IRIS Inventor, A 3D Graphics Toolkit

Paul S. Strauss

Silicon Graphics Computer Systems
2011 North Shoreline Blvd.
Mountain View, CA 94039-7311
pss@sgi.com
415-390-1091

Abstract

IRIS® Inventor™ is an object-oriented toolkit for developers of applications that incorporate interactive, three-dimensional graphics. Encapsulation of data and methods in high-level graphical objects allows Inventor to provide a significant improvement in usability to developers, as compared to the standard, low-level set of graphics primitives provided in most other graphics libraries. Inventor's object-oriented framework also facilitates extensions, which are necessary in the diverse and rapidly changing field of 3D graphics.

Introduction

Most three-dimensional graphics libraries available today provide very simple interfaces to application developers. *Immediate mode* libraries such as IRIS GL™ [3], Starbase™ [8], and RenderMan™ [10] provide a set of drawing commands that can be used by applications to create visual representations of modeled 3D objects. *Display lists* are used by packages such as GKS [4] and PHIGS+ [7] (and, to a lesser extent, some immediate mode libraries) to collect drawing commands into simple linear lists. Neither of these approaches truly exploits any correlation between modeled objects, such as chairs and airplanes, with the commands used to represent them visually; these commands usually consist of sequences of geometric primitives like polygons and lines. The Doré™ [2] and HOOPS™ [11] 3D graphics libraries present a slightly higher level abstraction of models, but they do not allow developers to take good advantage of key object-oriented techniques.

A problem with these conventional approaches is that developers are forced to maintain two distinct versions of their application models. Maintaining the relationships between the application data and the graphical "objects" used to represent them is a large burden that developers are forced to bear. In interactive applications, translating graphical gestures of a mouse or other input devices into 3D operations on a model is extremely difficult unless there is a tight coupling between application and graphical models. As a result, most 3D application writers using conventional graphics libraries do not provide much 3D interaction.

Developers may also be constrained by the set of graphical primitives provided by most libraries. For example, if a developer wants to include spheres as objects in an application, those spheres have to be converted into primitives that the library supports, such as polygons. While this conversion is acceptable for interactive rendering, it may not be suitable for operations such as picking. Picking a sphere, for example, can be done much more easily and exactly if the sphere is treated as an integral object, rather than as a collection of polygons.

Object-oriented techniques have proven to be a boon to graphics developers. The InterViews [1, 6], system has been successful for two-dimensional applications. However, few papers have been published on 3D systems. The work done at Brown University [12] presents a successful approach to this problem that is different from ours; they rely on a message-passing protocol to track changes to scenes.

OOPSLA'93, pp. 192-200

This paper presents IRIS Inventor, an object-oriented toolkit for developers of interactive, three-dimensional graphics applications. Object-oriented techniques provide two main advantages to developers:

- **Encapsulation**. Developers can embed rendering methods and other behaviors, including responses to input events, in higher-level objects. Many seemingly simple graphical operations require fairly complicated implementations; encapsulating them in objects relieves other developers of the need to code or even to understand them. Developers of proprietary algorithms can hide them safely in the objects' methods. There is also a simple solution to the duplicate data problem, since both application and graphical representations can be encapsulated in the same object.
- **Extensibility**. Because the state of the art of interactive 3D computer graphics changes rapidly, graphics development tools must be adaptable to new models and techniques. An object-oriented toolkit approach provides the necessary support and flexibility for its designers and its users.

IRIS Inventor is implemented in C++. The decision to use C++ was motivated by the language's availability, (relative) efficiency, and (relatively) short learning time for experienced C programmers. Another benefit is the ability to provide C bindings, in addition to the C++ bindings, using mostly automatic means. Developers can derive new C++ classes to extend the toolkit, although callback mechanisms are provided for easier extensibility in many cases.

