An Object-Oriented 3D Graphics Toolkit

Paul S. Strauss
Rikk Carey

Silicon Graphics Computer Systems
2011 North Shoreline Blvd.
Mountain View, CA 94039-7311
pss@sgi.com, rikk@sgi.com

Abstract

This paper presents an object-oriented toolkit for developers of interactive 3D graphics applications. The primary goal of the toolkit is to make it easier for programmers to create 3D graphics applications that employ direct manipulation techniques in addition to conventional 2D-widgets. Such techniques have generally been ignored in previous graphics packages and systems.

The toolkit provides a general and extensible framework for representing 3D scenes so that applications can integrate their data with graphical objects rather than having duplicate copies. A simple, integrated event model that enables direct interaction with 3D objects is also included.

CR Categories and Subject Descriptors

I.3.2 Graphics Systems; I.3.4 Graphics Utilities, Application Packages, Graphics Packages; I.3.6 Methodology and Techniques - Interaction Techniques; I.3.7 Three-Dimensional Graphics and Realism, Animation

Keywords

Interactive 3D graphics, object-oriented design, scene representation, direct manipulation.

Introduction

Writing interactive 3D graphics applications has traditionally been a tedious, time-consuming, and difficult task requiring a high level of expertise by the programmer. Because it has been so difficult, developers and researchers have either invented their own software abstractions above the low-level graphics commands or produced portable, "lowest common denominator" applications with little or no direct 3D interaction. The former choice yields short-term solutions that are rarely given sufficient design and implementation effort. The latter approach is more common in industry and results in disjoint user interfaces in which users view 3D areas but can interact only through remote user interface widgets or keyboard command languages.

A 3D graphics toolkit could facilitate writing interactive applications. A truly interactive 3D toolkit should provide a rich and extensible set of 3D objects and should support *direct manipulation*, allowing users to interact with those objects in the same window in which they appear. This technique is common in 2D graphics applications, yet it has rarely appeared in 3D applications. There are three fundamental areas in which a toolkit can enable the development of such interactive 3D programs:

- **Object representation**. Graphical data should be stored as editable objects and not just as collections of drawing primitives used to represent them. That is, applications should be able to specify *what it is* and not have to worry about *how to draw it*.
- **Interactivity**. An event model for direct interactive programming must be integrated with the representation of graphical objects.
- **Architecture**. Applications should not have to adapt to object representation or interaction policies imposed by the toolkit. Instead, the toolkit mechanisms should be used to implement the desired policies. Such flexibility should also be reflected in the ability to extend the toolkit when necessary.

Traditional 3D graphics application systems can be characterized as taking one of two approaches to object representation. GKS [3] and PHIGS+ [6] represent the *display list* approach in which objects are defined as sequences of drawing commands and cannot be treated as first-class 3D objects. Doré™ [1] and HOOPS™ [9] significantly improve the display list model by providing additional editing capabilities, but they still fall short of true 3D object representation.

Immediate mode libraries such as Iris GL™ [2], Starbase™ [7], and RenderMan™ [8] are streamlined drawing packages; they have no notion of retained or represented 3D objects (although GL and Starbase also support display lists). These libraries are focused on providing flexible and efficient interfaces to specific graphics devices or programs.

The main problem with both of these approaches is that they concentrate solely on the rendering or display aspects of application writing (a worthy task, especially considering advances in rendering hardware and algorithms during the 1970's and 1980's). However, rendering is only a small part of the task of writing an interactive application. 3D graphics packages, for the most part, have ignored user input; they often provide little more than gross-object picking. Furthermore, the programming models for these packages do not treat 3D objects as geometric, physical entities. Thus, previous systems are committed to display-oriented application development and provide little help for direct, 3D interaction support.

Another shortcoming of traditional graphics packages is that they proliferate the "duplicate database" problem. Applications store their objects in a form suited to their needs, but must convert them into structures required by the graphics package, which maintains an additional copy. Furthermore, graphics packages are typically closed systems that do not allow applications to add their own object geometries, properties, or operations.

There have been few published descriptions of 3D toolkits which attempt to solve the problems presented here. The InterViews [5] system successfully integrates rendering and interaction in the world of 2D graphics and text, but it does not extend to 3D. The Brown Animation Generation System [10] is one published system that successfully integrates abstract, 3D object representation and dynamics. It is designed primarily for animation but does include support for interactive techniques.

The toolkit described in this paper attempts to meet the needs described above. It defines an object-oriented framework for describing scenes containing 3D objects and operations on them. 3D objects are abstract representations that can render themselves when requested; they are not merely display lists. The methods by which objects are rendered can vary from machine to machine, but their representations are constant.

The framework of this toolkit is designed with application extensibility in mind. Application writers are able to add new, application-specific objects to the toolkit when necessary. By allowing application data to be incorporated into the toolkit, the need for duplicate databases is greatly reduced. Extensibility also includes object operations such as new rendering methods and geometric computations.

A simple, yet effective, model for handing events to 3D objects is integrated into the toolkit. Several interactive 3D objects are provided, allowing applications to add interactive operations, such as rotation, easily. This set of interactive objects is also extensible, and the toolkit provides several levels of support to make it easy for applications to create their own objects for direct interaction.

Overview

The 3D toolkit library consists of three main sections, as illustrated in Figure 1. Furthermore, there are two window system utility libraries built on top of the toolkit that provide window objects and handle event translation.

