WorldToolKit™ Release 8 Technical Overview

*High-Performance Visual Simulation
Development Software*

February 1998

:
:
:
:
:
:
:
:

Table of Contents

1. Introduction to WorldToolKit

Overview

WorldToolKit (or WTK) is a portable, cross-platform development system for building high-performance, real-time, integrated 3D applications for scientific and commercial use.

WorldToolKit 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. WorldToolKit also supports network-based distributed simulations and an array of interface devices, such as head-mounted displays, trackers, and navigation controllers.

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 Technical Overview is designed to help you learn about WorldToolKit – its capabilities and uses.

What is WorldToolKit?…

Simply stated, you build your virtual world by writing code to call WTK functions. WorldToolKit 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 have real-world properties and behavior. You control these worlds with a variety of input sensors, from a simple mouse to position and orientation-sensing 6D input devices. Users can experience these worlds through a computer display (which acts as a movable window into a world) or by using a position-tracked, head-mounted, stereoscopic display.

WorldToolKit 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. Functions are included for device instancing, display setup, collision detection, loading object geometry from file, dynamic geometry creation, specifying object behavior, and controlling rendering.

Scene Graph Architecture

The architecture of WorldToolKit has been designed to incorporate the power of scene hierarchies. WTK lets you build a simulation by assembling geometry 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.

While WTK Version 2.1 (the fifth release of WTK) offered object hierarchies, later releases provide a much broader range of nodes which are hierarchically arranged in the scene graph and each of which represents part of the simulation. This efficient visual representation of the database 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 its location in the scene graph so that light affects only 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 calls for easily modifying and reconfiguring scene graphs.

In addition, WTK's companion product, World Up, provides an interactive scene graph builder and integrated Modeler which can be used to construct your scene graph and models.

What WorldToolKit Does

WorldToolKit 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 which 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 user viewpoint or objects in the universe are continuously changing.

WorldToolKit incorporates the philosophy of OpenVR™ , which means that it is portable across platforms, including SGI, Sun, HP, DEC, Intel, PowerPC, 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 is optimized to make full use of the unique capabilities of each platform to deliver the fastest graphics possible.

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) and to interface with a variety of information sources.

Features

Sound

WorldToolKit provides a cross-platform API for creating 3D and stereo sound. On NT systems, WTK supports Windows-compatible sound cards and Crystal River Engineering products, and on Silicon Graphics Workstations, WTK supports the SGI system audio and Visual Synthesis 3D sound products. *(Please contact Sense8 for more detailed 3rd 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 mono, stereo and quad systems.

User-Interface Objects

You can add graphical user interface (GUI) elements to your simulations by using the cross-platform user-interface objects. These objects let you quickly and easily create a (2D) graphical user interface. These user-interface objects have been designed in both Motif and Windows styles, to match the native operating system. The user-interface 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 GUI elements automatically change to match the new operating system. For example, if you develop it 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 WorldToolKit 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 (CAVE™).

VRML Support

WorldToolKit provides support for reading and writing VRML 1.0 files.

Other Features

Other features of WorldToolKit 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 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 New Sensors** - 5DT glove, CyberMaxx2, Thrustmaster T-2 steering wheel, Thrustmaster serial joystick, and Polhemus Insidetrak.

- **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++.

System Configuration

The WorldToolKit development system can be integrated with over fifty third-party products, ranging from compilers to high resolution display devices. With these products, WTK users have constructed a variety of systems, ranging from DOS-based Pentium computers to CAVE-like environments running multiple Silicon Graphics Onyx RE2s.

Basic System Configuration

A basic WTK development system includes the following components:

- Host computer(s)

- Hardware graphics accelerator board [system dependent]

- WTK library

- C compiler

- 3D modeling program

- Bitmap editing software [system dependent]

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 above, including a 3D or 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).

Input Sensors and Displays Supported

WorldToolKit supports a wide range of 3D and 6D input sensors and displays, both desktop devices and devices that can be worn for sensing position and orientation of various body parts of the user. Routines to read the input devices are part of the WTK library. The following are some sensors and displays that WTK supports:

- Standard mouse.

- Spacetec IMC Spaceball.

- Spacetec Spaceball Space Controller (NT only).

- CIS Geometry Ball Jr.

- Polhemus Isotrak, Isotrak II, and Fastrak sensors.

- Polhemus InsideTrak (NT only).

- Ascension BIRD, Extended Range BIRD, and FLOCK of birds.

- Logitech 3D mouse and head tracker and also the Magellan Space Control mouse.

- StereoGraphics CrystalEyes and CrystalEyes VR LCD shutter glasses with built-in Logitech tracker.

- Fake Space Labs monochrome BOOM, two-color BOOM2C, and full-color BOOM3C—button models and joystick models.

- Precision Navigation Wayfinder head tracker.

- Virtual i-O i-glasses! head tracker.

- Thrustmaster serial joystick.

- Thrustmaster T-2 steering wheel (NT only).

- Victormaxx Technologies Cybermaxx2 HMD (NT only)

- Precision Navigation 5DT serial glove.

- Fakespace Pinch glove.

- Virtual Technologies CyberGlove (some platforms only).

Geometries: Building Blocks of Your Virtual World

Objects are the building blocks of your virtual world. These objects may include geometries, sensors, lights, viewpoints, serial ports, and others. Although there are several different types of objects you will encounter in a typical WTK application, geometries are perhaps the most interesting since they provide form and function in your virtual world. Only Geometry objects (Geometry Nodes) are visible in the scene. Examples of Geometry objects are balls, chairs, platforms, vehicles, cylinders, wheels, houses, and landscapes.

Geometry Construction

You can construct geometries in several ways. You can use a modeling program to design a geometry and then load it into WTK, or you can use the special functions WTK provides for creating geometries. WTK lets you create spheres, cones, cylinders, 3D text, as well as geometry built from the ground up with vertex and polygon primitives.

WTK supports the following CAD and modeling programs: AutoCAD, Pro/Engineer, and any 3D modeling tool that can produce a DXF file. WTK also supports other file formats including 3D Studio (3DS), Wavefront (OBJ), MultiGen/ModelGen (FLT), and VideoScape (GEO).

WTK can read and write VRML (WRL) files. WTK can also read and write neutral file format (NFF) files, which are ASCII text files with an easy-to-read syntax, and binary NFF files.

Textures

WTK lets you apply textures to geometrical objects or to their underlying polygonal surfaces. By using textures, geometries appear more realistic and lifelike. Judicious use of texturing results in more complex and pleasing environments with simpler 3D models, because textures provide image detail that no longer must be represented by individual geometric surfaces. Texturing is valuable because while it provides greater image detail, less time must be spent on actual modeling; it also improves run-time performance beyond what would have been possible if all scene details had been modeled in 3D.

Obtaining and Applying Textures

You can obtain textures by using a capture board and a video camera, a scanner, a digital camera, or paint (bitmap image editor) program. Many textures are available for free on the WWW or can be purchased.

Typically, textures you acquire from either a capture board, camera, or scanner will require some pixel editing. This is done using a pixel painting program. WTK supports transparent textures, that is, you can specify WTK to treat all black pixels in a texture as if they were transparent.

Textures can be applied to polygonal surfaces interactively with WTK function calls, and can be interactively rotated, translated, scaled, or mirrored once they are applied.

Building a Virtual World

In a WTK simulation, you create individual moving parts as separate nodes, using a Geometry Construction function. For example, to model a car with rotating wheels, the geometry nodes used for wheels are constructed separately from the node representing the car body. You can construct the car with individual nodes, or use self-contained groups of nodes called movable nodes, which can be easily repositioned.

To create geometries for a simple world using a CAD or modeling program you typically construct the geometries in the same CAD model, and then write out separate files for each one.

For example, suppose you have a simple model of a room with four walls, a floor and a ceiling. In this room, you have modeled five geometries: a table, chair, book, computer and bookshelf. If you created a single .dxf file (in ASCII text format) of this model complete with

the five geometries and started the WTK application that read in this file using the *WTgeometrynode_load* command, you would be launched into your virtual world with everything in the room treated as a single geometry (including the table, chair, book, computer and bookshelf). This form of application is really a static "walk-through" program, where you could interactively edit the polygon attributes (such as colors and textures) of any surface, but you cannot move the parts of the geometry with respect to each other.

If each of the geometries which represent the table, chair, book, etc., were individually loaded into WTK, then it would be possible to move the objects in relation to one another using functionality which is available in WTK. For example, you could create and place transform nodes between the individual geometry nodes to control the position and orientation of the geometries.

WTK also has the higher level concept of a movable node, which combines the characteristics of several different node types (for example, a geometry node and a transform node) so that your simulation can make use of self-contained entities which are easy to move around.

The example code below is a WTK application that allows the user to fly around a spinning planet.

A Sample WTK Application

```
/* SAMPLE PROGRAM */
/*
* 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;
     WTsensor *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_rotateaxis(planet, Y_AXIS, 0.01);
}
```
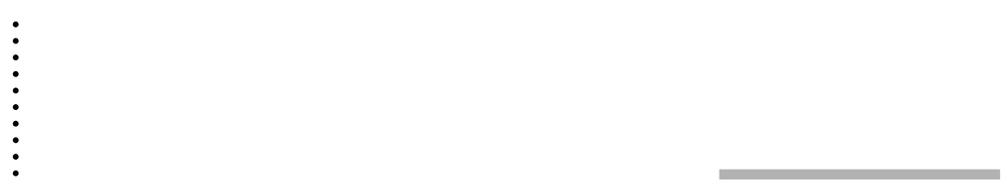
2. The Universe

The universe contains all graphical objects that appear in a simulation. These objects may include geometries, sensors, lights, viewpoints, serial ports, or other object types. Once these objects are created, they are automatically maintained by the WTK simulation manager.

Overview of the WorldToolKit Classes

WorldToolKit 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 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 placed in the 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 Nodes, and Switch Nodes are not visible, though they can affect the appearance of Geometry Nodes.

- *Vertices*: can be dynamically created along with the definition of a vertex normal for gouraud-shading.

- *Polygons*: can be dynamically created and texture-mapped using various sources of image data. Rendering is performed in either wireframe, smooth-shaded or textured modes.

- *Lights*: can be dynamically created or loaded from a file.

- *Viewpoints*: define the position and orientation 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 transforms, viewpoints, movables, etc. Multiple sensor objects are supported.

- *Paths*: objects or viewpoints can 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*: connects a *source* of position and orientation information with a *target* that moves to correspond with that changing set of information.

- *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*: this capability enables 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 method (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 methods defined in the WTK library.

All functions acting on a given class have by convention a name which 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 method 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);

Universe Construction and Destruction

The universe is the container of all WorldToolKit objects. Once these objects are created, they are automatically maintained by the WorldToolKit simulation manager. In a WorldToolKit 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.

Unlike the methods for other WTK objects, universe methods do not require a pointer as the first argument. This is because there is only one universe in existence at any given time.

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 *WTviewpoint_removesensor*. The *WTuniverse_delete* function also cleans up and closes the graphics hardware or WTK display.

