3.
Wtpoly_gettexturetype	Wtpoly_gettexturestyle
Wtpoly_intersectobject	Wtpoly_intersectnode
Wtpoly_intersectobjpolys	See Wtpoly_intersectnode, Note #4.

Table G-1: Version 2.1 functions mapped to current release (continued)

Version 2.1 Function	Current Release Function
WTPoly_intersectpoly	WTPoly_intersectpolygon
WTPoly_intersectuniverse	See Note #4.
WTPoly_intersectunivpolys	See Note #4.
WTPoly_setcolor	WTPoly_setrgb
WTPoly_settexturetype	WTPoly_settexturestyle
WTportal_*	See Note #3.
WTuniverse_getanimations	See WTswitchnode_new. See <i>Animation</i> on page G-31, and <i>How Do I Create A Simple Animation Using Switch Nodes?</i> on page A-26.
WTuniverse_getbgcolor	WTuniverse_getbgrgb
WTuniverse_getentryfn	See WTuniverse_setactions, Note #3.
WTuniverse_getexitfn	See WTuniverse_setactions, Note #3.
WTuniverse_getextents	WTnode_getextents
WTuniverse_getframe	See WTnodepath_gettransform, Note #2.
WTuniverse_getintersectedpolys	See Note #4.
WTuniverse_getlights	See <i>How Do I Associate A Task With a Particular Object?</i> on page A-21. See <i>Handling of Lights in This Release</i> on page G-26.
WTuniverse_getmidpoint	WTnode_getmidpoint
WTuniverse_getname	WTnode_getname
WTuniverse_getobjects	See WTuniverse_getrootnodes.
WTuniverse_getpolys	See WTgeometry_getpolys.
WTuniverse_getportaling	See Note #3.

Table G-1: Version 2.1 functions mapped to current release (continued)

Version 2.1 Function	Current Release Function
WTuniverse_getradius	WTnode_getradius
WTuniverse_getremovedobjects	See WTnode_getparent, Note #1.
WTuniverse_getviewpoint	WTuniverse_getviewpoints
WTuniverse_id2poly	WTgeometry_id2poly
WTuniverse_intersect	See Note #4.
WTuniverse_load	WTnode_load <i>See Loading In Objects on page G-22.</i>
WTuniverse_name2object	WTuniverse_findnodebyname <i>How Do I Get A Pointer To A Node Using Its Name? on page A-20</i>
WTuniverse_npolygons	WTwindow_numpolys
WTuniverse_pickobject	WTwindow_pickpoly <i>See Picking on page G-31 and How Do I Pick The Frontmost Polygon At A Specific Point In A Specific Window? on page A-6.</i>
WTuniverse_pickpolygon	WTwindow_pickpoly <i>See Picking on page G-31 and How Do I Pick The Frontmost Polygon At A Specific Point In A Specific Window? on page A-6.</i>
WTuniverse_save	WTnode_save
WTuniverse_setbgcolor	WTuniverse_setbgrgb
WTuniverse_setentryfn	See WTuniverse_setactions, Note #3.
WTuniverse_setexitfn	See WTuniverse_setactions, Note #3.
WTuniverse_setname	WTnode_setname
WTuniverse_setportaling	See Note #3.
WTuniverse_vacuum	WTnode_vacuum

Table G-1: Version 2.1 functions mapped to current release (continued)

Version 2.1 Function	Current Release Function
WTvertex_delete	See WTgeometry_recomputestats, Note #5.
WTvertex_getnormal	WTgeometry_getvertexnormal
WTvertex_getposition	WTgeometry_getvertexposition
WTvertex_new	WTgeometry_newvertex
WTvertex_setnormal	WTgeometry_setvertexnormal
WTvertex_setposition	WTgeometry_setvertexposition
WTvertex_setrgb	WTgeometry_setvertexrgb
WTviewpoint_addsensor	See WTmotionlink_new.
WTviewpoint_getasymmetric	WTwindow_getprojection
WTviewpoint_gethithervalue	WTwindow_gethithervalue
WTviewpoint_getviewangle	WTwindow_getviewangle
WTviewpoint_getwindowparams	WTwindow_getparams
WTviewpoint_getyonvalue	WTwindow_getyonvalue
WTviewpoint_removesensor	See WTmotionlink_delete.
WTviewpoint_setasymmetric	WTwindow_setprojection
WTviewpoint_sethithervalue	WTwindow_sethithervalue
WTviewpoint_setviewangle	WTwindow_setviewangle
WTviewpoint_setwindowparams	WTwindow_setparams
WTviewpoint_setyonvalue	WTwindow_setyonvalue
WTviewpoint_zoomall	WTwindow_zoomviewpoint
WTwindow_getbgcolor	WTwindow_getbgrgb
WTwindow_pickpolygon	WTwindow_pickpoly
WTwindow_setbgcolor	WTwindow_setbgrgb

NOTES

1. *WObject_add*, *WObject_insimulation*, *WObject_nextremoved*, *WObject_remove*, *WUniverse_getremovedobjects*:

WTK V2.1 allowed for objects to be created and yet be inactive in the simulation. This release also allows for entities (nodes) to be inactive. A node is effectively inactive if it is not contained within the scene graph of any root node (i.e., if it is not directly or indirectly connected to a root node). A node that is connected to a root node may also be inactive if its root node is not associated with an active window. While the current release allows inactive nodes, WTK does not actively track them. If it's necessary for you to track inactive nodes, you can create a root node that is not associated with any window and then associate all inactive nodes with the "inactive" root node. For example, whenever you remove nodes from an active root node, you can attach them to the inactive root node, thereby always keeping track of the inactive nodes.

2. *WObject_getaxis*, *WObject_setaxes*, *WObject_world2local*, *WUniverse_getframe*:

WTK V2.1 allowed for objects to be independently defined in their own coordinate frame and therefore required functions for manipulating an object's coordinate frame. The current release incorporates the concept of a scene graph, so the coordinate frame of each object in the scene graph is relative to the coordinate frame of its parent node. Because of this, the current release allows for the creation and modification of transformation nodes to affect the position and orientation of objects and therefore eliminates the need for functions that directly manipulate coordinate frames. (See *Notes on Using WTgeometry_translate and WTgeometry_transform to Change the Vertex Positions*, on page G-28 and *Rotating A Movable About Its Midpoint* on page G-28.)

3. *WPoly_getportal*, *WPortal_**, *WUniverse_getentryfn*, *WUniverse_getexitfn*, *WUniverse_getportaling*, *WUniverse_setentryfn*, *WUniverse_setexitfn*, *WUniverse_setportaling*:

One significant change from WTK V2.1 to this release involves portals. All portal information in NFF files is now ignored. WTK no longer associates a polygon with a portal. (See *Changes in Reading/Writing NFF Files* on page G-24. Also see *How Do I Handle Portals In This Release?* on page A-22.)

4. *Wtpoly_intersectobjpolys*, *Wtpoly_intersectuniverse*, *Wtpoly_intersectunivpolys*, *WTuniverse_getintersectedpolys*, *WTuniverse_intersect*:

WTK provides a number of functions for testing the intersection of two objects, either on the polygon or the bounding box level. (See *Intersection Testing* on page 4-85. Also see *How Do I Test For Objects Intersecting With Other Objects In The Universe?* on page A-25.) The intersection functions require that the two objects you are testing be contained within a common scene graph, and that neither object is a subset of the other object (i.e., the scene graph sub-tree corresponding to one object does not also contain the scene graph sub-tree of the other object). Some WTK V2.1 functions like *Wtpoly_intersectobjpolys* returned a list of all polygons of the object that intersected with the specified polygon. The current release does not have any functions that return a list of polygons. If you need to obtain a list of intersected polygons, do it by calling *Wtpoly_intersectnode* to test whether a polygon intersects an object, and if so, then manually test each polygon of the object to see if it intersects the specified polygon (see *Wtpoly_intersectpolygon* on page 4-85).

5. *WTvertex_delete*: The current release does not allow you to directly delete vertices from geometries. WTK does allow, however, geometries to contain unused vertices (i.e., no polygons within the geometry reference the vertex). Thus, you shouldn't worry about extra unused vertices within your geometries. If you want to remove a vertex from an existing polygon, you can use the *WTgeometry_beginpoly* function to create a new polygon and then use the *Wtpoly_addvertex* function to add only those vertices which are still needed.

You can then use *Wtpoly_delete* to delete the previously defined polygon. If you then call *WTgeometry_recomputestats* with the *clearverts* flag set to TRUE, WTK removes all unused vertices from the specified geometry.

Details on Mapping WTK V2.1 Functions to This Release

This section discusses and guides you through how to accomplish certain tasks using the paradigms of this current release. It explains the following topics:

- loading in objects
- changes in reading and writing NFF files
- attaching objects to one another
- handling of lights in this release
- moving from *WTxx_addsensor* to motion links
- rotating a movable about its midpoint
- using *WTgeometry_translate* and *WTgeometry_transform* to change the vertex positions
- differences in applying tasks
- positioning and moving objects in your scene
- picking
- animation
- the lack of *WTgroup_** functions
- Pivot Points and Handles
- Coordinate Frames

Loading In Objects

To load in your objects in this release, you might use *WTnode_load*, *WTmovnode_load*, or *WTgeometrynode_load*. The following sections provide some important “tips” on using these functions to import objects.

WTNODE_LOAD

The *WTnode_load* function creates individual nodes for each geometry in the file being read. If your file has only one geometry this is equivalent to *WTgeometrynode_load*. *WTnode_load* maintains the hierarchical information present in the file. If the file has no hierarchical information (e.g., it just lists a bunch of geometries), then a node is created for each geometry and added to the specified parent. You could then move each geometry node independently of the other. Also see *How Do I Get A Pointer To A Node Using Its Name?* on page A-20.

WTMOVNODE_LOAD

The *WTmovnode_load* function loads movables; it is the best way to simulate WTK V2.1 object behavior. When using *WTmovnode_load*, however, you need to be careful if you have a file containing multiple objects and you want to move these independently of each other. If that is what you require, you would need to break it down into multiple files so that each file contains a single object. You have to do this because multiple objects loaded through *WTmovnode_load* move together, as they are assumed to be associated with each other to form a single entity. *How Do I Get A Pointer To A Node Using Its Name?* on page A-20

WTGEOMETRYNODE_LOAD

When using *WTgeometrynode_load*, if the file being imported has multiple geometries, this function merges all geometries into a single one and returns one geometry node. Note that this would be meaningless for filetypes that maintain hierarchical information (such as MultiGen and VRML). Therefore, this function is not available for these filetypes. Refer to the *Scene Graphs* chapter (starting on page 4-1) for more information on the filetypes that this function supports. Also see *How Do I Get A Pointer To A Node Using Its Name?* on page A-20.

GENERAL NOTES

Once you load your object, you can position it in your scene using the *WTnode_settranslation*, *WTnode_setrotation*, and *WTnode_orientation* functions. Keep in mind that the *WTp3* you pass into the *WTnode_settranslation* function may not be the same as you would have passed to *Wtobject_setposition*. This is because *Wtobject_setposition* sets the object's position to that value irrespective of its initial position.

WTnode_settranslation, on the other hand, acts on a transform node or the transform component of a movable node, and therefore, adds the value to the geometry's current position. For example, if you position a geometry at 15 units along the x-axis, and you apply a +15 unit translation through a *WTnode_settranslation* to it, it ends up at 30 units along the x-axis.

Note: The functions *WTgeometry_translate* and *WTgeometry_transform* change the geometry's vertex positions, so you can also use these to position geometries in your scene without having to deal with transform nodes.

Changes in Reading/Writing NFF Files

The transition from WTK V2.1 to the current release has resulted in some changes in the way NFF files are read. The major differences between the way this release handles NFF files and the way they were handled in WTK V2.1 are the following:

- Portal information in NFF files is now ignored.
- You can't save out NFF files of your universe that have multiple objects which were grouped previously as a *WTgroup*. You still have the ability, however, to save out single geometries as NFF files.
- This release uses material table information to give color to polygons and vertices rather than hexadecimal color values.

The first significant change involves portals. All portal information in NFF files is now ignored. WTK no longer associates a polygon with a portal. This obviously affects the behavior of your earlier applications. You can correct this by adding code in your actions function to check whether your viewpoint has crossed a portal polygon.

WTK now has a function, *WTviewpoint_intersectpoly* (see page 16-26), that tests whether the viewpoint has crossed a given polygon due to its motion in the current frame. Since this function checks for intersection with the polygon on a frame by frame basis, you should call *WTviewpoint_intersectpoly* in your actions function, so that it is executed once every frame. This is not done by default within WTK to give you more control over what needs to be done if a portal is crossed. Also, from a performance perspective, you have the ability to turn off the test depending on where the viewpoint is. To determine which polygon to perform the test on, you should mark the portal polygon with a Polygon ID in your NFF file and retrieve it in your application. (See the example demonstration program *portal.c*, located in the demo directory of your WTK distribution. Also see *How Do I Handle Portals In This Release?* on page A-22.)

Another change is that this release does not have the concept of *WTgroup*. If you load in your objects into WTK using *WTnode_load*, you will not be able to use *WTgroup* functions like *WTgroup_saveobjects*. This prevents you from writing out NFF files of your universe that have multiple objects which were grouped previously as a *WTgroup*. You still have the ability, however, to save out single geometries as NFF files. You have two options for saving out a scene graph with multiple geometries. The more intuitive way, as far as this release is concerned, is to save it out as a VRML file. This option keeps the hierarchical structure intact, and preserves the different node types. The other option is to merge all of the geometries into one file, and then save the file in an NFF or DXF format. Remember that when you save a file using this option, and read it back in, you will have only one geometry with no hierarchy. (See the example on how to take into account the relative positions of the geometries before using *WTgeometry_merge* on page 6-27.)

The last major change in the way NFF files are read is with the use of material tables. In WTK V2.1 NFF files, you associated polygons and vertices with colors specified as 12-bit or 24-bit hexadecimal numbers. The current release uses material table information to give colors to polygons and vertices. WTK now associates each geometry with a material table. The material table has entries for every color and material property that the geometry uses. Each polygon (and vertex, if the geometry has vertex colors) is associated with a Material ID (*matid*) that references an entry in the material table. If the version number at the top of your NFF file says 3.0 or greater, the NFF reader will not accept hexadecimal colors specified for each polygon and vertex.

Attaching Objects To One Another

WTK V2.1 provided a function called *WTobject_attach*, which you could use to create a hierarchical relationship between objects. For example, you could have a base object with one or more objects “attached” to it, so that when the base object moved, all of the attached objects moved along with it, as one entity.

In this release, the *WTmovnode_load* (see page 5-5) function lets you create such a hierarchical relationship between geometrical entities. The basic steps are as follows:

1. Create your base object as a movable (using *WTmovnode_load* or *WTmovgeometrynode_new*).
2. Use *WTmovnode_attach* (see page 5-11) to attach any other movable or geometry node to this movable base object.

The following example illustrates how to attach geometrical entities:

```
{
    earth = WTmovnode_load( rootnode, "earth.nff", 1.0f);
    moon = WTnode_load( NULL, "moon.nff", 1.0f);
    WTmovnode_attach( earth, moon, 0);
}
```

After the attachment is made, the base node and the attached nodes move as one entity. The individual attachments can still move independently of each other, but a movement of the base node moves all of the attached nodes, as if they were rigidly connected.

You can create complex hierarchical structures through multiple calls to *WTmovnode_attach*. (See the example demonstration program *arm.c*, located in the demo directory of the WTK distribution.)

When a node in the hierarchy moves, all of the nodes that are hierarchically below it move with it. Nodes that are hierarchically above this node are not affected by its motion. (Remember, the bounding box of the base node now encompasses all of the attached nodes.)

Refer to the *Movables* chapter, starting on page 5-1, for more information about the other functions that concern movable node attachments. Also see *How Do I Associate A Task With a Particular Object?* on page A-21.

Handling Of Lights In This Release

One conceptual difference between the way WTK V2.1 handled lights and the way they are handled now, is that lights are now treated as nodes. This means when you create lights in your scene you have to specify a parent below which to add the light node.

In this release, a light node affects the elements that are to the right and below it in the scene graph. Thus, when you build your scene as a hierarchy of nodes, it matters where you place your light nodes in the scene graph. In WTK V2.1, however, lights illuminated every object in the universe.

You can take advantage of this state-dependent lighting to produce lighting effects which were not possible using WTK V2.1. You can use a separator node (see page 4-21) to constrain the areas that a light node may illuminate. A separator node placed (in the scene

graph) above a light node does not allow light information to “seep” past it to its parent node or to the parent node’s sibling nodes.

In the current release, WTK does not maintain a list of light nodes that are created. So, the WTK V2.1 function *WTuniverse_getlights* is no longer applicable. You will have to traverse through your scene graph and get a pointer to the light nodes as necessary (see *Traversing the Scene Graph Tree* on page 4-9).

An added feature in this release is that you can load and create your lights as movables. This gives you the ability to move light nodes around, attach nodes to them, and apply any of the transform functions to them. See *How Do I Associate A Task With a Particular Object?* on page A-21.

Moving from *WTxx_addsensor* to Motion Links

WTK V2.1 did not have the concept of motion links. Subsequent releases use motion links to associate a sensor with a viewpoint (or a transform node or a movable node). The advantage to this approach is you have more control over the way a sensor associates with an entity.

For example, previously if a viewpoint and an object were controlled by a sensor, constraints applied on the sensor would affect the behavior of both the viewpoint as well as the object. By replacing the following two lines of code:

```
WTviewpoint_addsensor( view, sensor);  
WTobject_addsensor( object, sensor);
```

with

```
link1 = Wtmotionlink_new( sensor, view, WTSOURCE_SENSOR,  
                          WTTARGET_VIEWPOINT);  
link2 = Wtmotionlink_new( sensor, node, WTSOURCE_SENSOR,  
                          WTTARGET_MOVABLE);
```

you have more control over the way the sensor moves each entity. You can apply constraints on each motion link rather than the sensor itself.

WTK maintains a list of the motion links that have been created in your application. If you need to modify or manipulate a particular motion link, you can access this list by calling

the *WTuniverse_getmotionlinks* (see page 2-17) function to access the first motion link. You can call *WTmotionlinks_next* to traverse this list. For more information on related functions, see the *Motion Links* chapter, starting on page 15-1.

Note: For backward compatibility, and ease-of-use in certain situations, the function *WTsensor_setconstraints* is still available.

Rotating A Movable About Its Midpoint

To simulate the WTK V2.1 *WtObject_alignaxis* function, you can use the new *WTmovnode_alignaxis* (see page 5-8) function. This function rotates the movable node about its midpoint in such a way that the specified axis of the movable now aligns with (i.e., points in the same direction as) the direction vector. Note that this function is not available for regular transform nodes or geometry nodes.

The following example aligns a graphical object with a light (it assumes that “flashlight” is a movable created with *WTmovnode_load*, and “lightnode” is a directional light node):

```
{
    WTP3 dir;
    WTlightnode_getdirection( lightnode, dir );
    /* X axis assumed to point along flashlight length */
    WTmovnode_alignaxis( flashlight, X, dir );
}
```

Changing Vertex Positions

This section discusses using *WTgeometry_translate* and *WTgeometry_transform* to change vertex positions. Before using these functions, keep in mind that they change the vertex positions of a geometry and do not retain the original vertex positions. Thus, if you are planning on saving out your scene graph and then reading the saved file back in, you should not use these functions in place of a function such as *WtObject_setposition*.

Also, *WTgeometry_translate* is not the same as *WTnode_translate*, since the latter just updates a transform node's matrix without modifying the vertex positions. You should not have calls to either *WTgeometry_translate* or *WTgeometry_transform* in your action function. This is because

1. it slows down your application (recalculating every vertex's position, and updating the geometry's extents), and
2. for such purposes, you should be using a transform node with a call to *WTnode_translate*.

In summary, you should use these functions as utility functions to modify your geometries and save them out, rather than having to make use of a modeler. However, if you are certain that altering the geometry's vertex positions will have no undesirable results, these functions become very useful in positioning static geometry in a scene as necessary. You could then eliminate the need for loading static geometries as movables.

Differences in Applying Tasks

The function *WtObject_settask* has been replaced with *WTtask_new*. This gives you more flexibility in creating tasks and assigning them to objects.

The *WtObject_settask* function imposed the restriction of associating only one task per object. You can now repeatedly call *WTtask_new* on an object and assign as many tasks as necessary to it. Note that this requires you to explicitly delete tasks that you no longer want active. In WTK V2.1, however, this was done for you since the new task *replaced* an existing one.

The following line:

```
WtObject_settask( object, taskptr );
```

should be replaced with:

```
WTtask_new( node, taskptr, 1.0 );
```

Use *WTtask_delete* to delete tasks.

See *How Do I Associate A Task With a Particular Object?* on page A-21.

Positioning And Moving Objects In Your Scene: *WObject* and *WTgeometry*

There is a fundamental difference between the concept of an object as used in WTK V2.1 and the concept of a geometry as used now. Both *WObject* and *WTgeometry* contain the geometric details of the graphical entity they represent. In WTK V2.1, to move an object around in the universe, you directly manipulated its position and orientation. To manipulate a geometry, however, you use transform nodes.

A geometry consists of vertices and their positions, polygonal information, and surface definitions. A geometry by itself cannot be a part of a scene graph. After you create a geometry, a “geometry node” is built with this geometry and added into the scene graph. If this node is not affected by any transformation nodes, it will be rendered exactly where the vertex positions are defined. To alter the rendering position of the geometry, you should insert a transform node before it, and call *translate* or *rotate* functions on the transform node. You cannot move a geometry node directly because there are no functions like *WObject_setposition* (or *WObject_move*) that can be applied on a geometry node.

If you have a static geometry (a geometry that does not move during a simulation) in your scene and you want to position it at a particular spot, you could use *WTgeometry_translate* (or *WTgeometry_transform*). Keep in mind that these functions alter the vertex positions, and saving out the scene graph results in the geometry being saved with these new positions. These functions are useful for geometries that are not going to move.

The process of creating a transform node for every geometry is simplified by using movables. Movables have “built-in” transform components, so all transform node functions are applicable to movable nodes as well. In converting your WTK V2.1 applications to this release, your task will become easier if you make all your objects movables. Refer to the *Movables* chapter (starting on page 5-1) for more information.

It is possible that a geometry node is affected by more than one transform node. The *Scene Graphs* chapter (starting on page 4-1) describes the use of separator nodes to block transformations. It also describes how transforms accumulate and the use of node paths to obtain the resultant (transformation) matrix for a series of transform nodes.

Picking

WTK V2.1 provided for selecting of polygons and objects based on their projection onto a 2D window through functions such as *WTuniverse_pickpolygon* and *WTuniverse_pickobject*.

There are two main differences in the way you perform picking in this release. The relevant functions are *WTwindow_pickpoly* and *WTscreen_pickpoly*. Both of these functions return the polygon that was picked. They do not return a pointer to the object (node) that was selected because, in a hierarchical structure, it makes more sense to return a node path to that node, rather than the node itself. This is because it is possible that the node is referenced multiple times in the scene graph, and a node path tells you exactly which instance of the node was picked. You should pass in a non-NULL node path as an argument to *WTwindow_pickpoly* or *WTscreen_pickpoly*, which will be appropriately filled in by WTK. You can then obtain the last node (the one that was selected) on this node path.

Also, in WTK V2.1, the picking functions returned NULL if the universe was being rendered in wireframe mode. This is no longer true. Both *WTwindow_pickpoly* and *WTscreen_pickpoly* return the selected polygon, if it is visible, irrespective of the rendering mode.

See *How Do I Pick The Frontmost Polygon At A Specific Point In A Specific Window?* on page A-6.

Animation

This release does not offer an automated procedure for animations. You have to build your animation sequences with the help of switch nodes. See the *Scene Graphs* chapter (starting on page 4-1) for more information about these node types.

A switch node allows you to determine which of its children is to be rendered in a particular frame. You can create a switch node with a sequence of geometry nodes as its children, so that each child corresponds to a frame in an animation sequence. You can then cycle through the children, switching once for every frame.

We have also included a code example to show how to use switch nodes to generate a simple animation (see *How Do I Create A Simple Animation Using Switch Nodes?* on page A-26). Apart from switch nodes, you might also use WTK pathing functions to record the motion of either the viewpoint, or an object, and play it back.

The Lack of *WTgroup_** Functions

In a hierarchical structure, every sub-tree of your scene graph is a group. Typically, separator nodes and group nodes are used to collect related nodes below them to form a group. See the *Scene Graphs* chapter, starting on page 4-1.

A change to the sub-tree at the head affects all the nodes below it. For example, if you have a transform node as the first child in a sub-tree, you can move the whole sub-tree as a collective entity, by just altering this transform node. You can also build a group by creating a movable separator node that has a number of associated nodes as children. These nodes can then be moved independently of each other, as well as one rigid entity.

WTgroup_saveobjects was a useful function to save multiple objects into a file. This has been replaced with *WTnode_save*, which can save the entire sub-tree below a specified node into a VRML file, maintaining the scene structure and hierarchical information. Another useful function is *WTgeometry_merge* which merges two geometries into one. Using node paths you can maintain the relative positions of the merged geometries. This way you can write out multiple objects as one geometry. (See page 6-27 for a description of the *WTgeometry_merge* function and an example.)

Calls to *WTgroup_getobjects* and *WTgroup_nextobject* should be replaced with the code that recursively traverses the relevant sub-tree and returns the geometry nodes sequentially. (See *Traversing the Scene Graph Tree* on page 4-9 and *How Do I Measure Performance On My Machine?* on page A-38.)

Pivot Points And Handles

In WTK V2.1, geometries were moved and oriented around a midpoint (the center of the bounding box of the object) by default. If you wanted to rotate an object around something other than the default pivot point (the center of the bounding box), you had to use the *WtObject_setpivot* function.

The most “natural” way to move a geometry, however, is to do it in reference to some coordinate reference frame. Typically, the object’s coordinate frame is used. Keeping this point in mind, geometries are usually created in what becomes their “local” coordinate frame (i.e., the origin in the modeling software defines this local coordinate reference frame).

This release uses this new, more natural paradigm, which will simplify your code and improve efficiency. In this release, when you load a geometry, what was the origin of the model becomes the local coordinate reference frame (and therefore the pivot point). In most instances, you will not need to change the pivot point.

This new approach may require you to take a look at the way geometries are modeled. Instead of setting the pivot point, you may need to model your geometries in such a way that its pivot coincides with the geometry's local coordinate reference frame origin. If this is not possible, you can use *WTgeometry_transform* or *WTgeometry_translate* to reposition the geometry so that its "pivot" point coincides with its local coordinate reference frame origin. In the past, most geometries were drawn by artists (not by engineers), who simply didn't care about the location of the origin.

EXAMPLE: SIMULATING *WTOBJECT_SETPIVOT*

To simulate *WtObject_setpivot*, you can use code similar to the following:

```
/*If mynode is your movable geometry node*/
WTp3_invert( pivot, invpivot );
WTgeometry_translate( WTnode_getgeometry( mynode ), invpivot );
WTnode_translate( mynode, pivot, WTFRAME_LOCAL );
```

SIMULATING *WTOBJECT_SETHANDLE*

There is no function in this release that directly corresponds to the WTK V2.1 *WtObject_sethandle* function. *WtObject_sethandle* was used so that subsequent calls to *WtObject_setposition* and *WtObject_moveto* would be relative to that handle rather than the midpoint of the object. In this release, however, you move an object or set its position using transform nodes, which will affect a geometry's vertices. Therefore, you need to modify the translational part of the transform nodes in order to position an object in its appropriate place.

If you want to move an object with a predefined offset, you can add a transform node before the object with the desired offset.

Coordinate Frames

Similar to WTK V2.1, this release provides local, world, and viewpoint coordinate frames.