The graphics-related features of Inventor have been documented elsewhere [5, 9]. The rest of this paper describes in more detail the objects provided by Inventor, noting the advantages and disadvantages of object-oriented design. The first section describes the application-programmer interface, using several examples for illustration. Following that are some details of the design and implementation of the toolkit and a discussion of extensibility.

Application-Programmer Interface

Components

Inventor provides a variety of classes to help developers of 3D graphics applications. At the highest level, there

```
main(int argc, char **)
{
      // Initialize Inventor and create window
      Widget appWindow = SoXt::init(argv[0]);

      // Read input file
      SoInput     in;
      SoSeparator *root = SoDB::readAll(&in);
      root->ref();

      // Set up and display viewer component
      SoXtExaminerViewer *viewer =
            new SoXtExaminerViewer;
      viewer->setSceneGraph(root);
      viewer->build(appWindow);
      viewer->show();
      SoXt::show(appWindow);
      SoXt::mainLoop();

}
```

Figure 1. Code for a simple application that reads any Inventor scene file and creates an instance of the `ExaminerViewer` component to view the scene. A view of of the component in action is shown in Color Plate 3.1.

are objects called *components*. These are pre-packaged, interactive application pieces that can be used as is within programs. Figure 1 illustrates the use of an `ExaminerViewer` component, which allows the user to interactively view a 3D scene from any direction, using the mouse to move the virtual camera. The set of Inventor components includes several other viewers, such as fly-through and walk-through, and editors for surface materials, colors, and textures.

Components are derived from the `RenderArea` class, which packages up a window, automatic rerendering, and event translation into a single object.

Nodes and Fields

The program in Figure 1 reads the representation of a 3D scene from a file in Inventor's file format. Once read in, a scene is stored as a directed acyclic graph of objects called *nodes*. Node classes can be divided into three basic categories:

- *shapes*[†], which represent geometric objects, such as cubes and spheres;
- *properties*, which are attributes of shapes, such as their surface materials and drawing styles; and
- *groups*, which have children and are used to collect nodes into hierarchies.

(Note that these categories are not restrictive; there could be a class of node that incorporates shape, proper-

[†]We use the word "shape" in this context to avoid confusion with the more generic term "object." We hope it works.

ty, and group characteristics.) A representative sampling of node classes supported by Inventor is given in Table 2.

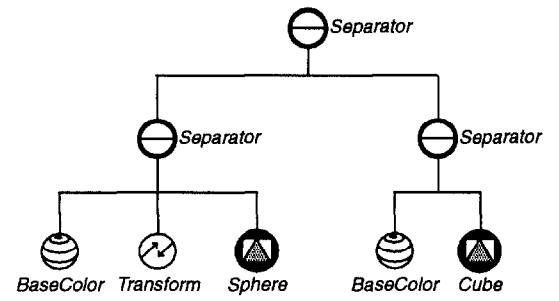
Instance data for nodes are stored in objects called *fields*. For example, the diffuse color, shininess, transparency, and other values in a `Material` node are stored in corresponding field instances. Field classes are categorized by the type of values (integer, float, vector of three floats, color, and so on) they contain. Fields provide a simple and consistent interface for setting and retrieving values, as well as automatic file read and write operations.

Several nodes may be required to specify fully a single 3D shape, since properties are stored separately from geometry. In Inventor, all aspects of a shape that are not specific to that particular shape class are implemented as separate properties, to maximize sharing and inheritance when possible. For example, coordinate values and surface normal vectors that are connected to form polyhedral shapes are stored in separate (property) nodes. Each shape node class is free to respond to whichever properties it wants to handle.

Applications can read scene graphs from files or build them at run-time. Figure 2 shows a sample of code that creates a very simple scene graph.