Simulation Management

The simulation loop is the heart of a WorldToolKit application. Every aspect of the simulation takes place in the universe. The WTK 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 1 shows the default order of events in the simulation loop. The order can be changed by using the function *WTuniverse_seteventorder*.
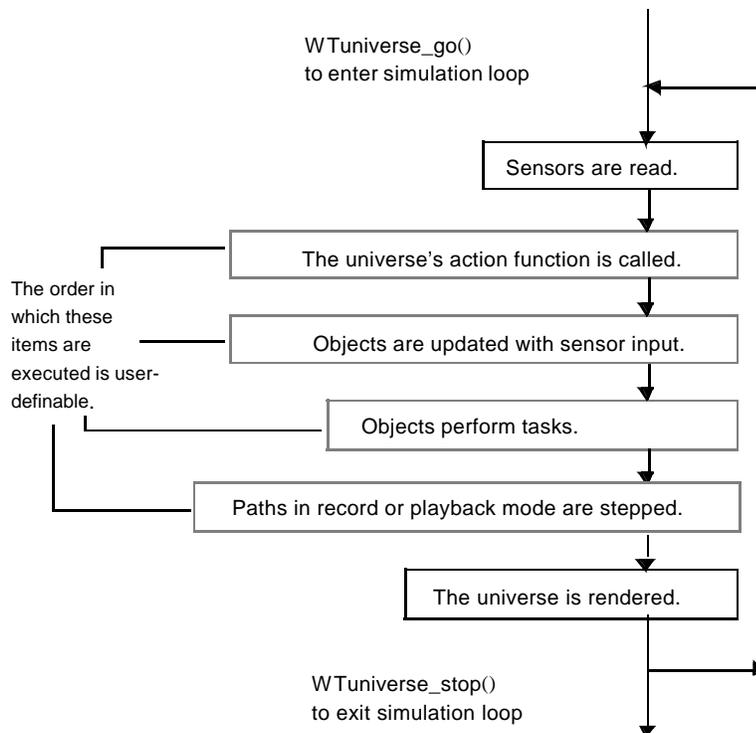
Figure 1: The default simulation loop.

The Universe Action Function

The universe action function is used to define and control the activity in the simulation. In the action function, actions involving any WorldToolKit objects, graphical or otherwise, can be specified. The action function is a user-defined function which is called by the simulation manager each time through the simulation loop. Figure 1 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 which you can specify in the universe action function are:

- Program termination by having a button press trigger a call to *WTuniverse_stop*.

- Simulation activities such as object manipulation, intersection tests, 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 interactively select a polygon, specifying what is to be done with the selected polygon, and processing keyboard input using *WTkeyboard* functions.

Other Global Concepts

Coordinate Systems

A coordinate system (sometimes called reference frame or context) refers to the X, Y, and Z coordinate axes used to describe position and orientation in space. Any location in space can be described by its XYZ coordinates relative to the origin. The origin is the center of the scene, which is located at XYZ coordinates 0,0,0. In a three-dimensional coordinate system, the X coordinate refers to the left-right location, the Y coordinate refers to the up-down location, and the Z coordinate refers to the near-far location.

The Scene Graph is a way of assembling objects hierarchically, so that the location of any object is relative to the coordinate system of its parent in the Scene Graph. For example, if you place an automobile Movable Geometry below a road Movable Geometry in the Scene Graph, then the translation property of the automobile specifies the position of the automobile in the reference frame of the road.

Universe Geometrical Properties

WorldToolKit functions provide access to three useful geometrical properties describing the graphical entities in a scene graph. These are the extents of these entities in the world coordinate frame, the midpoint of these extents, and the "radius" of this extents box. Figure 2 illustrates these properties.
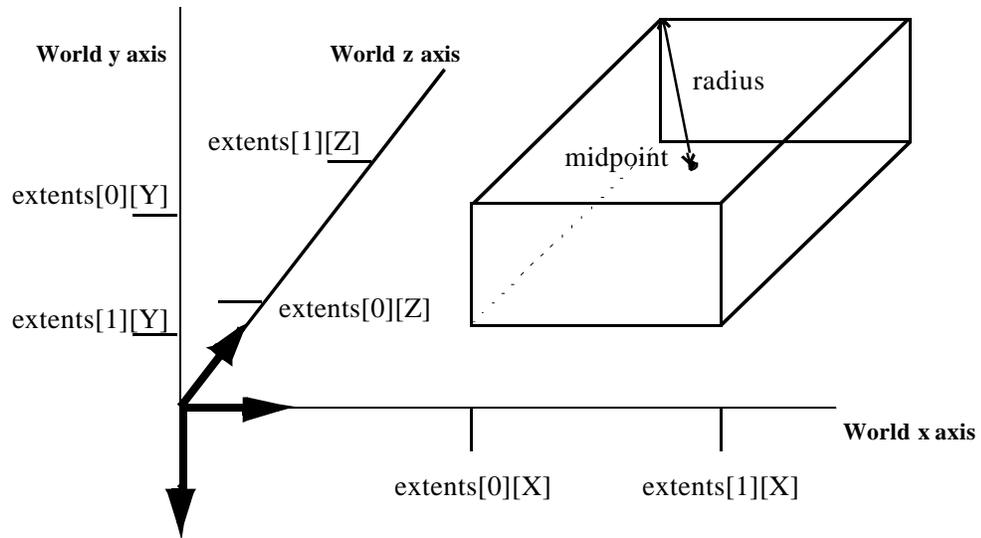
Figure 2: Scene graph geometric properties: extents, radius, and midpoint.

The extents box is the smallest world-coordinate frame aligned box which fits about the graphical entities in the scene graph. The radius is the distance from the midpoint of the extents box to one of its corners.

Accessing Objects in Your Simulation

You can access the objects in your simulation through a variety of functions in WorldToolKit. Generally, a call to the function which returns a pointer to the list of objects of a specified class is called first. These functions are *WTuniverse_getwindows*, *WTuniverse_getviewpoints*, and *WTuniverse_getsensors*, etc. Then to iterate through this list, the corresponding iterator functions *WTwindow_next*, *WTviewpoint_next*, or *WTsensor_next*, etc., are used.

Interacting with the Objects in Your Simulation

You can use the *WTwindow_pickpoly* function to select polygons in the scene graph associated with a window based on their projection into the 2D window or screen. *WTwindow_pickpoly* takes a 2D screen point and returns the front-most polygon at this point. You can modify the appearance of this polygon by passing it in to other functions, such as *WTpoly_settexture*.

Testing for Intersections

Many WorldToolKit applications require that the objects in the scene interact with each other. To do so often requires that an object know when it comes in contact with another object.

WTK provides functions which test for bounding box (extents box) intersections as well as intersections of polygons with other polygons. Keep in mind that polygon-level intersection testing, while more precise than bounding box intersection tests, is computationally intensive. You should use these functions only as often as needed and only when bounding box tests are insufficient.

3. Scene Graphs

The spatial organization and relationship of Node objects to each other is an important part of the simulation's content. In WorldToolKit, this organization and relationship is specified visually with the Scene Graph.

The Scene Graph is a hierarchical arrangement of nodes which you use to create a scene. Node objects make up a scene in a simulation. These Node objects can include Lights, Fog, Geometries, Groups, Transforms, Separators, and others, which are organized beneath a single Root node.

Figure 3 illustrates a simple scene graph used to display a car. The order in which nodes appear in the Scene Graph determines the order in which objects get rendered. This tree is traversed from top to bottom and left to right, in a "depth first" manner — the left-most branch is read completely before the next one is considered. Note that there is only one wheel in this scene graph; the second geometry node "Wheel" is just a reference to the first.
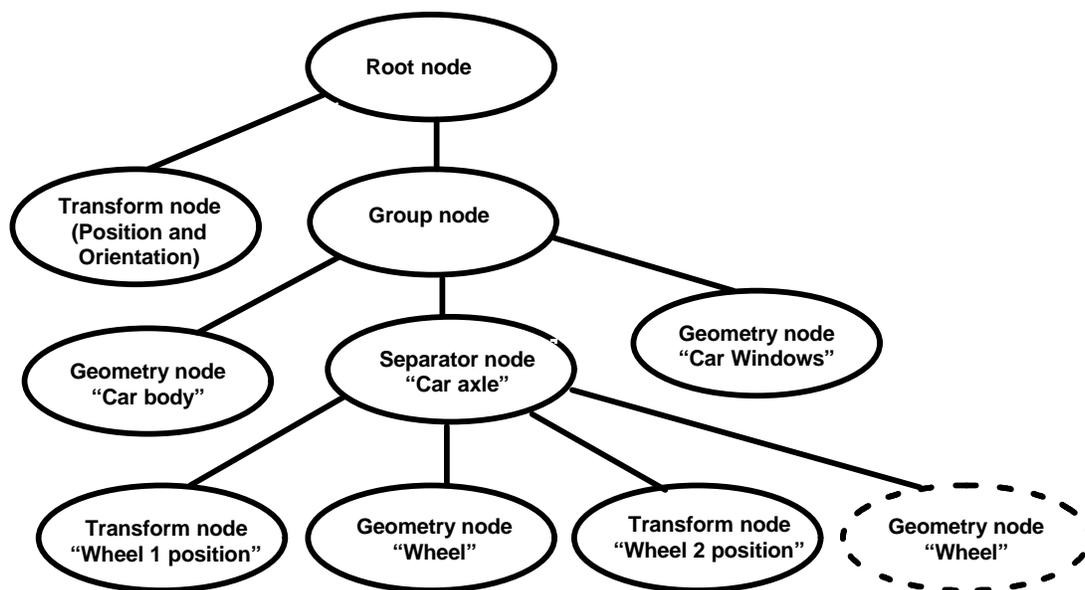
Figure 3: A simple scene graph.

Some nodes, like the Transform nodes, are used to affect the *state* of the scene you are creating. The state remains in effect until you modify it with another similar node. For example, the position and orientation set by the transform node in the upper-left quadrant of Figure 3 are used for this entire scene graph; the transform nodes used to display the wheels specify position and orientation relative to those initial settings.

Advantages of Scene Graphs

With earlier versions of WTK, programmers used the *WTobject_attach* function call to create hierarchical assemblages of graphical objects. This allowed child objects to move with — as well as independently of — the parent objects to which they were attached, so that, for example, the wheels on a vehicle would move when the vehicle moved, but could also themselves rotate independently of the vehicle.

WTK now supports a much more sophisticated scene database description, by using a scene graph. The following are some of the new features and advantages of the scene graph:

- Object grouping.

- Level of Detail switching.

- Instancing of geometry and entire scene graph sub-trees, providing better memory usage.

- More powerful culling and more efficient database representation, providing increased performance, i.e., frame rate.

- Support for VRML and MultiGen file formats.

- The enabling of lights, materials, and environmental effects in selective parts of the database.

- Multiple scene graphs.