For example, to get *mynode*'s position in world coordinates, use:

```
mynodepath = WTnodepath_new( mynode, WTuniverse_getrootnodes(), 0 );  
WTnodepath_gettranslation( mynodepath, worldpos );
```

See *Using Frames of Reference (Coordinate Frames)* on page 4-32.

New Functions to Facilitate Incorporation of WTK V2.1 Applications into the R6/R7/R8/R9 Paradigm

This section lists the new functions that have been added to facilitate the incorporation of WTK V2.1 applications into the Release 6/7/8/9 scene graph paradigm.

Scene Graphs and Nodes

WObject nodes are provided for backward compatibility with versions of WTK prior to Release 6/7/8/9. If you call any of the *WObject* constructor functions (*WObject_new*, *WTterrain_new*, *WObject_newcylinder*, etc.), WTK constructs the *WObject* and also automatically creates a *WObject* node.

The node created by WTK in the *WObject* constructor function can be obtained using the function *WObject_getnode*.

WObject_getnode

```
WTnode *WObject_getnode(  
    WObject *obj);
```

Returns the (automatically created) node associated with the specified object.

WTnode_getobject

```
WObject *WTnode_getobject(  
    WTnode *node);
```

This function returns a pointer to the *WObject* associated with the specified node. Note that *WObjects* are only associated with those nodes that WTK automatically created to correspond to newly created *WObjects*.

Material Colors

The functions below all create a new material of the specified color; defined fields other than *diffuse* will be filled with default values. These functions are backwardly compatible for existing applications. These functions should not be used in new WTK development; users should work directly with the material table functions instead.

These functions have no effect on a material table which has neither the diffuse nor the ambient-diffuse material property set.

- **Wtpoly_setcolor**
- **WTvertex_setcolor**
- **WObject_setcolor**
- **WTvertex_setrgb**
- **WObject_setrgb**

When you call one of these functions, WTK *generates a new material* for the polygons whose colors are being set, rather than editing the material that the polygon/vertex/object is referring to. This is to isolate changes to the intended entities. Multiple “sets” of the same color will re-use the same material; i.e., if *WTPoly_setcolor(white)* is called on six polygons in an object, only one new material will be created.

Note: Textures function independently from materials; texture information is not considered to be part of the material.

Transitioning From Release 6 To Release 7/8/9

To ensure WTK's consistency and ease-of-use, Release 7, Release 8 and Release 9 introduce some *new* functions (which directly correspond to existing Release 6 functions) in the areas of WTK UI, transformations, and C++ wrappers. Since these new functions replace the Release 6 functions, the WTK Release 7/8/9 documentation (that is, the Reference Manual and PDF files) only describes the Release 7/8/9 version.

If your application uses any of the equivalent Release 6 functions, you do not need to make any changes. However, when creating new applications, it is recommended that you use the new functions. The following section provides a listing (by area) of the new functions and the equivalent Release 6 functions.

Changed Functions from Release 6 to Release 7/8/9

WTK User-Interface (UI) Functions

The names of all the UI constructor functions (i.e., functions that create a UI object) have changed. Further the prototypes of certain functions have changed. Certain arguments have been removed because they were either ignored or it did not make sense to have them.

Table H-1 lists the functions whose names have changed only.

Table H-1: UI Functions Whose Names Have Changed in This Release

New Function	Equivalent R6 Function
WTui* WTuiform_new(WTui *parent, char *title, ...);	WTui* WTui_newform(WTui *parent, char *title, ...);
WTui *WTuiform_new(WTui *parent, char *label, FLAG labeltype, ...);	WTui *WTui_newlabel(WTui *parent, char *label, FLAG labeltype, ...);
WTui *WTuipushbutton_new(WTui *parent, char *label, ..);	WTui *WTui_newpushbutton(WTui *parent, char *label, ...);
WTui *WTuiscale_new(WTui *parent, char *label, int minimum, int maximum, int decimal_points, int value, ...);	WTui *WTui_newscale(WTui *parent, char *label, int minimum, int maximum, int decimal_points, int value, ...);
WTui* WTuiscrolledlist_new(WTui *parent, char *label, char *items[], int nitems, ...);	WTui* WTui_newscrolledlist(WTui *parent, char *label, char *items[], int nitems, ...);
WTui* WTuiscrolledtext_new(WTui *parent, char *text, FLAG editable, ...);	WTui* WTui_newscrolledtext(WTui *parent, char *text, FLAG editable, ...);

Table H-1: UI Functions Whose Names Have Changed in This Release

New Function	Equivalent R6 Function
WTui* WTuitextfield_new(WTui *parent, char *text, ...);	WTui* WTui_newtextfield(WTui *parent, char *text, ...);
WTwindow *WTuiwtkwindow_new(WTui *form, int window_config);	WTwindow *WTui_newwtkwindow(WTui *form, int window_config);

Table H-2 lists the UI functions whose names and prototypes have changed.

Table H-2: UI Functions whose Names and Prototypes have Changed in this Release

New Function	Equivalent R6 Function
WTui *WTuifileselection_new(WTui *parent, char *title, char *file, char *pat);	WTui *WTui_newfileselection(char *title, char *file, char *pat); ...);
WTui *WTuimessagebox_new(WTui *parent, char *message, char *title);	WTui *WTui_newmessagebox(WTui *parent, char *message, char *title); ...);
WTui* WTuitextinput_new(WTui *parent, char *msg);	WTui* WTui_newtextinput(WTui *parent, char *msg, FLAG modal, ...);
WTui* WTuimenuubar_new(WTui *parent);	WTui* WTui_newmenuubar(WTui *parent, ...);

Table H-2: UI Functions whose Names and Prototypes have Changed in this Release

New Function	Equivalent R6 Function
WTui* WTuimenupopup_new(WTui *parent, char *label);	WTui* WTui_newmenupopup(WTui *parent, char *label, ...);
WTui* WTuimenuitem_new(WTui *parent, char *label);	WTui* WTui_newmenuitem(WTui *parent, char *label, ...);
WTui* WTuitoolbar_new(WTui *parent, int items, char **bitmap_files);	WTui* WTui_newtoolbar(WTui *parent, int items, char **bitmap_files, ...);

Transformations

In WTK Release 6, there were several functions that required angle parameters to be specified in *degrees*. WTK Release 7/8/9 introduces some new functions which behave similarly, except that now all angles are specified in *radians*. Thus, in Release 7/8/9, all functions requiring angle parameters are now specified in radians.

Table H-3 lists the transformation functions that have changed in this release.

Table H-3: Release 7/8/9 Transformation Functions that are Now Specified in Radians

New Function (angles in radians)	Equivalent R6 Function (angles in degrees)
FLAG WTmovnode_axisrotation(WTnode *movnode, int axis, float angle);	FLAG WTmovnode_rotateaxis(WTnode *movnode, int axis, float angle);

Table H-3: Release 7/8/9 Transformation Functions that are Now Specified in Radians

New Function (angles in radians)	Equivalent R6 Function (angles in degrees)
FLAG Wtnode_axisrotation(Wtnode *node, int axis, float angle, int frame);	FLAG Wtnode_rotateaxis(Wtnode *node, int axis, float angle, int frame);
FLAG Wtnode_rotation(Wtnode *node, float angley, float anglex, float anglez, int frame);	FLAG Wtnode_rotate(Wtnode *node, float angley, float anglex, float anglez, int frame);

C++ WRAPPERS

Table H-4 lists the C++ wrapper functions for the above three transformation functions.

Table H-4: C++ Wrapper Functions for the above Transformation Functions

New Method	Equivalent R6 Method	Class
FLAG MovAxisRotation(int axis, float angle);	FLAG MovRotateAxis(int axis, float angle);	WtMovable
FLAG AxisRotation(int axis, float angle, int frame);	FLAG RotateAxis(int axis, float angle, int frame);	WtXform

Table H-4: C++ Wrapper Functions for the above Transformation Functions

New Method	Equivalent R6 Method	Class
FLAG Rotation(float angley, float anglx, float anglez, int frame);	FLAG Rotate(float angley, float anglx, float anglez, int frame);	WtXform

Third-party Software

This chapter contains information from other software providers for both image conversion and modeling packages. WTK also has a sister product, called World Up, which includes a built-in Modeler. The World Up Modeler converts images to the NFF format by importing the file and then saving it in the NFF format. The World Up Modeler supports the file formats shown in table I-1:

Table I-1: File formats supported by the World Up Modeler

File Extension	Description
NFF	SENSE8 Neutral File Format
BFF	SENSE8 Binary Neutral File Format
3DS	3D Studio file format
WRL	VRML file format
FLT	MultiGen Flight file format
DXF	AutoCAD file format
OBJ	Wave Front file format
GEO	Videoscape file format
SLP	ProEngineer "RENDER" file format

You can also use the World Up Modeler to create new images, apply materials and textures to the model, and do real-time rendering. World Up runs on NT/Windows 95 and SGI platforms. For more information, contact SENSE8 using the information in Appendix K, *Technical Support*.

Image Conversion (SGI)

We recommend using the Iris image utilities included with IRIX in the *ee2.sw.imagetools* subsystem.

The Irix operating system comes with approximately 100 image utilities for performing a variety of image operations and converting between various image formats. These are installed on your system in `/usr/sbin` when you load the subsystem of Irix called *ee2.sw.imagetools*.

One particularly useful utility is the one called *ipaste*, which displays a *.rgb* image in a window on your monitor. The syntax for this command is:

```
ipaste image.rgb
```

Table I-2 lists image conversion products.

Image Conversion (Windows 32-bit Platforms)

Table I-2: Image conversion (Windows 32-bit Platforms)

Product	Comments	Manufacturer
HiJaak Pro	Converts between more than 85 2D and 3D graphics formats including TGA, GIF, BMP, JPG, IFF, and EPS.	Quarterdeck Systems (800) 525-2580 (510) 548-0393 fax web site: www.Quarterdeck.com
Image Alchemy	Image Alchemy converts over 75 different image formats and includes all colorspace and compression variations of each format.	Handmade Software, Inc. PC/Sun/SGI (408) 358-1292, (408) 356-4143 fax web site: www.handmadesw.com

Table I-3 lists model conversion products.

Model Conversion

Table I-3: Model conversion

Product	Comments	Manufacturer
InterChange	Also available as a 3D Studio IPAS plug-in. Converts between: 3D Studio, Lightwave, Imagine, Wavefront, Alias polysets, SENSE8 NFF, RenderWare, Inventor, Vista Pro DEM, CADKey, DXF, Stereolithography, Sculpt, Envisage, Prisms, Vertigo, StyleGuide, POV, PLG, Swivel (some conversions are read only, others write-only).	Syndesis 235 South Main St. Jefferson, WI 53549 (414) 674-5200 (414) 674-6363 fax web site: www.threedee.com

Table I-4 lists 3D modelers.

3D Modelers

Table I-4: 3D modelers

Product	Platform	Comments	Manufacturer
3D Design "Plus"	MSWin	CSG and polygonal modeling, optional raytracing, imports and exports DXF (but not very well). Native format: "mdl", binary, proprietary.	ComputerEasy International, Inc. 414 E. Southern Avenue Tempe, AZ 85282 (800) 522-3279 (602) 829-9616 fax

Table I-4: 3D modelers (continued)

Product	Platform	Comments	Manufacturer
3D Studio	MSDOS	Modeler/non-realtime renderer. Triangular polygons only. Supports texture applications for WTK. Native formats: ASC (ASCII) and 3DS ("mesh" binary, proprietary). Also reads DXF and .flm ("Filmroll" files), and writes DXF.	Autodesk, Inc. 111 McInnis Parkway San Rafael, California 94903 (800) 964-6432 (415) 507-5000 (415)507-5100 fax web site: www.autodesk.com
Blob Sculptor	MSDOS, other platforms in progress	Freeware. Allows you to model 3D objects through the use of blobs. Output formats: Blob Sculptor (native), POV, Polyray, Rayshade, "RAW" polygons, "CTDS - Connect The Dots System", DXF Input: native Blob format	Contact authors at Pi Square BBS (301) 725-9080
Pioneer Pro and trueSpace /2 and trueSpace /SE	MSWin, Amiga	Modeler/renderer, can mix splines and faces and output DXF with color information	Caligari Corporation 1959 Landings Dr. Mountain View, CA 94043 (415) 390-9600 (415) 390-9755 fax web site: www.caligari.com
CyberVision	SGI, DEC-Alpha	Modeling and walkthrough rendering, native format: .cyb, imports AND exports: OBJ, NFF (non-SENSE8), DXF, SGI's ".sgm" and ".bin" formats, Inventor, DTED, Multigen/Modelgen support at extra cost	Computer Explorations Inc. (800) 443-8278

Table I-4: 3D modelers (continued)

Product	Platform	Comments	Manufacturer
LightWave	MSWin, Amiga	Modeler and non-realtime renderer. Reads and writes its own 3D file format, Lightwave. Also reads Videoscape files. Converters are provided for importing DXF 11 or higher, 3DStudio, PICT, Swivel.	NewTek, Inc. 1200 SW Executive Dr. Topeka, Kansas 66615 (800) 847-6111 (800) 854-7111 fax web site: www.newtek.com
Multigen, ModelGen	SGI only	Reads and writes their own .flt "flight" format which is readable by WTK-SGI. Supports textures in WTK. Imports DXF	MultiGen Inc. 550 South Winchester Blvd., Suite 500 San Jose, CA 95128 (408) 261-4100 (408) 247-4326 fax web site: www.multigen.com
Polywog	MSWin, Amiga	Freeware wireframe 3D modeler; no solid rendering or application of colors/textures. Reads and writes SENSE8 NFF, GEO (VScape), and PLG (VR386).	Author contact: iguana@crl.com
Ray Dream Designer and Ray Dream Studio	Mac, MSWin	Integrated modeler/non-realtime renderer, can output usable DXF files	Fractal Design Corporation P.O. Box 66959 Scotts Valley, CA 95067-6959 (800) 846-0111 web site: www.fractal.com

Table I-4: 3D modelers (continued)

Product	Platform	Comments	Manufacturer
3Design	SGI only	<p>Entertainment-based modeler that integrates NURBS, polygon, and metaball modeling functions within a single easy-to-use package.</p> <p>Uses ASCII formats: .obj for object geometry, .mtl for material list/ descriptions.</p> <p>There is also the .Geo format, which is inherited from the TDI Explore package.</p> <p>They have translators from 3Design to DXF or IGES (DXF will be the most accurate because of polygons but it seems that it will not write out color layers.</p> <p>Reads: OBJ, IGES, DXF, SDRC, STL, Catia, ProE SLA (use sla_obj command), 3DS (3DS conversion done by Inner Space Labs).</p> <p>Writes: OBJ (.obj)</p>	<p>Wavefront Technologies, A Silicon Graphics Company 110 Richmond Street East Toronto Ontario Canada (800) 441-2542 (416) 362-0630 fax web site: www.aw.sgi.com</p>
World Render 3D	MSWin	<p>Wireframe modeler and translator.</p> <p>Reads and writes: 3DS, 3D Workshop, DXF, Envisage 3D, Imagine, Lightwave, NapCad 3D, NFF, "RAW", Sculpt 3D/4D, VideoScape, Wavefront, native "World Render 3D" format.</p> <p>Writes: DBW 2.0, POV-Ray 1.0, TART, Vivid 2.0. Demo version on the net and on CI\$also.</p>	<p>MAZAR software corp. 1801 NE 197 Terrace N. Miami Beach, FL 33179 (305) 936-9290</p>

Sources of Components

This appendix lists the manufacturers of a variety of products used with WTK. You can contact the manufacturers listed here to determine the appropriateness of any particular component for use with WTK.

Input Devices

Table J-1: Input devices

Manufacturer	Product
Ascension Technology Corporation, P.O. Box 527, Burlington, VT 05402 (802) 860-6440 <i>email:ascension@world.std.com</i>	Ascension Bird, Flock of Birds (6D magnetic tracker)
Fakespace, 4085 Campbell Ave. Menlo Park, CA 94025 (415) 688-1940	Pinch Glove System
General Reality Company (Distributor) 124 Race St. San Jose, CA 95126 Phone: 408-289-8340 <i>e-mail: sales@genreality.com</i>	5DT Glove
Logitech, Inc., 6505 Kaiser Drive, Fremont, CA 94555 (510) 795-8500	3D Mouse (Red Baron), Space Control Mouse (Magellan), Head Tracker
Polhemus Inc., P.O. Box 560, Colchester, VT 05446 (802) 655-3159	ISOTRAK, ISOTRAK II, InsideTRAK and FASTRAK 6D magnetic trackers
Precision Navigation, 1235 Pear Ave., Ste. 111, Mountain View, CA 94043 (415) 962-8777	Precision Navigation Wayfinder-VR

Table J-1: Input devices (continued)

Manufacturer	Product
Spacotec IMC Corporation, 100 Foot of John Street, Lowell, MA 01852 (508) 970-0330	Spaceball and Spaceball SpaceController
ThrustMaster, Inc. 7175 N.W. Evergreen Parkway, #400, Hillsboro, OR 97124 (503) 615-3200	Serial Joystick and Formula T2 Steering Console

Output Devices

Table J-2: Output devices

Manufacturer	Product
Fakespace, 4085 Campbell Ave., Menlo Park CA 94025 (415) 688-1940	BOOM (monochrome and color stereo viewers on articulated arm)
StereoGraphics, Corp., 2171-H East Francisco Blvd., San Rafael, CA 94901 (415) 459-4500	CrystalEyes and CrystalEyesVR LCD Shutter Glasses (stereo viewing glasses and head-tracking)
Virtual Research, 2326 Walsh Ave., Santa Clara, CA 95051, (408) 748-8712	VR4 (head mounted display), and FS5 (head mounted display)
Crystal River Engineering, 4245 Technology Drive, Fremont, CA 94538 (800) 317-TRON	Beachtron, Alphanon, Acoustetron II (3D sound system)
Virtual i-O, 1000 Lenora St., Suite 600, Seattle, WA 98121 (206) 382-7410	i-glasses! (Head-mounted display)
VictorMaxx, 510 Lake Cook Rd., Suite 100, Deerfield, IL 60015 (708) 267-0007	CyberMaxx2 HMD (head-mounted display)

Video Accelerators

Table J-3: OpenGL graphics accelerators

Graphics card	Manufacturer	Comments
Artists 2000 I	Artists Graphics, Inc.	
Gloria-4, Gloria-8	ELSA, Inc.	8 MB Texture Memory 12.5 MPixels/sec Fill Rate
Freedom Graphics	Evans & Sutherland, Inc. 600 Komas Drive Salt Lake City, Utah 84108 USA (801) 588-1000	1MB - 16MB Texture Mem 8 MPixels/sec Fill Rate
Sapphire 2SX	Fujitsu Microelectronics, Inc. 30 Rio Robles San Jose, CA 95134-1807 (408) 922-9000	
PCI Glint Video 1	RPI Advanced Technology	No Texture Acceleration
Eclipse 100	RSR Engineering, Inc.	4 MB - 21 MB Texture Mem 30MPixels/sec Tex Fill Rate
Eagle	Creative Labs 1523 Cimarron Plaza, Stillwater, OK 74075 U.S.A. (800) 998-1000	GLiNT-based Hardware Texture Acceleration
Millennium	Matrox 1025 St. Regis Blvd. Dorval, Quebec Canada H9P 2T4 (800) 804-6243 FAX: (514) 969-6273	

Table J-3: OpenGL graphics accelerators (continued)

Graphics card	Manufacturer	Comments
Fire GL	SPEA Software AG	DM5000 15MPixels/sec Fill Rate No Texture Acceleration

The WTK Users' Group

A WorldToolKit Users' Group (SIG-WTK) has been organized by WTK customers with assistance from EAI/SENSE8. SIG-WTK provides a world-wide electronic forum for the discussion of WTK-related issues. In addition to the electronic forum, several SIG-WTK users' group meetings have been held and additional regular meetings are planned.

Participating in SIG-WTK

The following material comes from the original SIG-WTK chairman, Terry Fong.

Greetings, fellow WTK user!

I would like to cordially invite you to participate in SIG-WTK, the WorldToolKit Users' Group. This group provides a contact point for users of EAI's SENSE8 Product Line's WorldToolKit to discuss and exchange information on a variety of topics. Among these are:

- 3D objects: modeling, importing/exporting to WTK NFF, sharing.
- Sensor drivers: development, reducing lag and latency.
- Managing user interaction.
- Efficient development of virtual environments with WTK.
- Distribution and sharing of virtual environments.
- Improving simulation performance (e.g., frame rate, quality).
- Platform-specific issues (e.g., GL queues on SGI machines).
- Advocating WTK improvements/changes to EAI/SENSE8.

Communicating with SIG-WTK

To subscribe or unsubscribe, send e-mail to:

`sig-wtk-request@sense8.com`

In the body of the message, type the word *subscribe* or *unsubscribe* without any white space preceding the word.

To get help, or a list of possible commands, send e-mail to:

`majordomo@sense8.com`

In the body of the message, type the word *help*.

To send a message to all SIG-WTK members, please address it to:

`sig-wtk@sense8.com`

SIG-WTK:Email Archives

If you are interested in looking through the archives of sig-wtk email, there are several ways to do this:

By Thread (Oldest Thread Listed First):

<http://www.sense8.com/support/archive/sig-wtk.archive/threads.html>

By Date (Most Recent Messages Listed First):

<http://www.sense8.com/support/archive/sig-wtk.archive/maillist.html>

By Text Search:

<http://www.sense8.com/support/AT-SigWTKquery.html>.



Technical Support

If you encounter problems installing or using WTK, EAI/SENSE8 offers several methods for getting your problems answered. If the problem needs immediate assistance, please contact your EAI/SENSE8 authorized reseller or EAI/SENSE8 directly for Technical Support.

Whenever contacting EAI/SENSE8 for technical support, please have your WTK serial number ready. It's printed on your CD-ROM jewel case (or distribution tape). You can save yourself time and effort by reading the WTK support page on the World Wide Web before you call. It has links to download the latest patches and answers to FAQs (frequently asked questions) about WTK. The URL is listed below.

U.S. Technical Support

USA Headquarters

EAI's SENSE8 Product Line

100 Shoreline Highway, Suite 282

Mill Valley, CA 94941

Sales/Product Information: (415) 339-3200

Facsimile: (415) 339-3201

Technical Support (phone): (415) 339-3392

Technical Support (e-mail): support@sense8.com

WTKCODES (e-mail): wtkcodes@sense8.com

Web site: www.sense8.com

Web site (for Technical Support questions): www.sense8.com/wtksupport/index.html

Web site (for WTKCODES): www.sense8.com/wtklicense.html

Non-US Technical Support

Please contact your WTK reseller directly. They can provide you with local support in your time zone. If you have difficulties getting support from your dealers, then please contact USA headquarters.

SIG-WTK Users' Group

A WorldToolKit Users' Group (SIG-WTK) has been set up at the NASA Ames Research Center in Mountain View, California. SIG-WTK provides a world-wide electronic forum for the discussion of WTK-related issues as well as an anonymous ftp site for uploading and downloading WTK-related data. More information about SIG-WTK is provided in Appendix J, *The WTK Users' Group*.

M

Glossary

3x3 Matrix	A 3x3 array of floats that is type defined in WTK as WTm3. A 3x3 matrix is a mathematical entity that can be used to represent position and orientation in 2D space.
3D Sound	Spatialized sound that appears to the end-user to have a distinct location in the simulation.
3DS	The native file format of Autodesk's 3D Studio. You can use this binary file format to represent 3D geometry, lighting, and animation.
4x4 Matrix	A 4x4 array of floats that is type defined in WTK as WTm4. A 4x4 matrix is a mathematical entity that can be used to represent position and orientation in 3D space.
6D Sensor	Sensors that have six degrees of freedom of movement. That is, they can control movement in the X, Y, and Z direction as well as control <i>pitch</i> , <i>yaw</i> , and <i>roll</i> .
Absolute Record	Sensor values that correspond to a specific absolute spatial location (i.e. the position and orientation of the sensor). See also <i>Relative Record</i> .
Ambient Color	The material property that represents the color reflected from a material in ambient white light, specified in red, green, and blue floats in a range from 0.0 to 1.0.
Ambient Light	Background light that illuminates all surfaces equally, regardless of their position or orientation.
Ambient Light Node	A scene graph node used to store ambient light.
Ancestor Node	Any node whose sub-tree contains a node (N), is considered to be an ancestor of that node (N).

Anchor Node	A scene graph group node that contains a string property (URL) used to retrieve a data file. The data file associated with an anchor node will NOT be automatically loaded.
Anti-aliasing	Lines or edges, especially nearly horizontal or nearly vertical ones, appear jagged due to screen resolution. This jaggedness is called aliasing and anti-aliasing refers to techniques used to reduce this jaggedness.
API	Application Programmers Interface.
Aspect Ratio	The ratio of the horizontal and vertical drawing dimensions. This ratio is used to correct for any monitor or pixel distortion that causes round objects to look elliptical or square objects to look rectangular.
Asymmetric Projection	A window projection type. Asymmetric projection is useful in some stereo viewing configurations. By changing the viewpoint <i>convergence distance</i> , geometries can be made to appear either in front of or behind the display device (e.g., the screen). A geometry in the 3D world closer to the viewpoint than the convergence distance appears to be in front of the screen, while a geometry that is farther from the viewpoint than the convergence distance appears to be behind the screen. See also <i>Projection Type</i> .
Attenuation	The degree to which a point or spot light's intensity decreases with increasing distance from the position of the light.
Axis	One of the reference lines of a coordinate system.
BFF	The binary version of SENSE8's neutral file format used for representing 3D geometry. See NFF.
Back Face Rejection	The elimination of a single-sided polygon (that is, a polygon that can only be viewed from one side) from the rendering process. In the back face rejection process, those polygons whose normals face away from the viewpoint are not rendered.
Baud Rate	Data transmission speed in bits per second.

Bothsides (of a polygon)	Polygons have front and back sides (or faces). The side facing in the direction of the polygon normal is considered to be the front facing side. You can choose to display just the front side of a polygon or bothsides of a polygon. If a polygon is bothsided, it can be viewed from either side.
Bounding Box	Also known as Extents Box (smallest box that surrounds an object). The term bounding box sometimes refers to the fact that extents boxes can be made visible in the scene.
Callback Handler Function	A scheme used in event-driven programs where the program registers a callback handler for a certain event. The program does not call the handler directly but when the event occurs, the handler is called, possibly with arguments describing the event.
Centroid	The centermost position of a three-dimensional object.
Child Node	A scene graph node that is a direct descendent of another (parent) node. A child node can inherit state information from its ancestor nodes.
Collision Detection	Intersection testing of objects at either the bounding box level or at the polygon level.
Concave Polygons	Any polygon that has at least one interior greater than 180 degrees.
Constraining Motion	Movement of objects is generally allowed in all 6 degrees of freedom. Sometimes it is desirable to disallow movement in one or more degrees of freedom. This limitation is called a constraint.
Constructor Functions	Any WTK function used to construct new geometries, viewpoints, etc. For example, <i>WTviewpoint_new</i> , which creates a new viewpoint is a constructor function.
Content Nodes	Containers for the four basic elements of a scene: geometry, light, position, and fog.
Convergence	A horizontal offset in pixels, which is applied to both the left and right eye images. This offset is subtracted from the left eye and added to the right eye. See <i>WTviewpoint_setconvergence</i> on page 16-21.