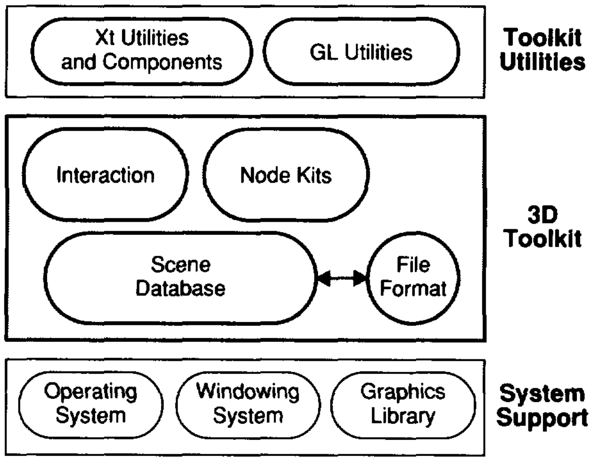

Figure 1. Overall toolkit architecture.

Each of the three toolkit sections provides a different level of programming support. They are introduced here in order from low-level to high-level.

The foundation of the 3D toolkit is the *scene database*. It stores dynamic representations of 3D scenes as graphs (typically directed acyclic graphs) of objects called *nodes*. Various classes of nodes implement different geometries, properties, and database traversal behaviors. The database provides a set of *actions* that can be applied to scenes or parts of scenes; examples of actions are rendering, picking, computing a bounding box, handling an event, and writing to a file. The format and methods for storing scenes in files and retrieving them are defined by the database. Also included are objects for adding animation to scenes and for tracking changes to them.

The *interaction* section of the toolkit introduces event classes and "smart" nodes that process events. Event classes define an extensible set of abstract events such as `ButtonEvent` and `Location2-Event`. An example of a smart node is the `Selection` node, which provides an easy way for applications to maintain lists of selected objects. Selection tracks picking, supports various policies for replacing and extending selections, and graphically highlights selected objects. Another type of smart node is a *manipulator*, which responds to interaction events and edits other nodes in a database. A manipulator typically employs a surrogate object (e.g., bounding sphere or box) that represents the manipulator visually and provides a means for translating events into changes to the database. For example, the `Trackball` manipulator uses a bounding sphere around an object to modify the rotation of that object. Other manipulators perform object-specific functions, such as changing the illumination direction of a light source or dragging a vertex of a polyhedral shape. Manipulators provide an easy way for applications to incorporate direct 3D interaction.

The third section of the toolkit defines *node kits*, which make it easier to create structured, consistent databases. Each node kit object combines some scene database subgraph, attachment rules, and other policies into a single class. For example, the `SphereKit` is a wrapper around a sphere node that adds material, geometric transformation, and other properties in the correct place when needed. Node kits also allow programmers to create higher-level objects that encapsulate application-specific behavior.

Note that the 3D toolkit itself does not include any objects that represent windows on the display screen. This decision was made to ensure window system independence and greater portability. *Utility libraries* tied to specific window systems (such as X and GL) are built on top of the toolkit to provide convenience to application programmers. Each utility library provides a basic `RenderArea` object that maintains a resizable window that handles automatic redrawing, translates window system events into toolkit events, and distributes events to objects rendered in the window. Furthermore, the utility libraries can also provide a set of application-level *components* that implement common interactive functions. Examples of components are color editors, surface material editors, light source editors, and viewers, which process user interaction to edit cameras.

Architecture

The 3D toolkit is implemented in C++, which supports many of the object-oriented paradigms essential to extensible systems. C bindings are also provided, although much of the extensibility is not accessible from C.

Nodes

Each node in a scene database performs some specific function. There are *shape nodes* that represent geometric or physical objects (see Figure 7), *property nodes* that describe various attributes of those objects (see Figure 8), and *group nodes*, which connect other nodes into graphs and subgraphs. Other nodes, such as cameras and lights, are also provided. A representative sampling of node classes is given in Table 1.

Shape nodes:
`Cone`
`Cube`
`Cylinder`
`FaceSet`
`IndexedFaceSet`
`IndexedLineSet`
`IndexedTriangleMesh`
`LineSet`
`NurbsCurve`
`NurbsSurface`
`PointSet`
`QuadMesh`
`Sphere`
`Text2`
`Text3`
`TriangleStripSet`

Light/camera nodes:
`OrthographicCamera`
`PerspectiveCamera`
`DirectionalLight`
`PointLight`
`SpotLight`

Group nodes:
`Group`
`Separator`
`Switch`
`Selection`
`Manipulator`
`LayerGroup`
`Array`
`MultipleCopy`

Property nodes:
`BaseColor`
`Complexity`
`Coordinate3`
`DrawStyle`
`Environment`
`Font`
`LightModel`
`Material`
`MaterialBinding`
`Normal`
`NormalBinding`
`Texture2`
`TextureCoordinate2`
`Transform`

Table 1. Some node classes.

Instance-specific information is stored within nodes in sub-objects called *fields*. Each node class defines some number of fields, each with a specific value type associated with it. For example, the `Cylinder` shape node contains two real-number (float) fields that represent the radius and height of a specific cylinder instance. Field objects provide a consistent mechanism for editing, querying, reading, writing, and monitoring instance data within nodes.

The set of nodes is designed to allow most of the high-volume data to be shared when possible. For example, coordinates and normal vectors are specified in separate (property) nodes that can be shared among various shapes. This scheme has the additional benefit of enforcing consistency of representation.

A variety of group node classes connect nodes into graphs. Each group node class determines if and how traversal of children is performed and how properties are inherited. A node typically inherits properties from its parent, and children of a group node usually inherit from prior siblings. Some groups provide inheritance from the group node to its parent, making insertion of properties in subgraphs simple. Other groups, such as `Separator` nodes, save state before and restore state after traversing children, isolating their effects from the rest of the graph.