Scene Graph Terminology

The following terms are important when learning how WorldToolKit implements scene graphs:

Scene graph tree	All of the nodes in a scene graph, arranged in a hierarchical order. The nodes in figure 4 are all in one scene graph tree.
Scene graph sub-tree	A node and all of its descendants. A sub-tree in figure 4 is shaded.
Ancestor	Since node A has a sub-tree that includes node E, A is an ancestor of node E. Note that node J is not an ancestor of node I.
Child node	A node's direct descendant. In figure 4, nodes B and C are both children of node A. J is not a child of A.
Parent node	A node's direct ancestor. Node A is a parent of node C, but not of node E. It is possible for a node to have several parents.
Sibling	Children of the same parent node are siblings. Nodes F, G, H, and I are siblings.
Root node	Each scene graph has only one root node. The root node in figure 4 is node A.
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 have been labeled so that their alphabetical order indicates the proper traversal order.

Predecessor   Since nodes B and C are processed before node J, they are its predecessors. A node's predecessors can affect the rendering of that node, even though they may not be ancestors.
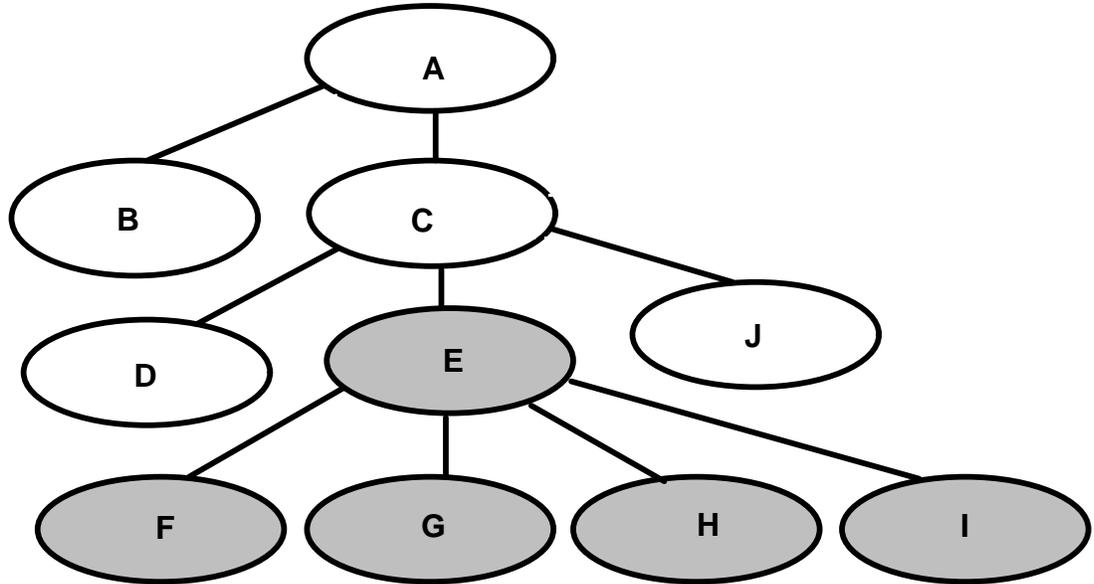
Figure 4: A schematic diagram of a scene graph.

Overview of Creating Scene Graphs

WorldToolKit provides two methods for constructing scene graphs. These two methods range from a high-level, automatic approach, providing the greatest ease of use, to a low-level approach which provides complete control of the scene graph on a node-by-node basis.

These techniques may also be combined. You can begin constructing a scene graph with one method and then decide to continue development using the other method. Regardless of how you construct your scene graph, WorldToolKit provides additional function calls for disassembling, reassembling, and rearranging it at any time.

You can create a scene graph by doing the following:

- Loading in a file that contains a scene graph description. VRML and MultiGen can create such files. For example, to read in a file named "myworld" and attach it to the root node, you would call *WTnode_load(root, "myworld", 1.0).*

- Using the WTK constructor and scene graph assembly functions to construct the graph node by node. This is the lowest-level approach and provides the most control. For example, you can create an LOD node and attach it as the last child node of the root node by calling *WTlodnode_new(root).*

Nodes Used in WTK Scene Graphs

Although there are many node types, nodes can be grouped into three classes:

- *Geometry nodes*, which are used for visible entities.

- *Attribute nodes*, which are used to affect the way geometry nodes are rendered.

- *Procedural nodes*, which are used to control the way a scene graph is put together and processed.

In WTK, 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.

Geometry and Attribute Nodes

Table 1 below lists geometry nodes and the set of attribute nodes which affect scene graph state.

Node	What it does
Geometry	Displays a set of polygons together with a WTK material. (The Geometry node does not affect scene graph state.)
Fog	A special effect that you can use to simulate fog, smoke, haze, smog, mist, etc.
Light	Specifies a WTK light. Light nodes illuminate geometry in a scene and are part of a scene graph's *state*. In other words, when a scene graph is traversed, as it is for rendering, geometry is lit according to the lighting state at the time a WTgeometry node is reached.
Transform	Provides position and orientation information.

Table 1: Geometry nodes and nodes that affect scene graph state.

Procedural Nodes

Table 2 lists procedural nodes that you use to organize the nodes in a scene graph.

Node	What it does
Anchor	A group node where children are read from a file. A user interaction is required before an anchor node's children are read in.
Group	Has child nodes, but no other properties.
Inline	A group node whose children are read from a file without user interaction.

Level of Detail (LOD)	You use an LOD node so that WTK can dynamically select between a set of different representations, each of which is a different level of detail.
	This is useful, for example, if your application involved a train that passed close to the viewer and then receded into the distance. After you created an LOD node, you would add several children to it– each of which was 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. Of course, your image doesn't have to be moving. LOD nodes are useful whenever the distance between the viewpoint and a geometric object will vary.
Root	Is the top node in a scene graph. Each scene has only one root node, which is not shared with any other graph. As the top node in its hierarchy, this node has no parent node.
Separator	Prevents state information from propagating from its descendant nodes to its sibling nodes. This is useful when you have lights and or transformations in one part of your scene graph and you do not want them to affect the remaining portion of the scene graph. Separator nodes also allow you to perform a quick "reject" test on the extents box of the separator node's sub-tree, which can result in a drastic improvement in performance.
Switch	Allows you to determine which of several children is processed. This is particularly useful if you are creating an animation sequence, in which case you could cycle through the switch node's children, switching once for each frame.
Transform	Consists of a 4x4 matrix which allows you to specify position and orientation information. You can set or retrieve various properties of transform nodes, like translation and rotation (orientation), by calling WTK functions.
Transform separator	Prevents just the transformation state from propagating from its descendant nodes to its sibling nodes. All other states are allowed to propagate.

Table 2: Procedural nodes.

Node Properties

Certain node properties are generic in that they can pertain to all node types. Such 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.

Node Geometrical Properties

WorldToolKit functions provide access to three useful parameters that describe the space occupied by the geometries in a scene graph. Figure 2 illustrates these parameters: the extents of the rectangular box ("the extents box") that encloses these geometries, the midpoint of the box, and its "radius."

The extents box of a node in a scene graph is relative to the coordinate system (the X, Y, and Z axes) defined by the transformations accumulated by traversing the scene graph as far as that node. A node's extents box encloses the geometries beginning at that node and including all nodes which are its children and grandchildren, etc. These comprise the node and its sub-tree. The radius is the distance from the midpoint of the extents box to one of its corners.

The extents box of the root node in a scene graph encloses all of the geometry referenced by the graph. The concept of an extents box is extremely useful because it allows WTK to quickly determine whether the geometry in a node sub-tree is in view or not.

How Scene Graphs are Rendered

Each WTK window has a scene graph associated with it. This graph is rendered into the window automatically as the simulation runs. WTK windows can each reference the same scene graph or use different ones. When windows reference the same scene graph, they can provide different views into the same scene. When they reference different scene graphs, they provide views into different scenes.

Different scene graphs may have common sub-trees, which means that the same geometry may be referenced by more than one scene graph.

Each scene graph has a single root node. When WTK processes a scene graph, it starts at the root node and traverses the tree top to bottom and left to right, so that the left-most branch (sub-tree) is completely processed before the next sub-tree is processed.

During the traversal, WTK evaluates each type of node and processes it. Depending upon the type of node encountered, WTK will do one of three things, as outlined in table 3 below:

Kind of Node	What WTK does when it encounters one of these nodes
Procedural	Processes the children of this node, depending on the type of traversal dictated by the node.
Attribute	Modifies the current state, which determines the appearance of subsequent geometries.
Geometry	Draws the specified set of polygons.

Table 3: Processing nodes in the scene graph.

Scene Graph State

The attribute nodes listed in table 3 affect the *state* of a scene graph, how your geometry is being rendered at any particular point in the graph. A light node placed just under the root node, for example, dictates the light state of the entire scene graph, until another light node is used to modify that state. Similarly, transform nodes affect the transform state (i.e., position and orientation) of geometries, until another transform node is used to modify the transform state.

Managing State

Certain nodes, known as separators and transform separators, are provided to help you manage state by isolating the effects of any of the attribute nodes. Note that neither of these separators actively modifies the state of a scene graph; they merely prevent the descendant attribute nodes from affecting the state of sibling and ancestor nodes.

In figure 3 on page 15, the position and orientation information contained in the first transform node below the root affects the entire scene graph. However, the separator node called "Car axle" prevents the information in the transforms below it from propagating upward, thus ensuring that the geometry node "Car windows" is affected only by the original transform node.

Uniquely Identifying Nodes in the Scene Graph

It is possible for a node to be referenced more than once in a scene graph. In other words, a node can occur as a child of multiple parent nodes. It is even possible for a node to occur multiple times as the child of the same parent node. Because of this, a mechanism is needed to uniquely specify a particular occurrence or instance of a node in the scene graph.

WTK uses the concept of a node path to uniquely identify nodes in the scene graph. A node path is a mathematical entity that allows you to distinguish between multiple occurrences of the same node due to instancing.

If you create a geometry node, for example, 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, and also re-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" or instance of that geometry— remember, there is only one such geometry— depends on the path you take through the scene graph tree to reach it.

Recall that the scene is rendered by starting at the root node and proceeding down the first child recursively. Consequently, if you want a node path to the instance of the geometry following the transform, you can create it by calling *WTnodepath_new* and specifying the geometry as the node, the scene graph's root node as the ancestor, and indicate the "second" path with a 1 (arrays start at 0). The newly created node path can then be used to uniquely specify a particular instance of a node.

Node Paths can be useful regardless of whether a node is referenced once or several times in the scene graph. Node paths enable you to do the following:

- Perform intersection tests between a specific instance of a node and other nodes in the scene graph.

- Pick graphical entities rendered into WTK windows. The WTK picking functions generate the node path of the picked geometry node.

Sample Node Functions

Table 4 below lists some sample node functions:

Function	Description
WTrootnode_new	Creates a new root node.
WTnode_save	Saves the data contained in a node sub-tree to a file.
WTnode_boundingbox	Allows you to highlight parts of your scene with a bounding box.
WTnode_addchild	Attaches a node as a child of another node.
WTnode_getparent	Returns the nth parent node of the specified node where n is a parameter of this function.

Table 4: Sample node functions.

4. Movable Nodes

A movable node is a self-contained entity; it is a combination of a separator node, a transform node, and a content node (such as a geometry node). Movable nodes are easy to move around in a scene graph and retain a sense of how they will behave. They also simplify the scene graph construction process.