Convergence Distance	The distance where a stereoscopic image is perceived to exist. This parameter determines the perceived location of an object relative to the plane of the computer screen.
Coordinate System	A positional system, containing X, Y, and Z components, by which three-dimensional entities can be described. See <i>Local Coordinate System</i> , <i>Parent Coordinate System</i> , and <i>World Coordinate System</i> .
Coplanar Polygon	Polygon surfaces that overlap and lie in the same plane.
Culling	See <i>Hierarchical Culling</i> .
Cylindrical Mapping	A technique for applying texture mapping coordinates so that the image appears to be wrapped around the object in a tube-like fashion. The application of a label to a can or bottle is an example of cylindrical mapping.
Descendant Node	Any node that is contained in the sub-tree of another node is considered to be a descendant of that node.
Diffuse Color	The material property that represents the color reflected from a material in diffuse white light, specified in red, green, and blue floats in a range from 0.0 to 1.
Diffuse Light	Positional or directional light that illuminates polygons as a function of the angle between the light direction and the polygon (or vertex) normal.
Directed Light	A light source that has direction but no (finite) position. A directed light can be used to emulate the effects of sunlight.
Distributed Simulation	An application that is shared between more than one computer over an Ethernet network.
DOF	Degrees of freedom. See <i>6D Sensor</i> .
DXF	Drawing Interchange Format. This file format was developed by Autodesk, Inc. as a way to transfer geometric data from one design application to another.
Emissive	A material property that represents the color of light produced (not reflected) by the material, even when there is no light. Emissive materials do not illuminate other objects.

Euler	A mathematical representation of a position and orientation in three-dimensional space.
Event Order	The order in which the simulation loop processes the universe's events.
Extents Box	The extents box is the smallest box that fits around an object. See also <i>Midpoint</i> and <i>Radius</i> .
Extrusion	The 3D outline or object created by taking a 2D contour and extending it into three dimensions.
Filtering Textures (mipmapping)	The process of (automatically) scaling down a texture so that it can be appropriately applied to a polygon regardless of the polygon's screen resolution.
Floating License	A WTK software license designed to be used on a network. See also <i>Node-locked License</i> .
FLT	ProEngineer file format used for representing 3D geometry.
Fog Node	A scene graph node used to simulate fog, smoke, etc.
Frame	An individual rendering loop during which each active window is redrawn after updating sensor input, path information, user-defined actions, and any foreground details. See also <i>Simulation Loop</i> .
Frame Rate	The number of times per second that WTK completes the simulation loop, i.e. renders a frame.
General Projection	A window projection type. Provides the greatest flexibility; useful when the viewer is not always perpendicular to the display surface, like in CAVE environments.
Geometry	A collection of polygons (composed of vertices) used to model physical objects.
Geometry Node	A scene graph node used to associate a geometry to a node, so that the geometry can be made part of the scene.
Geometry Optimization	A technique used to optimally organize the contents of a geometry so that it can be rendered in the shortest amount of time. Once a geometry has been optimized (using <i>WTgeometry_prebuild</i>) edits to the geometry will have no effect.

Gouraud Shading	A technique used for shading a 3D graphical object composed of polygons, by interpolating light intensities at the vertices of each polygon's face, rendering a smooth surface.
Graphics Pipeline	Many high performance systems utilize specialized graphics hardware (aka graphics pipeline) to substantially increase the system's ability to process and render geometric polygons.
Group Node	A scene graph node that has children, but no other properties.
Grouping Nodes	Grouping (organizational) nodes contain no content directly, however they are the essential structuring nodes used in building a scene graph. Grouping nodes, such as the group node, the separator node and the transform separator node, let you group together and encapsulate a set of nodes that share common states, such as position or lighting effects.
GUI	Acronym for graphical user interface. Also called user interface (UI).
Head Mounted Display	A display device that is worn on the head, which sometimes permits position and orientation tracking.
Heads Up Display	The static portion of an image rendered on a display device.
Hierarchy	Used in the context of scene graphs, hierarchy refers to how the nodes in a scene graph are organized and the relationship of one node to another.
Hierarchical Culling	WTK's automatic process of quickly and efficiently eliminating objects that are not visible from the current viewpoint so that they are not unnecessarily processed during the rendering process.
Hither Clipping Plane	The physical range in front of the viewpoint, before which objects are not rendered in that window. That is, objects that appear between the viewpoint and the hither clipping plane are not rendered. Objects are rendered only in the area between the hither clipping plane and the <i>yon clipping plane</i> . The hither clipping value is a window parameter.

Inline Node	A scene graph group node that contains a string property (URL) used to retrieve a data file. The data file associated with an inline node will NOT be automatically loaded unless the inline node is read in from a VRML file.
Instance	WTK's scene graph hierarchy allows nodes to be referenced multiple times within a single scene graph. Since it is sometimes necessary to identify a particular occurrence of a node to distinguish it from other occurrences, each occurrence is called an instance. The ability to use instances (instancing) saves system memory (since only one copy of each object is necessary) and therefore improves performance.
Interpolation	The method of determining a new value using two or more existing values. WTK uses interpolation when new paths are created from previously defined paths.
Intersection Testing	See Collision Detection.
Iterators	Functions used to "walk through" lists of objects of a given type. For example, <i>WTwindow_next</i> gets the next window in the universe's list of windows.
Leaf Node	A scene graph node that has no descendants.
Level of Detail (LOD) Node	A scene graph node used to automatically select between different representations (levels of detail) of an object based upon the distance between the object and the viewpoint position.
Level of Detail Switching	The process of swapping less detailed objects for more detailed objects as the distance between the viewpoint and the object decreases (or vice-versa).
Light Node	A scene graph node used to specify a WTK light (ambient, point, directional, or spot).
Local coordinate system	The coordinate system specific to a particular object, usually the one in which the object was modeled.
Material	A material is used to define the appearance of graphical objects and consists of the following material properties: ambient color, diffuse color, specular color, shininess, emissive, and opacity.

Material Table	Used to store the material properties of any number of materials. Each geometry references a number of materials from the material table that is associated with that geometry.
Matrix	See <i>4x4 Matrix</i> and <i>3x3 Matrix</i> .
Midpoint	The center of a node's extents box. See also <i>Extents Box</i> and <i>Radius</i> .
Mipmapping	See <i>Filtering Textures</i> .
Motion Link	Used to connect a source of position and orientation information (a path or sensor) with a target that moves to correspond with that changing set of information. A target can be a movable node, a node path, a transform node, or a viewpoint.
Movable Node	A scene graph node that represents self-contained entities that can be easily moved around in the scene. The three basic components of a movable node are a <i>separator</i> , a <i>transform</i> , and a <i>content</i> (which corresponds to either a Geometry, Light, Separator, Switch, or LOD node).
Network Daemon	The daemon "wtklsd" is required for the floating license version of WTK.
NFF	Neutral File Format, SENSE8's neutral ASCII file format used for representing 3D geometry. It is a very compact format, which WTK can use to store geometric data.
Node	The fundamental element or building block used to construct a scene graph. A node is simply an element of content or a grouping/procedural element used to maintain scene hierarchy.
Node path	An entity that represents the path through a scene graph from the root node to another node.
Node-locked License	A WTK software license designed to be used on a stand-alone computer. See also <i>Floating License</i> .

Normal	A direction vector used for shading and rendering. Normals can be applied at both the vertex and polygon level. A polygon normal is perpendicular to the polygon surface and extends outward from the visible side of the polygon. A vertex normal represents the direction that is perpendicular to the tangent vector at the vertex position of the polygon.
Normalized	A normalized vector is a vector whose magnitude is 1.0.
NTSC	Acronym for National Television Standards Committee and the standard defining the television video signal format used in the USA. The UK equivalent is PAL.
OBJ	Wavefront/Alias file format used for representing 3D geometry.
Opacity	A material property that dictates the extent to which the material is opaque/transparent. An opacity value of 0.0 indicates that the material is completely transparent, while an opacity value of 1.0 implies that the material is completely opaque.
OpenGL Coordinate Conventions	The WTK coordinate convention differs from the OpenGL convention. (The WTK convention has X pointing to the right, Y pointing down, and Z pointing straight ahead.) WTK coordinates are obtained by simply negating Y and Z OpenGL coordinate values. See also <i>Right-hand Rule</i> .
Orphaned Nodes	Nodes that are not associated with any scene graph cannot be rendered and are considered to be orphaned nodes.
Orthographic Projection	Orthographic projection is a window projection type that is useful for plan views or anytime a perspective distortion is not desired; parallel lines remain parallel regardless of viewpoint position. Translations in the X and Y directions work as before, but translations along the Z-axis do not affect the scene.
Parallax	The distance between the left eye and the right eye position when using a stereo viewing device.
Parent coordinate system	The coordinate system of the parent object in the scene graph.

Parent Node	A node's direct ancestor in the scene graph.
Parity Bit	An extra bit added to a byte or word to reveal errors in transmissions over a serial port.
Path	Stores a series of position and orientation records in absolute world coordinates. A path can be used to pre-program a flight path through a scene, or to pre-define the motion of an object within the scene.
Path Element	A single position and orientation record. A sequence of path elements defines a path.
Picking	The ability to select the front-most rendered polygon in a window.
Pitch	The orientation of an object about the X axis.
Pixel	A contraction of "picture element," it refers to one point in a graphics image on a computer display. A standard VGA display might have 640 x 480 pixels. The number of bits per pixel determines how many colors can be represented on the image. VGA displays typically have eight bits per pixel. "Truecolor" displays typically use 24 bits per pixel.
Planar Mapping	A technique for applying texture mapping coordinates so that the image appears to be projected onto a surface. This type of mapping is used to create buildings, pictures, signs, and many other graphical entities.
Planar Polygon	Polygons whose vertices are all positioned within the allowable distance (defined by the WTK constant WTFUZZ (0.004)) from the plane passing through the vertices.
Point Light	An omni-directional source of lighting capable of being positioned by the user.
Polygon	A flat plane figure with multiple sides. It is the basic building block of geometries.
Polygon ID	Polygon ID's are read from and written to NFF files, and are set with the function WTPoly_setid. Polygon ID's provide a handy way of obtaining pointers to polygons, which can then be passed in to other functions.
Polygon Normal	See <i>Normal</i> .

Port	A logical channel in a communications system. See also <i>Serial Port</i> .
Portability	The ability to move an application built on one hardware platform to another platform without having to make extensive changes to the application source code.
Portals	An outdated concept from older versions of WTK. Portals allowed users to move from one universe to another. This is now accomplished by using multiple scene graphs.
Position	The current X, Y, and Z coordinates of an object.
Predecessor Node	Any node in a scene graph that directly affects how a specific node (N) is processed is considered to be a predecessor of that node (N), even though that node is not an ancestor node.
Primitive (Geometric)	A three-dimensional basic geometric form (such as a block, sphere, or cylinder) stored as a collection of polygons.
Projection Type	Defines how the scene is projected onto the display device. The projection type is either <i>symmetric</i> , <i>asymmetric</i> , <i>general</i> , or <i>orthographic</i> . The default is symmetric.
Propagation of state	When traversing a scene graph tree, the lighting and transformation state created by each transform and light node accumulates (propagates) as the remainder of the scene graph is traversed. See <i>Separator Nodes</i> for a description of how to avoid propagation of state.
Quads	Four-sided <i>polygons</i> .
Quaternion	A mathematical representation of an orientation.
Radiosity Preprocessing	A radiosity preprocessor program takes a model and a light source specification as input and generates a new model with lighting information (such as for shadows or reflections) built into it.
Radius	The distance from the midpoint of a node's extents box to a corner of the box. This is the same as the length of the extents vector. See also <i>Extents Box</i> and <i>Midpoint</i> .

Ray Casting	A ray is a vector representing a direction. Ray casting is the process of calculating a ray that emanates from a position (for example, a viewpoint) and which then passes through a specified point. Ray casting can be used for terrain following or intersection testing.
Real-time Simulation	A 3D application that responds to input and displays the corresponding change (almost) instantly. When measured in frames per second, real-time usually means at least 10 fps.
Relative Record	Sensor values that correspond to a sensor's change in spatial location (position and orientation) since the last time through the simulation loop. See also <i>Absolute Record</i> .
Rendering	Generation of a graphical image from mathematical models of three-dimensional objects, i.e. a scene.
Rendering Flags	Rendering options for displaying geometries in a simulation. Types of rendering include wireframe, textured, smooth, etc. Rendering options can be set globally for the universe or individually for geometries
Resource File	Sets certain parameters from a file when your application starts up. For example, you can specify background color, viewing angle, window size, and window position this way.
RGB	RGB stands for the red, green, and blue components of a color specification. Valid values for color components range from 0 to 255. An RGB triple of (255, 0, 0) represents the color red while an RGB triple of (255, 255, 0) is yellow.
Right-hand Rule	The WTK coordinate system obeys the right-hand rule. The default coordinate system has the X axis pointing to the right, the Y axis pointing down, and the Z axis pointing straight ahead.
Roll	The orientation of an object about the Z axis.
Root Node	A scene graph node that is the top most node in any scene graph. Each scene graph has only one root node. All other nodes in the scene graph are descendants of the root node.

Rotation	The turning of an object so that it has a different orientation.
RTS signal	On some platforms, certain serial devices require that the serial port's RTS signal be kept in either a high or low state or the device will not communicate with the serial port.
Scene	The virtual world being displayed.
Scene Graph	A scene graph is a hierarchical arrangement of nodes (such as geometry, light, fog, and positional information) representing objects in a simulation. A universe can contain more than one scene graph.
Scene Graph Tree	The scene graph is arranged as an upside down tree, where the root is on the top and the branches and leaves are on the bottom.
Sensor	A device that responds to physical movement and that transmits the resulting position and (possibly) orientation information.
Separator Nodes	A scene graph node that prevents lighting and transformation state information from propagating from its descendant nodes to its sibling nodes.
Serial Port	A connector on a computer where you can attach a serial line connected to peripherals that communicate using a serial protocol.
Shading	The process of rendering polygons, especially when using lighting effects.
Shininess	A material property that controls the narrowness of focus of specular highlights. Shininess can range from 0.0 to 128.0. The lower the shininess value, the more "spread out" the highlight is; the higher the shininess value, the sharper (shinier) the highlight.
Sibling Node	Children of the same parent node are siblings.
Simulation	Your WTK application.

Simulation Loop	When a WTK application is running, the simulation loop is repeatedly executed. WTK reads input sensors, executes universe action functions, updates objects with sensor input, executes object tasks, steps any paths, and renders a new view of your scene onto the display during each pass through the simulation loop. Each pass through the simulation loop is called a <i>frame</i> .
Six Degrees of Freedom	See <i>6D Sensor</i> .
SLP	Pro/Engineer file format used for representing 3D geometry.
Spatialized Sound	See <i>3D Sound</i> .
Specular Color	A material property that represents the color of the highlights that are reflected off a shiny surface.
Spherical Mapping	A technique for applying texture mapping coordinates so that the image appears to be wrapped around the object in a spherical fashion. A good example of spherical mapping would be a world globe.
Spot Light	A light that illuminates a small area, within a cone of specified angle. For example, an automobile headlight.
State	Refers to the accumulated lighting and transformation state that results during the traversal of a scene graph. The scene graph state affects how and where geometry is rendered at any particular point in the scene graph.
Stereoscopic Viewing	The visual effect achieved when part of your scene appears to be in front of your display screen, and part of the scene appears to be behind your display screen, giving the illusion that the image is a 3 dimensional image.
Stipple Pattern	The 2D or 3D line style. The default line style is solid.
Stop Bits	In serial communications, where each bit of the message is transmitted in sequence, stop bits are extra "1" bits that follow the data and any parity bit. They mark the end of a unit of transmission (normally a byte or character).

Subfaces	ModelGen and MultiGen permit “subfaces,” polygons that generally are oriented in the same plane as another polygon, but that are intended to appear as if they are on top of the other polygon. When polygons with subfaces are translated literally into the WTK viewing format, Z-buffer roundoff becomes pronounced, resulting in flickering between the coplanar faces as the object is rendered. When WTK encounters subfaces in an OpenFlight file, it translates them by a constant amount in the direction of the parent polygon’s normal vector.
Sub-tree	A node and all its descendants in the scene graph is called a sub-tree of the overall scene graph tree.
Switch Node	A scene graph node that allows the user to control which of its children to traverse.
Switch-out Distances	An array of floats that specify the distances where WTK will switch to a lower level of detail when using an LOD node. By default, an LOD node has no switch-out (range) values.
Symmetric Projection	A window projection type. Commonly used projection, especially for monoscopic and flat screen displays. This is the default projection type.
Task	A user-defined function assigned to WTK or user objects.
Tessellation	Refers to the manner in which the surface of a geometric object is modeled via polygons. Finer tessellations usually require the use of more polygons than a rough tessellation. For example, a cone that was tessellated using 100 polygons would, when rendered, appear much superior to a cone that was tessellated using only 10 polygons, since the 10 polygon tessellated cone would appear very faceted.
Texels	A contraction of “Texture element”, it refers to the individual texture elements of a texture image.
Text Fields	A text field user interface object is a simple object that allows a user to enter text using the keyboard. Text field objects are normally used as a single line data entry field. It gives the user text editing capabilities and also provides the point and click functionality expected of GUI applications.

Texture	A bitmap image usually created for the purpose of applying complex images to simple polygons to increase performance of 3D graphics applications.
Texture Draping	The process of applying a texture bitmap image stored in a file to a polygon or an entire geometry.
Texture Mapping	The process of applying a digitized image onto a polygon or structure composed of polygons.
Texture uv Coordinates	WTK allows you to specify how the texture is mapped onto a polygon, by allowing you to specific texture (uv) coordinates in polygon definitions.
Toolbars	A toolbar object is a group of push buttons, each painted with an appropriate bitmap.
TPS	Triangles per second. A commonly used statistic used to compare performance characteristics of graphics hardware.
Transform Node	A scene graph node used to specify position and orientation information.
Transform Separator Node	A scene graph node used to “encapsulate” the effects of all transform nodes below it in the scene graph. In other words, the transform nodes below a transform separator only affect the portion of the scene graph below the transform separator, and nowhere else.
Transformation Matrix	See <i>4x4 Matrix</i> and <i>3x3 Matrix</i> .
Translation	A change in position.
Transparency	If a texture is transparent, you will be able to see through portions of the geometry to which the texture is applied. Transparency is achieved by not rendering black texels.
Traversal Order	The order in which nodes in a scene graph are processed while the simulation is running. WTK starts at the root node and traverses the scene graph tree from top to bottom and left to right.
Universal Resource Locator (URL)	String properties used in VRML files to specify a file location and file name that contains data to be imported.

Universe	The container of all entities considered by the WorldToolKit simulation manager – including graphical objects and other entities: lights, sensors, viewpoints, etc.
Universe Action Function	The action function is a user-defined function that is called by the simulation manager each time through the simulation loop. Using the action function, you can specify actions involving any WTK objects, graphical or otherwise. For example, program termination, or simulation activities like terrain-following, object manipulation, intersection testing, or changes to rendering parameters like lighting conditions or background color can be modified through this function.
Vertex	A single point in three-dimensional space that defines a corner of a polygon. A sequence of vertices defines a polygon.
Vertex Normals	A direction vector used for shading and rendering. You can generate vertex normals with a modeling program or use the NFF automatic normal generation feature. When WTK reads a vertex with a normal associated with it, it automatically renders the associated polygon as Gouraud-shaded. See also <i>Normals</i> .
Viewpoint	The observer's point of view (position and orientation) from which the scene is viewed.
VRML 1.0	An acronym for Virtual Reality Modeling Language. A specification for the design and implementation of a platform-independent language for virtual reality scene description.
Window Projection Types	See <i>Projection Type</i> .
Wireframe	A rendering style in which textures, materials, and shading is not visible because only the outlines of polygons will be rendered, i.e. polygons will not be solid-filled. Rendering using the wireframe style typically achieves the highest frame rate of any of the rendering styles.
World coordinate system	A coordinate system used to notate a world or universe. The origin (0,0,0) of the WCS is typically located at the center of a world or universe.

World Origin	Defined as 0, 0, 0 in X, Y, Z world coordinates. The world origin is the default location of new geometries.
WRL	The VRML 1.0 file format used for representing hierarchical 3D geometry and other data.
WTK Coordinate System	The WTK coordinate convention has X pointing to the right, Y pointing down, and Z pointing straight ahead. See also <i>Right-hand Rule</i> .
WTK Tolerance Factor	The WTK tolerance factor is specified with WTFUZZ (a defined constant equal to 0.004).
X Resources	On UNIX platforms, WTK provides the ability to set certain parameters from an X Resources file when your application starts up. For example, you can specify background color, viewing angle, window size, and window position this way.
Yaw	The orientation of an object about the Y axis.
Yon Clipping Plane	The physical range in front of the viewpoint, beyond which objects are not rendered in that window. That is, objects appearing beyond the yon clipping plane are not rendered. Objects are rendered only in the area between the <i>hither clipping plane</i> and the yon clipping plane.
Z-buffer	A software or hardware buffer that stores Z coordinate information.

- 2D
 - graphical entities 17-24
 - vectors 25-4
 - window points, usage example A-6
 - 2D fonts 17-24
 - WTwindow_draw2Dtext 19-7
 - WTwindow_get2Dtextextents 19-7
 - WTwindow_set2Dfont 19-6
 - 3D
 - geometry 6-1
 - matrices 25-21
 - vectors 25-5
 - 3D fonts
 - 3D font file 9-5
 - ASCII character set 9-6
 - base points 6-20, 9-3, 9-6
 - character placement 6-20
 - creating text strings 6-20
 - default spacing 9-5
 - diagram, extents box 9-4
 - diagram, origin and spacing 9-2
 - extents definition 9-4
 - origin 6-20
 - overview 9-1
 - spacing 6-20, 9-3
 - WFont3d_charexists 9-5
 - WFont3d_delete 9-3
 - WFont3d_getextents 9-4
 - WFont3d_getspacing 9-3
 - WFont3d_load 9-2
 - WFont3d_setspacing 9-3
 - WGeometry_newtext3d 6-20
 - 3D matrix functions
 - WTm3_init 25-21
 - WTm3_multm3 25-22
 - WTm3_transpose 25-22
 - 3D Mouse
 - defined constants 13-76
 - list of functions 13-8
 - manufacturer J-1
 - overview 13-73
 - raw data 13-74
 - scaling records 13-75
 - suspend button 13-76
 - update function 13-76
 - warning message D-6, D-13
 - WTbaron_update 13-76
 - 3D sound
 - see Sound support library
 - 3D Studio
 - 3DS file format 6-2
 - warning message D-10, D-14
 - 4D matrix functions
 - WTm4_copy 25-23
 - WTm4_init 25-23
 - WTm4_invert 25-24
 - WTm4_multm4 25-24
 - WTm4_rotatep3 25-25
 - WTm4_transpose 25-23
 - WTm4_xformp3 25-24
 - 5DT Glove
 - calibrating 13-66
 - changing the hand model 13-66
 - defined constants 13-65
 - list of functions 13-8
 - manufacturer J-1
 - overview 13-63
 - raw data 13-63
 - scaling records 13-64
 - update function 13-64
 - WTglove5dt_calibrateclosed 13-66
 - WTglove5dt_calibrateopen 13-66
 - WTglove5dt_loadhandmodel 13-66
 - WTglove5dt_rawupdate 13-65
 - WTglove5DT_update 13-64
 - WTglove5dt_updatefingers 13-64
 - 6D
 - coordinate frame structure 25-19
 - sensors 13-16
- ## A
- Absolute sensor records
 - definition 13-2
 - generating relative records for 13-25
 - setting 13-25
 - Accessing 13-71
 - Accumulated transformation 4-84
 - Acoustetron 20-2
 - Action function 2-7
 - setting 2-12
 - stopping the simulation 2-9

- with keyboard function 24-2, 24-3
 - Ambient light node
 - and performance 12-4
 - creating 12-5
 - definition 12-1
 - Ancestor node 4-7
 - Anchor nodes
 - description 4-13
 - functions 4-63
 - overview 4-28
 - setting the URL path 4-62
 - WTanchornode_getlocation 4-63
 - WTanchornode_setlocation 4-63
 - Angular rate
 - getting from a sensor 13-13
 - setting for a sensor 13-12
 - Animation
 - example using switch nodes A-26
 - Anti-aliasing
 - definition M-2
 - enabling 6-34
 - WTgeometry_setrenderingstyle 6-33
 - Appearance of geometries
 - changing (example of) A-13
 - Application development process 1-13
 - Ascension Bird
 - see Bird
 - Ascension Extended Range Bird
 - see Extended Range Bird
 - Ascension Flock of Birds
 - see Flock of Birds
 - Aspect ratio
 - and viewpoints 16-18
 - and windows 17-5
 - definition M-2
 - Asynchronous connections 21-22
 - WTconnection_issynchronous 21-31
 - WTconnection_setsynchronous 21-30
 - AutoCAD 6-1
 - Automatic normal generation
 - with NFF files F-8
- B**
- Back faces
 - also see Polygons
 - and geometry-constructor functions 6-11
 - default 6-23
 - definition 6-14
 - rejection 6-10, 6-14
 - Backward compatibility
 - aligning object axis G-28
 - animation G-31
 - assigning tasks to objects G-29
 - attaching objects G-25
 - coordinate frames G-34
 - diagram, new vs. old universe G-3
 - equivalent functions G-9
 - group files in NFF G-25
 - group functions G-32
 - handles G-32
 - instancing G-5
 - lights G-6, G-26
 - loading objects G-22
 - mapping 2.1 functions G-22
 - material colors G-35
 - materials G-5
 - materials in NFF G-25
 - new functions G-34
 - new paradigms G-2
 - NFF files G-24
 - nodes G-4
 - object axes G-20
 - overview G-1
 - picking G-31
 - pivot points G-32
 - positioning and moving objects G-23, G-30
 - positioning static objects G-28
 - replaced features G-8
 - scene graph concepts G-2
 - sensors G-27
 - terrains G-8
 - Version 2.1 to Release 6/7/8/9 G-1
 - Base points 6-20, 9-3, 9-6
 - Baud rates
 - error message D-4
 - Benchmarks
 - obtaining A-38
 - BFF
 - also see NFF
 - binary file format 6-3
 - overview F-1
 - Billboard effect
 - usage example A-33
 - Bird
 - constants C-12
 - DIP switch settings 13-39

- interference 13-42
 - list of functions 13-7
 - manufacturer J-1
 - overview 13-39
 - raw data 13-41
 - scaling records 13-41
 - setting Bird hemisphere 13-43
 - synchronization with CRT 13-42
 - update function 13-42
 - warning message D-7, D-11
 - WTbird_getabsoluterecord 13-41
 - WTbird_gethemisphere 13-43, 13-44
 - WTbird_sethemisphere 13-43
 - WTbird_setsync 13-42
 - WTbird_update 13-42
 - Blocks (boxes)
 - creating 6-15
 - Bluebook Ethernet standard 22-2
 - BOOM
 - BOOM2C versus BOOM3C 13-55
 - defined constants 13-57
 - example 13-55
 - joystick 13-58
 - list of functions 13-7
 - manufacturer J-2
 - overview 13-55
 - raw data 13-56
 - scaling records 13-56
 - update function 13-57
 - warning message D-6
 - WTboom_update 13-57
 - WTsensor_getmiscdata 13-57
 - Boston Dynamic's DiGuy, using with WTK ...
 - A-39
 - Bounding boxes
 - also see Intersection testing
 - making visible 4-73
 - overview 4-72
- ## C
- C++ wrappers
 - transitioning from R6 to R7/8/9 H-1
 - using 26-1
 - CADMOVER program 6-3
 - Callback functions 2-7
 - Center of gravity
 - definition 7-5
 - example 7-9
 - of a texture 10-27
 - Child node 4-7
 - CIS Graphics Geometry Ball Jr.
 - see Geometry Ball Jr.
 - Classes
 - access methods 1-4
 - constructors/destructors 1-5
 - WTK classes 1-3
 - Client-server networking (via World2World) .
 - 21-1
 - Close functions for sensors
 - closefn 13-9
 - customizing E-6
 - Clouds A-13
 - Collision detection
 - see Intersection testing
 - Color
 - also see Materials
 - as related to light 12-3
 - setting background color from a file ... 2-28
 - Composite transformations 4-30
 - Cones
 - creating 6-16
 - Connections
 - callbacks 21-23
 - functions 21-26
 - overview 21-22
 - synchronous vs. asynchronous 21-22
 - update rates 21-23
 - WTconnection_addcallback 21-31
 - WTconnection_connect 21-28
 - WTconnection_delete 21-27
 - WTconnection_deleteallenumtrees ... 21-36
 - WTconnection_deleteenumtreebyid .. 21-36
 - WTconnection_disconnect 21-28
 - WTconnection_getcallback 21-32
 - WTconnection_getclockdiff 21-30
 - WTconnection_getconnections 21-27
 - WTconnection_getdata 21-27
 - WTconnection_getenumtree 21-37
 - WTconnection_getenumtreebyid 21-37
 - WTconnection_getenumtreeid 21-38
 - WTconnection_getlatency 21-30
 - WTconnection_getmyid 21-28
 - WTconnection_getmyname 21-29
 - WTconnection_getroot 21-32
 - WTconnection_getstatus 21-29
 - WTconnection_getupdaterate 21-31