These groups represent traditional, hierarchical grouping objects found in most 3D systems. However, other behaviors can be implemented. For example, the `Switch` node selects one of its children to traverse; this can be useful for implementing level-of-detail, for example. The `Array` node traverses its children multiple times, applying a transformation before each traversal to arrange the results in a 3D array.

Figure 2 depicts a scene graph whose rendered result appears in Figure 9.

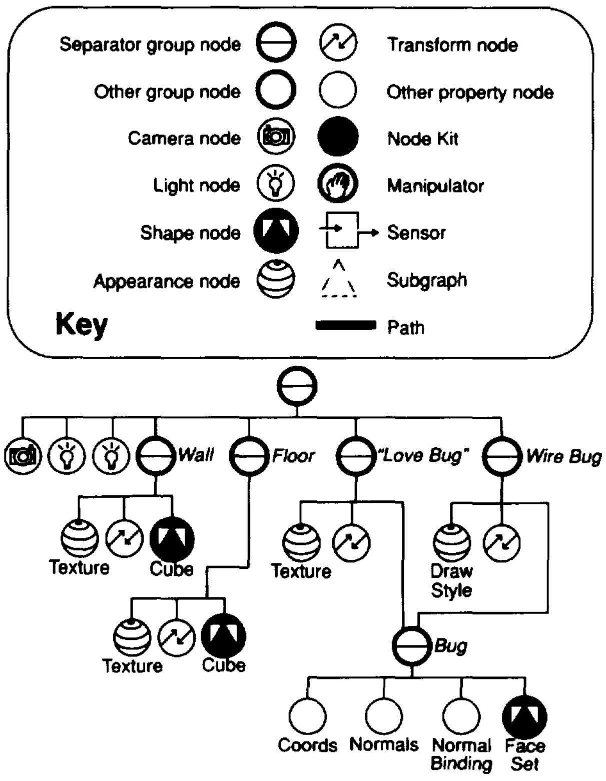

Figure 2. A simple scene graph.

Paths

A node may be a child of more than one group, allowing common subgraphs (multiple instances) to be shared. For example, a model of a bicycle may use a subgraph representing a wheel twice, with different transformation nodes applied to each instance of the wheel. This scheme can result in more compact and manageable scene representations in many cases. The downside is that it is not always possible to refer unambiguously to an object (such as the rear bicycle wheel) in the 3D scene simply by pointing to a single node. To remedy this problem, the toolkit supports *path* objects, which point to nodes in a chain from some node in the graph down to the node in question (see Figure 3). For example, performing a pick operation returns a path from the root of the graph to the shape node under the cursor, unambiguously indicating the object that was picked.

Note that a path actually defines a subgraph consisting of more than just the connected chain of nodes. The subgraph also includes all nodes (if any) below the last node in the chain and all nodes (typically to the left of the chain) that have an effect on these nodes. This definition is extremely important when performing graph editing such as cut-and-paste; all of the subgraph nodes are necessary to fully represent the selected object.

Actions

Objects called *actions* traverse scene graphs to perform specific operations, such as rendering, computing a bounding box, searching, or writing to a file. Several currently supported actions are listed in Table 2. An application performs an action on a scene in a database by applying it to a node in the scene graph, typically the root. Actions may also be applied to paths. The next section discusses the mechanism of applying actions in more detail.

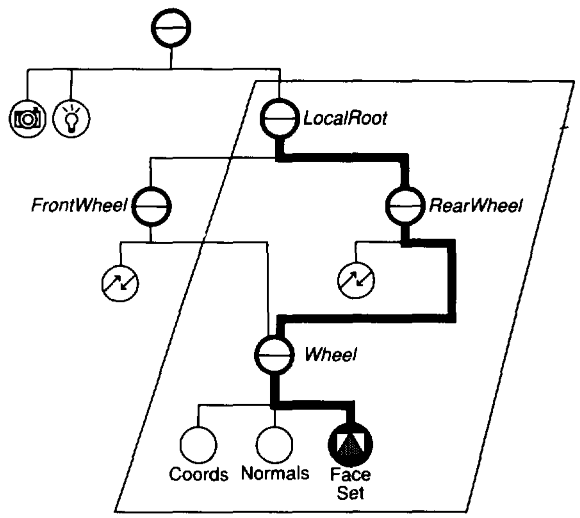

Figure 3. The *Wheel* node is multiply instanced, since it is a child of both *FrontWheel* and *RearWheel*. The path (heavy line) refers unambiguously from the local graph root to the shape defining the rear wheel. Within the parallelogram is the subgraph defined by the path.

`CallbackAction`	Generic scene traversal with user callbacks.
`GLRenderAction`	Renders objects in database.
`GetBoundingBoxAction`	Computes bounding box of scene or part of scene.
`GetMatrixAction`	Computes cumulative transformation matrix for object in scene.
`HandleEventAction`	Gives nodes in database a chance to handle an event.
`RayPickAction`	Returns frontmost object or all objects intersected by a ray cast into scene.
`PrintAction`	Produces PostScript primitives or other printable form of rendered scene.
`SearchAction`	Looks for specific node or type of node in database.

Table 2. Action classes.

The 3D toolkit is designed to be extensible in many areas. It is assumed that users will often extend the set of nodes to create new shapes, properties, manipulators (which are derived from nodes), and other classes. Similarly, it may be useful in some applications to create actions to perform new tasks while traversing a scene graph. Unfortunately, the standard virtual method table implementation in C++ makes this two-way extensibility very difficult: it is relatively easy to create a new node class that supports all of the standard actions, but it is not possible for application writers to add new action methods to existing standard node classes. This could be done by creating subclasses of all nodes that supported the new actions, but this is an unattractive solution.