Shape nodes:	**Group nodes:**
Cone	Group
Cube	Separator
Cylinder	Switch
FaceSet	Selection
IndexedFaceSet	Array
IndexedLineSet	MultipleCopy
LineSet	
NurbsCurve	**Property nodes:**
NurbsSurface	BaseColor
PointSet	Complexity
QuadMesh	Coordinate3
Sphere	DrawStyle
Text2	Environment
Text3	Font
TriangleStripSet	LightModel
	Material
Light/camera nodes:	MaterialBinding
OrthographicCamera	Normal
PerspectiveCamera	NormalBinding
DirectionalLight	ShapeHints
PointLight	Texture2
SpotLight	TextureCoordinate2
	Transform

Table 1. Some Inventor node classes.




```
SoSeparator      *root, *sep1, *sep2;
SoBaseColor      *b1, *b2;
SoSphere         *sphere;
SoTransform      *xf;

// Create the subgraph with the sphere
sep1 = new SoSeparator;
b1 = new SoBaseColor;
b1->rgb.setValue(1.0, 0.2, 0.2);
xf = new SoTransform;
xf->translation.setValue(0.0, 3.0, 0.0);
sphere = new SoSphere;
sphere->radius = 0.3;
sep1->addChild(b1);
sep1->addChild(xf);
sep1->addChild(sphere);

// Create the subgraph with the cube
sep2 = new SoSeparator;
b2 = new SoBaseColor;
b2->rgb.setValue(0.2, 0.2, 1.0);
sep2->addChild(b2);
sep2->addChild(new SoCube);

// Put them together
root = new SoSeparator;
root->ref();
root->addChild(sep1);
root->addChild(sep2);
```

```
#Inventor V2.0 ascii
Separator {
      Separator {
            BaseColor {
                  rgb   1 .2 .2
            }
            Transform {
                  translation0 3 0
            }
            Sphere {
                  radius.3
            }
      }
      Separator {
            BaseColor {
                  rgb   .2 .2 1
            }
            Cube {}
      }
}
```

Figure 2. *Top*: a very simple scene graph containing a small red sphere positioned above a blue cube. *Center*: ASCII file format representing the scene. *Bottom*: code that builds the scene within an application. Color Plate 3.2 shows the result of rendering the scene with appropriate camera and light nodes added.

Actions and Traversal

Operations on a scene or part of a scene are applied using objects called *actions*. For example, a scene is rendered by applying an instance of the `RenderAction` object to the root node of the scene graph, like so:

```
renderAction->apply(sceneRoot);
```

Similarly, a scene may be written to a file by applying a `WriteAction` to the graph. Other actions include searching, computing bounding boxes, and picking. Another important action is the `HandleEventAction`, which lets objects in a scene process input or window-system events, allowing interaction handling to be built into scenes. A list of action classes provided by Inventor is shown in Table 2.

When an action is applied to a graph, the action traverses each node in the graph, maintaining *traversal state*. This state contains all current property values, such as surface material, lighting model, and drawing style, allowing these properties to be inherited and shared among shapes in the scene.

Different classes of group nodes implement different behaviors for traversing their children. The base `Group` class merely traverses all of its children in order (from left to right in the scene graph diagrams). The `Switch` node, however, usually traverses only the child indicated by the value in a field; this can be used, for example, to choose one of several representations of a shape. The ubiquitous `Separator` group node saves the current traversal state before traversing its child nodes and restores it afterwards, effectively isolating any state changes that occur in its children from the rest of the scene. Separators allow different parts of a scene to be modeled independently.

`CallbackAction`	Generic traversal with user callbacks
`GLRenderAction`	Renders
`GetBoundingBoxAction`	Computes 3D bounding box
`GetMatrixAction`	Computes cumulative transformation matrix
`HandleEventAction`	Offers nodes a chance to handle an input event
`RayPickAction`	Returns frontmost shape or all shapes intersected by a ray cast into scene
`SearchAction`	Looks for specific node or type of node
`WriteAction`	Outputs ASCII or binary representation

Table 2. Action classes

Animation

Inventor provides objects and mechanisms for adding animation to scenes and for tracking changes to scenes. A class of objects called *sensors* implements a callback mechanism. *Data sensors* invoke callbacks when a particular node or any node below it has changed. The `RenderArea` class uses a data sensor on the root node of a graph to detect when to re-render. *Timer sensors* invoke callbacks when a particular time of day is reached, or regularly at some specified interval. Since sensors operate through callbacks, they require programming, can be defined only during run-time, and cannot be stored in files.