- WTconnection_getuserid 21-33
- WTconnection_getuseridbyname 21-33
- WTconnection_getusername 21-33
- WTconnection_getusernamebyid 21-33
- WTconnection_issynchronous 21-31
- WTconnection_new 21-26
- WTconnection_next 21-28
- WTconnection_numcallbacks 21-32
- WTconnection_numenumtrees 21-37
- WTconnection_numusers 21-32
- WTconnection_print 21-29
- WTconnection_remove 21-32
- WTconnection_setdata 21-27
- WTconnection_setsynchronous 21-30
- WTconnection_setupdaterate 21-31
- WTconnection_synch 21-30
- WTconnection_update 21-29
- WTconnection_updateconnections 21-29
- WTsharegroup_getconnection 21-18
- WTuniverse_deleteconnections 21-28
- Constants
 - also see WTCONSTANTS by name
 - appendix C-1
 - X, Y, Z, and W defined values 25-1
- Constraining
 - sensor input 13-21
 - sensor records E-3
 - through constants C-1
- Constraint frame
 - setting for a motion link 15-10
- Constructor functions
 - creating custom geometries 6-21
 - creating predefined geometries 6-14
- Content nodes
 - definition 4-5
 - encountering 4-10
 - fog 4-11
 - geometry 4-11
 - light 4-11
 - transform 4-11
- Convergence distance
 - diagram 16-22
 - getting for a display 16-21
 - getting for a viewpoint 16-23
 - setting for a display 16-21
 - setting for a viewpoint 16-4, 16-22
- Conversion functions
 - 4D matrix to pq 25-30, 25-31
 - direction vector to quaternion 25-30, 25-31
 - euler to matrix 25-27
 - euler to quaternion 25-27
 - getting a quaternion's rotation 25-17
 - matrix to euler 25-27
 - matrix to euler (nearest) 25-32
 - matrix to quaternion 25-26
 - normal to slope (radians) 25-33
 - overview 25-25
 - pq to 4D matrix 25-31
 - quaternion to 3D matrix 25-26
 - quaternion to 4D matrix 25-32
 - quaternion to direction vector 25-30
 - quaternion to euler 25-29
 - quaternion to euler (nearest) 25-32
 - setting a quaternion's rotation 25-17
- Conversion programs
 - geometries 6-3
- Coordinate
 - conventions 17-23
 - frame example 4-33
 - frames 4-32
 - sensor coordinate axes 13-16
 - systems, definition M-4
 - transformations for viewpoints 16-24
- Coordinate frame structures
 - see Math library
- Coordinate systems
 - local M-7
 - parent M-9
 - world M-17
- Coplanar
 - polygons 6-12
 - testing 25-11
- CRE sounds 20-15
- CRT
 - CRT based-device (BOOM) 13-55
 - interference 13-42
- Crystal River Engineering
 - manufacturer J-2
- CrystalEyesVR
 - defined constants 13-110
 - list of functions 13-8
 - manufacturer J-2
 - overview 13-108
 - raw data 13-109
 - reference frame diagram 13-108
 - scaling records 13-109
 - update function 13-110
 - with Logitech tracker 13-77

WTCrystaleyesVR_update 13-110
 Culling
 overview 4-4, 4-56
 Cullmode constants C-8
 Current element
 pointer 14-17
 Current node
 pointer 14-16
 Current viewpoint 14-28
 Custom graphical objects 6-1
 Custom sensor drivers
 writing 13-24
 Customer support
 see Technical support
 CyberGlove
 introduction 13-123
 Cyberglove
 accessing bend angle data 13-132
 accessing graphical hand model objects
 13-130
 calibration 13-126
 defined constants 13-134
 graphical hand model 13-127
 graphical hand model visibility 13-130
 initialization 13-124
 on windows platforms 13-135
 WTcybglove_deletehandmodel 13-129
 WTcybglove_getanglearray 13-132
 WTcybglove_getfingers 13-131
 WTcybglove_getforearm 13-130
 WTcybglove_getpalm 13-131
 WTcybglove_new 13-124
 WTcybglove_setvisibility 13-130
 WTcybglove_showcalibrationpanel 13-126
 WTcybglove_usehandmodel 13-127
 CyberMaxx2 HMD 13-119
 example 13-119
 list of functions 13-9
 manufacturer J-2
 overview 13-119
 raw data 13-120
 scaling records 13-120
 WTcybermaxx2_rawupdate 13-121
 WTcybermaxx2_update 13-120
 Cylinders
 creating 6-15

D

Data types for properties
 listing of 3-14
 WTproperty_getdatatype 3-16
 Debugging
 math print functions 25-1
 Default
 color of polygons 6-23
 geometry rendering constant 6-34
 material-table entry 8-14
 sampling rate of viewpoint position and
 orientation 14-14
 sensor sensitivity 13-11
 universe rendering options 2-18
 Defined constants C-1
 Depth-first traversal of scene graph 4-84
 Descendant node 4-7
 Device-specific constructor functions
 example 13-6
 overview 13-6
 Directed light 4-11
 Directed light node
 definition 12-1
 Direction constants C-9
 Direction vector
 convert to quaternion 25-30
 Directories, reading 24-4
 Display
 also see Windows
 characteristics 2-18
 manufacturers J-2
 WTDISPLAY constants 2-2, C-2
 Distributed simulations
 see Networking
 Documentation
 available for WTK 1-7
 Drawing constants C-2
 Drawing functions
 2D 19-1
 3D 19-8
 user-defined 19-1
 DXF
 and WTuniverse_getinitialview 2-16
 error message D-2, D-3
 file format 6-2
 texture specification 10-22
 warning message D-9, D-12

E

Electromagnetic sensors
 overview 13-2

Enumeration
 overview 21-34
 WTconnection_deleteallnumtrees 21-36
 WTconnection_deleteenumtreebyid .. 21-36
 WTconnection_getenumtree 21-37
 WTconnection_getenumtreebyid 21-37
 WTconnection_getenumtreeid 21-38
 WTconnection_numenumtrees 21-37
 WTsharegroup_enumerate 21-36

Environment VariablesB-1

Error messages
 appendix D-1
 application function 24-8

Ethernet 22-1, 22-2

Euler angles, converting 25-25
 also see Conversion functions

Event order constantsC-3

Events
 detecting keyboard events A-8
 event handlers 3-23
 overview 3-23
 WTbase_removeallhandlers 3-26
 WTproperty_addhandler 3-25
 WTproperty_gethandler 3-26
 WTproperty_numhandlers 3-25
 WTproperty_removeallhandlers 3-26
 WTproperty_removehandler 3-25
 WTuniverse_processevents 3-26

Explosions A-15

Extended Range Bird
 constantsC-12
 list of functions 13-7
 overview 13-51
 raw data 13-52
 scaling records 13-52
 update function 13-53

Extents
 3D font 9-4

Extents box
 defined 4-52
 radius 4-52

Extruded geometry
 creating 6-19

Eye constantsC-3

F

Fakespace BOOM
 see BOOM

Fakespace Pinch Glove System
 see Pinch Glove

FALSE constantC-6

FAQsA-1
 animation exampleA-26
 associating a task with an objectA-21
 changing the event orderA-34
 detecting button events using "misc data"
 functionsA-10
 detecting keyboard eventsA-8
 displaying multiple instances of an object ...
 A-5
 dynamically changing a geometry's
 appearanceA-13
 explanation of WTmovnode_load versus
 WTnode_loadA-4
 getting a pointer to a nodeA-20
 getting pointer to a WTK displayA-38
 getting the object positionA-25
 getting transparencies in a textureA-12
 handling portalsA-22
 keeping an object perpendicularA-33
 loading lights as movablesA-15
 LOD exampleA-29
 measuring performanceA-38
 objects following a lightA-16
 objects following the viewpointA-16
 orienting sensors differentlyA-36
 picking the frontmost polygonA-6
 recursively walking down the scene graph ..
 A-19
 terrain followingA-31
 testing for intersections between viewpoint
 and universeA-24
 testing for objects intersecting with objects .
 A-25
 using material tables for colorA-11
 window usageA-35
 WTmovnode_load versus WTnode_load
 A-4
 WTnode_load versus
 WTgeometrynode_loadA-3

FASTRAK
 adaptive filtering 13-95
 DIP switch settings 13-92

- error message D-9
 - list of functions 13-8
 - manufacturer J-1
 - overview 13-92
 - raw data 13-93
 - scaling records 13-94
 - signal filtering 13-94
 - unit argument 13-9
 - update function 13-94
 - warning message D-9
 - WTfastrak_afilter 13-95
 - WTfastrak_afilteroff 13-95
 - WTfastrak_pfilter 13-95
 - WTfastrak_pfilteroff 13-95
 - WTfastrak_update 13-94
 - Field sequential mode, stereo viewing 2-34
 - Fifth Dimension Technologies' 5DT Glove
 - see 5DT Glove
 - File formats
 - loading unsupported ones 6-3
 - used with WTgeometrynode_load 4-47
 - used with WTK 6-2
 - File selection box object
 - see User interface
 - Filetype constants C-3
 - Filtering (mipmapping)
 - textures overview 10-24
 - Flashing
 - between coplanar surfaces 6-12, 6-13
 - Flipbook method, animation A-26
 - Floating point
 - comparisons 25-33
 - tolerances 6-13
 - Flock of Birds
 - constants C-12
 - Extended Range Bird 13-51
 - list of functions 13-7
 - manufacturer J-1
 - overview 13-51
 - unit argument 13-9
 - warning message D-6
 - FLT file format 6-3
 - Fog
 - attributes 4-64
 - constants C-8
 - construction functions 4-45
 - overview 4-64
 - WTfognode_getcolor 4-65
 - WTfognode_getlinearstart 4-67
 - WTfognode_getmode 4-66
 - WTfognode_getrange 4-66
 - WTfognode_new 4-45
 - WTfognode_setcolor 4-65
 - WTfognode_setlinearstart 4-66
 - WTfognode_setmode 4-66
 - WTfognode_setrange 4-65
 - Fog nodes
 - see Fog
 - description 4-11
 - Fonts
 - see 2D fonts
 - see 3D fonts
 - Form object
 - see User interface
 - Formula T2
 - list of functions 13-8
 - manufacturer J-2
 - overview 13-111
 - raw data 13-112
 - scaling records 13-112
 - update function 13-112
 - WTformula_drive 13-112
 - WTformula_rawupdate 13-113
 - Frame of reference
 - constants C-4
 - description 4-31
 - overview 4-32
 - rotating sensor input 13-16
 - viewpoint functions 16-14
 - Frame rate
 - back face rejection 6-10
 - Frames per second 2-23
 - Frequently Asked Questions
 - see FAQs A-1
 - Front faces
 - see Polygons
- ## G
- Gameport Joystick 13-8
 - defined constants 13-71
 - range 13-72
 - raw data 13-69
 - reinitializing 13-73
 - scaling records 13-69
 - update functions 13-69
 - WTjoystick_fly 13-70

- WTjoystick_getdrift 13-73
- WTjoystick_getrange 13-72
- WTjoystick_rawupdate 13-71
- WTjoystick_readcalibrationfile 13-73
- WTjoystick_setdrift 13-72
- WTjoystick_walk 13-70
- WTjoystick_walk2 13-70
- Gas Clouds A-13
- GEO
 - file format 6-3
- Geographically disbursed simulations
 - see Networking
- Geometry
 - 3D text string 6-20
 - also see WTGeometry functions by name
 - animating A-26
 - back faces 6-14
 - blocks (boxes) 6-15
 - bounding box extents 6-29
 - cones 6-16
 - construction functions 4-44
 - constructor functions 6-11
 - creating custom geometries 6-21
 - creating predefined types 6-14
 - creation and deletion 6-26
 - cylinders 6-15
 - dynamic construction 7-10
 - editing (example of) A-13
 - extents box 4-51
 - extents,radius, and midpoint diagram .. 4-52
 - extrusions 6-19
 - functions 6-14
 - getting rendered position of A-25
 - hemispheres 6-17
 - midpoint 4-51, 6-28
 - modification 6-37
 - moving example A-25
 - nodes 6-1
 - optimization 6-39
 - overview 6-1
 - prebuild 6-39
 - properties 4-51, 6-28
 - pyramid 6-16
 - radius 4-51
 - rectangles 6-17
 - referencing vertices 7-10
 - spheres 6-16
 - static and dynamic 4-2
 - stretching and scaling 6-37
 - terrain example A-31
 - tetrahedon 6-16
 - triangular prisms 6-15
 - truncated cones 6-18
 - using materials tables with 8-18
 - using materials with 6-30
 - vertex-level editing 6-42
 - warning message D-6
 - WTnode_load versus
 - WTgeometrynode_load A-3
- Geometry Ball Jr.
 - defined constants 13-55
 - example 13-17
 - list of functions 13-7
 - overview 13-53
 - raw data 13-53
 - reference frame diagram 13-18
 - scaling records 13-54
 - update function 13-54
 - warning message D-11
 - writing a driver E-10
 - WTgeoball_present 13-55
 - WTgeoball_update 13-54
- Geometry motion reference frames
 - rotating sensor input 13-19
- Geometry nodes
 - description 4-11
- Global options
 - constants C-9
 - rendering parameters 2-18
- Gouraud shading 6-8, F-4
- Graphic boards
 - manufacturers J-3
- Group movable nodes
 - creation 5-4
 - see also Movables
- Group node
 - description 4-13
- Grouping nodes
 - anchor 4-13
 - definition 4-6
 - group 4-13
 - inline 4-13
 - level of detail 4-13
 - root 4-13
 - switch 4-13
 - transform separator 4-14
- GUI elements
 - see User interface

H

Hardware Guides	1-10
Head Tracker	
defined constants	13-81
diagram	13-79
overview	13-77
raw data	13-79
scaling records	13-80
update function	13-80
WTlogitech_update	13-80
Head-mounted display	
setting convergence	16-21
with viewpoint paths	14-20
Heads-up display	
creating	17-24
Helmet orientation	
with Logitech tracker (diagram)	13-79
Hemispheres	
creating	6-17
Hither clipping	17-18
Host-specific window	
usage example	A-35
Http	
specifying a link to	4-46
support for	4-46

I

i-glasses!

list of functions	13-9
manufacturer	J-2
overview	13-121
raw data	13-122
scaling records	13-122
update function	13-122
WTiglasses_rawupdate	13-123
WTiglasses_update	13-122

Image file

conversion	I-2
loading into window	17-25
warning message	D-7, D-10

Image flashing

overlapping polygons	6-12
----------------------------	------

Inline nodes

description	4-13
functions	4-63
overview	4-28
setting the URL path	4-62

WTinlinenode_getlocation	4-64
WTinlinenode_setlocation	4-63

Input sensors

also see Sensors	
manufacturers	J-1
supported by WTK	1-12
Inside surfaces of geometries	6-14

InsideTRAK

list of functions	13-8
manufacturer	J-1
overview	13-90
raw data	13-91
scaling records	13-91
update function	13-91
WTinsidettrak_update	13-91

Instance

defined	4-37
usage example	A-5

Instancing

backward compatibility	G-5
node types supported	6-4
scheme supported	6-4
with the scene graph	4-37

Interlaced mode, stereo viewing

hardware interlaced	2-37
overview	2-36
stencil interlaced	2-36

Internal fault

error message	D-2
---------------------	-----

Internet

IP and UDP guidelines	22-3
TCP	22-3

Internet protocol

time to live value	22-8
--------------------------	------

Interpolating

paths	14-1, 14-6
-------------	------------

Intersection testing

collision detection example	A-25
during simulation loop	2-11
overview	4-85
usage example	A-24
using bounding boxes	4-86
using node instances	4-86
using polygons	4-85
using polygons in node paths	4-86
using rays and nodes	4-88
using rays and polygons	4-88
using viewpoints	4-89
using WTviewpoint_getlastposition	16-9

-
- Introduction
 - what is WorldToolKit? 1-1
 - Invalid
 - polygons (warning message) D-5
 - quaternion (warning message) D-11
 - texture name (error message) D-2
 - Invert a 4D matrix
 - WTm4_invert 25-24
 - IP address
 - default 22-8
 - ipaste (Iris utility) I-2
 - ISOTRAK
 - DIP switch settings 13-85
 - list of functions 13-8
 - manufacturer J-1
 - overview 13-85
 - raw data 13-86
 - reference frame diagram 13-86
 - scaling records 13-86
 - update function 13-88
 - WTPolhemus_update 13-88
 - ISOTRAK II
 - DIP switch settings 13-88
 - list of functions 13-8
 - manufacturer J-1
 - overview 13-88
 - raw data 13-89
 - scaling records 13-89
 - update function 13-90
 - WTisotrak2_update 13-90
- ## K
- Keyboard 24-1
 - constants C-4
 - example 24-3
 - input buffer 24-2
 - using to detect events A-8
 - WTkeyboard_close 24-3
 - WTkeyboard_getkey 24-2
 - WTkeyboard_getlastkey 24-2
 - WTkeyboard_open 24-1
- ## L
- Label object
 - see User interface
 - Latency
 - see Networking
 - Leaf node 4-7
 - Level of detail
 - see LOD nodes
 - License
 - server (error message) D-2
 - type (error message) D-2
 - type (warning message) D-11
 - warning message D-8
 - Light nodes
 - see Lights
 - Lights
 - attributes 12-2
 - backward compatibility G-6
 - constants C-5
 - constructing light nodes 4-43, 12-5
 - description 4-11
 - directed 4-11
 - direction 12-13
 - disabling 12-4
 - example file 12-11
 - intensity 12-3
 - list of light nodes G-27
 - loading lights from file 12-9
 - management 12-5
 - maximum number 12-4
 - object following a light A-16
 - overview 12-1
 - performance impact 12-4
 - point light 4-11
 - properties 12-12
 - saving to a file 12-11
 - shadows 12-4
 - specifying ambient lighting 12-10
 - specifying directed lights 12-9
 - spot light 4-11
 - spot light diagram 12-8
 - types 12-1
 - WTlightnode_getambient 12-15
 - WTlightnode_getangle 12-19
 - WTlightnode_getattenuation 12-18
 - WTlightnode_getdiffuse 12-16
 - WTlightnode_getdirection 12-13
 - WTlightnode_getexponent 12-20
 - WTlightnode_getintensity 12-14
 - WTlightnode_getposition 12-12
 - WTlightnode_getspecular 12-16
 - WTlightnode_gettype 12-18
 - WTlightnode_load 12-9

- WTlightnode_newambient 12-5
- WTlightnode_newdirected 12-6
- WTlightnode_newpoint 12-7
- WTlightnode_newspot 12-8
- WTlightnode_save 12-11
- WTlightnode_setambient 12-14
- WTlightnode_setangle 12-19
- WTlightnode_setattenuation 12-17
- WTlightnode_setdiffuse 12-15
- WTlightnode_setdirection 12-13
- WTlightnode_setexponent 12-19
- WTlightnode_setintensity 12-14
- WTlightnode_setposition 12-12
- Links
 - see Motion Links
- List
 - of paths 14-8, 14-10
 - of sensors 13-10
 - of vertices 7-10
 - of viewpoints 16-4
 - of windows 17-8
- Local coordinate system
 - defined M-7
- Local networks
 - see Networking
- Locked properties
 - description 21-3
 - WTproperty_islocked 21-10
 - WTproperty_islockedbyme 21-11
 - WTproperty_lock 21-10
 - WTproperty_unlock 21-10
- Locked sharegroups
 - description 21-12
 - WTsharegroup_islocked 21-19
 - WTsharegroup_islockedbyme 21-19
 - WTsharegroup_lock 21-19
 - WTsharegroup_unlock 21-19
- LOD nodes
 - description 4-13
 - overview 4-26
 - usage example A-29
 - WTlodnode_getcenter 4-56
 - WTlodnode_getrange 4-55
 - WTlodnode_numranges 4-56
 - WTlodnode_setcenter 4-56
 - WTlodnode_setrange 4-55
- Logitech 3D Mouse
 - see 3D Mouse
- Logitech Head Tracker
 - see Head Tracker
- Logitech Magellan
 - see Space Control Mouse
- Logitech Red Baron
 - see 3D Mouse
- Logitech Space Control Mouse
 - see Space Control Mouse
- M**
- Magellan
 - see Space Control Mouse
- Material ID
 - usage example A-11
- Material index table entries
 - WTgeometry_getvertexmatid 6-48
 - WTgeometry_setmatid 6-31
 - WTgeometry_setvertexmatid 6-48
 - WTPoly_getmatid 7-3
 - WTPoly_setmatid 7-3
- Material properties
 - ambient 8-2
 - ambient-diffuse 8-3
 - calculations made for color 8-3
 - determining color 8-3
 - diagram 8-4
 - diffuse 8-2
 - emissive 8-2
 - opacity 8-2
 - overview 8-1
 - shininess 8-2
 - specular 8-2
 - with 3DS files 6-2
 - WTmtable_newentry 8-14
- Material tables
 - constants C-5
 - entry functions 8-14
 - file formats used 8-19
 - NFF F-4
 - out-of-range indices 8-17
 - overview 8-5
 - results of creating a table 8-9
 - sample material file 8-6
 - saving 8-12
 - usage example A-11
 - WTmtable_copyentry 8-14
 - WTmtable_delete 8-8
 - WTmtable_getbyname 8-13

WTmtable_getdata	8-13	WTm3_init	25-21
WTmtable_getentrybyname	8-17	WTm3_multm3	25-22
WTmtable_getentryname	8-16	WTm3_transpose	25-22
WTmtable_getname	8-13	WTm4_2pq	25-31
WTmtable_getnumentries	8-8	WTm4_copy	25-23
WTmtable_getproperties	8-11	WTm4_init	25-23
WTmtable_getvalue	8-15	WTm4_invert	25-24
WTmtable_load	8-11	WTm4_multm4	25-24
WTmtable_merge	8-8	WTm4_rotatp3	25-25
WTmtable_new	8-7	WTm4_transpose	25-23
WTmtable_newentry	8-14	WTm4_xformp3	25-24
WTmtable_save	8-12	WTnormal_2slope	25-33
WTmtable_setdata	8-13	WTP2_copy	25-4
WTmtable_setentryname	8-16	WTP2_dot	25-5
WTmtable_setname	8-12	WTP2_mag	25-4
WTmtable_setproperties	8-9	WTP2_norm	25-4
WTmtable_setvalue	8-15	WTP3_add	25-7
WTPoly_getmatid	7-3	WTP3_coplanar	25-11
WTPoly_setmatid	7-3	WTP3_copy	25-6
Materials		WTP3_cross	25-8
advanced topics	8-17	WTP3_distance	25-11
backward compatibility	G-5	WTP3_diststovector	25-11
material tables	8-5	WTP3_dot	25-7
overview	8-1	WTP3_equal	25-8
properties	8-1	WTP3_exact	25-9
Math conventions		WTP3_frame2frame	25-34
overview	25-2	WTP3_init	25-5
used with custom sensor drivers	E-2	WTP3_invert	25-6
Math library		WTP3_local2worldframe	25-34
also see Conversion functions		WTP3_mag	25-6
2D vectors	25-4	WTP3_multm3	25-10
3D matrices	25-21	WTP3_multm4	25-10
3D vectors	25-5	WTP3_mults	25-10
4D matrices	25-22	WTP3_norm	25-6
conversion functions	25-25	WTP3_print	25-12
floating point comparisons	25-33	WTP3_rotate	25-9
math data types	25-1	WTP3_rotatp3	25-9
overview	25-1	WTP3_subtract	25-7
WorldToolKit conventions	25-2	WTP3_world2localframe	25-34
WTDdir_2q	25-30	WTP3_xform	25-10
WTDdirandtwist_2q	25-31	WTPq_2m4	25-31
WTEuler_2m3	25-27	WTPq_copy	25-20
WTEuler_2q	25-27	WTPq_frame2frame	25-36
WTFUZZ	25-33	WTPq_init	25-20
WTm3_2euler	25-27	WTPq_local2worldframe	25-36
WTm3_2eulernear	25-32	WTPq_print	25-20
WTm3_2q	25-26	WTPq_world2localframe	25-36
WTm3_copy	25-22	WTq (quaternions)	25-12
WTm3_copyt	25-22	WTq_2dir	25-30

- WTq_2dirandtwist 25-30
- WTq_2euler 25-29
- WTq_2eulernear 25-32
- WTq_2m3 25-26
- WTq_2m4 25-32
- WTq_construct 25-17
- WTq_copy 25-15
- WTq_dot 25-18
- WTq_equal 25-16
- WTq_exact 25-16
- WTq_frame2frame 25-35
- WTq_getangle 25-17
- WTq_getvector 25-16
- WTq_init 25-14
- WTq_interpolate 25-18
- WTq_invert 25-15
- WTq_local2worldframe 25-35
- WTq_mag 25-15
- WTq_mult 25-17
- WTq_multinv 25-18
- WTq_norm 25-16
- WTq_print 25-19
- WTq_scale 25-17
- WTq_world2localframe 25-35
- WTzero 25-33
- Mathematical constants C-6
- Matrix
 - 3D functions 25-21
 - 4D functions 25-22
 - conversion, see Conversion functions
 - diagram 4-35
- Maximum angular rate
 - sensors 13-13
- Maximum rotation
 - sensors 13-12
- Memory
 - efficiency 6-13
- Menu bar object
 - see User interface
- Menu item button object
 - see User interface
- Menu pop-up button object
 - see User interface
- Mesh file (3DS) 6-2
- Meshing color information
 - radiosity preprocessing 6-10
- Message box object
 - see User interface
- Messages
 - constants C-6
 - latency 22-5
 - overview 24-5
 - WTErrors 24-8
 - WTmessage 24-5
 - WTmessage_sendto 24-6
 - WTwarning 24-6
- Midpoint
 - of a geometry 6-28
 - usage example A-25
- Mipmapping
 - also see Filtering
 - textures 10-24
- Miscellaneous data
 - example of detecting button events A-10
 - sensors 13-15, 13-25
- Missile plumes A-14
- Modeling
 - 3DS file format 6-2
 - back face rejection 6-10
 - BFF file format 6-3
 - considerations for geometries 6-2
 - converting files 6-11, I-3
 - coplanar considerations 6-12
 - DXF file format 6-2
 - error message 24-8, D-3
 - file formats supported 6-2
 - FLT file format 6-3
 - generating vertex normals 6-9
 - GEO file format 6-3
 - gouraud shading 6-8
 - NFF file format 6-3
 - OBJ file format 6-2
 - scale factor 6-14
 - scaling factors 6-13
 - setting model paths (resource files) 2-33
 - SLP file format 6-3
 - software programs I-3
 - subface considerations 6-7
 - texture uv considerations 10-13
 - textures in model files 10-22
 - using textures 10-2
 - vertex colors 6-9
 - vertex colors with FLT files 2-25
 - VRML file format 6-3
 - with a CAD program 6-7
 - World Up Modeler I-1
- Monitor distortion
 - correcting 16-18