To solve the two-way extensibility problem, the toolkit uses a two-dimensional virtual table to implement node/action methods. Each entry in the table is a method that implements a certain action for a particular node. Adding a new node is equivalent to adding a column to this table, whereas adding a new action involves adding a new row. For convenience, standard action methods are implemented as regular virtual functions on nodes.

This method lookup scheme is the justification for implementing actions as separate objects. The resulting syntax for applying an action to a graph, `action->apply(node)`, is fairly easy to use and understand.

Applying Actions

Applying an action to a scene graph usually results in a traversal of the graph. Although group node subclasses are free to define their own traversal behavior, standard groups visit their children in left-to-right order. Therefore, scene graph traversal is usually depth-first.

Most actions require *state* to be accumulated during traversal. For example, when a shape node is encountered during rendering traversal, it may need to know the current material, set of coordinates, and drawing style. Each of these properties is stored in an *element* in the state. Typically, property nodes change one or more state elements, and shape nodes interpret the current values of the elements.

When a `Separator` node (a type of group) is encountered, traversal state is saved before visiting its children and is restored afterwards. Because this node is used very often in scene graphs to segregate objects, state saving and restoration must be efficient. The state puts off saving element values until those values are modified after the save; this "lazy" scheme avoids copying data unnecessarily.

Rendering and Picking

Because the toolkit is designed for interactive graphics applications, the action that performs rendering must be capable of interactive speeds. Therefore, the rendering methods built into nodes have been tuned for maximum performance. The toolkit can also boost rendering performance by caching state at crucial points. For example, once it has been determined how to render a subgraph, caching a display list can make the next traversal fast and easy. Caching is built into `Separator` nodes and can be activated by an application when desired. References to nodes used to build a cache are stored with it, so any changes to those nodes will automatically invalidate the cache. Additionally, it is possible to abort a rendering action during traversal; this is useful when implementing real-time applications and for minimizing the wait for event processing in interactive applications.

Some rendering features, such as texture mapping, are not implemented on all supported platforms. Users of the toolkit do not need to test whether a particular machine supports a feature; they can just specify the feature by inserting nodes in the scene database, and the toolkit does the rest. On architectures that do not support the feature, the toolkit can simulate it in software (if this would not greatly compromise interactive performance) or elide it.

Interactive applications require flexible and efficient picking. The `RayPickAction` can be used to find a path to the frontmost object under the cursor. If an application requires more information, it can query the action for more details, such as the world-space and object-space points of intersection and the surface normals at those points. Each shape node class can define additional, specific information that can be queried. For example, objects constructed from sets of vertices can return the face containing the intersection point and the edge and vertex closest to it.

Picking performance, like that of rendering, is improved by caching. `Separator` nodes automatically cache bounding boxes, which are used by the pick action for culling. This caching speeds picking enough to make locate highlighting (dynamic highlighting of the object under a moving cursor) possible.

Sensors

Sensor objects are used to track changes to nodes and to implement simple animation. There are two major types of sensors, each of which calls a user-defined callback function when triggered. A *data sensor* is attached to a node and is triggered when it detects changes to data in that node or any descendant node. These sensors can be used by an interactive editor to update sliders or other indicators whenever any other program element changes values used by the editor. A data sensor can also be attached to the root of a graph to track changes to the entire scene (e.g., to determine when to redraw).

The other type of sensor is a *timer sensor*, which is triggered at a specified time or at regular intervals. For example, an application can set a timer sensor to trigger 30 times a second to animate parts of a graph. Figure 4 illustrates how sensors can be connected to a scene database.

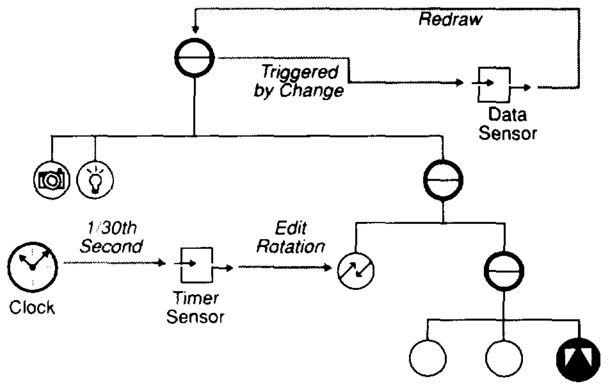

Figure 4. A scene graph with sensors attached. The data sensor at the top of the graph is triggered whenever a change is made to any node below the top; it is used to determine when to redraw the graph. The timer sensor connected to the clock fires at regular intervals, animating the rotation of the object at the lower right.

3D Event Model

The toolkit and utility libraries use a very simple algorithm to distribute user events to manipulators and other smart nodes. An event specific to the window system is generated as the result of some user action (mouse motion, keyboard button press, etc.) and is passed to the instance of the utility library's `RenderArea` object that corresponds to the window in which the event occurred. If the `RenderArea` is part of a component, the component might handle the event itself. For example, a viewer component might process all mouse motion to edit the position and orientation of a camera in the scene.

If the component does not process the event, the `RenderArea` translates the window system event into a toolkit event (independent of any window system) and distributes it to the nodes in the 3D scene. The `RenderArea` applies an instance of the `HandleEventAction` to the root node of the graph to distribute the event. This action performs a standard traversal of the graph. Any node that is interested in events may process the event and indicate that it has handled it. As with all other actions, each node class is free to define its own behavior when handling events.