What Makes Up a Movable Node?

As shown in figure 5, the three basic components of a movable node are a separator, a transform, and a content.
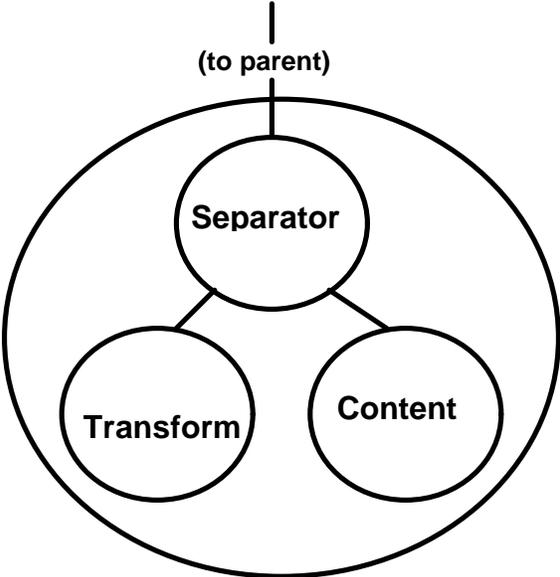
Figure 5: The basic structure of a movable node.

Separator

A separator prevents the transformation within a movable from affecting sibling nodes.

Transform

Each movable node has a transformation component which allows you to control the position and orientation of a movable node. This means that you do not have to explicitly create a transformation node which is a predecessor to the movable node in order to control the position and orientation of the movable node, since a transformation component is built into

every movable node. To set the position and/or orientation of a movable node, you can use any of the WTK position and orientation functions which are applicable to transform nodes.

By default, the WTK functions used to position a geometry use the geometry midpoint, the midpoint of the geometry's extents box, as the location of the geometry moved to the specified position. For example, when *WTnode_setposition* is called, the geometry is translated so that its midpoint is placed at the 3D world coordinate location passed in to this function.

The following example shows how a movable geometry node can be created and positioned at the world coordinates (100.0, 0.0, 0.0) given that the corresponding WTgeometry has previously 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);
```

Content

The content controls what the movable displays or accomplishes. The five types of content components are:

- **Geometry**: a series of vertex positions and polygon definitions.

- **Light**: a defined source of illumination.

- **Separator**: prevents state information from propagating from its descendant nodes to its sibling nodes.

- **Switch**: a group that allows the user to control which of its children is in the simulation at any given time.

- **Level of Detail**: a switch that chooses the active child automatically, based on the distance to the viewpoint.

Movable Node Hierarchies

A movable hierarchy is a group of nodes which moves together as a whole but whose sub parts can move independently. As an example, consider a hierarchically assembled robot arm such as is illustrated in figure 6.
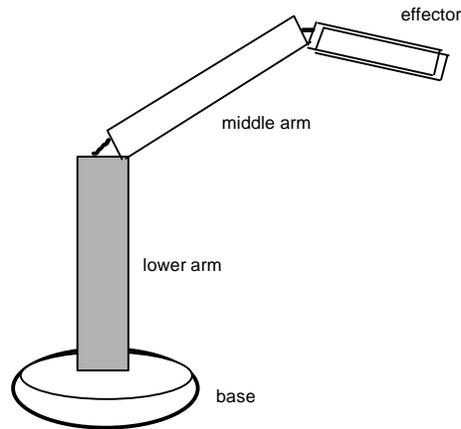
Figure 6: Hierarchically assembled robot arm.

Each part of the robot arm— the base, the lower arm, the middle arm, and the effector— must be created as a separate node, using, for example, the function *WTmovnode_load*. Let's say that pointers to these four movable nodes are called base, lower, middle, and effector. Then to assemble the robot arm as in the preceding diagram, you would make the following calls to *WTmovnode_attach*:

    WTmovnode_attach(base, lower);
    WTmovnode_attach(lower, middle);
    WTmovnode_attach(middle, effector);

These calls result in a geometry hierarchy in which base is the root, and moving down through the hierarchy you find lower, then middle, and then effector. (Don't be confused by the fact that "down" in the hierarchy corresponds to "up" in figure 6) When a geometry in the hierarchy moves, it moves all of the geometries that are hierarchically below it, as if the geometries were rigidly attached.

Geometries that are hierarchically above the 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.

Any type of sensor could be attached to the various robot arm segments to cause the arm to move. Most likely, only rotational input from these sensors would be applied to the robot arm segments so that each segment would simply rotate and not become detached from the arm segment hierarchically above it.

For example, if using a Spaceball to control part of the robot arm, you could constrain the motion link (discussed in Chapter 11) connecting the device to the robot arm to return only rotations with these calls:

```
WTmotionlink *link;
WTmotionlink_addconstraint(link, WTCONSTRAIN_X, 0.0, 0.0);
WTmotionlink_addconstraint(link, WTCONSTRAIN_Y, 0.0, 0.0);
WTmotionlink_addconstraint(link, WTCONSTRAIN_Z, 0.0, 0.0);
```

Sample Movable Functions

Table 5 below lists some sample movable functions:

Function	Description
WTmovgeometrynode_new	Creates a movable geometry node from the existing geometry and adds it to the scene graph after the last child of the parent you specify.
WTmovlightnode_newpoint	Creates a movable point light node and adds it to the scene graph after the last child of the parent you specify.
WTmovnode_rotateaxis	Rotates a movable node in its local frame.

Table 5: Sample movable functions.

.

5. Geometry Nodes

In WTK, three-dimensional graphical entities are rendered to the screen as the scene graph is traversed and geometry nodes in the scene graph are encountered. Geometry nodes contain a pointer to the actual geometry (or WTgeometry structure), which consists of a set of polygonal faces.

You can create WorldToolKit geometries using the following approaches:

- WTK's neutral file format (NFF) import facility.

- The Sense8 World Up Modeler that ships with WTK.

- A CAD or other modeling program such as AutoCAD, with the 3D geometry written out in one of the formats supported by WTK.

- WTK's functions for dynamically constructing predefined geometry types such as cylinders, blocks, cones, truncated cones, spheres, hemispheres, and extrusions.

- WTK's polygon and vertex functions for dynamically constructing custom graphical objects.

- WTK's functions for creating 3D text objects.

- Copying an existing geometry with *WTgeometry_copy*, with the option of modifying this copy by using WTK's polygon and vertex editing functions.

File Formats Supported by WTK

The following file formats containing descriptions of 3D geometry and attributes can be read by WTK. Geometrical entities are constructed when you call *WTgeometrynode_load*.

- **Autodesk (DXF) file format**: This common format is generated by many 3D modeling programs. WTK can also output files in DXF format.

- **Wavefront (OBJ) file format**: This is generated by the Wavefront modeling tool. WTK imports the 3D polygonal geometry and curved surfaces which have been polygonalized. Vertex normals and texture vertices are supported for gouraud shading and texture draping.

- **Autodesk 3D Studio mesh (3DS) file 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.

- **Pro/Engineer RENDER (SLP) file format**: WTK reads the facets in an SLP file as colored polygons with vertex normals for smooth shading. An SLP file contains only one geometry.

- **MultiGen/ModelGen Flight (FLT) file format**: WTK supports textures, subfaces, external references, transforms, instances and replicas. An FLT file can contain multiple objects.

- **VideoScape (GEO) file format**: This is a simple 16-color format in which all polygons are backface-rejected. A GEO file describes a single geometry.

- **WorldToolKit neutral file format (NFF) and binary (BFF) file format**: 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. An NFF or BFF file can contain multiple geometries.

- **VRML (WRL) file format**: WTK can read and write VRML 1.0 files.

Many other file formats can be loaded into WorldToolKit through the use of third-party geometry conversion programs capable of writing formats which WorldToolKit can read. A program such as KANDU software's CADMOVER reads and writes most popular 3D file formats.

Modeling Considerations

The way in which geometrical entities are modeled affects the appearance as well as the run-time performance of a simulation. There are several important factors you should consider when modeling geometry for use in a WTK application.

Constructing a World with Multiple Objects

Using a CAD program, you can create a graphical environment for your WorldToolKit application in which the various graphical entities have the desired spatial relationships. One technique for accomplishing this is to initially build all of the geometry into one CAD file, positioning the various entities as desired, and then to save out each portion of the model from which you wish to create a separate graphical entity into a separate file. Alternatively, you can save them out as separate objects in a single multi-object file.

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, suppose that you want only the chair and the book to be movable (dynamic) objects. What you would do is construct the model containing all of these components and then save it out to a file. Then to create the file which contains the stationary scene, you would start from the original file, erase the book and the chair, and then save to file the resulting model which contains just the walls and the desk. Similarly, to create the file for the chair, you would load in the original file, erase the walls, desk, and book, and then save the result to a separate file. 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 which you wish to load in separately to WorldToolKit on a separate layer. Once your model is constructed, you can successively, for each object, turn off all layers but the one that the object is on and save out the model to 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.

The following are a few important points about Gouraud 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.

WorldToolKit automatically uses Gouraud shading when rendering geometric objects which contain vertex normals.

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, and Pro/Engineer RENDER (.slp) files.

- Enter them yourself in an NFF file (this is difficult).

- Use the NFF automatic-normal-generation feature to make them for you.

- Call a geometry constructor such as *WTgeometry_newsphere* or *WTgeometry_newcylinder* with the gouraud parameter set to TRUE.

- Create your own objects in your application code and set vertex normals with *WTgeometry_setvertexnormal*.

Vertex Colors and Radiosity

As with gouraud shading, you can use vertex colors to increase the visual realism of your virtual scene.

For example, vertex color support enables you to render models which have been radiosity preprocessed. A radiosity-preprocessed model stores lighting information such as shadows and reflections as vertex colors. Since this lighting doesn't have to be computed at run-time, you can achieve complex lighting with real-time performance.

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 which computes the appropriate vertex colors, and then pass them to WTK.

Vertex colors can be set for geometry in the following ways:

- In an NFF file.

- Using a radiosity preprocessing program. ATMA's program called Real Light is a radiosity preprocessor which reads and writes NFF files.

- With the function *WTgeometry_setvertexmatid*.

Backface Rejection

Another important modeling consideration is backface rejection. By eliminating the rendering of polygons which face away from the viewer, frame rates can be significantly increased.

In WorldToolKit, the front face of a polygon is the side of the polygon for which the vertices are ordered counterclockwise. It is also the side from which the polygon normal points.

WTK's NFF format is very flexible in its support for specifying the back face and front face of polygons and whether they are backface rejected. The keyword "both" can be used 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.

The function *WTpoly_setbothsides* 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*, programmatically with the functions *WTgeometry_getpolys* or *WTgeometry_id2poly*, or with the constructor function *WTpoly_begin*.

Coplanar Polygons

When building models it is best to avoid the use of coplanar polygons or surfaces— that is, surfaces which 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 can result in image flashing between coplanar surfaces where they overlap.

To avoid this problem, your model should be constructed 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. How far separated the planes must be to avoid flashing depends on the resolution of the Z-buffer and on the locations of the window's hither and yon clipping planes.

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.

Sample Geometry Functions

Table 6 below lists some sample geometry functions:

Function	Description
WTgeometry_newcone	Creates and returns a new cone geometry.
WTgeometry_newtext3d	Creates and returns a new 3D text geometry.
WTgeometry_copy	Creates and returns a new geometry by copying an existing geometry.
WTgeometry_begin	Starts the construction of a custom geometry.
WTgeometry_getradius	Returns the radius of the specified geometry.
WTgeometry_save	Saves a geometry to file.

WTgeometry_setvertexrgb	Assigns a color to a vertex of a geometry.
WTgeometrynode_new	Creates a geometry node.

Table 6: Sample geometry functions.

Polygons

Geometries in WorldToolKit are made up of polygonal surfaces which can be colored, shaded, and textured. These polygons are created automatically when geometries are constructed with functions such as *WTgeometry_newcone* and *WTgeometry_newtext3d*. Polygons can also be constructed explicitly, vertex by vertex.

Other functions enable you to modify polygon attributes, access their geometrical properties and vertices, define polygon ID's, and test for intersections of polygons with other graphical entities.

Nearly all of the polygon functions take a pointer to a polygon. Polygon pointers are obtained in a variety of ways. They can be obtained interactively, using *WTwindow_pickpoly*; through polygon ID values using *WTgeometry_id2poly*; using the polygon access functions *WTgeometry_getpolys* and *WTpoly_next*, or with the dynamic constructor function *WTgeometry_beginpoly*.

WTK provides polygon functions that allow you to do the following:

- Set, get, and change polygon attributes, such as color, normals, and material index.

- Assign polygon ID numbers.

- Iterate through a list of polygons.

- Dynamically construct geometries from vertices and polygons.

- Delete polygons.

- Test for intersections of polygons with other polygons.

Sample Polygon Functions

Table 7 below lists some sample polygon functions:

Function	Description
WTpoly_setrgb	Specifies the 24-bit color value of a polygon.
WTpoly_setmatid	Changes the index into the current material table.
WTpoly_setbothsides	Specifies whether both sides of a polygon are visible.
WTpoly_addvertex	Adds a vertex to a polygon.
WTpoly_intersectpolygon	Tests whether two polygons intersect.

Table 7: Sample polygon functions.

6. Materials and Textures

A material is a combination of light and color attributes that you use to define the appearance of a geometry or collection of geometries. WorldToolKit functions let you create, edit, and save material information.

Surfaces of objects in the real world are not smooth and featureless— they have pattern, grain, and detail. To emulate this, WorldToolKit lets you give polygons a surface texture. For example, instead of depicting a table top with a uniform brown shaded polygon, WTK's texturing capability lets you create a table top with an actual wood-grain image mapped onto it.

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. Each type of color has three components (red, green, and blue), where each component is a floating point number between 0.0 and 1.0.

Each material has 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.*

Specular *The color reflected from the highlights of the geometry. The specular material property is what makes a geometry appear to by "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).*

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.*

Emissive   *The color of light produced (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. It is specified in red, green, and blue floats in range 0 to 1.*

Opacity/
(Translucency)   *The extent to which the color value of a pixel is combined with the color value behind it. Opacity ranges from 0 to 1 (float), where 0 is completely invisible and 1 is completely opaque.*

An object does not need to have all of its material properties specified. Using fewer fields can generate a moderate improvement in performance.

Using Existing Materials

You can obtain materials by reading in files from a modeler which specifies material properties in its export file format. Wavefront's .obj, 3D Studio's .3ds, and VRML's .wrl formats all have material information in them and are supported by WTK. When the file is read in, the geometry is automatically rendered using the modeler-specified material properties. For example, a material will look shiny in WTK if it looked that way in the modeler.

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.

Material tables are indexed from 0 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. Once a geometry file has been loaded into a scene, you can use the material table functions 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 may be affected when you modify a material.

To create a new material and then modify the copy, use *WTmtable_copyentry* and then *WTmtable_setvalue*.

Sample Material Functions

Table 8 below lists some sample material functions:

Function	Description
WTmtable_merge	Merges two material tables and returns a new material table which contains the materials from both of the tables.
WTmtable_save	Writes a material table to a file.
WTmtable_getnumentries	Returns the number of table entries contained in the material table specified by mtable.
WTmtable_load	Reads a material table from the specified filename.

Table 8: Sample material functions.

Adding Textures to a Polygon

Textures add to the realism of geometries. Pattern, grain, and detail on the surface of a polygon give it texture. You can create textures with a bitmap image editor or derive them from video images. Basically, anything on a computer screen can be converted to a texture format. WTK supports RGB, TGA, and JPEG texture files.

Texture makes an object look more authentic, more real. The more realistic the image, the more pleasing the environment. A virtual world becomes "virtual reality" when it succeeds in momentarily suspending the user's disbelief. Textures play a significant role in this reality.

Judicious use of textures increases 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 all of the windows of a distant building with 3D details, you can apply a digital image of a real building to a single polygon, which then serves as an entire side of the building. Modeling labor is conserved and rendering speed is increased dramatically compared to what would have been necessary to model all of these details in 3D.

Texture Application

WTK provides functions that let you drape textures over geometries and/or apply textures to individual polygons. The functions *WTpoly_settexture* and *WTgeometry_settexture*, automatically compute texture coordinates for geometries and polygons, thereby providing automatic texture application to geometric entities. WTK also provides functions such as *WTpoly_settextureuv* and *WTgeometry_settextureuv* which allow you to specify how the texture is to be applied to the polygon or geometry.

Texture Manipulation

You can modify textures using any of the texture manipulation functions. WTK texture manipulation functions let you modify the texture application using intuitive calls to translate, rotate, scale, etc., the texture on a polygon. WTK internally modifies the polygon's texture uv values when these functions are called. WTK also lets you modify the texture application by accessing the texture uv information directly.

Texture Filtering

Depending on their distance from the viewpoint, polygons appear at different sizes in your simulation. 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.

A large texture applied to a small polygon (in terms of display size) or a small texture applied to a large polygon can also produce unwanted results. WTK can eliminate these undesired effects by performing additional computations. During this processing (called *filtering*) the texture is scaled down to a size which is appropriate for the polygon's display size.

The function *WTtexture_setfilter* lets you specify the manner in which you want WTK to filter the texture that has been applied to a polygon. You can use the function *WTtexture_setfilter* to specify the quality of the filtering desired– higher quality requires more computations (and rendering time).

Sample Texture Functions

Table 9 below lists some sample texture functions:

Function	Description
WTtexture_load	Reads in a texture bitmap file.
WTtexture_replace	Dynamically replaces the image associated with a texture.
WTpoly_settexturestyle	Allows for shaded, transparent, and blended texturing.

Table 9: Sample texture functions.

7. Sensors

Sensor objects in WorldToolKit generate position, orientation, and other kinds of data by reading inputs which 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 WorldToolKit application to be directly coupled to the viewpoints, graphical objects, and lights in the universe.

WorldToolKit supports many of the 3D and 6D (position/orientation) sensors that are available. There are two principal classes of such sensors: desk-based sensors and sensors that are worn on various parts of 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.

In the former category are conventional devices, such as the mouse, joystick and isometric balls such as the CIS Geometry Ball Jr. and Spaceball Technology's Spaceball that respond to forces and torques applied by the user. Using such devices, a 3D object can be directly manipulated, displaced or rotated, with the ball acting as if directly connected to the object. Ball sensors are also useful for moving the viewpoint, by applying displacements and rotational forces to move and rotate the viewpoint.

The second category of sensor (sensors generating absolute records) includes electromagnetic 6D trackers such as the Polhemus Fastrak and Ascension Bird. This type of sensor can be used for viewpoint tracking when affixed 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 the underlying hardware technology by which they operate, WorldToolKit's sensor objects are treated homogeneously and can be used interchangeably in an application. Once a sensor object is created, it is automatically maintained by the simulation manager, as are the objects to which the sensor is attached. In this way, the developer does 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.

Supported Sensors and Displays

The WTK library directly supports a wide variety of devices. See page 5 in Chapter 1, "Introduction to WorldToolKit."

Sensor Object Construction and Destruction

You can create sensor objects with either the generic sensor constructor function *WTsensor_new* or with one of WorldToolKit's device-specific constructor macros such as *WTspaceball_new*, *WTfastrak_new*, or *WTbird_new*.

To consider a specific case, the device-specific constructor function for the Spaceball is a macro defined as follows:

    #define WTspaceball_new(port)

        WTsensor_new(WTspaceball_open, WTspaceball_close, \

            WTspaceball_update, WTserial_new(port, 9600))

To use this macro, you would make the call:

        WTsensor *ball;

        ball = WTspaceball_new(SERIAL1);

where the constant SERIAL1 is defined on all systems for portability.

All of the device-specific constructors are simply macro calls to *WTsensor_new*, which takes as its first three arguments, pointers to WorldToolKit functions that respectively open, close, and update the particular device.

Sample Sensor Functions

Table 10 below lists a some sample sensor functions:

Function	Description
WTsensor_setsensitivity	Sets the sensitivity value of the sensor. A sensor's sensitivity value defines the maximum magnitude of the translation input from the sensor along each axis, in the same distance units as the 3D geometry making up the virtual world.
WTsensor_setangularrate	Sets the scale factor for rotational records. The angular rate is the maximum rotation (in radians) around any axis that a sensor will return in any pass through the simulation loop.
WTsensor_getmiscdata	Returns a short in which miscellaneous data pertaining to the sensor has been stored. For example, button press information is retrieved this way.
WTsensor_getrawdata	This function returns the sensor-specific raw data structure. This should be typecast into an appropriate value for a user or WorldToolKit-defined sensor. For example, for the mouse as implemented in WorldToolKit, the raw data is WTp2 containing the current mouse cursor position in screen coordinates.

Table 10: Sample sensor functions.

Functions for Writing Your Own Sensor Driver

WorldToolKit includes a complete set of functions for writing your own sensor driver. These provide a hardware-independent interface to various input and output devices. Table 11 below lists a few of these functions:

Function	Description
WTsensor_setrecord	Use this function to store the current relative position and orientation record with your sensor.
WTsensor_setmiscdata	Use this function to store miscellaneous sensor data with the sensor object.
WTsensor_setupdatefn	Use this function to change a sensor's update function. A sensor object's update function is initially set in the sensor object constructor function *WTsensor_new*.

Table 11. Sample WorldToolKit functions for writing sensor drivers.

An Example of a Sensor Driver: The Mouse

WorldToolKit provides the following sensor driver functions for using the mouse:

- A function for opening the mouse device.

- A function for closing the mouse device.

- Several update functions.

These functions are only used when calling *WTsensor_new* to create a new mouse sensor object, or when calling *WTsensor_setupdatefn* to change the mouse's update function.

When creating a mouse sensor object, you can use one of the update functions provided (such as *WTmouse_moveview1* or *WTmouse_moveview2*), or you can write your own. Your update function should first call *WTsensor_getrawdata* to obtain the raw mouse record. 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*.

WTmouse_moveview2 is a mouse update function which is useful for moving a viewpoint through a 3D environment. *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 further away from the middle of the screen, the faster the movement.

WorldToolKit Technical Overview

8. Lights

WorldToolKit supports several types of lighting: ambient, directed, point, and spot. Each type of lighting illuminates geometries in a different way.

Introduction to Light Nodes

WTK supports the following four types of light nodes:

- **Ambient Light Node**

  Ambient light is background light which illuminates all polygons equally regardless of their position or orientation.

- **Directed Light Node**

  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**

  Point light emanates radially from the light position.

- **Spot Light Node**

  Spot light emanates radially from the light position, within a cone of specified angle centered about the spot light direction.

Light Node Attributes

Excluding ambient light nodes, all other light nodes display three types of color: ambient, diffuse, and specular. Once you create a light node, you can set these color attributes for it—or accept existing default values if you are not concerned about a particular type of color. If you are not concerned about the ambient and specular components of a particular light, the easiest way of setting the light's color is to specify a diffuse color value, leaving the other colors in the light set to 0.

There are many attributes available for different types of lights, however, not all of these attributes are applicable to all types of light nodes. Table 12 lists the full set of attributes available for modifying WTK light nodes:

Attribute	Description
Position	The location of the light in 3D space, as affected by any existing transformation.
Direction	The direction of the light rays, as affected by any existing transformation.

Intensity	The brightness of the light, with a maximum value of 1.0.
Ambient color	The color of the portion of the light which illuminates all polygons equally regardless of their position or orientation.
Diffuse color	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.
Specular color	The color of the highlights that are reflected off a shiny surface.
Attenuation	Degree to which light intensity decreases with increasing distance from the position of the light.
Angle	The half-angle of the spot light cone. This attribute only applies to spot lights.
Exponent	Specifies how the intensity of a spot light falls off within the spot light cone. This attribute only applies to spot lights.

Table 12: Attributes available for modifying light nodes.

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 which affect the polygon. 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.

Basic Light Management

WTlightnode_newdirected creates a new directed light and adds it to the scene graph, returning a handle to the new light. The intensity of the light should be between 0.0 and 1.0. *WTlightnode_load* reads in a list of light descriptions from an ASCII file and creates the corresponding light node objects. Each light must be specified on a separate line of the file. For each light, seven floating point values must be specified: the X, Y, and Z coordinates of the light position, the X, Y, and Z coordinates of the light direction, and the light intensity. Depending upon the type of light being defined, other values may also be specified, such as a spot light half angle, attenuation values, and ambient, diffuse, and specular color components.

Performance

The maximum number of non-ambient (directed, point, and spot) lights which may be added to the simulation is system dependent. However, the greater the number of lights, the greater the performance impact of lighting computations. The time it takes 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, it's better to do so with a call to *WTnode_enable* (with the enable flag set to FALSE) to disable the light node than by setting the light's intensity to 0.0 using *WTlightnode_setintensity*. In the former case the light is disabled from the simulation and no longer enters into shading computations; in the latter case the light remains part of the simulation. You can also remove a light node from a simulation by detaching the node from the scene graph.

Other Important Aspects of Lights

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. Ambient and directed lights do not attenuate with distance from the polygon while point and spot lights may attenuate with distance. Also, there is no limit on the number of lights that you may add to the simulation. By default, a simulation always contains a white ambient light whose intensity is 0.4.

Light Functions

Table 13 below lists some of the functions available for manipulating lights:

Function	Description
WTlightnode_newdirected	Adds a new directed light to the scene graph.
WTlightnode_load	Reads in light settings from an ASCII file.
WTlightnode_setambient	Sets ambient components of a light's color.
WTlightnode_setdirection	Sets the direction a particular light points to.
WTlightnode_setposition	Sets the location of a particular light.

Table 13: Sample light functions.
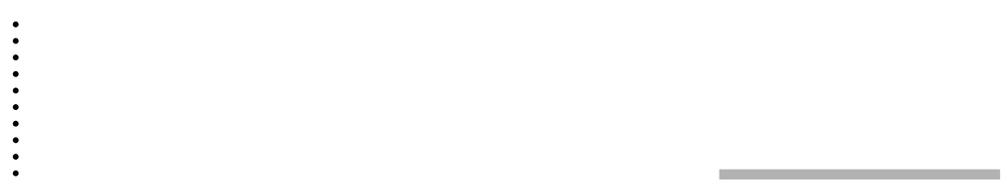
9. Windows

A WTK window object corresponds to a region of the screen in which a view of the graphical scene is displayed. With the window class, you can display multiple views simultaneously and to different parts of the screen.

WTK provides functions that let you create and delete windows with system-specific characteristics (such as border type), set a window's background color (*WTwindow_setbgrgb*), reposition and resize a window (*WTwindow_setposition*), define the way in which the scene is projected to the window when rendered (*WTwindow_setprojection*), assign a textured backdrop to a window (*WTwindow_loadimage*), and others.

Window Construction and Destruction

When you create a universe using *WTuniverse_new*, WTK automatically constructs a default window and associated viewpoint for you. You can construct additional windows using the *WTwindow_new* function. When constructing an additional window, you also need to use the *WTwindow_setviewpoint* function to assign a viewpoint to the new window.

When you create a window, the following parameters are set:

- **Projection Type**: The way in which the scene is projected into the window. You can specify a symmetric, asymmetric, general, or orthographic projection.

- **Viewpoint**: The viewpoint from which the scene is projected into the window.

- **Eye**: Specifies whether the scene is rendered as if projected by the left or right eye.

- **Background Color**: The background color (red, green, and blue) of the window.

- **View Angle:** The horizontal viewing angle (in radians) of the window.

- **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 those portions of graphical entities on the opposite side of the hither plane from the viewpoint are drawn.

- **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 those portions of graphical entities on the side of the yon clipping plane closest to the viewpoint are drawn.

Sample Window Functions

Table 14 below lists some of the window functions available:

Function	Description
WTwindow_delete	Deletes a WTK window object.
WTwindow_next	Iterates through the universe's list of WTK window objects.
WTwindow_setrootnode	Associates a scene graph with a particular WTwindow.
WTwindow_enable	Lets you enable or disable rendering to a specified window.
WTwindow_zoomviewpoint	Zooms the viewpoint of the given window so that all geometry in the scene graph associated with that window is visible.

Table 14: Sample windows functions.

:
:
:
:
:
:

10. Viewpoints

A WorldToolKit viewpoint defines the position and orientation of a simulation's geometries on the computer screen. Each WTK window has a viewpoint associated with it, from which the scene graph associated with the window is drawn.

Introduction to Viewpoints

When the universe is created with *WTuniverse_new*, a viewpoint is automatically created for it. WTK lets you construct additional viewpoints and switch between them. For example, you may wish to create a "bird's-eye view," an "out-the-window view," or a "rear view." An analogy to changing viewpoints in this way is cutting between various cameras in a movie.

To display several viewpoints simultaneously, you create multiple windows (and possibly additional viewpoints - using *WTviewpoint_new*) and use the function *WTwindow_setviewpoint* to specify the viewpoint from which the scene is rendered in each window. Each of these windows is associated with a particular scene graph; alternate views of the same scene would use the same scene graph, while windows depicting different scenes would use different scene graphs. Note that, 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.

The position and orientation of a viewpoint can be set explicitly through function calls such as *WTviewpoint_setposition* and *WTviewpoint_setorientation*. Alternatively, a viewpoint's position and orientation can be controlled by attached sensors. For example, if you construct a mouse sensor object and attach it to a viewpoint, as you move the mouse, the viewpoint moves automatically. You can also manage a viewpoint's motion by using a motion link. *(See Chapter 11, "Motion Links.")*

Figure 7 and figure 8 illustrate monoscopic and stereoscopic viewing geometries for symmetric window projections. Note that the view position and orientation is relative to the global (i.e., world) coordinate frame.

In figure 7, 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 is not shown.
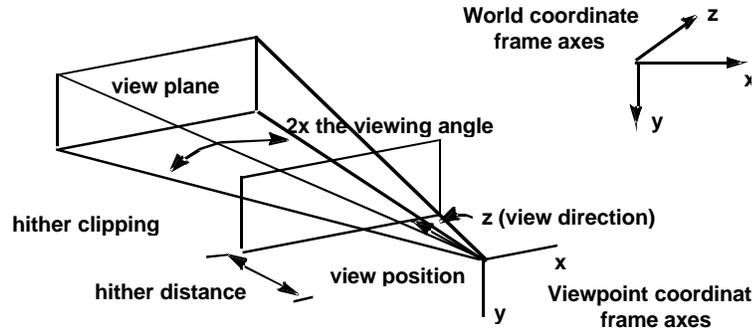
Figure 7: Monoscopic viewing geometry.
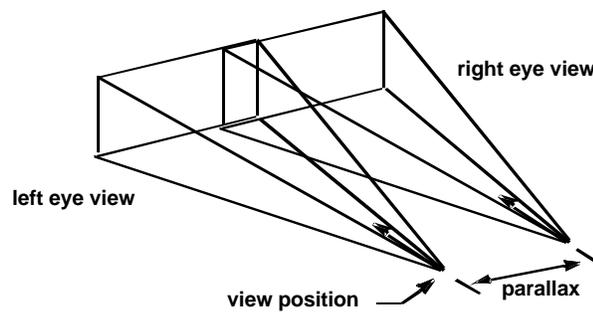
Figure 8: Stereoscopic viewing.

Figures 8 illustrates how stereoscopic viewing has the same parameters as monoscopic viewing, except that there are two view pyramids, linearly offset by the parallax distance.

Sample Viewpoint Functions

Table 15 below shows some sample viewpoint functions:

Function	Description
WTviewpoint_new	Creates a new viewpoint.
WTviewpoint_addsensor	Attaches a sensor to the viewpoint allowing viewpoint movement to be controlled by a device.
WTviewpoint_moveto	Moves the viewpoint to a particular location and orientation in the 3D world.
WTviewpoint_setdirection	Sets the viewpoint direction.
WTviewpoint_setparallax	Adjusts the parallax of the image.

Table 15. Sample viewpoints functions.

Attaching a Sensor to a Viewpoint

It is possible to attach a sensor to a viewpoint, so that the sensor's translation and orientation records automatically cause a corresponding translation and rotation of the viewpoint. If a viewpoint has more than one sensor attached to it, each one contributes to the motion of the viewpoint.

In the following example, Polhemus and Spaceball sensor objects are created and attached to the viewpoint. This is a useful sensor configuration in setups in which head-tracking with an absolute sensor such as the Polhemus is desired, but where you also want to independently control the viewpoint with a joystick-like device such, as the Spaceball.