- Monoscopic viewing
 - diagram 17-6
- Morph
 - usage example A-13
- Motion Links
 - constants C-7
 - constraints 15-3
 - default constraint frames 15-11
 - example of constraining 15-13
 - functions 15-3
 - reference frames 15-2, 15-4
 - sources 15-1
 - targets 15-1
 - valid constraint frames 15-10
 - valid reference frames 15-8
 - WTmotionlink_addconstraint 15-11
 - WTmotionlink_delete 15-5
 - WTmotionlink_enable 15-5
 - WTmotionlink_getconstraintframe 15-11
 - WTmotionlink_getdata 15-6
 - WTmotionlink_getreferenceframe 15-9
 - WTmotionlink_getsource 15-6
 - WTmotionlink_gettarget 15-6
 - WTmotionlink_isenabled 15-5
 - WTmotionlink_next 15-8
 - WTmotionlink_removeconstraint 15-12
 - WTmotionlink_setconstraintframe 15-10
 - WTmotionlink_setdata 15-6
 - WTmotionlink_setreferenceframe 15-8
 - WTpath_setrecordlink 14-15
 - WTuniverse_deletelink 2-17
 - WTuniverse_getmotionlinks 2-17
- Mouse
 - cursor E-8
 - defined constants 13-33
 - example of update function E-8
 - list of functions 13-7
 - overview 13-26
 - platform-independent sensor macro ... 13-26
 - raw data 13-27
 - scaling records 13-27
 - screen location 13-27
 - update functions 13-28
 - using as a trackball 13-36
 - warning message D-12
 - WTmouse_close 13-7
 - WTmouse_drawcursor 13-28
 - WTmouse_gettrackballdrift 13-37
 - WTmouse_gettrackballsnap 13-39
 - WTmouse_gettrackballsnapangle 13-38
 - WTmouse_inwindow 13-35
 - WTmouse_move2D 13-29
 - WTmouse_moveview1 13-29
 - WTmouse_moveview2 13-31
 - WTmouse_new 13-7
 - WTmouse_open 13-7
 - WTmouse_rawupdate 13-32
 - WTmouse_settrackballdrift 13-37
 - WTmouse_settrackballsnap 13-38
 - WTmouse_settrackballsnapangle 13-38
 - WTmouse_trackball 13-36
 - WTmouse_trackballreset 13-39
 - WTmouse_trackballvpoint 13-37
 - WTmouse_whichwindow 13-35
- movable node constants C-8
- Movable Nodes
 - see Movables
- Movables
 - attaching to parent node 5-11
 - components 5-2
 - creating 5-3
 - creating group nodes 5-4
 - creating LOD nodes 5-5
 - creating movable switch nodes 5-5
 - creating separator nodes 5-4
 - deleting attachments 5-12
 - detaching child nodes 5-12
 - diagram 5-1
 - directed light nodes 5-4
 - following a light A-16
 - geometry nodes 5-3
 - hierarchies 5-9
 - instancing 5-13
 - loading movable nodes 5-5
 - node position and orientation 5-7
 - number of attachments 5-13
 - overview 5-1
 - point light nodes 5-3
 - rotate about midpoint 5-8
 - rotating movable nodes 5-8
 - spot light nodes 5-4
 - WTmovenode_load versus WTnode_load ..
A-4
 - WTmovgeometrynode_new 5-3
 - WTmovgeometrynode_newdirected 5-4
 - WTmovgeometrynode_newpoint 5-3
 - WTmovgeometrynode_newspot 5-4
 - WTmovlodnode_new 5-5

WTmovnode_alignaxis	5-8
WTmovnode_attach	5-11
WTmovnode_axisrotation	5-8
WTmovnode_deleteattachment	5-12
WTmovnode_detach	5-12
WTmovnode_getattachment	5-13
WTmovnode_instance	5-13
WTmovnode_load	5-5
WTmovnode_numattachments	5-13
WTmovsepnodes_new	5-4
WTmovswitchnode_new	5-5
Multicast networking	22-1
Multicasting	22-3
IP address	22-8
MultiGen/ModelGen Flight file	6-5
format	6-3
subfaces	6-7
MultiPipe/Multi-Processor	
introduction	1-6
Multi-user simulations	
see Networking	

N

Naming conventions	
WTK functions	1-1, 1-4
Networking, client-server	
connection callbacks	21-23
connection functions	21-26
connections	21-22
enumeration	21-34
locked properties	21-3
locked sharegroups	21-12
overview	21-1
persistent properties	21-3
persistent sharegroups	21-14
property sharing functions	21-5
registered interest	21-13
Server Manager	21-1
sharegroup functions	21-15
sharegroups	21-11
sharing properties	21-2
Simulation Server	21-2
time-sensitive properties	21-4
unsupported object types	21-5
update frequencies for properties	21-3
update rates for connections	21-23
World Up compatible properties	21-38

WTbase objects	21-5
WTbase_unshare	21-11
WTconnection_addcallback	21-31
WTconnection_connect	21-28
WTconnection_delete	21-27
WTconnection_deleteallenumtrees	21-36
WTconnection_deleteenumtreebyid	21-36
WTconnection_disconnect	21-28
WTconnection_getcallback	21-32
WTconnection_getclockdiff	21-30
WTconnection_getconnections	21-27
WTconnection_getdata	21-27
WTconnection_getenumtree	21-37
WTconnection_getenumtreebyid	21-37
WTconnection_getenumtreeid	21-38
WTconnection_getlatency	21-30
WTconnection_getmyid	21-28
WTconnection_getmyname	21-29
WTconnection_getroot	21-32
WTconnection_getstatus	21-29
WTconnection_getupdate rate	21-31
WTconnection_getuserid	21-33
WTconnection_getuseridbyname	21-33
WTconnection_getusername	21-33
WTconnection_getusernamebyid	21-33
WTconnection_issynchronous	21-31
WTconnection_new	21-26
WTconnection_next	21-28
WTconnection_numcallbacks	21-32
WTconnection_numenumtrees	21-37
WTconnection_numusers	21-32
WTconnection_print	21-29
WTconnection_removecallback	21-32
WTconnection_setdata	21-27
WTconnection_setsynchronous	21-30
WTconnection_setupupdate rate	21-31
WTconnection_synch	21-30
WTconnection_update	21-29
WTconnection_updateconnections	21-29
WTproperty_getsharegroup	21-8
WTproperty_gettimesensitive	21-9
WTproperty_getupdatefreq	21-8
WTproperty_islocked	21-10
WTproperty_islockedby me	21-11
WTproperty_iss hared	21-7
WTproperty_lock	21-10
WTproperty_numshares	21-7
WTproperty_sendupdate	21-9
WTproperty_settimesensitive	21-9

-
- WTproperty_setupdatefreq 21-8
 - WTproperty_share 21-5
 - WTproperty_unlock 21-10
 - WTproperty_unshare 21-7
 - WTsharegroup_delete 21-16
 - WTsharegroup_enumerate 21-36
 - WTsharegroup_findchildbyname 21-21
 - WTsharegroup_getchild 21-20
 - WTsharegroup_getconnection 21-18
 - WTsharegroup_getdata 21-18
 - WTsharegroup_getname 21-18
 - WTsharegroup_getparent 21-20
 - WTsharegroup_getproperty 21-21
 - WTsharegroup_islocked 21-19
 - WTsharegroup_islockedbyme 21-19
 - WTsharegroup_issshared 21-17
 - WTsharegroup_lock 21-19
 - WTsharegroup_new 21-15
 - WTsharegroup_numchildren 21-20
 - WTsharegroup_numproperties 21-21
 - WTsharegroup_print 21-18
 - WTsharegroup_registerinterest 21-20
 - WTsharegroup_setdata 21-17
 - WTsharegroup_share 21-17
 - WTsharegroup_unlock 21-19
 - WTuniverse_deleteconnections 21-28
 - Networking, multicast
 - asynchronous communication 22-1
 - byte ordering 22-2, 22-6
 - default address 22-8
 - functions 22-7
 - group address 22-3
 - latency 22-5
 - local 22-1
 - net_actions 22-4, 22-5
 - netdemo.c 22-1, 22-4
 - network layers diagram 22-2
 - private 22-1
 - public 22-1
 - remote 22-1
 - sample transaction 22-4
 - tag field 22-7, 22-9
 - type field 22-7, 22-9
 - UDP 22-3
 - UDP packets 22-3
 - warning message D-7, D-12
 - WTnet_additem 22-9
 - WTnet_addstring 22-10
 - WTnet_close 22-9
 - WTnet_flush 22-13
 - WTnet_getport 22-13
 - WTnet_getrange 22-13
 - WTnet_next 22-11
 - WTnet_open 22-7
 - WTnet_removeitem 22-11
 - WTnet_removestring 22-12
 - WTnet_skip 22-13
 - Networks
 - warning message D-7
 - NFF 2-16, 6-1
 - 3D font files 9-5
 - automatic normal generation F-8
 - backward compatibility G-24
 - both (NFF keyword) F-6
 - both flag 6-11, 7-4
 - convenience of 6-11
 - end-of-line conventions F-2
 - file format 6-3
 - file structure F-1
 - header format F-2
 - ID (NFF token) F-8
 - material tables F-4
 - multi-object file 9-5
 - norm keyword F-4
 - number of polygons F-6
 - overview F-1
 - polygon color F-6
 - polygon ID 7-6
 - sample file F-10
 - syntax F-2
 - texture specification 10-22
 - using a 3D font 9-1
 - version history F-9
 - vertex normals 6-9
 - vertices F-4
 - viewdir token F-2
 - viewpos token F-2
 - warning message D-10
 - Node path
 - diagram 4-81
 - getting an accumulated transformation 4-84
 - intersection testing 4-85
 - locating a node 4-80
 - overview 4-79
 - WTnode_rayintersect 4-88
 - WTnodepath_addsensor 4-93
 - WTnodepath_delete 4-82
 - WTnodepath_gettextents 4-83

-
- WTnodepath_getnode 4-82
 - WTnodepath_getorientation 4-85
 - WTnodepath_gettransform 4-84
 - WTnodepath_gettranslation 4-84
 - WTnodepath_gettraversal 4-83
 - WTnodepath_intersectbbox 4-87
 - WTnodepath_intersectnode 4-87
 - WTnodepath_intersectpoly 4-86
 - WTnodepath_new 4-81
 - WTnodepath_numnodes 4-82
 - WTnodepath_removesensor 4-93
 - WTPoly_intersectnode 4-86
 - WTPolyintersectbbox 4-86
 - Nodes
 - accessing a node A-20
 - backward compatibility G-4
 - constants C-7
 - content 4-5
 - diagram of parent, child, siblings 4-6
 - enabling 4-50
 - extents box 4-52
 - grouping 4-6
 - how to get a pointer using node's name
 - A-20
 - properties 4-48
 - radius 4-54
 - root 4-52
 - root node 4-7
 - scene graph hierarchy 4-6
 - tracking inactive nodes G-20
 - transform nodes 4-5, 4-58
 - types 4-5
 - utility functions 4-76
 - warning message D-9
 - WTanchornode_new 4-40
 - WTgeometrynode_load 4-46
 - WTgetname 4-49
 - WTgroupnode_new 4-40
 - WTinlinenode_new 4-40
 - WTlodnode_new 4-41
 - WTnode_addchild 4-74
 - WTnode_addsensor 4-92
 - WTnode_axisrotation 4-62
 - WTnode_boundingBox 4-73
 - WTnode_canaddchild 4-51
 - WTnode_delete 4-75
 - WTnode_deletechild 4-75
 - WTnode_enable 4-49
 - WTnode_getchild 4-77
 - WTnode_getdata 4-51
 - WTnode_getextents 4-53
 - WTnode_getmidpoint 4-54
 - WTnode_getorientation 4-60
 - WTnode_getparents 4-78
 - WTnode_getradius 4-54
 - WTnode_getrotation 4-60
 - WTnode_gettransform 4-58
 - WTnode_gettranslation 4-59
 - WTnode_gettype 4-50
 - WTnode_hasboundingbox 4-73
 - WTnode_insertchild 4-74
 - WTnode_isenabled 4-50
 - WTnode_ismovable 4-50
 - WTnode_load 4-46
 - WTnode_load versus
 - WTgeometrynode_load A-3
 - WTnode_load versus WTMovnode_load
 - A-4
 - WTnode_numchildren 4-77
 - WTnode_numparents 4-77
 - WTnode_numpolys 4-78
 - WTnode_print 4-76
 - WTnode_remove 4-75
 - WTnode_removechild 4-74
 - WTnode_removesensor 4-92
 - WTnode_rotatem3 4-61
 - WTnode_rotatem4 4-62
 - WTnode_rotateq 4-61
 - WTnode_rotation 4-61
 - WTnode_save 4-48
 - WTnode_setdata 4-51
 - WTnode_setname 4-49
 - WTnode_setorientation 4-60
 - WTnode_setrotation 4-60
 - WTnode_settransform 4-58
 - WTnode_settranslation 4-59
 - WTnode_translate 4-59
 - WTnode_vacuum 4-76
 - WTrootnode_new 4-39
 - WTrootnode_next 4-77
 - WTsepnode_new 4-41
 - WTswitchnode_new 4-42
 - WTuniverse_findnodebyname 4-49
 - WTuniverse_getrootnodes 2-17
 - WTxformnode_new 4-42
 - WTxformsepnode_new 4-43
 - Non-coplanar polygons
 - warning message D-5

- Normals
 see Polygons
- NTSC 2-3
- O**
- OBJ file format 6-2
- Object/Property/Event architecture
 and World2World 21-1
 events 3-23
 overview 3-1
 properties 3-14
 supplied properties 3-2
 supported types 3-2
 time 3-27
 Wtbase functions 3-7
 Wtbase objects and functions 3-7
 Wtbase_addparent 3-8
 Wtbase_delete 3-11
 Wtbase_deleteproperties 3-13
 Wtbase_find 3-13
 Wtbase_findchild 3-10
 Wtbase_getchild 3-10
 Wtbase_getdata 3-11
 Wtbase_getname 3-12
 Wtbase_getparent 3-9
 Wtbase_getproperty 3-12
 Wtbase_gettype 3-11
 Wtbase_ismchild 3-10
 Wtbase_new 3-8
 Wtbase_next 3-8
 Wtbase_nfind 3-13
 Wtbase_nfindproperty 3-13
 Wtbase_numchildren 3-9
 Wtbase_numparents 3-9
 Wtbase_numproperties 3-12
 Wtbase_print 3-11
 Wtbase_removeallhandlers 3-26
 Wtbase_removeparent 3-9
 Wtbase_setdata 3-11
 Wtbase_setname 3-12
 Wtnode properties 3-3
 Wtpath properties 3-6
 WTproperty_addhandler 3-25
 WTproperty_delete 3-15
 WTproperty_exists 3-16
 WTproperty_get 3-20
 WTproperty_getasString 3-22
 WTproperty_getdata 3-16
 WTproperty_getdatatype 3-16
 WTproperty_gethandler 3-26
 WTproperty_getsizeofdata 3-17
 WTproperty_new 3-15
 WTproperty_numhandlers 3-25
 WTproperty_removeallhandlers 3-26
 WTproperty_removehandler 3-25
 WTproperty_set 3-17
 WTproperty_setat 3-19
 WTproperty_setdata 3-16
 WTsensor properties 3-5
 WTime_getcurrent 3-27
 WTime_getcurrentlocal 3-27
 WTime_getcurrentmsec 3-28
 WTime_getcurrentmseclocal 3-28
 WTime_getcurrentsec 3-27
 WTime_getcurrentseclocal 3-27
 WTime_getdouble 3-28
 WTime_getmsec 3-28
 WTime_getsec 3-28
 WTime_update 3-27
 WTuniverse_getbases 3-8
 WTuniverse_processevents 3-26
 WTvalue_tostring 3-23
 WTviewpoint properties 3-4
 WTwindow properties 3-4
- Objects
 associating a task with A-21
 axis G-20
 collision detection example A-25
 editing (example of) A-13
 following a light A-16
 getting midpoint of A-25
 getting the rendered position of A-25
 keeping perpendicular to viewpoint A-33
 moving example A-25
 naming conventions 1-4
- Octahedron
 creating 6-17
- Open functions
 customizing E-5
 openfn 13-9
- Open GL Callback Node 4-67
- OpenFlight file
 subfaces 6-7
- OpenGL
 embedding drawing routines 19-1
 material properties specification 8-19

Optimization
 geometry 6-39
 LOD example of scene optimization .. A-29
Organizational grouping nodes 4-6
Orientation records
 overview 25-1
Out of memory
 error message D-3
Output devices
 manufacturers J-2
Over/under mode, stereo viewing 2-35
Overlapping polygons
 image flashing 6-12

P

Packets
 see Networking
Parallax
 definition 16-19
 value 16-20, 17-12
Parameters
 setting from a file 2-28
Parent coordinate system
 defined M-9
Parent node 4-8
Path elements
 current 14-17
 management 14-24
 WTpathelement_copy 14-25
 WTpathelement_delete 14-24
 WTpathelement_getorientation 14-26
 WTpathelement_getpath 14-26
 WTpathelement_getposition 14-25
 WTpathelement_new 14-24
 WTpathelement_next 14-26
 WTpathelement_remove 14-25
 WTpathelement_setorientation 14-26
 WTpathelement_setposition 14-25
Path file format 14-12
Path nodes
 see Paths
Paths
 also see Path elements
 constant (maximum filename length) ..C-22
 constants C-9
 constraints 14-4
 construction 14-2

 copying a sequence of path nodes 14-5
 creating a sequence of path nodes 14-15
 current element 14-17
 current node 14-17
 current node pointer 14-16
 diagram 14-1
 direction constants 14-4
 editing 14-27
 element list 14-10
 element management 14-24
 example of copying 14-5
 interpolating 14-6
 interpolation 14-1
 interpolation methods diagram 14-7
 list of elements 14-10
 loading and saving 14-11
 management 14-8
 methods of interpolating 14-6
 overview 14-1
 play mode constants 14-4, 14-21
 playback 14-13
 recording 14-13, 14-15
 sample rate 14-23
 speed 14-4, 14-22
 toggling visibility 14-8
 universe list 2-13
 universe list of paths 14-5, 14-8, 14-10
 user-specifiable data 14-29
 uses for paths 14-2
 viewpoint paths 14-2
 visibility 14-4
 WTpath_appendelement 14-27
 WTpath_copy 14-5
 WTpath_delete 14-5
 WTpath_getconstraints 14-21
 WTpath_getcurrentelement 14-18
 WTpath_getdata 14-30
 WTpath_getdirection 14-20
 WTpath_getelements 14-10
 WTpath_getmarker 14-10
 WTpath_getmode 14-22
 WTpath_getplayspeed 14-23
 WTpath_getsamples 14-23
 WTpath_getvisibility 14-9
 WTpath_insertelement 14-27
 WTpath_interpolate 14-6
 WTpath_isplaying 14-16
 WTpath_isrecording 14-16
 WTpath_load 14-11

- WTpath_new 14-4
- WTpath_next 14-10
- WTpath_numelements 14-10
- WTpath_play 14-14
- WTpath_play1 14-14
- WTpath_record 14-14
- WTpath_record1 14-15
- WTpath_rewind 14-16
- WTpath_save 14-12
- WTpath_seek 14-18
- WTpath_setconstraints 14-20
- WTpath_setcurrentelement 14-17
- WTpath_setdata 14-29
- WTpath_setdirection 14-19
- WTpath_setmarker
 - description 14-9
- WTpath_setmode 14-21
- WTpath_setplayspeed 14-22
- WTpath_setrecordlink 14-15
- WTpath_setsamples 14-23
- WTpath_setvisibility 14-8
- WTpath_showcurrentelement 14-17
- WTpath_stop 14-16
- Performance
 - coordinate storage 6-13
 - geometry optimization 6-39
 - how to test for A-38
 - impact of lighting 12-4
 - modeling considerations 6-2
 - polygon complexity 6-10
 - radiosity preprocessor 6-10
 - system timer functions 2-22
 - texture filters 10-26
 - tip 6-45
- Perpendicular
 - usage example A-33
- Persistent Properties 21-3
- Persistent sharegroups
 - description 21-14
- PI constants C-6
- Picking polygons
 - overview 4-91, 17-20
 - usage example A-6
 - WTscreen_pickpoly 4-91
 - WTwindow_pickpoly 17-20
- Pinch Glove
 - defined constants 13-62
 - list of functions 13-7
 - manufacturer J-1
 - overview
 - raw data 13-60
 - scaling records 13-61
 - update function 13-61
 - WTpinch_update 13-61
- Pixel distortion
 - correcting the viewpoint aspect ratio . 16-18
- Play mode 14-21
 - constants C-10
 - path's playback speed 14-22
- Playback
 - see Play mode
- Point light 4-11
- Point light node
 - definition 12-2
- Pointer
 - getting a pointer to a nodeA-20
 - getting pointer to a WTK displayA-38
 - naming conventions 1-4
- Polhemus
 - FASTRAK, see FASTRAK
 - InsideTRAK, see InsideTRAK
 - ISOTRAK II, see ISOTRAK II
 - ISOTRAK, see ISOTRAK
 - Stylus, see Stylus
- Polygons
 - adding a vertex 7-10
 - also see Picking polygons
 - attributes 7-2
 - back face rejection 7-4
 - back faces 6-11, 7-4
 - back faces defined 7-4
 - coplanar 6-12
 - creation 7-10
 - default color 6-23
 - default ID 7-6
 - definition of front face 6-10
 - deleting 7-12
 - dynamic creation 7-10
 - front faces 7-4
 - generating vertex normals 6-9
 - get ID 7-7
 - ID numbers 7-6
 - intersection testing 7-13
 - non-coplanar (warning message)D-5
 - normals 6-10, 7-4, 7-5
 - obtaining pointers 7-6
 - optimization 6-39
 - overview 7-1

- picking 4-91
- picking frontmost polygon A-6
- pointers 7-1
- set ID 7-6
- texture information 10-31
- vertex access 7-8
- vertex order 6-10
- warning message D-5, D-13
- WTgeometry_beginpoly 6-23
- WTgeometry_setuv 10-32
- Wtpoly_addvertex 7-10
- Wtpoly_addvertexptr 7-10
- Wtpoly_close 7-12
- Wtpoly_delete 7-12
- Wtpoly_deletetexture 10-23
- Wtpoly_getbothsides 7-4
- Wtpoly_getcg 7-5
- Wtpoly_getgeometry 7-7
- Wtpoly_getid 7-7
- Wtpoly_getmatid 7-3
- Wtpoly_getnormal 7-4
- Wtpoly_getrgb 7-2
- Wtpoly_gettextureinfo 10-31
- Wtpoly_gettexturestyle 10-24
- Wtpoly_getuv 10-32
- Wtpoly_getvertex 7-8
- Wtpoly_intersectnode 4-86
- Wtpoly_intersectpolygon 4-85
- Wtpoly_mirrortexture 10-29
- Wtpoly_next 7-8
- Wtpoly_numvertices 7-9
- Wtpoly_rayintersect 4-88
- Wtpoly_rotatetexture 10-27
- Wtpoly_setbothsides 7-4
- Wtpoly_setid 7-6
- Wtpoly_setmatid 7-3
- Wtpoly_setrgb 7-2
- Wtpoly_settexture 10-11
- Wtpoly_settexturestyle 10-23
- Wtpoly_settextureuv 10-13
- Wtpoly_setuv 10-32
- Wtpoly_stretchtexture 10-30
- Wtpolyintersectbbox 4-86
- Wtpolytranslatetexture 10-29
- Wtpoyscaletexture 10-28
- POLYLINE entities
 - warning message D-5
- Portability
 - overview 24-1
 - reading directories 24-4
 - reading keyboard 24-1
 - WTdirectory_close 24-5
 - WTdirectory_getentry 24-4
 - WTdirectory_open 24-4
- Portals
 - handling in R6/7 A-22
 - using WTviewpoint_intersectpoly 4-90
- Prebuild function 6-39
- Precision Navigation Wayfinder-VR
 - see Wayfinder-VR
- Precision of coordinates
 - see Scaling 6-13
- Predecessor node 4-8
- Printing
 - math functions 25-1
 - scene graph 4-76
- Prism
 - constructing 6-15
- Private application data
 - see User-defined data field
- Private networks
 - see Networking
- Pro/Engineer file 6-3
- Procedural grouping nodes 4-6
- Programming with C++ wrappers 26-1
- Projection type constants C-10
- Properties
 - compatibility with World Up 21-38
 - data types 3-14
 - overview 3-14
 - sharing 21-2
 - WTnode 3-3
 - WTpath 3-6
 - WTproperty_delete 3-15
 - WTproperty_exists 3-16
 - WTproperty_get 3-20
 - WTproperty_getasstring 3-22
 - WTproperty_getd 3-21
 - WTproperty_getdata 3-16
 - WTproperty_getdatatype 3-16
 - WTproperty_getf 3-21
 - WTproperty_geti 3-21
 - WTproperty_getp 3-22
 - WTproperty_getp2 3-21
 - WTproperty_getp3 3-22
 - WTproperty_getq 3-22
 - WTproperty_gets 3-22
 - WTproperty_getsizeofdata 3-17

- WTproperty_getui 3-21
 - WTproperty_new 3-15
 - WTproperty_set 3-17
 - WTproperty_setat 3-19
 - WTproperty_setd 3-18
 - WTproperty_setdata 3-16
 - WTproperty_self 3-18
 - WTproperty_seti 3-18
 - WTproperty_setp 3-19
 - WTproperty_setp2 3-18
 - WTproperty_setp3 3-19
 - WTproperty_setq 3-19
 - WTproperty_sets 3-19
 - WTproperty_setui 3-18
 - WTsensor 3-5
 - WTvalue_tostring 3-23
 - WTviewpoint 3-4
 - WTwindow 3-4
 - Public networks
 - see Networking
 - Pushbutton object
 - see User interface
 - Pyramid
 - constructing 6-16
- ## Q
- Quad-buffering 2-34
 - Quaternions
 - also see Conversion functions
 - and the right-hand rule 25-3
 - convert to 3D matrix 25-26
 - convert to 4D matrix 25-32
 - convert to direction vector 25-30
 - convert to direction vector and twist .. 25-30
 - convert to euler 25-29, 25-32
 - overview 25-12
 - used with rotation sensor records 13-14
 - warning message D-11
 - Quick reference guide 1-11
 - Quick-reject test 4-57
 - overview 4-56
 - separator nodes 4-42
- ## R
- Radiosity preprocessing 6-10, 12-4
 - Raw sensor data
 - getting 13-15
 - setting 13-26
 - Ray casting
 - overview 17-20
 - WTwindow_getray 17-21
 - WTwindow_pickpoly 17-20
 - WTwindow_projectpoint 17-21
 - Realism
 - textures 10-2
 - vertex colors 6-9
 - Rear-view mirror
 - example 10-20
 - Recording path information
 - sample rate 14-23
 - Rectangles
 - creating 6-17
 - Recursive
 - scene graph example A-19
 - Red Baron
 - see 3D Mouse
 - Reference frame
 - and rotating a viewpoint 16-12
 - diagram 13-19, 16-10
 - for geometry motion 13-19, 16-10
 - sensors 13-16
 - viewpoint functions 16-14
 - Reflection Mapping 6-41
 - Registered interest
 - description 21-13
 - WTsharegroup_registerest 21-20
 - Relative sensor records 13-25
 - Remote networks
 - see Networking
 - RENDER file format 6-3
 - Rendering
 - constants C-10
 - geometry options 6-33
 - global parameters 2-18
 - universe options 2-18
 - window usage example A-35
 - Replaced Features
 - overview G-8
 - Replaced functions G-9
 - Resource files
 - overview 2-28
 - setting image paths 2-33
 - setting model paths 2-33
 - Restricting viewpoint motion 4-53
 - RGB