Some nodes, such as manipulators, may be interested in mouse events only when the cursor is over their rendered objects. Such nodes may ask the `HandleEventAction` for a path to the object under the cursor. To process this request, the action automatically performs a pick (using a `RayPickAction`) to determine this path; it is cached so further inquiries do not require additional picks. Therefore, pick correlation is performed only when requested, speeding up the event handling process.

Database traversal for the `HandleEventAction` stops as soon as a node is found to handle the event. Nodes that want to monitor events without ending the traversal can do so by omitting the step that marks the event as handled. Also, nodes that are interested in future events can perform a "grab," meaning that they will receive all events until further notice. Event grabbing is useful for processing mouse press-move-release sequences.

An example of the workings of the event model is illustrated in Figure 5.

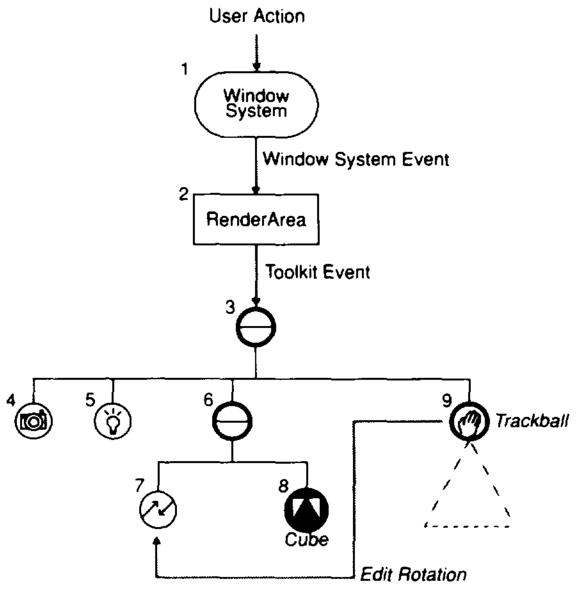

Figure 5. Event model example. (1) The window system processes a user action and produces a window system event. (2) The `RenderArea` associated with the window in which the action occurred translates the event into a toolkit event and passes it to a `HandleEventAction` which it applies to the root of its graph. (3) The root node of the graph passes the event to its children. (4)-(5) The camera and light nodes are not interested in events, and therefore they ignore this one. (6) The next child, a separator group, passes the event to its children. (7)-(8) The transform and cube nodes ignore the event. (9) The trackball manipulator checks if the event is a left-mouse-down, and, if so, determines whether the cursor is over the trackball object defined in its subgraph. If the cursor is over the object, the trackball grabs future events and indicates that it has handled the event. Subsequent events are sent immediately to the trackball manipulator, which processes mouse motion events and edits the rotation field in the transform node. This process continues until a left-mouse-up event is processed by the trackball, which then releases its grab.

Manipulators

The event model described above is designed to allow manipulators and other interactive nodes to be integrated easily into graphics applications. As an illustration, this section explores the implementation of the trackball manipulator (Figure 10) in more detail.

As mentioned earlier, the trackball manipulator places an invisible surrogate bounding sphere around the 3D object it is manipulating. The sphere is used to translate mouse motion into rotational changes to this object. Three cylindrical bands around the sphere make it easy to specify rotations constrained to the principal axes.

When a trackball is activated, it is given a path to the geometric transformation node it is to edit to achieve the rotation. The application inserts a `Trackball` node into its scene graph, ensuring that the trackball is rendered with the rest of the scene and that it is given an opportunity to handle events. The `Trackball` node (derived from `Separator`) has as children a scene graph that defines the surrogate sphere and constraint bands.

When a `Trackball` node is encountered during traversal by a `HandleEventAction`, the node checks if the event is a left-mouse-down. If so, it asks the action for the object under the cursor (computed by picking). If the cursor is over the trackball object, the node does two things. First, it announces to the action that it has handled the event, so traversal need not continue. Second, it performs a grab so that all future events will go directly to the trackball. The grab will be released when a left-mouse-up event is processed by the trackball. Each intervening mouse-motion event is processed by the trackball to compute rotational changes to its target object.

The toolkit includes many *simple* manipulators, each of which performs a single task, such as translation in one dimension or cylindrical rotation. Also in the toolkit are *compound* manipulators (such as the trackball) that combine several simple manipulators into a more complex, integrated tool. Figure 11 illustrates several manipulators.

Node Kits

Because the scene graph library is general enough to provide maximum performance and flexibility, it can often be confusing to novice users. There are no strict rules for forming scene graphs, so it is possible to create bizarre and sometimes meaningless collections of nodes unless some sort of structural guidelines are imposed. (This situation is reminiscent of assembly code and structured programming.) Furthermore, class-specific traversal and inheritance rules make it difficult to examine a scene graph and determine exactly how subgraphs of nodes relate to "objects" (chairs, bicycles) in the 3D scene.

Node kits provide one way to make these tasks easier by enforcing a consistent policy for database construction, editing, and inquiry. Each node kit effectively contains some structured subgraph of database nodes. A template associated with the node kit determines which nodes can be added when necessary, and where they should be added. For example, the `SphereKit` node represents a sphere object; its template allows a material, geometric transformation, and other properties to be inserted in the correct place when needed (Figure 6).