```c
#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;
}
```

11. 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. For example, you can attach a mouse to a viewpoint using a motion link. Motion links are necessary, at least in part, because of the concept of instancing.

If you have a geometry with many instances, you may want to modify the position, orientation, or both of one instance of that geometry. For example, if you have 100 trees, where 99 of the trees are instances of the tree geometry, you may want to modify the position, orientation, or both of one tree by a sensor. You can accomplish this by creating a nodepath to the instance of the desired tree, and then creating a motion link between the sensor and the newly created nodepath to the instance of the tree.

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.

Motion Link Sources and Targets

The motion link source can be a path or a sensor. Motion link targets include the following:

- **A Viewpoint**: Use this as your target when you want the user's viewpoint controlled by the source you've specified.

- **A 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.

- **A 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.

- **A Movable Node:** Use this as your target when you want your source to affect a movable node (with or without attachments).

Figure 9 shows some of the things that can be attached with a Motion Link. Although a sensor (a mouse) is shown on one end of the Motion Link, sensors are not the only things that can be attached using Motion Links. As mentioned before, you may have a viewpoint on one end of the Motion Link and a path on the other end.
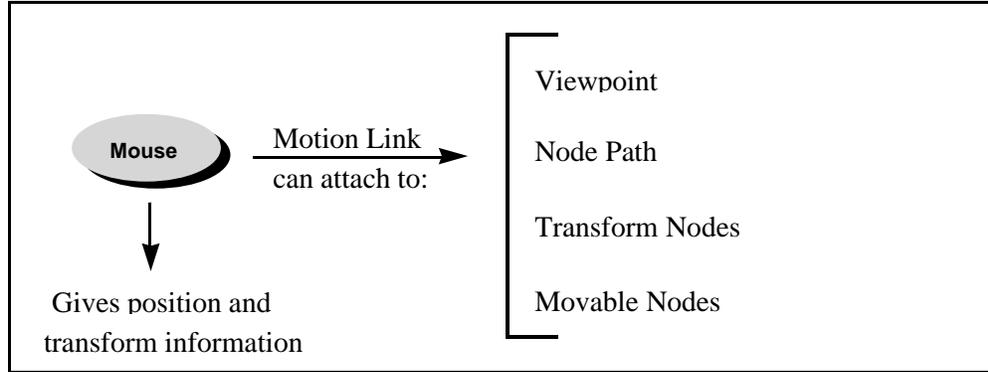
Figure 9: Some ways to use motion links.

Once a motion link is created (*WTmotionlink_new*), 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 functions like *WTpath_recordmotionlink* to record a path from the position and orientation information being transmitted by a motion link. This path can then be used as a source for other motion links.

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 references frames are as listed below:

Target type	Default reference frame
VIEWPOINT	WTFRAME_LOCAL
TRANSFORM	WTFRAME_LOCAL
NODEPATH	WTFRAME_WORLD
MOVABLE	WTFRAME_LOCAL

It is possible to change the reference frame in which the position and orientation information is applied to the motion link's target by using the function *WTmotionlink_setreferenceframe*. 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_setreference*, you could apply a sensor's position and orientation information to the movable in the current viewpoint frame.

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* function.

Sample Motion Link Functions

Table 16 below shows some sample motion link functions:

Function	Description
WTmotionlink_enable	Enables or disables the specified motion link.
WTmotionlink_gettarget	Finds the target and target type for a given link.
WTmotionlink_setreferenceframe	Sets the reference frame in which the indicated motion link will operate. It is the frame in which motion of the motion link's target is expected.
WTpath_recordmotionlink	Records the motion of the target of a motion link.

Table 16: Sample motion link functions.

12. Paths

A WorldToolKit path stores a series of position and orientation records. You can use these paths to guide the viewpoint or move other entities in the scene graph. You can dynamically create, record, save, load, and playback paths in a variety of ways. You can also use interpolation to smooth a roughly defined path.

As shown in figure 10, paths are made up of a set of discrete elements, where each element stores position and orientation. You might construct a path by recording the position and orientation of the viewpoint each frame, or creating one element each time through the simulation loop or at a specified sample rate.
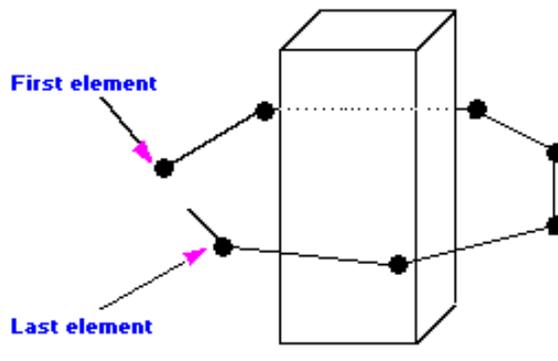
Figure 10: A path around an object.

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. You can also use viewpoint paths whenever an application requires that a viewpoint be moved from one location to another and you wish to provide a smooth transition.

Similarly, there are many uses for paths associated with other entities in the Scene Graph. Consider a simple case in which you want to have a door swing open and shut. You can 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 a 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

There are 3 ways to create and define a new path: by recording, manually constructing, or interpolating an existing path. The first method of recording a path, uses *WTpath_record* as follows:

1. Call *WTpath_new* to obtain a pointer to a new, empty path.

2. Call *WTpath_record* to start recording your viewpoint location.

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.

If you use the *WTpath_setrecordlink* function, you can record the position and orientation of a motion link's target instead of the current viewpoint's position and orientation.

The second method of manually constructing a path, uses *WTpathelement_new* and *WTpath_insertelement* or *WTpath_appendelement* to construct a path one element at a time, as follows:

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.

The third method of creating a path is to create an interpolated ("smooth") version of an existing path, using *WTpath_interpolate*. (You can also copy and delete paths.)

WTpath_interpolate creates and returns a pointer to a new path, based on an interpolation of the elements of another path. The *WTpath_interpolate* function takes an argument *npoints*, which specifies the number of elements to insert between each of the elements of the original path. This number must be 1 or greater and the original path must have at least two elements for the interpolation to be successful. The original path is unaffected by this operation.

The *WTpath_interpolate* function also allows you to specify the approach to be used to generate the positions of the interpolated points. The possible approaches are as follows:

WTPATH_BEZIER   For a Bezier curve.

WTPATH_BSPLINE   For a B-spline curve.

WTPATH_LINEAR   For a straight line path between elements.

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.

The **Bezier** option may be the most useful since it gives a smooth curve which 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. The **linear path** interpolation option places the interpolated points along a straight line between each pair of points in 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 above call 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.

Sample Path Functions

Table 17 below shows some sample path functions:

Function	Description
WTpath_play	Begins the playback of the indicated path, starting from the path's current element.
WTpath_save	Saves a path to the specified filename.
WTpath_setmarker	Sets the geometry which is to be used to display a path element. This function lets you visualize a path that has been recorded, loaded, or created.
WTpath_setvisibility	Toggles the visibility of a path's graphical representation.

Table 17: Sample path functions.

13. Special Effects and Sound

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 and the distance at which objects begin to be obscured.

WorldToolKit also allows you to enhance the realism of your scenes through the use of 3D sound. WTK supports 3D spatialization of sounds, Doppler shifts, and volume and roll-off controls. WTK supports WAV sound files on NT platforms and AIFF sound files on SGI platforms.

Fog Node Attributes

You can control the effects generated by a fog node by setting the following attributes:

- **fogcolor**: The color to which objects in the scene are blended to. The default fog color is black (0.0, 0.0, 0.0).

- **range:** The distance upon which all objects blend (completely) into the fogcolor. The default range is 0.0 which means that the range is set to the window yon plane distance.

- **mode:** The fog blending ramp (linear, exponential, exponential-squared). The default mode is linear.

- **linearstart:** The distance at which objects begin to be affected by the fog color. (Only applicable if the fog mode is linear.) The default linearstart is 0.0.

Introduction to WTK 3D Sound Support

WorldToolKit 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 and set up the hardware parameters (such as output type, rolloff).

2. Load various sound samples from disk.

3. Assign properties to sounds (such as volume, pitch, priority, position).

4. Cue the sounds to play on events or loop continuously during the simulation.

5. Close the audio hardware device, which removes the sound samples from memory.

Supported Devices

WTK supports the following sound 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.

- **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. WorldToolKit communicates with the Sound Server through a null-modem serial cable. Using this device you can play up to 4 hardware-spatialized sounds at a time.

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.

- **VSI Synthesiser (VSI)**

  This device requires a synthesiser 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 for Windows 95/NT, with the exception that the SGI requires an Acoustetron Server rather than an AudioReality NT Server.

Sample Sound Functions

Table 18 below shows some sample sound functions:

Function	Description
WTsound_load	Creates a new sound from a source.
WTsound_play	Cues a sound to begin playing
WTsounddevice_open	Opens an audio hardware device.
WTsound_stop	Stops a currently playing sound.
WTsound_setposition	Sets a sound's position.

Table 18: Sample sound functions.

. . . .

14. Tasks

In addition to the user-defined Universe Action function, which typically describes the overall activity of a WTK application, 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 and a pointer to the structure or WTK object with which the task is associated. It also contains a priority value specifying the order in which the task is executed relative to other tasks as the simulation runs.

You can add tasks, remove, and delete them from a simulation, as shown in Table 19.

Sample Task Functions

Table 19 below lists a few sample task functions:

Function	Description
WTtask_new	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 (see figure 1, on page 11).
WTtask_remove	This function removes a task from the simulation (deactivates it) without deleting the WTtask. A task which has been de-activated is no longer executed as the simulation runs. A task that has been de-activated can be re-activated by calling *WTtask_add*.
WTtask_delete	This function deletes a task. 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.
WTtask_setpriority	Sets the priority of a task.

Table 19: Sample task functions.

Sample Code

For example, to add a task to a light, your application would include code such as 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 */

}
```