- in WTK V2.1 A-11
- setting via material table A-11
- Right-hand rule
 - defined 25-2
 - diagram 25-3
- Robot arm
 - diagram 5-9
 - example 5-9
- Root nodes 4-7, 4-8
 - building a scene graph 4-29
 - description in table 4-13
- Rotating an object
 - usage example A-21
- Rotating sensor input
 - example 13-17, 13-20
 - geometry motion reference frames 13-19
 - overview 13-16
- Rotation operators
 - column vector 25-3
 - defined 25-3
 - row vector 25-3
- Rotation records
 - sensors scale factor 13-12
- Roundoff
 - see Scaling
- RTS
 - signal E-5

S

- Sample rate
 - path 14-23
 - path value 14-4
- Scale object
 - see User interface
- Scaling
 - 3D Mouse 13-75
 - 5DT Glove 13-64
 - Bird 13-41
 - BOOM devices 13-56
 - CrystalEyesVR 13-109
 - CyberMaxx2 HMD 13-120
 - FASTRAK 13-94
 - Formula T2 13-112
 - gameport joystick 13-69
 - Geometry Ball Jr. 13-54
 - i-glasses! 13-122
 - InsideTRAK 13-91

- ISOTRAK 13-86
- ISOTRAK II 13-89
- Logitech Head Tracker 13-80
- model problems 6-13
- overview 6-13
- Pinch Glove 13-61
- scaling factor with geometries 6-13
- sensor records E-3
- Serial Joystick 13-115
- Space Control Mouse 13-83
- Spaceball 13-101
- Spaceball SpaceController 13-105
- Wayfinder-VR 13-97
- Scene graphs
 - accessing a node A-20
 - advantages 4-4, 4-14
 - ancestor node 4-7
 - anchor nodes 4-28
 - assembly 4-74
 - backward compatibility G-2
 - building 4-29
 - child node 4-7
 - composite car diagram 4-32
 - concepts in detail 4-5
 - constructor functions 4-39
 - content nodes (table) 4-11
 - coordinate frames 4-32
 - culling 4-4
 - cyclic tree diagram 4-30
 - depth first traversal 4-84
 - descendant node 4-7
 - description 4-3
 - diagram of a simple scene graph 4-4
 - directed light 4-11
 - elements 4-2
 - encapsulation diagram 4-23
 - example 4-18
 - grouping nodes (table) 4-13
 - inline nodes 4-28
 - instancing 4-37
 - introduction 1-2, 4-2
 - leaf node 4-7
 - light encapsulation 4-22
 - loading a file 4-45
 - LOD diagram 4-27
 - multiple 4-7
 - nodes 4-5
 - order of child nodes 4-15
 - overview 4-2

- parent node 4-8
- point light 4-11
- predecessor node 4-8
- printing 4-8, 4-76
- rendering 4-9
- root node 4-7, 4-8
- saving 4-48
- scenes 4-2
- schematic diagram 4-7
- separator node diagram 4-21
- separator nodes 4-21
- sibling node 4-8
- spot light 4-11
- state 4-10, 4-17
- state accumulation diagram 4-17
- state propagation diagram 4-19
- state separation 4-21
- static and dynamic geometries 4-2
- structure inquiry 4-76
- sub-tree 4-8
- switch diagram 4-25
- terminology 4-7
- transform separator 4-24
- transform separator diagram 4-24
- traversal 4-9, 4-78
- traversal diagram 4-9
- traversal example A-19
- traversal order 4-8
- tree 4-8
- using frames of reference 4-32
- viewing 4-8
- WTviewport_intersectpoly 4-89
- Screen
 - blanking interval 2-21
 - pixel units 16-21
 - vertical blanking interval 2-20
 - WTscreen_getyblank 2-21
 - WTscreen_load 10-33
 - WTscreen_pickpoly 4-91
 - WTscreen_setyblank 2-20
- Scrolled list object
 - see User interface
- Sensitivity value
 - default 13-11
 - sensors 13-12
- Sensors
 - 3D Mouse 13-73
 - 5DT Glove 13-8, 13-63
 - absolute records 13-25
 - absolute sensor 13-13
 - absolute sensor records 13-2
 - attaching to node paths 4-92
 - attaching to transform nodes 4-92
 - Bird 13-39
 - BOOM 13-55
 - close function (custom drivers) E-1
 - constants 13-15, C-11
 - constraining input 13-21
 - construction and destruction 13-5
 - CrystalEyesVR 13-108
 - current rotation record 13-14
 - custom drivers 13-24, E-1
 - CyberMaxx2 HMD
 - example of detecting button events A-10
 - example of how to orient differently A-36
 - FASTRAK 13-92
 - Flock of Birds 13-51
 - Formula T2 13-111
 - frame-rate 13-5
 - Geometry Ball Jr. 13-53
 - Head Tracker 13-77
 - i-glasses! 13-121
 - InsideTRAK 13-90
 - ISOTRAK 13-85
 - ISOTRAK II 13-88
 - lag 13-5
 - manufacturers J-1
 - mouse 13-26
 - multi-sensor devices (custom drivers) ... E-1
 - open function (custom drivers) E-1
 - overview 13-2
 - Pinch Glove 13-59
 - raw data 13-26
 - reference frame 13-16
 - relative records 13-25
 - rotating sensor input 13-16
 - sensor data, user-specifiable 13-23
 - sensor state, accessing 13-11
 - Serial Joystick 13-113
 - setting absolute sensor records 13-25
 - Space Control Mouse (Magellan) 13-81
 - Spaceball 13-100
 - Spaceball SpaceController 13-104
 - Stylus 13-93
 - universe list 2-13
 - update function 13-10
 - update function (custom drivers) E-1
 - update functions 13-7

- Wayfinder-VR 13-96
- WTsensor_delete 13-10
- WTsensor_getangularrate 13-13
- WTsensor_getconstraints 13-22
- WTsensor_getdata 13-24
- WTsensor_getlastrecord 13-25
- WTsensor_getmiscdata 13-15
- WTsensor_getrawdata 13-15
- WTsensor_getrotation 13-14
- WTsensor_getsensitivity 13-12
- WTsensor_getserial 13-16
- WTsensor_gettranslation 13-13
- WTsensor_getunit 13-16
- WTsensor_new 13-7
- WTsensor_next 13-10
- WTsensor_relativizerecord 13-24
- WTsensor_rotate 13-20
- WTsensor_setangularrate 13-12
- WTsensor_setconstraints 13-21
- WTsensor_setdata 13-23
- WTsensor_setlastrecord 13-25
- WTsensor_setmiscdata 13-25
- WTsensor_setrawdata 13-26
- WTsensor_setrecord 13-24
- WTsensor_setsensitivity 13-11
- Separator nodes
 - overview 4-56
 - WTsepnode_getcullmode 4-57
 - WTsepnode_setcullmode 4-57
- Serial Joystick
 - constants C-17
 - defined constants 13-117
 - drift 13-118
 - list of functions 13-9
 - manufacturer J-2
 - overview 13-113
 - range 13-118
 - raw data 13-114
 - reinitializing 13-119
 - scaling records 13-115
 - update functions 13-115
 - WTjoyserial_fly 13-116
 - WTjoyserial_getdrift 13-119
 - WTjoyserial_getrange 13-118
 - WTjoyserial_rawupdate 13-117
 - WTjoyserial_readcalibrationfile 13-119
 - WTjoyserial_setdrift 13-118
 - WTjoyserial_walk 13-116
 - WTjoyserial_walk2 13-116
- Serial ports
 - constants C-17
 - construction and destruction 23-1
 - error message D-3, D-4
 - example using constant 13-6
 - getting an object 13-16
 - pointer to object 13-9
 - reading and writing characters 23-3
 - warning message D-5
 - WTserial_delete 23-2
 - WTserial_new 23-1
 - WTserial_ntoread 23-4
 - WTserial_read 23-3
 - WTserial_write 23-4
- SERIAL1 constant C-17
- SERIAL2 constant C-17
- Server Manager 21-1
- Setting event order
 - example 2-10
- Shading
 - Gouraud 6-8
- Shared properties
 - functions 21-5
 - locked 21-3
 - overview 21-2
 - persistent 21-3
 - time-sensitive 21-4
 - update frequencies 21-3
 - WTbase_unshare 21-11
 - WTproperty_getsharegroup 21-8
 - WTproperty_gettimesensitive 21-9
 - WTproperty_getupdatefreq 21-8
 - WTproperty_islocked 21-10
 - WTproperty_islockedbyme 21-11
 - WTproperty_issshared 21-7
 - WTproperty_lock 21-10
 - WTproperty_numshares 21-7
 - WTproperty_sendupdate 21-9
 - WTproperty_settimesensitive 21-9
 - WTproperty_setupupdatefreq 21-8
 - WTproperty_share 21-5
 - WTproperty_unlock 21-10
 - WTproperty_unshare 21-7
- Sharegroups
 - functions 21-15
 - locked 21-12
 - overview 21-11
 - persistent 21-14
 - registered interest 21-13

- WTsharegroup_delete 21-16
- WTsharegroup_enumerate 21-36
- WTsharegroup_findchildbyname 21-21
- WTsharegroup_getchild 21-20
- WTsharegroup_getconnection 21-18
- WTsharegroup_getdata 21-18
- WTsharegroup_getname 21-18
- WTsharegroup_getparent 21-20
- WTsharegroup_getproperty 21-21
- WTsharegroup_islocked 21-19
- WTsharegroup_islockedbyname 21-19
- WTsharegroup_isshared 21-17
- WTsharegroup_lock 21-19
- WTsharegroup_new 21-15
- WTsharegroup_numchildren 21-20
- WTsharegroup_numproperties 21-21
- WTsharegroup_print 21-18
- WTsharegroup_registerinterest 21-20
- WTsharegroup_setdata 21-17
- WTsharegroup_share 21-17
- WTsharegroup_unlock 21-19
- Sibling node 4-8
- SIG-WTK users' group 1-11, K-1, L-2
- Simple objects
 - creating 6-14
 - overview 6-1
- Simulation
 - development process 1-13
 - frames per second 2-23
 - management 2-5, 2-11
 - manager 2-1
 - running speed 2-22
- Simulation loop
 - creating path elements 14-1
 - definition 2-5
 - diagram 2-6, 11-2
 - preparing for 2-6
 - setting order of events 2-9
 - starting 2-7
 - starting for one frame only 2-7
 - stopping 2-8
- Simulation Server 21-2
- SLP file format 6-3
- Sound constants C-18
- Sound support library
 - CRE device parameters 20-7
 - device constants C-19
 - device spatializing functions 20-9
 - overview 20-1
 - sound device options 20-6, 20-7
 - sound parameter options 20-13, 20-14
 - sound spatializing functions 20-17
 - supported devices 20-1
 - WTK feature 1-5
 - WTsound_delete 20-10
 - WTsound_getdata 20-16
 - WTsound_getdonefn 20-17
 - WTsound_getname 20-15
 - WTsound_getnodepath 20-18
 - WTsound_getparam 20-15
 - WTsound_getposition 20-17
 - WTsound_isplaying 20-12
 - WTsound_load 20-10
 - WTsound_next 20-11
 - WTsound_play 20-11
 - WTsound_setdata 20-15
 - WTsound_setdonefn 20-16
 - WTsound_setnodepath 20-18
 - WTsound_setparam 20-12
 - WTsound_setposition 20-17
 - WTsound_stop 20-10
 - WTsounddevice_close 20-3
 - WTsounddevice_getdata 20-9
 - WTsounddevice_getlistener 20-9
 - WTsounddevice_getparams 20-8
 - WTsounddevice_getsounds 20-4
 - WTsounddevice_name2sound 20-5
 - WTsounddevice_numplayable 20-4
 - WTsounddevice_open 20-3
 - WTsounddevice_setdata 20-8
 - WTsounddevice_setlistener 20-9
 - WTsounddevice_setparam 20-5
- Space Control Mouse
 - defined constants 13-84
 - dominant mode 13-84
 - list of functions 13-8
 - manufacturer J-1
 - overview 13-81
 - pick button 13-84
 - raw data 13-82
 - reference frame diagram 13-82
 - scaling and constraints 13-83
 - scaling records 13-83
 - special modes 13-84
 - update function 13-83
 - WTspacecontrol_rawupdate 13-83
 - WTspacecontrol_update 13-83
- Spaceball

- avoiding unwanted motion 13-102
 - defined constants 13-103
 - dominant mode 13-102
 - error message D-4
 - list of functions 13-8
 - manufacturer J-2
 - Model 3003 13-104
 - overview 13-100
 - raw data 13-101
 - redefining the center 13-103
 - reference frame diagram 13-100
 - scaling records 13-101
 - update functions 13-102
 - warning message D-6, D-14
 - WTprecision_rawupdate 13-102
 - WTspaceball_dominant 13-102
 - WTspaceball_rezero 13-103
 - WTspaceball_update 13-102
 - Spaceball SpaceController
 - avoiding unwanted motion 13-106
 - defined constants 13-107
 - dominant mode 13-106
 - list of functions 13-8
 - manufacturer J-2
 - overview 13-104
 - raw data 13-105
 - redefining the center 13-107
 - scaling records 13-105
 - update functions 13-106
 - WTspaceballSC_dominant 13-106
 - WTspaceballSC_rezero 13-107
 - WTspaceballSC_setwindow 13-107
 - WTspaceballSC_update 13-106
 - Spaceship exhaust A-14
 - Spacotec IMC Spaceball
 - see Spaceball
 - Spacotec IMC Spaceball SpaceController
 - see Spaceball SpaceController
 - Spatial culling
 - see Culling
 - Special Effects A-13
 - Special effects
 - see Fog
 - Spheres
 - creating 6-16
 - Spherical linear interpolation
 - math function 25-18
 - Spot light 4-11
 - Spot light node
 - definition 12-2
 - State information
 - separator nodes 4-13, 5-2, 5-4
 - Statistics
 - universe 2-22
 - Stereo viewing
 - configurations 16-23
 - diagram 16-3
 - modes 2-34
 - setting convergence 16-21
 - StereoGraphics CrystalEyesVR
 - see CrystalEyesVR
 - Streaming-Mode Flock of Birds Driver . 13-44
 - Stylus
 - constants C-15
 - overview 13-93
 - Subfaces
 - with MultiGen files 6-7
 - Surfaces
 - coplanar 6-12
 - non-planar 6-14
 - Switch nodes
 - animation 4-26
 - animation example using switch nodes A-26
 - defined in table 4-13
 - overview 4-25, 4-57
 - portalling 4-26
 - using 4-26
 - WTswitchnode_getwhichchild 4-58
 - WTswitchnode_setwhichchild 4-57
 - switchnode constants C-8
 - Synchronous connections 21-22
 - WTconnection_issynchronous 21-31
 - WTconnection_setsynchronous 21-30
 - System configuration
 - overview 1-12
- ## T
- Tag field
 - see Networking, multicast
 - Tasks
 - behavior of objects 11-1
 - diagram 11-2
 - overview 11-1
 - task object 11-1
 - usage example A-21

- with data structures 11-1
- WTtask_add 11-4
- WTtask_delete 11-4
- WTtask_getfunction 11-5
- WTtask_getpriority 11-5
- WTtask_new
 - description 11-2
- WTtask_remove 11-4
- WTtask_setpriority 11-5
- WTuniverse_gettaskbypointer 11-6
- TCP 22-3
- Technical support
 - Non-U.S. L-2
 - U.S. L-1
- Terrain following
 - usage example A-31
- Terrains
 - creating in this release G-8
- Test
 - machine performance A-38
 - quick reject 4-42
 - quick-reject 4-56, 4-57
- testing for intersections (example of) A-24
- Tests
 - viewpoint intersection 16-26
- Tetrahedron
 - creating 6-16
- Text input object
 - see User interface
- Text object
 - see User interface
- Text string
 - see 3D fonts
- Text-field object
 - see User interface
- Texture filtering
 - constants C-20
- Textures 10-1
 - 3DS uv values 6-2
 - animating 10-18
 - application 10-5
 - applying 10-4
 - assigning 10-22
 - bitmap image 10-2
 - changing properties 10-23
 - deleting 10-23
 - filtering 10-24
 - illustration 10-2
 - image formats (converting between) I-2
 - image utilities I-2
 - manipulating uv values 10-32
 - manipulation 10-22
 - memory use 10-18
 - methods for applying 10-4
 - mipmapping overview 10-24
 - mirrored illustration 10-30
 - on non-rectangular polygons (diagram) 10-6
 - on rectangular polygons (diagram) 10-5
 - overview 10-2
 - pattern and textures 10-2
 - procedure for applying 10-5
 - rotation 10-27
 - scaling 10-27
 - scaling illustration 10-28
 - setting default filter 10-25
 - shading 10-22
 - translation illustration 10-29
 - transparent 10-22
 - usage example A-12
- WTgeometry_deletetexture 10-23
- WTgeometry_setuv 10-32
- WTPoly_deletetexture 10-23
- WTPoly_gettextureinfo 10-31
- WTPoly_gettexturestyle 10-24
- WTPoly_getuv 10-32
- WTPoly_mirrortexture 10-29
- WTPoly_rotatetexture 10-27
- WTPoly_scaletexture 10-28
- WTPoly_settexture 10-11
- WTPoly_settexturestyle 10-23
- WTPoly_settextureuv 10-13
- WTPoly_setuv 10-32
- WTPoly_stretchtexture 10-30
- WTPoly_translatetexture 10-29
- WTtexture_getfilter 10-27
- WTtexture_getmemory 10-18
- WTtexture_setfilter 10-25
- ThrustMaster Formula T2 Steering Console
 - see Formula T2
- ThrustMaster Mark II Flight Control System ..
 - 13-113
- ThrustMaster Mark II Weapons Control
 - System 13-113
- ThrustMaster Serial Joystick
 - see Serial Joystick
- Time
 - overview 3-27
 - WTime_getcurrent 3-27

WTime_getcurrentlocal 3-27
 WTime_getcurrentmsec 3-28
 WTime_getcurrentmseclocal 3-28
 WTime_getcurrentsec 3-27
 WTime_getcurrentseclocal 3-27
 WTime_getdouble 3-28
 WTime_getmsec 3-28
 WTime_getsec 3-28
 WTime_update 3-27
 Time sensitivity
 overview 21-4
 WTproperty_gettimesensitive 21-9
 WTproperty_settimesensitive 21-9
 Time to Live value 22-8, 22-13
 Token Ring 22-2
 Tool bar object
 see User interface
 Transform nodes
 creating 4-42
 description 4-11
 overview 4-58
 WTnode_axisrotation 4-62
 WTnode_getorientation 4-60
 WTnode_getrotation 4-60
 WTnode_gettransform 4-58
 WTnode_gettranslation 4-59
 WTnode_rotatem3 4-61
 WTnode_rotatem4 4-62
 WTnode_rotateq 4-61
 WTnode_rotation 4-61
 WTnode_setorientation 4-60
 WTnode_setrotation 4-60
 WTnode_settransform 4-58
 WTnode_settranslation 4-59
 WTnode_translate 4-59
 WTxformnode_new 4-42
 Transform separator nodes
 creating 4-43
 defined in table 4-14
 WTxformsepnodes_new 4-43
 Transformation
 accumulated 4-84
 changed functions H-1
 composite 4-30
 Transition guide
 see Backward compatibility
 Transitioning
 from R6 to R7/8/9 H-1
 Transparent Textures 10-8

Transparent textures 10-22
 usage example A-12
 Traversal order
 see Scene graphs
 Triangular prisms
 constructing 6-15
 TRUE constant C-6
 Truncated cones
 creating 6-18
 Type definitions
 WTK naming conventions 1-4
 Type field
 see Networking, multicast

U

U constant C-6
 UDP
 see Networking
 Ultrasonic sensor
 CrystalEyesVR 13-108
 Logitech Head Tracker 13-77
 overview 13-2
 Universe A-24
 action function 2-11
 collision detection example A-25
 construction 2-1
 destruction 2-1
 list of paths 14-8, 14-10
 list of viewpoints 16-4
 list of windows 17-8
 overview 2-1
 performance and statistics function 2-22
 rendering and display 2-18
 rendering options 2-18
 stereoscopic viewing 2-34
 warning message D-10
 WTuniverse_avgframerate 2-24
 WTuniverse_delete 2-5
 WTuniverse_deletelink 2-17
 WTuniverse_findnodebyname 4-49
 WTuniverse_framecount 2-23
 WTuniverse_framerate 2-23
 WTuniverse_getbgrgb 2-21
 WTuniverse_getcurrscridx 2-14
 WTuniverse_getcurrwindow 2-14
 WTuniverse_getcurrwindx 2-14
 WTuniverse_geteventorder 2-10

- WTuniverse_getinitialview 2-16
- WTuniverse_getmotionlinks 2-17
- WTuniverse_getoption 2-27
- WTuniverse_getpaths 2-13
- WTuniverse_getrendering 2-20
- WTuniverse_getrootnodes 2-17
- WTuniverse_getsensors 2-13
- WTuniverse_getsubfaceoffset 2-22
- WTuniverse_gettaskbypointer 11-6
- WTuniverse_getviewpoints 2-15
- WTuniverse_getwindows 2-13
- WTuniverse_go 2-7
- WTuniverse_go1 2-7
- WTuniverse_new 2-2
- WTuniverse_ready 2-6
- WTuniverse_resetframecount 2-23
- WTuniverse_resettime 2-23
- WTuniverse_setactions 2-12
- WTuniverse_setbboxrgb 2-21
- WTuniverse_setbgrgb 2-21
- WTuniverse_seteventorder 2-9
- WTuniverse_setopt 2-24
- WTuniverse_setrendering 2-18
- WTuniverse_setsubfaceoffset 2-22
- WTuniverse_setviewpoint 2-15
- WTuniverse_stop 2-8
- WTuniverse_time 2-22
- UNIX
 - getting a pointer to a WTK display A-38
- Update frequencies
 - for shared properties 21-3, 21-8
- Update functions
 - called by simulation manager 13-9
 - changing 13-10
 - custom E-1
 - example 13-10
 - sensors 13-7
 - unsupported sensors 13-9
 - updatefn E-6
 - WTPolhemus_update 13-88
- Update rates
 - for connections 21-23, 21-31
- URL
 - function for anchor and inline nodes ... 4-62
- User interface
 - changed functions H-1
 - checkboxbuttons 18-16
 - constants C-20
 - example accessing scrolled list items 18-22
 - example GUI application 18-2
 - example of accessing a file 18-14
 - example of creating a scale object 18-20
 - example of creating a toolbar 18-10
 - example of menu system creation 18-26
 - example simulating WTuniverse_go1 18-38
 - example using variable argument list .. 18-7
 - file selection box object 18-13
 - form object 18-6
 - label object 18-17
 - loading images in form objects 18-17
 - menu bar button object 18-26
 - menu bar object 18-25
 - menu pop-up button object 18-25
 - overview 18-1
 - pushbutton object 18-18
 - scale object 18-19
 - scale object example 18-20
 - scrolled list object 18-21
 - text input object 18-15
 - text object 18-23
 - text-field object 18-24
 - tool bar object 18-28
 - variable arguments 18-7
 - WTinit_usewindow 17-6
 - WTui_check 18-35
 - WTui_delete 18-40
 - WTui_deleteitem 18-33
 - WTui_enable 18-34
 - WTui_getcallback 18-41
 - WTui_getid 18-37
 - WTui_getparent 18-40
 - WTui_getposition 18-36
 - WTui_getselecteditem 18-31
 - WTui_gettext 18-30
 - WTui_go 18-12
 - WTui_init 18-5
 - WTui_insertitem 18-33
 - WTui_isspace 18-35
 - WTui_isenabled 18-34
 - WTui_isswtkrunning 18-38
 - WTui_manage 18-11
 - WTui_setcallback 18-8
 - WTui_setposition 18-36
 - WTui_settext 18-29
 - WTui_settoolbarcallback 18-9
 - WTui_unmanage 18-40
 - WTui_wtkstart 18-38
 - WTui_wtkstop 18-38

- WTuicheckbutton_new 18-16
 - WTuifileselection_new 18-13
 - WTuilabel_new 18-17
 - WTuimenuubar_new 18-24
 - WTuimenuitem_new 18-26
 - WTuimenupopup_new 18-25
 - WTuimessagebox_new 18-15
 - WTuipushbutton_new 18-18
 - WTuiradiobox_new 18-18
 - WTuiscale_new 18-19
 - WTuiscrollledlist_new 18-21
 - WTuiscrollledtext_new 18-23
 - WTuitextfield_new 18-24
 - WTuitextinput_new 18-15
 - WTuitoolbar_new 18-28
 - WTuiwtkwindow_new 18-11
 - User-defined data field
 - material tables 8-13
 - nodes 4-51
 - sensors 13-23
 - viewpoints 16-25
 - User-defined draw function
 - loading an image 17-25
 - overview 19-1
 - Users' group (SIG-WTK) K-1, L-2
 - Utility functions 4-76
- V**
- V constant C-6
 - Vertex
 - access 7-8
 - add to geometry 6-22
 - adding to a polygon 7-10
 - colors 6-9
 - editing example A-13
 - index 7-10
 - normals 6-9
 - normals in NFF file F-4
 - warning message D-5
 - WTgeometry_newvertex 6-22
 - WTPoly_addvertex 7-10
 - WTPoly_addvertexptr 7-10
 - WTPoly_close 7-12
 - WTPoly_getvertex 7-8
 - WTPoly_numvertices 7-9
 - WTPoly_rayintersect 4-88
 - WTvertex_next 6-33
 - Vertical blanking interval 2-20
 - Vertical sync problem 2-20
 - VictorMaxx Technologies' CyberMaxx2 HMD
 - see CyberMaxx2 HMD
 - Video accelerator cards
 - manufacturers J-3
 - VideoScape 6-3
 - View matrix 17-23
 - Viewing angle 17-19
 - setting from a file 2-28
 - Viewpoints
 - accessing position and orientation 16-8
 - aspect ratio 16-18, 16-19
 - convergence 16-4
 - convergence distance diagram 16-22
 - coordinate transformations 16-24
 - creation 2-2
 - current 14-28
 - default orientation 16-3
 - default position 16-3
 - default sound position 20-17
 - delete 2-15, 16-5
 - getting convergence values 16-21
 - handling portals A-22
 - hither and yon clipping 17-17
 - information 2-16
 - intersection test 16-26
 - keeping an object perpendicular to A-33
 - management 16-3
 - monoscopic viewing diagram 16-2
 - overriding attachments 20-17
 - overview 16-1
 - paths 14-2
 - reference frame 16-24
 - reference frame diagram 16-10
 - sensor attachment 16-6
 - stereo modes 2-34
 - stereo viewing 16-19
 - stereo viewing diagram 16-3
 - testing for intersections A-24
 - universe list of viewpoints 16-4
 - user-specifiable data 16-25
 - viewing angle 17-19
 - WTviewpoint_addsensor 16-7
 - WTviewpoint_alignaxis 16-14
 - WTviewpoint_copy 16-5
 - WTviewpoint_delete 16-5
 - WTviewpoint_getaspect 16-19
 - WTviewpoint_getaxis 16-14