Another use of node kits is to define application-specific objects and semantics. For example, consider a flight simulation package that includes a variety of objects representing airplanes. Each of these airplanes consists of the same general scene graph structure (e.g., fuselage, wings, and landing gear) as well as some airplane-specific methods (e.g., *bank-left*, *raise-landing-gear*). To an application writer using this package, each type of airplane can be dealt with in a similar way. There is no need to know the details of the structure of the subgraph representing the landing gear to raise it, since there is a general method for doing so.

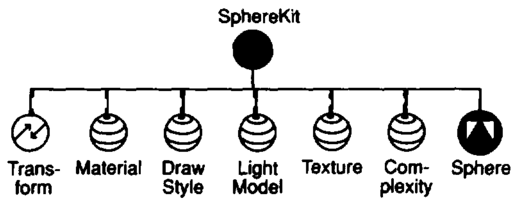

Figure 6. Sphere node kit. When an application creates an instance of the kit, a template of the above graph is constructed, opaque to the application. The application makes changes to the sphere only though the node kit. Nodes connected by dashed lines are created only when necessary. For example, applying a transformation to the node kit is implemented internally as a change to the transform node, which is created first if necessary.

Callbacks

No toolkit can ever be complete; users will always want to add something. One way to extend the set of objects provided by the toolkit is to derive new classes from existing ones. Although the toolkit is designed to make such extensions as easy as possible, sometimes an easier mechanism is useful. For this reason, several objects in the toolkit provide callback mechanisms for quick prototyping of new features.

The `Callback` node, for example, invokes a user-defined function whenever it is encountered during traversal for any action; this node makes it easy to introduce specialized behavior into a scene graph or to prototype new nodes without subclassing. Similarly, the `CallbackAction` allows functions to be called before and after each node encountered during traversal, giving them access to the current traversal state.

Other classes provide callbacks during various interactive processes, such as direct manipulation and selection of objects.

Conclusions and Future Work

The toolkit presented here is an attempt to make it easier for graphics application developers to design and implement 3D applications with direct interaction. The flexible and extensible 3D database makes it possible for applications to use only one representation of application data. By integrating interactive objects into the data with a simple event model, the toolkit allows applications to support direct manipulation of 3D data, which is often more friendly and intuitive than conventional widget-driven interfaces. Of course, the 3D toolkit still allows standard widgets to be integrated into applications in addition to the direct interfaces. Sample applications showing a variety of interactive techniques are shown in operation in Figure 12 through Figure 15.

An obvious direction for the future is extending the toolkit to have more nodes, actions, manipulators, and components. Chief among these are volumes, more 2D shapes, shape-specific manipulators, photorealistic rendering, and the ability to generate and return primitives (polygons, lines, etc.) for all shapes. This last feature can be used to implement radiosity, simulation, and analysis.

Other potential directions for the toolkit include:

Animation. Sensors provide the mechanism upon which to build more complex animation and constraint schemes. The toolkit could provide higher-level classes to facilitate the creation and storage of constrained, timed, and scripted motion.

Shared databases. Running distinct components or applications as separate processes requires databases to be stored in shared memory to avoid duplicating large amounts of data. The challenge is to design a means for exclusive access into shared memory that does not greatly compromise performance.

Acknowledgments

The other members of the team that designed, implemented, and documented the 3D toolkit are Thad Beier, Gavin Bell, Alain Dumesny, David Immel, Paul Isaacs, Howard Look, David Mott, Nick Thompson, and Josie Wernecke. Thanks to Derrick Burns for NURBS support, Paul Haeberli for help with printing algorithms, and Delle Maxwell for assistance with images and aesthetics.

Thanks to Wei Yen and Silicon Graphics for their support of the Iris Inventor™ 3D toolkit project [4].

References

[1] *Doré Programmer's Guide, Release 5.0*, Kubota Pacific Computer, Incorporated, Santa Clara, Calif., 1991.

[2] *Graphics Library Programming Guide*, Silicon Graphics Computer Systems, Mountain View, Calif., 1991.

[3] International Standards Organization, *International Standard Information Processing Systems — Computer Graphics — Graphical Kernel System for Three Dimensions (GKS-3D) Functional Description*, ISO Document Number 8805:1988(E), American National Standards Institute, New York, 1988.

[4] *Iris Inventor Programming Guide*, Silicon Graphics Computer Systems, Mountain View, Calif., 1992.

[5] Mark Linton, Paul Caulder, John A. Interrante, Steven Tang, and John M. Vlissides, *InterViews Reference Manual, Version 3.0.1.*, Stanford University, October 1991.

[6] PHIGS+ Committee, Andries van Dam, chair, "PHIGS+ Functional Description, Revision 3.0," *Computer Graphics*, **22**(3), pp. 125-218 (July 1988).

[7] *Starbase Graphics Techniques and Display List Programmer's Guide*, Hewlett-Packard Company, Fort Collins, Colo., 1991.

[8] Steve Upstill, *The RenderMan Companion*, Addison-Wesley, Reading, Mass., 1990.

[9] Garry Wiegand and Bob Covey, *HOOPS Reference Manual, Version 3.0*, Ithaca Software, 1991.

[10] Robert C. Zeleznik, D. Brookshire Conner, Matthias M. Wloka, Daniel G. Aliaga, Nathan T. Huang, Philip M. Hubbard, Brian Knep, Henry Kaufman, John F. Hughes, and Andries van Dam, "An Object-Oriented Framework for the Integration of Interactive Animation Techniques," *Computer Graphics (SIGGRAPH '91 Proceedings)* **25**(4) pp. 105-111 (July, 1991).