A better animation mechanism is implemented using *field connections* and *engines*. A field in a node may be connected directly to another field, providing a simple constraint. Alternatively, a field may be connected to an output of an engine, which has other fields as inputs. For example, an engine that adds a 3D vector to a 3D location can be used to offset one shape from another by a given vector. A *global field* containing the current real clock time is always maintained by Inventor; connecting fields or engines to this field creates clock-based animation.

One advantage of using engines over sensors is that engines and their connections can be stored in files. For example, it is possible to save a spinning windmill in a file. When it is read in, the rotation of the mill is connected automatically to the real time field, so it spins continuously.

There are also some node classes that contain instances of engines to define moving properties. Figure 3 illustrates how an instance of the `Rotor` class can be used to spin the cube in the graph from Figure 2. The `Rotor` node is derived from the static `Rotation` node; on construction, it creates an instance of an `ElapsedTime` engine and connects its input to the real time clock and its output to the `rotation` field of the node. Therefore, this

node creates animated rotation when added to a scene. If we were to read the scene in Figure 3 into the application from Figure 1, the cube would spin automatically.

Paths

A node that is the child of more than one group is said to be *multiply instanced*. This feature allows for common subgraphs to be shared and re-used, reducing memory requirements, and is essential for large graphics applications. For example, a model of a bicycle might instance a subgraph representing a wheel twice. An unfortunate result of multiple instancing is that it is not possible to traverse upwards in a graph from a particular node. It also makes it impossible to refer unambiguously to a shape in a 3D scene, such as the bicycle's front wheel, with a single node pointer.

To make such references possible, Inventor includes a *path* object that contains a contiguous chain of nodes from a root of a graph down to some node. A path unambiguously refers to a node in a graph in a particular context, such as the hub of the front wheel of the third bicycle. Paths are returned by the picking operation, which is used to correlate 2D locations in a rendered window to 3D locations on surfaces of shapes; in typical use it returns a path to the shape visible under the cursor.

```
#Inventor V2.0 ascii
Separator {
      Separator {
            BaseColor {
                  rgb   1 .2 .2
            }
            Transform {
                  translation0 3 0
            }
            Sphere {
                  radius.3
            }
      }
      Separator {
            BaseColor {
                  rgb   .2 .2 1
            }
            Rotor {
                  # 1 revolution every 2 seconds
                  speed .5
            }
            Cube {}
      }
}
```

Figure 3. Adding an instance of a `Rotor` node to the graph from Figure 1. The rotor defines an animated transformation that is applied to subsequent objects. In this case, the cube spins.

Paths are also needed to provide context for removing or replacing nodes in graphs, since nodes do not provide access to their parents. This feature is important for supporting a class of objects called *manipulators*. A manipulator is an interactive 3D object that is used to edit another object. The most common types of manipulators are those that edit transformations applied to shapes; they typically employ some sort of surrogate shapes to provide a handle for users to interact with, as well as for displaying feedback. For example, the `Trackball` manipulator places a virtual sphere around the affected shape and converts mouse motion input into 3D rotations applied to that shape. A manipulator of this type replaces the appropriate transformation node while operating, then restores it when it is done, as illustrated in Figure 4. A path to the transformation node is used to supply the context so the manipulator can replace it.