- WTviewpoint_getconvdistance 16-23
 - WTviewpoint_getconvergence 16-21
 - WTviewpoint_getdata 16-26
 - WTviewpoint_getdirection 16-13
 - WTviewpoint_getdirectionframe 16-18
 - WTviewpoint_getframe 16-15
 - WTviewpoint_getlastorientation 16-11
 - WTviewpoint_getlastposition 16-9
 - WTviewpoint_getorientation 16-11
 - WTviewpoint_getorientationframe 16-16
 - WTviewpoint_getparallax 16-20
 - WTviewpoint_getposition 16-8
 - WTviewpoint_getpositionframe 16-15
 - WTviewpoint_intersectpoly 4-89
 - WTviewpoint_local2world 16-24
 - WTviewpoint_move 16-12
 - WTviewpoint_moveframe 16-17
 - WTviewpoint_moveto 16-13
 - WTviewpoint_movetoframe 16-17
 - WTviewpoint_new 16-3
 - WTviewpoint_next 16-5
 - WTviewpoint_remoovesensor 16-8
 - WTviewpoint_rotate 16-12
 - WTviewpoint_rotateframe 16-16
 - WTviewpoint_setaspect 16-18
 - WTviewpoint_setconvdistance 16-22
 - WTviewpoint_setconvergence 16-21
 - WTviewpoint_setdata 16-25
 - WTviewpoint_setdirection 16-13
 - WTviewpoint_setdirectionframe 16-17
 - WTviewpoint_setorientation 16-11
 - WTviewpoint_setorientationframe 16-16
 - WTviewpoint_setparallax 16-19
 - WTviewpoint_setposition 16-8
 - WTviewpoint_setpositionframe 16-15
 - WTviewpoint_translate 16-9
 - WTviewpoint_translateframe 16-15
 - WTviewpoint_world2local 16-24
 - WTwindow_gethithervalue 17-18
 - yon clipping 17-19
 - Viewports 17-30
 - Virtual i-O i-glasses!
 see i-glasses!
 - Virtual reality
 hardware components 1-12
 - Virtual Research
 manufacturer J-2
 - VRML
 exporting a file 6-4
 - file format 6-3
 - limitations 6-4
 - loading file 4-46
 - loading in using an http URL 6-3
 - material properties 2-3
 - new viewpoints 8-15
 - overview for anchor nodes 4-28
 - overview for inline nodes 4-28
 - saving 4-41, 4-48
 - scene graph advantages 4-4
 - setting a URL path 4-62
 - support for 1-6
 - used to create in-line nodes 4-41
 - WTvrm1_seturl 4-62
- ## W
- W constant C-6
 - Warning messages D-5
 overview D-1
 - Wavefront
 file format 6-2
 - Wayfinder-VR
 example 13-96
 - list of functions 13-8
 - manufacturer J-1
 - overview 13-96
 - raw data 13-96
 - scaling records 13-97
 - special notes 13-98
 - update function 13-97
 - WTprecision_new macro 13-96
 - WTprecision_rawupdate 13-97
 - WTprecision_update 13-97
 - Windows
 angular field of view 17-19
 - configuration 2-3
 - constants 2-3, 17-2, C-21
 - creation and deletion 17-2
 - hither and yon clipping 17-17
 - loading an image file 17-25
 - monoscopic viewing diagram 17-6
 - overview 17-1
 - picking frontmost polygon A-6
 - rendering properties 17-22
 - setting position from file 2-28
 - setting size from file 2-28
 - size and placement 17-10

- stereo viewing diagram 16-3
- universe list of windows 17-8
- usage example A-35
- user-defined drawing functions 19-1
- viewpoints 17-11
- yon clipping 17-19
- World coordinate frame
 - example 4-33
- World coordinate system
 - defined M-17
 - example 4-33
- World coordinates
 - paths 14-1
- World Up compatible properties 21-38
- World Up Modeler I-1
 - file formats supported I-1
- World2World server product 21-1
 - also see Networking, client-server
- WorldToolkit
 - additional features 1-5
 - documentation available 1-7
 - Hardware Guides 1-10
 - introduction 1-1
 - naming conventions 1-4
 - quick reference guide 1-11
 - sample animation 1-13
 - Special Interest Group 1-11
 - transitioning from R6 to R7/8/9 H-1
 - users' group L-2
- Writing a sensor driver E-1
- WTanchornode_getlocation 4-63
- WTanchornode_new 4-40
- WTanchornode_setlocation 4-63
- WTanimation functions G-9
- WTbaron_update 13-76
- WTbase objects
 - functions for all supported types 3-10
 - functions for WTbase objects 3-7
 - in World2World-compliant simulations 21-5
 - overview 3-7
- WTbase_addparent 3-8
- WTbase_delete 3-11
- WTbase_deleteproperties 3-13
- WTbase_find 3-13
- WTbase_findchild 3-10
- WTbase_getchild 3-10
- WTbase_getdata 3-11
- WTbase_getname 3-12
- WTbase_getparent 3-9
- WTbase_getproperty 3-12
- WTbase_gettype 3-11
- WTbase_iscchild 3-10
- WTbase_new 3-8
- WTbase_next 3-8
- WTbase_nfind 3-13
- WTbase_nfindproperty 3-13
- WTbase_numchildren 3-9
- WTbase_numparents 3-9
- WTbase_numproperties 3-12
- WTbase_print 3-11
- WTbase_removeallhandlers 3-26
- WTbase_removeparent 3-9
- WTbase_setdata 3-11
- WTbase_setname 3-12
- WTbase_unshare 21-11
- WTBIRD constants 13-43
- WTbird_autohemisphere 13-44
- WTbird_getabsoluterecord 13-41
- WTbird_gethemisphere 13-43
- WTbird_new macro 13-51
 - Bird 13-39
 - Flock of Birds 13-51
- WTbird_sethemisphere 13-43
- WTbird_setsync 13-42
- WTbird_update 13-42
- WTBIRDDELAY environment variable ... B-5
- WTBOOM constants
 - for BOOM with buttons 13-57
 - for BOOM with joystick 13-58
- WTboom_new macro
 - example 13-55
- WTboom_update 13-57
- WTcalloc 24-9
- WTconnection_addcallback 21-31
- WTconnection_connect 21-28
- WTconnection_delete 21-27
- WTconnection_deleteallenumtrees 21-36
- WTconnection_deleteenumtreebyid 21-36
- WTconnection_disconnect 21-28
- WTconnection_getcallback 21-32
- WTconnection_getclockdiff 21-30
- WTconnection_getdata 21-27
- WTconnection_getenumtree 21-37
- WTconnection_getenumtreebyid 21-37
- WTconnection_getenumtreeid 21-38
- WTconnection_getlatency 21-30
- WTconnection_getmyid 21-28

-
- WTconnection_getmyname 21-29
 - WTconnection_getroot 21-32
 - WTconnection_getstatus 21-29
 - WTconnection_getupdaterate 21-31
 - WTconnection_getuserid 21-33
 - WTconnection_getuseridbyname 21-33
 - WTconnection_getusername 21-33
 - WTconnection_getusernamebyid 21-33
 - WTconnection_issynchronous 21-31
 - WTconnection_new 21-26
 - WTconnection_next 21-28
 - WTconnection_numcallbacks 21-32
 - WTconnection_numenumtrees 21-37
 - WTconnection_numusers 21-32
 - WTconnection_print 21-29
 - WTconnection_removecallback 21-32
 - WTconnection_setdata 21-27
 - WTconnection_setsynchronous 21-30
 - WTconnection_setupdaterate 21-31
 - WTconnection_synch 21-30
 - WTconnection_update 21-29
 - WTconnevent 21-24
 - WTCONSTRAIN constants
 - constraining a path 14-20
 - constraining a sensor 13-21
 - writing sensor drivers E-3
 - WTcrystaleyesVR_new macro
 - example 13-109
 - WTcrystaleyesVR_update 13-110
 - WTcybermaxx2_new macro
 - example 13-119
 - WTcybermaxx2_rawupdate 13-121
 - WTcybermaxx2_update 13-120
 - WTcybglove_deletehandmodel 13-129
 - WTcybglove_getanglearray 13-132
 - WTcybglove_getfingers 13-131
 - WTcybglove_getforearm 13-130
 - WTcybglove_getpalm 13-131
 - WTcybglove_new 13-124
 - WTcybglove_setvisibility 13-130
 - WTcybglove_showcalibrationpanel ... 13-126
 - WTcybglove_usehandmodel 13-127
 - WDir_2q 25-30
 - WDirandtwist_2q 25-31
 - WTDIRECTION constants
 - getting a path direction 14-20
 - setting a path direction 14-19
 - Wtdirectory_close 24-5
 - Wtdirectory_getentry 24-4
 - Wtdirectory_open 24-4
 - WTDISPLAY constants 2-2, C-2
 - example of WTDISPLAY_DEFAULT 16-6
 - example of WTDISPLAY_STEREO .. 17-8
 - WTDRAW2D constants C-2
 - WTercbird_init macro 13-52
 - Wterror 24-8
 - WTeuler_2m3 25-27
 - WTeuler_2q
 - description 25-27
 - example 13-20, E-14
 - WTEVENT constants 2-9
 - WTEYE constants C-3
 - WTfastrak_afilter 13-95
 - WTfastrak_afilteroff 13-95
 - WTfastrak_pfilter 13-95
 - WTfastrak_pfilteroff 13-95
 - WTFASTRAK_STYLUSBUTTON_DOWN
 - constant C-15
 - example 13-93
 - WTfastrak_update 13-94
 - WTFILE_DELIM constant C-21
 - WTFILE_PATHDELIM constant C-21
 - WTFILETYPE constants
 - saving a geometry 6-26
 - saving a node 4-48
 - WTFILTER constants 10-26
 - Wtflock_close 13-46
 - Wtflock_deviceopen 13-45
 - Wtflock_getabsmat 13-50
 - Wtflock_getabsoluterecord 13-50
 - Wtflock_getabspos 13-50
 - Wtflock_getcrtsyncdata 13-47
 - Wtflock_getdefaulthemisphere 13-46
 - Wtflock_gethemisphere 13-47
 - Wtflock_getlastmat 13-49
 - Wtflock_getlastpos 13-49
 - Wtflock_getorgmat 13-49
 - Wtflock_new 13-45, 13-51
 - Wtflock_open 13-45
 - Wtflock_resetorigin 13-48
 - Wtflock_setcrtsync 13-48
 - Wtflock_setdefaulthemisphere 13-46
 - Wtflock_sethemisphere 13-47
 - Wtflock_update 13-48
 - WTFOG constants 4-66
 - Wtfognode_getcolor 4-65
 - Wtfognode_getlinearstart 4-67
 - Wtfognode_getmode 4-66

- WtFognode_getrange 4-66
 WtFognode_new 4-45
 WtFognode_setcolor 4-65
 WtFognode_setlinearstart 4-66
 WtFognode_setmode 4-66
 WtFognode_setrange 4-65
 WtFont3d_charexists 9-5
 WtFont3d_delete 9-3
 WtFont3d_gettextents 9-4
 WtFont3d_getspacing 9-3
 WtFont3d_load
 description 9-2
 example 6-21
 warning message D-7, D-10
 WtFont3d_setspacing 9-3
 WtFont3d_textobject G-9
 WtFormula_drive 13-112
 WtFormula_rawupdate 13-113
 WtFRAME constants C-4
 description 13-19
 used by Wtviewpoint_translate 16-9
 WtFree 24-10
 WtFUZZ
 and other mathematical constants C-6
 and overlapping polygons 6-12
 and Wtwindow_sethithervalue 17-18
 and Wtzero 25-33
 WtGEOBALL defined constants 13-55
 WtGeoball_present 13-55
 WtGeoball_update 13-54
 WtGeometry_begin 6-22
 WtGeometry_beginedit 6-42
 WtGeometry_beginpoly
 description 6-23
 example 6-24
 WtGeometry_changetexture 10-16
 WtGeometry_close 6-23
 WtGeometry_closesmooth 6-25
 WtGeometry_computevertexnormal 6-46
 WtGeometry_copy 6-26
 WtGeometry_delete 6-26
 WtGeometry_deleteprebuild 6-40
 WtGeometry_deletetexture 10-23
 WtGeometry_endedit 6-43
 WtGeometry_gettextents 6-29
 WtGeometry_getmidpoint
 description 6-28
 usage example A-25
 WtGeometry_getmtable 6-31
 WtGeometry_getname 6-30
 WtGeometry_getpolys
 description 6-32
 example 7-8
 WtGeometry_getradius 6-29
 WtGeometry_getrenderingstyle 6-35
 WtGeometry_getvertexmatid 6-48
 WtGeometry_getvertexnormal 6-46
 WtGeometry_getvertexposition
 description 6-44
 example 7-9
 WtGeometry_getvertexrgb 6-47
 WtGeometry_getvertices 6-33
 WtGeometry_id2poly
 description 6-33
 used with NFF files F-8
 WtGeometry_load G-9
 WtGeometry_merge 6-27
 WtGeometry_newblock 6-15
 WtGeometry_newcone 6-16
 WtGeometry_newcylinder 6-15
 WtGeometry_newextrusion 6-19
 WtGeometry_newhemisphere 6-17
 WtGeometry_newrectangle 6-17
 WtGeometry_newsphere 6-16
 WtGeometry_newtext3d
 description 6-20
 example 6-21
 WtGeometry_newtrunccone 6-18
 WtGeometry_newvertex 6-22
 WtGeometry_numpolys 6-32
 WtGeometry_prebuild 6-40
 WtGeometry_prebuildreflectmap 6-41
 WtGeometry_recomputeustats 6-44
 WtGeometry_save 6-26
 WtGeometry_scale 6-38
 WtGeometry_setmatid 6-31
 WtGeometry_setmtable 6-30
 WtGeometry_setname 6-29
 WtGeometry_setrenderingstyle 6-33
 WtGeometry_setrgb 6-31
 WtGeometry_settexture
 description 10-12
 usage example A-12
 WtGeometry_settextureuv 10-15
 WtGeometry_setuv 10-32
 WtGeometry_setvertexmatid 6-48
 WtGeometry_setvertexnormal 6-45
 vertex normals and Gouraud shading 6-9

-
- WTgeometry_setvertexposition 6-44
 - WTgeometry_setvertexrgb 6-47
 - WTgeometry_stretch 6-37
 - WTgeometry_transform 6-39
 - WTgeometry_translate 6-38
 - WTgeometrynode_load
 - description 4-46
 - versus WTnode_load A-3
 - WTgeometrynode_new 4-44
 - WTglnode_getflags 4-72
 - WTglnode_new 4-70
 - WTglnode_replacecallback 4-71
 - WTglnode_setcullingbox 4-71
 - WTglnode_setflags 4-71
 - WTGLOVE5DT constants 13-65
 - WTglove5dt_calibrateclosed 13-66
 - WTglove5dt_calibrateopen 13-66
 - WTglove5dt_loadhandmodel 13-66
 - WTglove5dt_rawupdate 13-65
 - WTglove5DT_update 13-64
 - WTglove5dt_updatefingers 13-64
 - WTgroup functions G-9
 - WTgroupnode_new 4-40
 - WTHOSTS, warning message D-7
 - WTiglasses_new macro 13-121
 - WTiglasses_rawupdate 13-123
 - WTiglasses_update 13-122
 - WTIMAGE constants C-20
 - WTIMAGES environment variable
 - description B-2
 - warning message D-7, D-14
 - WTinit_defaults 2-32
 - WTinit_setimages 2-33
 - WTinit_setmodels 2-33
 - WTinit_usewindow 17-6
 - WTinlinenode_getlocation 4-64
 - WTinlinenode_new 4-40
 - WTinlinenode_setlocation 4-63
 - WTinsidetrak_update 13-91
 - WTisotrak2_update 13-90
 - WTJOYSERIAL constants 13-117
 - WTjoyserial_fly 13-116
 - WTjoyserial_getdrift 13-119
 - WTjoyserial_getrange 13-118
 - WTjoyserial_rawupdate 13-117
 - WTjoyserial_readcalibrationfile 13-119
 - WTjoyserial_setdrift 13-118
 - WTjoyserial_walk 13-116
 - WTjoyserial_walk2 13-116
 - WTJOYSTICK 13-71
 - WTjoystick_fly 13-70
 - WTjoystick_getdrift 13-73
 - WTjoystick_getrange 13-72
 - WTjoystick_rawupdate 13-71
 - WTjoystick_readcalibrationfile 13-73
 - WTjoystick_setdrift 13-72
 - WTjoystick_walk 13-70
 - WTjoystick_walk2 13-70
 - WTK functions
 - backward compatibility G-1
 - overview 1-14
 - WTKALPHAENABLE environment variable
 - B-5
 - WTKALPHATEST environment variable B-3
 - WTKCODES
 - warning message D-8
 - WTKCODES environment variable
 - description B-1
 - WTKCODES file
 - error message D-1
 - WTKCPU environment variable B-6
 - WTKDISPLAY environment variable B-7
 - WTKEY constants C-4
 - WTkeyboard_close 24-3
 - WTkeyboard_getkey
 - description 24-2
 - keyboard constants C-4
 - WTkeyboard_getlastkey
 - description 24-2
 - keyboard constants C-4
 - WTkeyboard_open
 - description 24-1
 - usage example A-8
 - WTKLS environment variable B-5
 - error message D-2
 - wtklsd daemon
 - warning message D-11
 - WTKMAXTEXTSIZE environment variable ...
 - B-4
 - WTKMULTISAMPLE environment variable .
 - B-6
 - WTKPROXY environment variable B-4
 - WTKSHMEM environment variable B-8
 - WTKSQRTTEX environment variable B-4
 - WTKZBUFFERSIZE environment variable ...
 - B-3
 - WTlight functions
 - mapping to current release functions G-9

- WTLightnode_getambient 12-15
- WTLightnode_getangle 12-19
- WTLightnode_getattenuation 12-18
- WTLightnode_getdiffuse 12-16
- WTLightnode_getdirection 12-13
- WTLightnode_getexponent 12-20
- WTLightnode_getintensity 12-14
- WTLightnode_getposition 12-12
- WTLightnode_getspecular 12-16
- WTLightnode_gettype 12-18
- WTLightnode_load 12-9
- WTLightnode_newambient 12-5
- WTLightnode_newdirected 12-6
- WTLightnode_newpoint 12-7
- WTLightnode_newspot 12-8
- WTLightnode_save 12-11
- WTLightnode_setambient 12-14
- WTLightnode_setangle 12-19
- WTLightnode_setattenuation 12-17
- WTLightnode_setdiffuse 12-15
- WTLightnode_setdirection 12-13
- WTLightnode_setexponent 12-19
- WTLightnode_setintensity 12-14
- WTLightnode_setposition 12-12
- WTLightnode_setspecular 12-15
- WTLIGHTTYPE constants C-5
- WTLINE constants C-2
- WTlodnode_getcenter 4-56
- WTlodnode_getrange 4-55
- WTlodnode_new 4-41
- WTlodnode_numranges 4-56
- WTlodnode_setcenter 4-56
- WTlodnode_setrange 4-55
- WTLOGITECH constants
 - example, WTLOGITECH_FLYING 13-74
 - for 3D Mouse 13-76
 - for CrystalEyesVR 13-110
 - for Head Tracker 13-81
- WTlogitech_new macro 13-78
- WTlogitech_update 13-80
- WTm3 (3D matrix)
 - functions 25-21
 - math data type 25-1
- WTm3_2euler 25-27
- WTm3_2eulernear 25-32
- WTm3_2q 25-26
- WTm3_copy 25-22
- WTm3_init 25-21
- WTm3_multm3 25-22
- WTm3_transpose 25-22
- WTm4 (4D matrix)
 - functions 25-22
 - math data type 25-1
- WTm4_2pq 25-31
- WTm4_copy 25-23
- WTm4_init 25-23
- WTm4_invert 25-24
- WTm4_multm4 25-24
- WTm4_rotatemp3 25-25
- WTm4_transpose 25-23
- WTm4_xformp3 25-24
- WTmalloc 24-9
- WTMAT constants C-5
- WTmessage
 - description 24-5
 - printing 25-12
- WTMESSAGE constants
 - description 24-6
 - printing to WTMESSAGE_USER 25-12
- WTmessage_sendto
 - description 24-6
 - printing 25-12
- WTmessage_setcallback 24-7
- WTMODELS environment variable
 - description B-2
 - error message D-7
- WTmotionlink_addconstraint 15-11
- WTmotionlink_delete 15-5
- WTmotionlink_enable 15-5
- WTmotionlink_getconstraintframe 15-11
- WTmotionlink_getdata 15-6
- WTmotionlink_getreferenceframe 15-9
- WTmotionlink_getsource 15-6
 - description 15-6
 - example 15-7
- WTmotionlink_gettarget 15-6
 - description 15-6
 - example 15-7
- WTmotionlink_isenabled 15-5
- WTmotionlink_new 15-3
- WTmotionlink_next 15-8
- WTmotionlink_removeconstraint 15-12
- WTmotionlink_setconstraintframe 15-10
- WTmotionlink_setdata 15-6
- WTmotionlink_setreferenceframe 15-8
- WTMOUSE constants 13-33
- WTmouse_drawcursor 13-28, E-8
- WTmouse_gettrackballdrift 13-37

-
- WTmouse_gettrackballsnap 13-39
 - WTmouse_gettrackballsnapangle 13-38
 - WTmouse_inwindow 13-35
 - WTmouse_move2D
 - description 13-29
 - example of setting update function 13-10
 - platform-independent macro example 13-27
 - WTmouse_moveview1 13-29
 - WTmouse_moveview2 13-31
 - WTmouse_new macro 13-26
 - WTmouse_rawupdate
 - description 13-32
 - example E-9
 - WTmouse_settrackballdrift 13-37
 - WTmouse_settrackballsnap 13-38
 - WTmouse_settrackballsnapangle 13-38
 - WTmouse_trackball 13-36
 - WTmouse_trackballreset 13-39
 - WTmouse_trackballvpoint 13-37
 - WTmouse_whichwindow 13-35
 - WTmovgeometrynode_new 5-3
 - WTmovlightnode_newdirected 5-4
 - WTmovlightnode_newpoint 5-3
 - WTmovlightnode_newspot 5-4
 - WTmovlodnode_new 5-5
 - WTmovnode_attach
 - description 5-11
 - usage example A-16
 - WTmovnode_axisrotation 5-8
 - WTmovnode_deleteattachment 5-12
 - WTmovnode_detach 5-12
 - WTmovnode_getattachment 5-13
 - WTmovnode_instance 5-13
 - WTmovnode_load
 - description 5-5
 - usage example A-16
 - versus WTnode_load A-4
 - WTmovnode_numattachments 5-13
 - WTmovsepnode_new 5-4
 - WTmovswitchnode_new 5-5
 - WTmsleep
 - description 24-8
 - example E-11
 - WTmtable_copyentry 8-14
 - WTmtable_delete 8-8
 - WTmtable_getbyname 8-13
 - WTmtable_getdata 8-13
 - WTmtable_getentrybyname 8-17
 - WTmtable_getentryname 8-16
 - WTmtable_getname 8-13
 - WTmtable_getnumentries 8-8
 - WTmtable_getproperties 8-11
 - WTmtable_getvalue 8-15
 - WTmtable_load 8-11
 - WTmtable_merge 8-8
 - WTmtable_new 8-7
 - WTmtable_newentry 8-14
 - WTmtable_save 8-12
 - WTmtable_setdata 8-13
 - WTmtable_setentryname 8-16
 - WTmtable_setname 8-12
 - WTmtable_setproperties 8-9
 - WTmtable_setvalue 8-15
 - WTnet_additem 22-4, 22-9
 - sending message items 22-3
 - WTnet_addstring 22-10
 - WTnet_close 22-9
 - WTnet_flush 22-13
 - WTnet_getport 22-13
 - WTnet_getrange 22-13
 - WTnet_next
 - checking for message items 22-4
 - description 22-11
 - WTnet_open
 - description 22-7
 - initializing network communications .. 22-3
 - warning message D-6, D-7, D-9
 - WTnet_removeitem 22-4, 22-11
 - receiving message items 22-3, 22-4
 - WTnet_removestring 22-12
 - WTnet_skip 22-13
 - WTNODE constants C-7, C-8
 - WTnode properties 3-3
 - WTnode_addchild 4-74
 - WTnode_addsensor
 - description 4-92
 - example 13-17
 - WTnode_axisrotation 4-62
 - WTnode_boundingbox 4-73
 - WTnode_canaddchild 4-51
 - WTnode_delete 4-75
 - WTnode_deletechild 4-75
 - WTnode_enable 4-49
 - WTnode_getchild 4-77
 - WTnode_getdata 4-51
 - WTnode_gettextents 4-53
 - WTnode_getgeometry 4-45
 - WTnode_getmidpoint 4-54

-
- WTnode_getname 4-49
 - WTnode_getobject G-35
 - WTnode_getorientation 4-60
 - WTnode_getparent 4-78
 - WTnode_getradius 4-54
 - WTnode_getrotation 4-60
 - WTnode_gettransform 4-58
 - WTnode_gettranslation 4-59
 - WTnode_gettype 4-50
 - WTnode_hasboundingbox 4-73
 - WTnode_insertchild 4-74
 - WTnode_isenabled 4-50
 - WTnode_ismovable 4-50
 - WTnode_load
 - description 4-46
 - versus WTgeometrynode_load A-3
 - versus WTmovnode_load A-4
 - WTnode_numchildren 4-77
 - WTnode_numparents 4-77
 - WTnode_numpolys 4-78
 - WTnode_print 4-76
 - WTnode_rayintersect 4-88
 - WTnode_remove 4-75
 - WTnode_removechild 4-74
 - WTnode_removesensor 4-92
 - WTnode_rotatem3 4-61
 - WTnode_rotatem4 4-62
 - WTnode_rotateq 4-61
 - WTnode_rotation 4-61
 - WTnode_save 4-48
 - WTnode_setdata 4-51
 - WTnode_setname 4-49
 - WTnode_setorientation 4-60
 - WTnode_setrotation 4-60
 - WTnode_settransform 4-58
 - WTnode_settranslation 4-59
 - WTnode_translate 4-59
 - WTnode_vacuum 4-76
 - WTnodepath_addsensor 4-93
 - WTnodepath_delete 4-82
 - WTnodepath_getextents 4-83
 - WTnodepath_getnode 4-82
 - WTnodepath_getorientation 4-85
 - WTnodepath_gettransform 4-84
 - WTnodepath_gettranslation 4-84
 - WTnodepath_gettraversal 4-83
 - WTnodepath_intersectbbox 4-87
 - WTnodepath_intersectnode 4-87
 - WTnodepath_intersectpoly 4-86
 - WTnodepath_new 4-81
 - WTnodepath_numnodes 4-82
 - WTnodepath_removesensor 4-93
 - WTnormal_2slope 25-33
 - WTOBJECT functions
 - mapping to current release functions ...G-11
 - WTOBJECT_getnodeG-35
 - WTOBJECT_setcolorG-35
 - WTOBJECT_setrgbG-35
 - WTOPTION constants 2-24
 - WTOUTPUT constants C-19
 - WTp2 (2D vectors)
 - functions 25-4
 - math data type 25-1
 - WTp2_copy 25-4
 - WTp2_dot 25-5
 - WTp2_init 25-4
 - WTp2_mag 25-4
 - WTp2_norm 25-4
 - WTp2_subtract 25-5
 - WTp3 (3D vectors)
 - functions 25-5
 - math data type 25-1
 - WTp3_add 25-7
 - WTp3_coplanar 25-11
 - WTp3_copy 25-6
 - WTp3_cross 25-8
 - WTp3_distance 25-11
 - WTp3_disttovector 25-11
 - WTp3_dot
 - description 25-7
 - example 7-5
 - WTp3_equal 25-8
 - WTp3_exact 25-9
 - WTp3_frame2frame 25-34
 - WTp3_init 25-5
 - WTp3_invert
 - description 25-6
 - example 13-58
 - WTp3_local2worldframe 25-34
 - WTp3_mag 25-6
 - WTp3_multm3 25-10
 - WTp3_multm4 25-10
 - WTp3_mults 25-10
 - WTp3_norm 25-6
 - WTp3_print 25-12
 - WTp3_rotate 25-9
 - WTp3_rotatept 25-9
 - WTp3_subtract 25-7
-