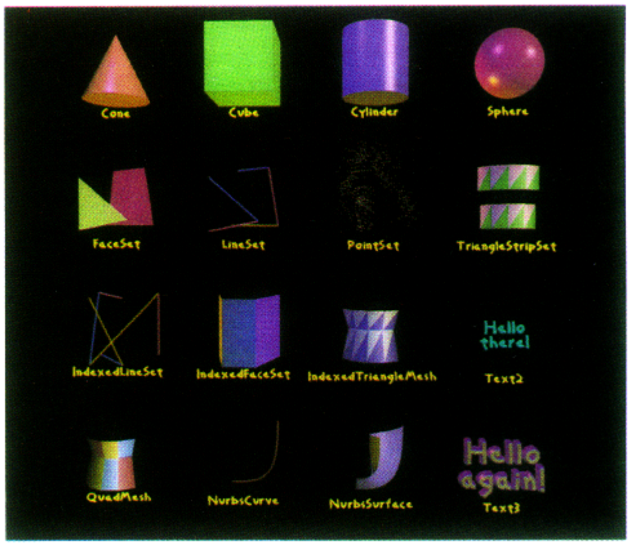

Figure 7. Examples of shape nodes provided by the toolkit.

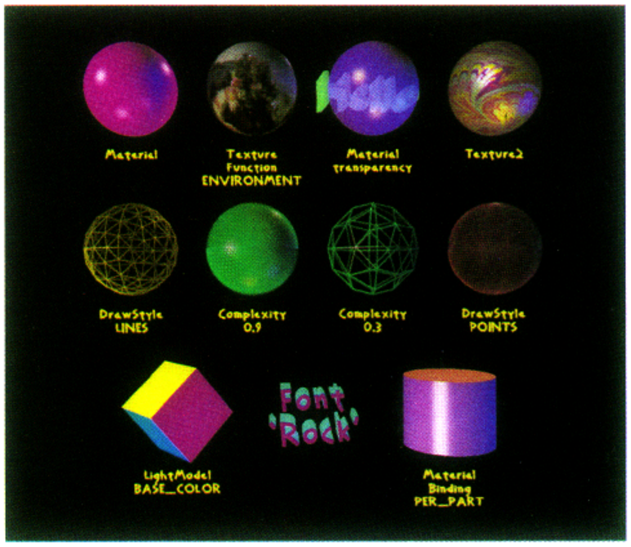

Figure 8. Examples of property nodes provided by the toolkit.

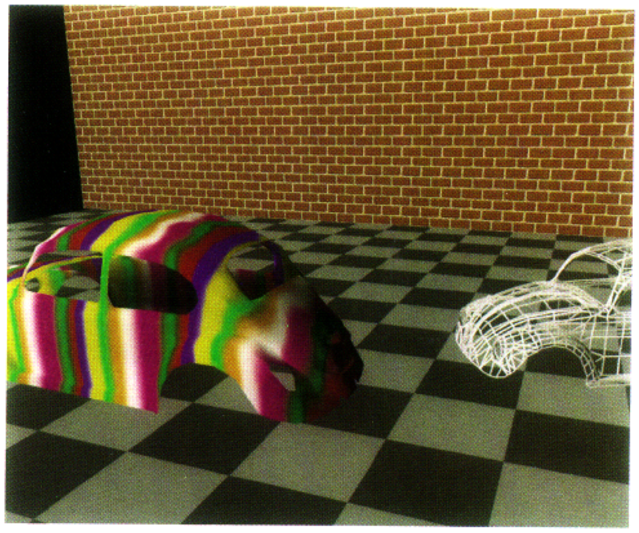

Figure 9. The result of rendering the scene graph in Figure 2.

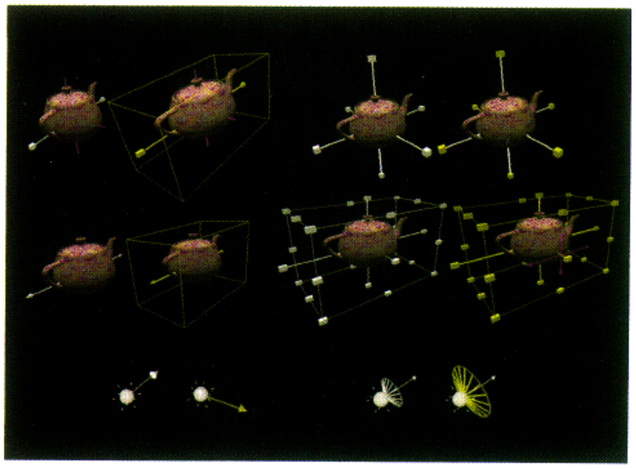

Figure 11. Examples of simple and compound manipulators. Each manipulator is shown in the inactive (left) and active (right) states. Clockwise, from upper left, the manipulators are: one-axis scale, jack, handle box, spot light, directional light, and one-axis translate.

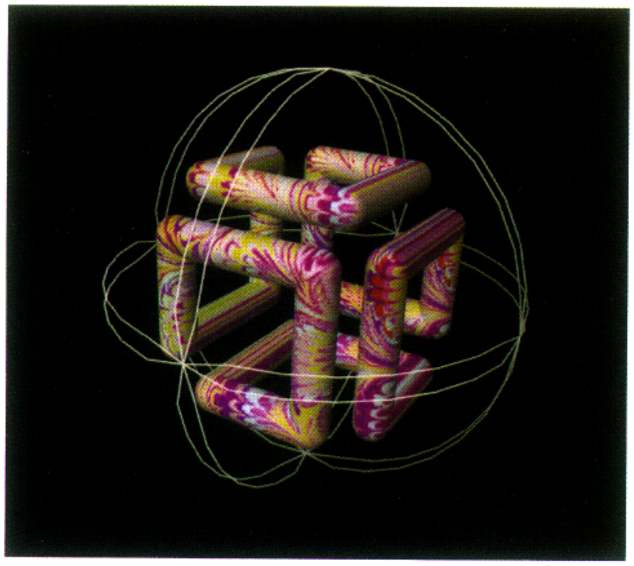

Figure 10. A trackball manipulator being used to rotate an object directly.