Design and Implementation Details

Several decisions in the design and implementation of the Inventor toolkit were motivated by the need to satisfy concurrently the goals of simplicity, efficiency, and extensibility. These goals are often at odds with each other, so compromises must be made. As a result, some of the object classes and methods are not always intuitive. Some of the more important decisions are explained below.

```
// Assume we have a path to the transformation
// node in pathToXf ...
// Save the current transformation node - the
// tail of the path. Also save the parent info
// so we can restore it later.
SoTransform  *saveXf =
            (SoTransform *) pathToXf->getTail();
int    pathLen = pathToXf->getLength();
SoNode *parent = pathToXf->getNode(pathLen - 2);
int   childIndex = pathToXf->getIndex(pathLen-1);
// Create a new instance of the trackball
// manipulator
SoTrackballManip*manip = new SoTrackballManip;
manip->ref();
// Replace the existing transformation node in the
// path with the manipulator
manip->replaceNode(pathToXf);
// ... use the manipulator ...
// When done, the manipulator can be replaced
// as follows:
manip->replaceManip(parent, childIndex, saveXf);
```

Figure 4. Code to create and activate a `Trackball` manipulator. Color Plate 3.3 is a snapshot of one in action.

Shapes, Properties, and Groups

Some structured graphics libraries (such as PHIGS+) define geometric entities in terms of primitives, attributes applied to those primitives, and some hierarchical structures that hold them; sometimes the hierarchy is built into the primitives themselves. In Inventor, these things are all nodes. The decision to define scenes this way was based on several considerations.

The storage and traversal models have to be simple and consistent. If each class is derived from the same base class (`Node`), there are fewer inter-object relations that programmers have to understand. Creating or modifying any aspect of a scene uses the same paradigm, setting values of field instances within nodes.

The sets of shapes, properties, and traversal behaviors (groups) have to be extensible, and a consistent subclassing mechanism is advantageous. This scheme also allows hybrid classes to be created. For example, Inventor supplies a set of encapsulated sets of nodes called node kits, each of which bundles up a set of node instances into one entity; these classes provide a simpler scene construction and editing interface for less-experienced programmers.

Traversal behavior is the responsibility of each group class. The base `Group` node class does the most obvious thing, traversing all of its children from left to right. It allows property nodes under it to affect nodes to its right. This scheme allows a single group node containing several properties to be added to affect a scene or part of a scene. The `Separator` node adds save/restore to this behavior, which makes it useful for defining independent subgraphs. Other types of groups, such as the `Switch` and `Array` classes, implement traversal behavior for specialized purposes.

Inventor uses reference counting for nodes in the database because C++ does not provide garbage collection. Users must explicitly increment the reference count for nodes that are roots of graphs or for those nodes to which they maintain static pointers.

Nodes and Actions

Actions are implemented as separate objects, rather than as methods on nodes, for two reasons, both of which have to do with extensibility. The first reason is that an instance of an action object provides a convenient mechanism for setting parameters of the implemented operation, as well as for accessing the results of that operation. For example, the `RayPickAction` contains methods that let users set up a picking ray as a world-space vector or as a ray from the current camera through a pixel in a rendering window on the screen. It also contains methods to access the paths to the shapes that were picked. Separating these methods into the action subclass allows for greater leeway in the design of actions, including their parameters and return values.

The other main reason for having action classes is to allow developers to extend more easily the set of operations that can be applied to graphs. If these operations were methods on node classes, developers would not be able to add new ones, since that would require modifications to the base `Node` class, which they do not have access to. Instead, they would have to derive new node classes, one subclass for each node that was to implement behavior for the operation. This would rapidly become a mess for any developer intrepid enough to try it.

Because actions are separate objects, the standard C++ (single dispatch) virtual function mechanism cannot be used to determine the correct function to call for a specific action class and a specific node class. Instead, Inventor implements a multiple dispatch scheme using a two-dimensional table in which the rows represent action classes and the columns represent node classes. Each cell in the table contains a pointer to the function to call to apply an instance of the action class to an instance of the node class. This scheme is totally extensible, since rows and columns can be added to the table fairly easily. However, adding a new class requires some (hidden) work to maintain inheritance of functions, filling in empty cells in the table.