-
- WTp3_world2localframe 25-34
 - WTp3_xform 25-10
 - WTPATH constants
 - for WTPath_interpolate 14-6
 - for WTPath_seek 14-18
 - WTPath_properties 3-6
 - WTPath_appendelement 14-27
 - WTPath_copy 14-5
 - WTPath_delete 14-5
 - WTPath_getconstraints 14-21
 - WTPath_getcurrentelement
 - description 14-18
 - example 14-19
 - WTPath_getdata 14-30
 - WTPath_getdirection 14-20
 - WTPath_getelements 14-10
 - WTPath_getmarker 14-10
 - WTPath_getmode 14-22
 - WTPath_getname 14-29
 - WTPath_getobject G-16
 - WTPath_getplayspeed 14-23
 - WTPath_getsamples 14-23
 - WTPath_getvisibility 14-9
 - WTPath_insertelement 14-27
 - WTPath_interpolate 14-6
 - WTPath_isplaying 14-16
 - WTPath_isrecording 14-16
 - WTPath_load
 - description 14-11
 - warning message D-10
 - WTPath_new 14-4
 - WTPath_next 14-10
 - WTPath_numelements 14-10
 - WTPath_play
 - description 14-14
 - example 14-19
 - WTPath_play1 14-14
 - WTPath_record 14-14
 - WTPath_record1 14-15
 - WTPath_rewind 14-16
 - WTPath_save 14-12
 - WTPath_seek
 - description 14-18
 - example 14-19, 14-28
 - WTPath_setconstraints 14-20
 - WTPath_setcurrentelement 14-17
 - WTPath_setdata 14-29
 - WTPath_setdirection
 - description 14-19
 - example 14-19
 - WTPath_setmarker 14-9
 - example 14-5
 - WTPath_setmode 14-21
 - WTPath_setname 14-29
 - WTPath_setnodeobject G-16
 - WTPath_setobject G-16
 - WTPath_setplayspeed 14-22
 - WTPath_setrecordlink 14-15
 - WTPath_setsamples 14-23
 - WTPath_setvisibility 14-8
 - WTPath_showcurrentelement 14-17
 - WTPath_stop 14-16
 - WTPathelement_copy 14-25
 - WTPathelement_delete 14-24
 - WTPathelement_getorientation
 - description 14-26
 - example 14-28
 - WTPathelement_getpath 14-26
 - WTPathelement_getposition 14-25
 - WTPathelement_new
 - description 14-24
 - example 14-28
 - WTPathelement_next
 - description 14-26
 - example 14-26
 - WTPathelement_remove 14-25
 - WTPathelement_setorientation 14-26
 - WTPathelement_setposition 14-25
 - WTPATHLEN constant C-22
 - WTPathnode_getobject G-16
 - WTPINCH constants 13-62
 - WTPinch_update 13-61
 - WTPLAY constants 14-21
 - WTPolhemus_new
 - example 16-6
 - WTPoly_addvertex
 - description 7-10
 - example 6-24
 - WTPoly_addvertextr 7-10
 - WTPoly_begin G-16
 - WTPoly_close
 - description 7-12
 - example 6-24
 - WTPoly_delete 7-12
 - WTPoly_deletetexture 10-23
 - WTPoly_getbothsides 7-4
 - WTPoly_getcg 7-5
 - WTPoly_getcolor G-16

-
- WTPoly_getgeometry 7-7
 - WTPoly_getid
 - description 7-7
 - with NFF file F-8
 - WTPoly_getmatid 7-3
 - WTPoly_getnormal
 - description 7-4
 - example 7-5, 25-33
 - WTPoly_getobject G-16
 - WTPoly_getportal G-16
 - WTPoly_getrgb 7-2
 - WTPoly_gettextureinfo 10-31
 - WTPoly_gettexturestyle 10-24
 - WTPoly_gettexturetype G-16
 - WTPoly_getuv 10-32
 - WTPoly_getvertex
 - description 7-8
 - example 7-9
 - WTPoly_intersectbbox 4-86
 - WTPoly_intersectnode 4-86
 - WTPoly_intersectobject G-16
 - WTPoly_intersectobjpolys G-16
 - WTPoly_intersectpoly G-17
 - WTPoly_intersectpolygon 4-85
 - WTPoly_intersectuniverse G-17
 - WTPoly_intersectunivpolys G-17
 - WTPoly_mirrortexture 10-29
 - WTPoly_next 7-8
 - WTPoly_numvertices
 - description 7-9
 - example 7-8
 - WTPoly_rayintersect 4-88
 - WTPoly_rotatetexture 10-27
 - WTPoly_scaletexture 10-28
 - WTPoly_setbothsides
 - description 7-4
 - example 6-25
 - WTPoly_setcolor G-17, G-35
 - WTPoly_setid 7-6
 - WTPoly_setmatid 7-3
 - WTPoly_setrgb 7-2
 - WTPoly_settexture
 - description 10-11
 - error message D-2
 - usage example A-12
 - WTPoly_settexturestyle 10-23
 - WTPoly_settexturetype G-17
 - WTPoly_settextureuv 10-13
 - WTPoly_setuv 10-32
 - WTPoly_stretchtexture 10-30
 - WTPoly_translatetexture 10-29
 - WTportal functions G-17
 - WTPq (WTP3 and WTq)
 - functions 25-19
 - math data type 25-1
 - WTPq_2m4 25-31
 - WTPq_copy 25-20
 - WTPq_frame2frame 25-36
 - WTPq_init 25-20
 - WTPq_local2worldframe 25-36
 - WTPq_print 25-20
 - WTPq_world2localframe 25-36
 - WTprecision_new macro 13-96
 - WTprecision_rawupdate 13-97
 - WTprecision_update 13-97
 - WTPROJECTION constants
 - and WTwindow_setparams 17-16
 - description 17-14
 - WTproperty_addhandler 3-25
 - WTproperty_delete 3-15
 - WTproperty_exists 3-16
 - WTproperty_get 3-20
 - WTproperty_getasString 3-22
 - WTproperty_getd 3-21
 - WTproperty_getdata 3-16
 - WTproperty_getdatatype 3-16
 - WTproperty_getf 3-21
 - WTproperty_gethandler 3-26
 - WTproperty_geti 3-21
 - WTproperty_getp 3-22
 - WTproperty_getp2 3-21
 - WTproperty_getp3 3-22
 - WTproperty_getq 3-22
 - WTproperty_gets 3-22
 - WTproperty_getsharegroup 21-8
 - WTproperty_getsizeofdata 3-17
 - WTproperty_gettimesensitive 21-9
 - WTproperty_getui 3-21
 - WTproperty_getupdatefreq 21-8
 - WTproperty_islocked 21-10
 - WTproperty_islockedbyme 21-11
 - WTproperty_issared 21-7
 - WTproperty_lock 21-10
 - WTproperty_new 3-15
 - WTproperty_numhandlers 3-25
 - WTproperty_numshares 21-7
 - WTproperty_removeallhandlers 3-26
 - WTproperty_removehandler 3-25
-

-
- WTproperty_sendupdate 21-9
 - WTproperty_set 3-17
 - WTproperty_setat 3-19
 - WTproperty_setd 3-18
 - WTproperty_setdata 3-16
 - WTproperty_setf 3-18
 - WTproperty_seti 3-18
 - WTproperty_setp 3-19
 - WTproperty_setp2 3-18
 - WTproperty_setp3 3-19
 - WTproperty_setq 3-19
 - WTproperty_sets 3-19
 - WTproperty_settimesensitive 21-9
 - WTproperty_setui 3-18
 - WTproperty_setupdatefreq 21-8
 - WTproperty_share 21-5
 - WTproperty_unlock 21-10
 - WTproperty_unshare 21-7
 - WTq (quaternions)
 - math data type 25-1
 - overview and functions 25-12
 - WTq_2dir 25-30
 - WTq_2dirandtwist 25-30
 - WTq_2euler 25-29
 - WTq_2eulernear 25-32
 - WTq_2m3 25-26
 - WTq_2m4 25-32
 - WTq_construct 25-17
 - WTq_copy 25-15
 - WTq_dot 25-18
 - WTq_equal 25-16
 - WTq_exact 25-16
 - WTq_frame2frame 25-35
 - WTq_getangle 25-17
 - WTq_getvector 25-16
 - WTq_init 25-14
 - WTq_interpolate 25-18
 - WTq_invert 25-15
 - WTq_local2worldframe 25-35
 - WTq_mag 25-15
 - WTq_mult
 - description 25-17
 - example 13-20, 25-29
 - WTq_multinv 25-18
 - WTq_norm 25-16
 - WTq_print 25-19
 - WTq_scale 25-17
 - WTq_world2localframe 25-35
 - WTrealloc 24-10
 - WTRENDER constants
 - used with geometries 6-34
 - used with the universe 2-18
 - WTrootnode_new 4-39
 - WTrootnode_next 4-77
 - WTSAMPLERATE constants C-18
 - WTscreen_getyblank 2-21
 - WTscreen_load 10-33
 - WTscreen_pickpoly
 - description 4-91
 - example 13-16
 - WTscreen_setyblank 2-20
 - WTsensor properties 3-5
 - WTSENSOR_DEFAULT constant
 - example 13-6
 - used with WTsensor_new 13-9
 - WTsensor_delete 13-10
 - WTsensor_getangularrate
 - description 13-13
 - example E-10, E-14
 - scaling sensor records E-3
 - WTsensor_getconstraints 13-22
 - WTsensor_getdata 13-24
 - WTsensor_getlastrecord 13-25
 - WTsensor_getmiscdata
 - description 13-15
 - example 2-9, E-10
 - WTsensor_getname 13-23
 - WTsensor_getrawdata
 - description 13-15
 - example E-9, E-13
 - used with mouse raw data 13-27
 - WTsensor_getrotation 13-14
 - WTsensor_getsensitivity
 - description 13-12
 - example E-9, E-14
 - scaling sensor records E-3
 - WTsensor_getserial
 - description 13-16
 - example E-11
 - WTsensor_gettranslation 13-13
 - WTsensor_getunit 13-16
 - WTsensor_new
 - description 13-7
 - example E-1
 - overview 13-5
 - WTsensor_next
 - description 13-10
 - example 13-11

-
- WTsensor_relativizerecord 13-24
 - description 13-24
 - example E-7
 - WTsensor_rotate
 - description 13-20
 - rotating sensor input 13-17
 - WTsensor_setangularrate
 - description 13-12
 - scaling sensor records E-3
 - WTsensor_setconstraints 13-21
 - WTsensor_setdata 13-23
 - WTsensor_setlastrecord
 - description 13-25
 - example E-5, E-7
 - WTsensor_setmiscdata
 - description 13-25
 - example E-14
 - WTsensor_setname 13-23
 - WTsensor_setrawdata
 - description 13-26
 - example E-11
 - WTsensor_setrecord
 - description 13-24
 - example E-6, E-10, E-14
 - WTsensor_setsensitivity
 - description 13-11
 - example 4-54
 - scaling sensor records E-3
 - WTsensor_setupdatefn 13-10
 - WTsepnode_getcullmode 4-57
 - WTsepnode_new 4-41
 - WTsepnode_setcullmode 4-57
 - WTserial_delete
 - description 23-2
 - error message D-3
 - WTserial_new 23-1
 - description 23-1
 - error message D-1, D-4
 - warning message D-5, D-11
 - WTserial_ntoread 23-4
 - WTserial_read
 - description 23-3
 - warning message D-13
 - WTserial_write
 - description 23-4
 - error message D-4
 - WTsharegroup_delete 21-16
 - WTsharegroup_enumerate 21-36
 - WTsharegroup_findchildbyname 21-21
 - WTsharegroup_getchild 21-20
 - WTsharegroup_getconnection 21-18
 - WTsharegroup_getdata 21-18
 - WTsharegroup_getname 21-18
 - WTsharegroup_getparent 21-20
 - WTsharegroup_getproperty 21-21
 - WTsharegroup_islocked 21-19
 - WTsharegroup_islockedbyme 21-19
 - WTsharegroup_issshared 21-17
 - WTsharegroup_lock 21-19
 - WTsharegroup_new 21-15
 - WTsharegroup_numchildren 21-20
 - WTsharegroup_numproperties 21-21
 - WTsharegroup_print 21-18
 - WTsharegroup_registerinterest 21-20
 - WTsharegroup_setdata 21-17
 - WTsharegroup_share 21-17
 - WTsharegroup_unlock 21-19
 - WTSOUND constants 20-14
 - WTsound_delete 20-10
 - WTsound_getdata 20-16
 - WTsound_getdonefn 20-17
 - WTsound_getname 20-15
 - WTsound_getnodepath 20-18
 - WTsound_getparam 20-15
 - WTsound_getposition 20-17
 - WTsound_isplaying 20-12
 - WTsound_load 20-10
 - WTsound_next 20-11
 - WTsound_play 20-11
 - WTsound_setdata 20-15
 - WTsound_setdonefn 20-16
 - WTsound_setnodepath 20-18
 - WTsound_setparam 20-12
 - WTsound_setposition 20-17
 - WTsound_stop 20-10
 - WTSOUNDDEVICE constants C-19
 - WTsounddevice_close 20-3
 - WTsounddevice_getdata 20-9
 - WTsounddevice_getlistener 20-9
 - WTsounddevice_getparam 20-8
 - WTsounddevice_getsounds 20-4
 - WTsounddevice_name2sound 20-5
 - WTsounddevice_numplayable 20-4
 - WTsounddevice_open 20-3
 - WTsounddevice_setdata 20-8
 - WTsounddevice_setlistener 20-9
 - WTsounddevice_setparam 20-5
 - WTsounddevice_update 20-4

-
- WTSOURCE constantsC-7
 - WTSPACEBALL constantsC-15
 - WTspaceball_dominant 13-102
 - WTspaceball_new
 - example 13-13, 16-7
 - WTspaceball_rezero 13-103
 - WTspaceball_update 13-102
 - WTSPACEBALLSC constants 13-107
 - WTspaceballSC_dominant 13-106
 - WTspaceballSC_rezero 13-107
 - WTspaceballSC_setwindow 13-107
 - WTspaceballSC_update 13-106
 - WTSPACECONTROL constants 13-84
 - WTspacecontrol_new macro 13-81
 - WTspacecontrol_rawupdate 13-83
 - WTspacecontrol_update 13-83
 - WTSPATIALIZE constants
 - for WTsound_setparam 20-14
 - for WTsounddevice_setparam 20-7
 - WTSPATIALIZE_OFF 20-7, 20-14
 - WTSWITCH constantsC-8
 - WTswitchnode_getwhichchild 4-58
 - WTswitchnode_new 4-42
 - WTswitchnode_setwhichchild 4-57
 - WTTARGET constantsC-7
 - WTtask_add 11-4
 - WTtask_delete 11-4
 - WTtask_getfunction 11-5
 - WTtask_getpriority 11-5
 - WTtask_new 11-2
 - usage example A-21
 - WTtask_remove 11-4
 - WTtask_setpriority 11-5
 - WTtexture_cache 10-17
 - WTtexture_getfilter 10-27
 - WTtexture_getmemory 10-18
 - WTtexture_iscached 10-18
 - WTtexture_load 10-17
 - WTtexture_replace 10-16
 - WTtexture_setfilter 10-25
 - WTtime_getcurrent 3-27
 - WTtime_getcurrentlocal 3-27
 - WTtime_getcurrentmsec 3-28
 - WTtime_getcurrentmseclocal 3-28
 - WTtime_getcurrentsec 3-27
 - WTtime_getcurrentseclocal 3-27
 - WTtime_getdouble 3-28
 - WTtime_getmsec 3-28
 - WTtime_getsec 3-28
 - WTime_update 3-27
 - WTUI constantsC-20
 - WTui_check 18-35
 - WTui_checkbuttonstate 18-35
 - WTui_delete 18-40
 - WTui_deleteitem 18-33
 - WTui_enable 18-34
 - WTui_getcallback 18-41
 - WTui_getid 18-37
 - WTui_getitemtext 18-32
 - WTui_getmenutext 18-30
 - WTui_getnumitems 18-32
 - WTui_getparent 18-40
 - WTui_getposition 18-36
 - WTui_getscalefactor 18-29
 - WTui_getselecteditem 18-31
 - WTui_gettext 18-30
 - WTui_go 18-12
 - WTui_init 18-5
 - WTui_insertitem 18-33
 - WTui_isChecked 18-35
 - WTui_isconsolevisible 18-41
 - WTui_isenabled 18-34
 - WTui_iswtkrunning 18-38
 - WTui_manage 18-11
 - WTui_setcallback 18-8
 - WTui_setitemtext 18-32
 - WTui_setmenutext 18-30
 - WTui_setposition 18-36
 - WTui_setscalefactor 18-29
 - WTui_settext 18-29
 - WTui_settoolbarcallback 18-9
 - WTui_showconsole 18-41
 - WTui_unmanage 18-40
 - WTui_wtkstart 18-38
 - WTui_wtkstop 18-38
 - WTUIATT constants 18-7
 - WTuicheckbutton_new 18-16
 - WTuifileselection_new 18-13
 - WTuiform_new 18-6
 - WTuilabel_new 18-17
 - WTuimenubar_new 18-24
 - WTuimenuitem_new 18-26
 - WTuimenupopup_new 18-25
 - WTuimessagebox_new 18-15
 - WTuipushbutton_new 18-18
 - WTuiradiobox_new 18-18
 - WTuiscale_new 18-19
 - WTuiscrolllist_new 18-21

-
- WTuisrolledtext_new 18-23
 - WTuitextfield_new 18-24
 - WTuitextinput_new 18-15
 - WTuitoolbar_new 18-28
 - WTuiwtkwindow_new 18-11
 - WTuniverse_avgframerate 2-24
 - WTuniverse_delete 2-5
 - WTuniverse_deleteconnections 21-28
 - WTuniverse_deletelink 2-17
 - WTuniverse_findnodebyname
 - description 4-49
 - getting a pointer to a node using its name ...
A-20
 - WTuniverse_framecount 2-23
 - WTuniverse_framerate 2-23
 - WTuniverse_getanimations G-17
 - WTuniverse_getbases 3-8
 - WTuniverse_getbgcolor G-17
 - WTuniverse_getbgrgb 2-21
 - WTuniverse_getconnections 21-27
 - WTuniverse_getcurrscriidx 2-14
 - WTuniverse_getcurrwindow 2-14
 - WTuniverse_getcurrwinidx 2-14
 - WTuniverse_getentryfn G-17
 - WTuniverse_geteventorder 2-10
 - WTuniverse_getexitfn G-17
 - WTuniverse_getextents G-17
 - WTuniverse_getframe G-17
 - WTuniverse_getinitialview 2-16
 - WTuniverse_getintersectedpolys G-17
 - WTuniverse_getlights G-17
 - WTuniverse_getmidpoint G-17
 - WTuniverse_getmotionlinks 2-17
 - WTuniverse_getname G-17
 - WTuniverse_getobjects G-17
 - WTuniverse_getoption 2-27
 - WTuniverse_getpaths
 - description 2-13
 - example 14-11
 - WTuniverse_getpolys G-17
 - WTuniverse_getportaling G-17
 - WTuniverse_getradius G-18
 - WTuniverse_getremovedobjects G-18
 - WTuniverse_getrendering 2-20
 - WTuniverse_getrootnodes 2-17
 - WTuniverse_getsensors
 - description 2-13
 - example 13-11
 - WTuniverse_getsubfaceoffset 2-22
 - WTuniverse_gettaskbypointer 11-6
 - WTuniverse_getviewpoint G-18
 - WTuniverse_getviewpoints
 - description 2-15
 - example 2-16, 16-7
 - WTuniverse_getwindows 2-13
 - WTuniverse_go
 - description 2-7
 - example 16-7
 - WTuniverse_go1 2-7
 - WTuniverse_id2poly G-18
 - WTuniverse_intersect G-18
 - WTuniverse_load G-18
 - WTuniverse_name2object G-18
 - WTuniverse_new
 - description 2-2
 - example 17-8
 - WTuniverse_npolygons G-18
 - WTuniverse_pickobject G-18
 - WTuniverse_pickpolygon G-18
 - WTuniverse_processevents 3-26
 - WTuniverse_ready 2-6
 - WTuniverse_resetframecount 2-23
 - WTuniverse_resettime 2-23
 - WTuniverse_save G-18
 - WTuniverse_setactions 2-12
 - WTuniverse_setbboxrgb 2-21
 - WTuniverse_setbgcolor G-18
 - WTuniverse_setbgrgb 2-21
 - WTuniverse_setentryfn G-18
 - WTuniverse_seteventorder 2-9
 - WTuniverse_setexitfn G-18
 - WTuniverse_setname G-18
 - WTuniverse_setopt 2-24
 - WTuniverse_setportaling G-18
 - WTuniverse_setrendering 2-18
 - WTuniverse_setsubfaceoffset 2-22
 - WTuniverse_setviewpoint 2-15
 - WTuniverse_stop 2-8
 - WTuniverse_time 2-22
 - WTuniverse_updateconnections 21-29
 - WTuniverse_vacuum G-18
 - WTurl_download 4-47
 - WTvalue_tostring 3-23
 - WTvertex_delete G-19
 - WTvertex_getnormal G-19
 - WTvertex_getposition G-19
 - WTvertex_new G-19
 - WTvertex_next 6-33
-

-
- WTvertex_setcolor G-35
 - WTvertex_setnormal G-19
 - WTvertex_setposition G-19
 - WTvertex_setrgb G-19, G-35
 - WTviewpoint properties 3-4
 - WTviewpoint_addsensor 16-7, G-19
 - example 13-17, 13-77
 - WTviewpoint_alignaxis 16-14
 - WTviewpoint_copy 16-5
 - WTviewpoint_delete 16-5
 - WTviewpoint_getaspect 16-19
 - WTviewpoint_getasymmetric G-19
 - WTviewpoint_getaxis 16-14
 - WTviewpoint_getconvdistance 16-23
 - WTviewpoint_getconvergence 16-21
 - WTviewpoint_getdata 16-26
 - WTviewpoint_getdirection
 - description 16-13
 - example 13-57
 - WTviewpoint_getdirectionframe 16-18
 - WTviewpoint_getframe 16-15
 - WTviewpoint_gethithervalue G-19
 - WTviewpoint_getlastorientation 16-11
 - WTviewpoint_getlastposition 16-9
 - WTviewpoint_getname 16-25
 - WTviewpoint_getorientation 16-11
 - WTviewpoint_getorientationframe 16-16
 - WTviewpoint_getparallax 16-20
 - WTviewpoint_getposition
 - description 16-8
 - example 16-20
 - WTviewpoint_getpositionframe 16-15
 - WTviewpoint_getviewangle G-19
 - WTviewpoint_getwindowparams G-19
 - WTviewpoint_getyonvalue G-19
 - WTviewpoint_intersectpoly
 - description 4-89
 - usage example A-22
 - WTviewpoint_local2world 16-24
 - WTviewpoint_move 16-12
 - WTviewpoint_moveframe 16-17
 - WTviewpoint_moveto
 - description 16-13
 - example 2-16
 - restricting viewpoint motion 4-53
 - WTviewpoint_movetoframe 16-17
 - WTviewpoint_new 16-3
 - WTviewpoint_next 16-5
 - WTviewpoint_removesensor 16-8, G-19
 - example 13-77
 - WTviewpoint_rotate 16-12
 - WTviewpoint_rotateframe 16-16
 - WTviewpoint_setaspect 16-18
 - WTviewpoint_setasymmetric G-19
 - WTviewpoint_setconvdistance
 - description 16-22
 - example 16-23
 - WTviewpoint_setconvergence 16-21
 - WTviewpoint_setdata 16-25
 - WTviewpoint_setdirection 16-13
 - WTviewpoint_setdirectionframe 16-17
 - WTviewpoint_sethithervalue G-19
 - WTviewpoint_setname 16-25
 - WTviewpoint_setorientation 16-11
 - WTviewpoint_setorientationframe 16-16
 - WTviewpoint_setparallax 16-19
 - WTviewpoint_setposition 16-8
 - WTviewpoint_setpositionframe 16-15
 - WTviewpoint_setviewangle G-19
 - WTviewpoint_setwindowparams G-19
 - WTviewpoint_setyonvalue G-19
 - WTviewpoint_translate
 - description 16-9
 - example 13-58
 - WTviewpoint_translateframe 16-15
 - WTviewpoint_world2local 16-24
 - WTviewpoint_zoomall G-19
 - WTvml_seturl 4-62
 - WTwarning 24-6
 - WTWINDOW constants
 - for WTuniverse_new 2-3
 - for WTwindow_new 17-2
 - WTWINDOW_DEFAULT, example .. 16-6
 - WTwindow properties 3-4
 - WTwindow_delete 17-7
 - WTwindow_draw2Dcircle 19-3
 - WTwindow_draw2Dline 19-4
 - WTwindow_draw2Dpoint 19-4
 - WTwindow_draw2Drectangle 19-3
 - WTwindow_draw2Dtext 19-7
 - WTwindow_draw2Dtexture 19-5
 - WTwindow_draw3Dlines 19-10
 - WTwindow_draw3Dpoints 19-10
 - WTwindow_enable 17-9
 - WTwindow_get2Dtextextents 19-7
 - WTwindow_getbgcolor G-19
 - WTwindow_getbgrgb 17-22
 - WTwindow_getdata 17-28

-
- WTwindow_geteye 17-12
 - WTwindow_gethithervalue 17-18
 - WTwindow_getidx 17-29
 - WTwindow_getimage 10-33, 17-27
 - WTwindow_getname 17-27
 - WTwindow_getparams 17-17
 - WTwindow_getposition 17-10
 - WTwindow_getprojection 17-16
 - WTwindow_gettray 17-21
 - WTwindow_getrootnode 17-9
 - WTwindow_getscreen 17-13
 - WTwindow_getviewangle 17-20
 - WTwindow_getviewport 17-11
 - WTwindow_getviewport2 17-13
 - WTwindow_getviewport 17-32
 - WTwindow_getwidget 17-29
 - WTwindow_getyonvalue 17-19
 - WTwindow_isenabled 17-9
 - WTwindow_loadimage
 - description 17-25
 - with 2D drawing functions 17-24
 - WTwindow_new 17-2
 - WTwindow_newuser 17-7
 - WTwindow_newviewport 17-32
 - WTwindow_next 17-8
 - WTwindow_numpolys 17-25
 - WTwindow_pickpoly 17-20
 - WTwindow_pickpolygon G-19
 - WTwindow_projectpoint 17-21
 - WTwindow_saveimage 17-26
 - WTwindow_set2Dcolor 19-1
 - WTwindow_set2Dfont 19-6
 - WTwindow_set2Dlinestyle 19-2
 - WTwindow_set2Dlinewidth 19-2
 - WTwindow_set3Dcolor 19-8
 - WTwindow_set3Dlinestyle 19-8
 - WTwindow_set3Dlinewidth 19-9
 - WTwindow_set3Dpointsize 19-9
 - WTwindow_setbgcolor G-19
 - WTwindow_setbgrgb 17-22
 - WTwindow_setdata 17-28
 - WTwindow_setdrawfn 17-23
 - WTwindow_seteye 17-12
 - WTwindow_setfgactions 17-24
 - WTwindow_sethithervalue 17-18
 - WTwindow_setname 17-27
 - WTwindow_setparams 17-16
 - WTwindow_setposition 17-10
 - WTwindow_setprojection 17-14
 - WTwindow_setrootnode 17-8
 - WTwindow_setviewangle 17-19
 - WTwindow_setviewport 17-11
 - WTwindow_setviewport2 17-12
 - WTwindow_setviewport 17-31
 - WTwindow_setyonvalue 17-19
 - WTwindow_zoomviewport 17-13
 - WTwindow_zoomviewportnode 17-14
 - WTxformnode_new 4-42
 - WTxformsepnode_new 4-43
 - WTzero 25-33
 - constant C-6
- ## X
- X constant C-6
 - X Resources 2-28
- ## Y
- Y constant C-6
 - Yon clipping
 - setting 17-19
- ## Z
- Z constant C-6
 - Z flashing
 - overlapping polygons 6-12
 - Z-buffer
 - error message D-3
 - rendering 6-13
 - roundoff 6-12
 - systems 6-12
-

NOTES

Notes

Notes

Notes