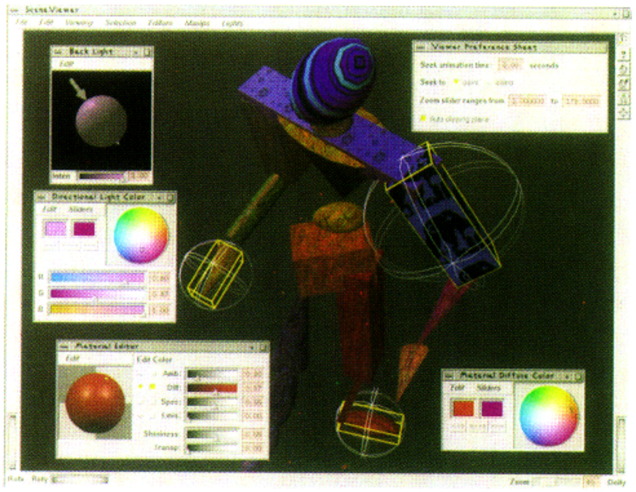

Figure 12. A sample application showing several forms of interaction. The large 3D window is a viewer component. Selected parts of the humanoid figure being viewed are highlighted with yellow bounding boxes. These parts are also enclosed by `Trackball` manipulators, which are used to rotate and scale objects directly. Other components in the figure are used to edit materials, light sources, and viewing options.

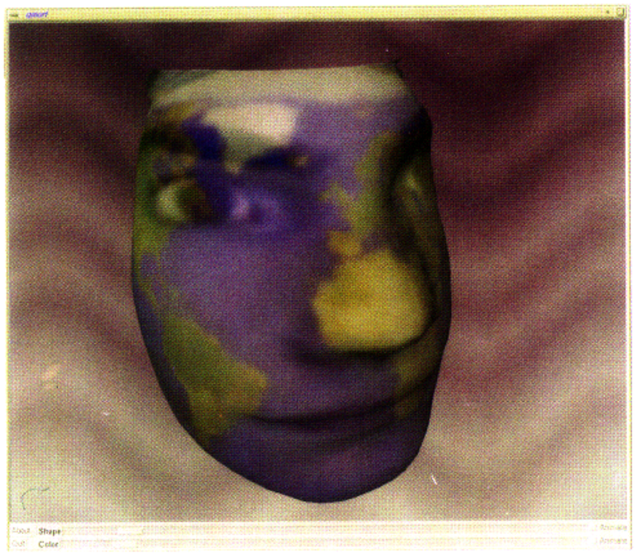

Figure 13. A three-dimensional morphing application. Geometry and color/texture can be interpolated independently. (Thanks to Rick Pasetto for letting us morph his face.)

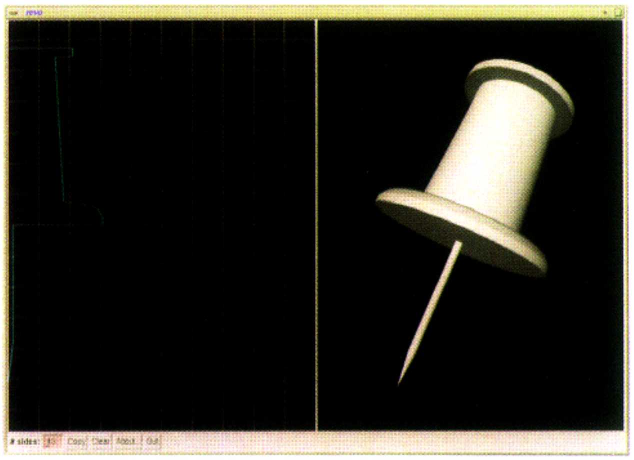

Figure 15. An application for creating objects of revolution. The same profile manipulator as shown in Figure 14 is used to edit the revolution profile.

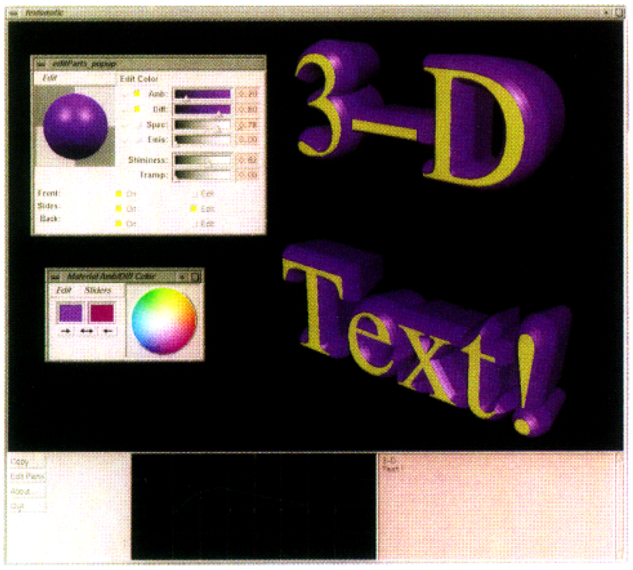

Figure 14. A three-dimensional text editing application. The small window at the bottom contains a profile curve manipulator which is used to extrude the 3D text.