15. User Interface Elements

WorldToolKit offers a complete set of high-level functions for creating common GUI elements for both X/Motif and Microsoft Windows environments. You can use this library to make an application that runs on both X Windows and MS Windows systems, while preserving the look and feel of each native environment.

WTK's GUI-suite includes the following elements:

- Toolbars

- Menus

- Message Boxes

- Text Input Dialogs

- Scales

- File Request Dialogs

- Scrolled Lists

- And Others

Adding GUI Elements

In addition to making common GUI elements, you can also easily add functionality to each user interface element by attaching callback handler functions for each desired event type to the element. Here's the basic procedure:

1. Make the interface element, using the appropriate *WTui* Create function.

2. Write the handler function for the specific event type.

3. Use the *WTui_setcallback* function to make the connection between the UI element, the handler function, the application data value that will be passed to the callback, and the event type.

Differences in the way Motif and Windows handle events and messages are taken care of by the underlying *WTui* library.

All UI elements are encapsulated in the generic *WTui* class. This is actually a structure which describes the UI element entirely, including the following parameters:

- The type of object (push-button, menu bar, frame, etc.).

- The object's generic ID.

- A pointer to the object's parent.

- A pointer to the object's list of children (if any).
- The object's platform-independent coordinates.
- A pointer to a callback function.

Sample User Interface Functions

Table 20 below lists a few user interface functions:

Function	Description
Wtuiform_new	Creates a form object.
WTuimessagebox_new	Creates a message box.
WTuipushbutton_new	Creates a push-button object.
WTuitextinput_new	Creates a simple input box composed of a label and text field.

Table 20: Sample user interface functions.

16. Math and Drawing Functions

WorldToolKit contains math functions for managing position and orientation data, and a set of user-defined drawing functions for embedding your own OpenGL drawing routines into your WTK applications.

Introduction to the Math Library

The WTK math functions include the following data types:

- **WTp2** Two-dimensional floating point vector.
- **WTp3** Three-dimensional floating point vector.
- **WTq** Quaternion — array of 4 floating point values.
- **WTpq** Structure containing WTp3 and WTq.
- **WTm3** 3x3 array of floats.
- **WTm4** 4x4 array of floats.

In WorldToolKit, 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 which returns orientation records in one of these representations, then you will need to convert the records into the quaternion representation. Conversion functions are provided for going between matrix, euler angle, and quaternion representations of orientation. WTK also provides functions to initialize, copy, add, subtract, multiply, invert, normalize, and compute magnitude, as well as dot products for the above data types where meaningful.

It may be convenient, when indexing mathematical quantities in WorldToolKit, 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 enable you to easily print out the value of position and orientation variables, for debugging an application or other purposes.

WorldToolKit Math Conventions

The WorldToolKit coordinate system obeys the right-hand rule. A viewpoint in the default orientation has the X axis pointing to the right, the Y axis pointing down, and the Z axis pointing straight ahead.

Rotations also obey the right-hand rule. For example, if a vector pointing along the Z axis is rotated by 90 degrees about the X axis, it will end up pointing in the negative Y direction.

Sample Math Functions

Table 21 below lists a few math functions:

Function	Description
WTp3_rotate	Rotates a 3D vector through a specified rotation.
WTp3_multm3, WTp3_multm4	Multiplies a direction vector by a WTm3 matrix and a WTm4 matrix respectively.
WTp3_distance	Returns the distance between two 3D points.

Table 21: Sample math functions.

User-defined Drawing Functions

WTK lets you embed your own OpenGL drawing routines into your WTK application. With WTK's user-defined drawing functions, you can combine the full rendering capabilities of your workstation with the functionality of WTK. WTK provides the ability for the user to specify a user-defined 2D drawing function (through the function *WTwindow_setfgactions*), as well as a user-defined 3D drawing function (through the function *WTwindow_setdrawfn*).

WTK also offers predefined 2D and 3D drawing functions such as *WTwindow_draw2Dline* and *WTwindow_set3Dlinewidth*, which can be called from the user-defined function.

Sample Drawing Functions

Table 22 below lists a few drawing functions:

Function	Description
WTwindow_set2Dcolor	Specifies the color to be used by subsequent draw2D functions.
WTwindow_draw2Dpoint	Draws a point at the coordinates specified by the x and y values.
WTwindow_draw2Dcircle	Draws a circle whose center is specified by the xc, yc parameters.
WTwindow_draw3Dlines	Draws a set of lines.

Table 22: Sample drawing functions.

17. Object/Property/Event Architecture

Overview

WorldToolKit (WTK) Release 8 has been enhanced through the addition of an Object/Property/Event 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.

- 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. 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 server product. If you have not purchased World2World, contact Sense8 to learn more about this client/server networking solution.

The Object/Property/Event 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 Object/Property/Event programming paradigm. Otherwise, you can use either programming paradigm.

For new WTK applications, we recommend that you make use of the Object/Property/Event 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 Object/Property/Event paradigm.

Supported Types and Supplied Properties

The Object/Property/Event architecture supports the following WTK object types:

- WTnode

- WTviewpoint

- WTwindow

- WTsensor

- WTpath

- WTbase

WTK supplies pre-defined properties for each of the supported object types, such as WTLIGHT_INTENSITY for the WTnode object type, and WTVIEWPOINT_POSITION for the WTviewpoint object type.

WTbase Objects

The WTbase object type 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 Object/Property/Event architecture, 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 Object/Property/Event paradigm cannot contain properties and, thus, do not generate events or allow for the sharing of data over a network. To extend the Object/Property/Event paradigm, create WTbase objects and user-defined properties to represent the desired attributes of the unsupported objects.

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

Sample Base Functions

Table 23 below lists some sample base functions:

Function	Description
WTbase_new	Creates a new, generic base object.
WTbase_addparent	Adds the specified WTbase object as a new parent of the object.
WTbase_getchild	Returns a pointer to the childnum'th child of the specified WTbase object, where childnum is a parameter of this function.
WTbase_getproperty	Returns the propnum'th property of the specified object, where propnum is a parameter of this function.

Table 23: Sample base functions.

Properties and Events

Properties

Objects that are supported by the Object/Property/Event architecture are distinguished from one another by their properties. Properties describe characteristics of an object. For example, WTviewpoint objects have the following pre-defined properties:

WTVIEWPOINT_ASPECT

WTVIEWPOINT_CONVDISTANCE

WTVIEWPOINT_CONVERGENCE

WTVIEWPOINT_ORIENTATION

WTVIEWPOINT_PARALLAX

WTVIEWPOINT_POSITION

One of the advantages of the Object/Property/Event 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.

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.

Sample Property Functions

Table 24 below lists some sample property functions:

Function	Description
WTproperty_new	Creates a new user-defined property.
WTproperty_set	Sets the specified property's value.
WTproperty_get	Retrieves the specified property's value.

| WTproperty_addhandler | Adds an event handler callback to the specified property. |

Table 24: Sample property functions.

:
:
:
:
:
:

18. Client-Server Networking (via World2World)

Overview

The Object/Property/Event programming paradigm, in conjunction with the high level networking functionality available in WTK, provides programmers with the ability to easily develop multi-user 3D/VR networked applications for use over LANs or the Internet. The 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.

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.

The mechanism by which property value changes are transmitted to all clients who are sharing the property is the World2World server product.

How World2World Works

The World2World server product consists of the following:

- a Server Manager

- one or more Simulation Servers

- 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 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.

Shared Properties

As previously described, 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 also sharing that property.

Properties are shared under specific sharegroups. Each Simulation Server can have a hierarchical arrangement of sharegroups that are used to organize the properties stored on a Simulation Server. 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. 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).

Persistent Properties

Shared properties can be flagged as being persistent. By making a shared property persistent, you ensure that the property will not be 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 that property.

Update Frequencies For Shared Properties

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 that controls how often the queued updates are actually sent across the network.

The default for shared properties is to queue updates each time the property value changes. You can override this default and set the update frequency to queue updates at a set time interval, specified in seconds.

Sample Shared Property Functions

Table 25 below lists some sample shared property functions:

Function	Description
WTproperty_share	Shares an object's property under a specified sharegroup of a Simulation Server.
WTproperty_setupdatefreq	Sets the frequency with which data updates for an object's property will be queued for transmission to a Simulation Server.
WTproperty_lock	Requests a property lock for the local client so that other clients cannot modify the specified object property.
WTproperty_sendupdate	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.

Table 25: Sample shared property functions.

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 connection has, by default, a root sharegroup. All other sharegroups created on that connection will be direct descendants (children) or indirect descendants of the root sharegroup.

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 lock is released. 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.

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. Any time a property or sharegroup is added or removed from a sharegroup in the connection's data tree, the Simulation Server is responsible for sending notification of these changes to any client that is interested in that sharegroup.

Persistent Sharegroups

Sharegroups, like properties, can be flagged as being 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 has the same effect as making each of the sharegroup's properties persistent.

Sample Sharegroup Functions

Table 26 below lists some sample sharegroup functions:

Function	Description
WTsharegroup_new	Creates a new sharegroup as a child of another specified sharegroup.
WTsharegroup_lock	Requests a sharegroup lock for the local client so that other clients cannot modify this sharegroup's subtree.
WTsharegroup_registerinterest	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.
WTsharegroup_getproperty	Returns the name of the propertynum'th property of a sharegroup, where propertynum is a parameter of this function.

Table 26: Sample sharegroup functions.

Connections

When a client starts a World2World-compliant simulation, the simulation 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 and sharing properties.

Connection Update Rates

A connection's update rate determines the maximum updates per second for the connection. This is the maximum number of times per second that the client will send data packets to the Simulation Server and the maximum 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.

Connection Events

For any connection, you can respond to certain changes to the Simulation Server, such as the addition of a user, sharegroup, or shared property to the connection.

As an example of how this may be useful, suppose that every time a new user connects to the simulation, you want to create a graphical avatar to represent the new user. You would simply trigger a reaction to be executed (the creation of the graphical object) each time a user is added to the connection.

Sample Connection Functions

Table 27 below lists some sample connection functions:

Function	Description
WTconnection_new	Defines a new connection to the World2World Server Manager at a specified port. Based on the port number, the Server Manager will direct the client to the appropriate Simulation Server as configured.
WTconnection_connect	Attempts to connect to the World2World Server Manager and port represented by the specified connection object.
WTconnection_setupdaterate	Sets the rate at which data packets are sent over the connection.
WTconnection_addcallback	Adds a callback function to a connection.

Table 27: Sample connection functions.

19. Pier-to-Pier Networking

WorldToolKit lets you create and distribute simulations in environments where a mixture of PCs and Unix platforms exist. Since WTK applications share a common API, a single application can be run on both PC and Unix workstations without modification. *(Note that additional licenses are required to use WTK's networking features.)*

You can add networking functionality to your applications by using functions 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 which allows multiple users to share the same virtual world. Graphical objects located at the other users' viewpoints are used to represent the other people.

The following terminology is useful when discussing distributed simulations:

- **Local**: The objects residing on the local simulation hardware.

- **Remote**: The objects residing on simulation hardware other than the local machine's.

- **Private**: Objects that only exist on the specific machine's hardware. They are not part of the distributed simulation.

- **Public**: Objects that are part of the distributed simulation. They may be controlled from a single machine, but their state is updated on all machines participating in the simulation.

How the Transport Layer Works

At the transport layer, multiple PCs or Unix workstations must be connected with standard Ethernet hardware and cabling. WorldToolKit only supports the DEC/Intel/Xerox (DIX or referred to as the Bluebook Ethernet) standard. PCs can either have their own physical, independent network, or be connected to the same Ethernet line used for the Unix workstations. Byte-ordering problems between PCs and workstations are handled invisibly by WTK.

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.

How the WorldToolKit Layer Works

You initialize network communications by calling the *WTnet_open* function. You can establish communication with other machines on the network by using a valid multicast group address. In addition to the group address, simulation machines only communicate with

other machines that share identical port addresses. This information is passed to WTK through the *WTnet_open* call.

Upon establishing a network connection, you can pass application-specific information between machines by using discrete message items that are assembled into valid UDP packets. WTK assembles these message items using the *WTnet_additem* function, and then automatically sends them out onto the network. 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. Whereas WTK provides the substrate for communications to occur, your application defines the type of 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.

Sample Transaction

Local Machine

In this example, a local user changes his viewpoint and moves his virtual body to a different position and orientation. The graphical object representing the local user is an example of a local, public object. When the local machine enters its user-defined action function, 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. 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 graphical object representing the remote user to the updated location. This cycle of sending and receiving application-specific messages is endlessly repeated, resulting in a distributed simulation.

Sample Networking Functions

Table 28 below lists some networking functions:

Function	Description
WTnet_open	Opens up the network if it has not already been opened.
WTnet_close	Closes the network, if it is currently open, and deletes private data structures.
WTnet_getrange	Returns the current range (Time to Live) value.
WTnet_getport	Returns the current port value.

Table 28: A sample of some networking functions.

WorldToolKit Technical Overview