Because graphics developers are much more likely to add new node classes than action classes, virtual functions are used to make node subclassing much easier. For each supported action, the base `Node` class enters in the lookup table a method that calls a virtual function to implement that action. For example, the `Node` class enters a pointer to a static method that implements the `GetBoundingBoxAction`; this method just calls the virtual `getBoundingBox()` function. Developers can then create new node classes without ever having to deal

with the lookup table.

The table lookup scheme makes it easy to change the behavior of a node for a particular action. It also makes it possible to add new actions, including those that are derived from existing action classes. A developer can enter in the table a function that implements the action for the base `Node` class, so all nodes will use that method by default. The developer can override this behavior for specific node classes by entering other functions in the appropriate columns.

Of course, the implementation of the node/action lookup table requires the ability to determine class types at run time, a feature which C++ should but does not provide. Therefore, run-time typing is implemented explicitly for most classes in Inventor, using the `Type` class, which provides a set of class inquiry methods. It also includes a `createInstance()` method to make it possible to create an instance of a (non-abstract) class knowing only its type. Coupled with a name registry in the typing information, this method allows the Inventor file reader to create a node of the correct type after reading its name. The association of names with types also makes debugging and error reporting easier.

Event Handling

Distributing an input event such as a mouse button press to shapes in a 3D scene is much more complicated than it is in two dimensions. In a 2D application, it is fairly easy to determine which object is visible under the cursor, whereas it is not possible in 3D for a shape to know if it is visible without performing a pick operation on all objects in the scene.

The event handling mechanism in Inventor is designed to be as simple and flexible as possible without incurring a significant performance cost. The basic mechanism is implemented by the `HandleEventAction`, which is applied to the root of a scene to determine how to handle an event. (This action is typically initiated by the `RenderArea` class.) The action traverses the graph, allowing each node, in turn, to respond to the event. Most nodes, such as properties and shapes, do not care about events and ignore them. However, there are classes of nodes that process events. Each of these nodes may choose to handle the event and terminate the traversal, or it may do something with the event and let traversal continue.

The interactive manipulator objects have already been described. Another type of interactive object is the `Selection` node, a group node that maintains a list of paths to all objects under it that have been selected interactively. When given a mouse-down event, the `Selection` node first lets all of its children get a crack at handling it. If none of its children handled the event, the `Selection` node determines (by asking the `HandleEventAction`) which shape, if any, is under the cursor. It then updates its list of selected objects accordingly. The `Selection` node also highlights selected objects automatically.

State Elements and Caching

Interactive graphics applications should be fast enough that there is little or no lag between user gestures and rerendering of a scene. Although achieving this performance goal is not always possible for complex scenes, developers want to take advantage of every efficiency improvement. One way that an object-oriented graphics toolkit like Inventor can assist is to use *render caching*, which involves storing a faster representation of parts of scenes that are not changing from frame to frame. Inventor includes heuristics that build caches automatically when they are likely to improve rendering performance.

The implementation of caching is based on the objects used to represent traversal state, which are called *state elements* in Inventor. Each element represents a single, simple property of the state, such as the diffuse color of the surface material or the current list of coordinates.

Each type of element is stored in a stack in the state, allowing save and restore operations (as are required for `Separator` nodes) to be implemented easily. The essential feature of an element is that when its value is changed, the value is replaced completely. (There are some exceptions to this rule, such as geometric transformations, which are accumulated. These cases are handled through a slightly more complicated mechanism.) When a node being traversed changes the value of an element, it stores in the element a unique identifier that indicates who changed it and when. Given this information, it is fairly easy for a given state to determine how all of its elements' current values came to be set.

Caches are stored primarily in `Separator` group nodes, since they are effectively isolated from the rest of the scene. As is always the case, cache invalidation is the hardest part. In Inventor, caches are typically invalidated in one of two ways. If a node under a separator changes, the cache in that separator is invalidated immediately. This mechanism is implemented using the internal notification process built into all Inventor nodes and fields; when a field in a node changes, that node is notified, and it then notifies all parent nodes and paths that refer to it.

A more insidious case of cache invalidation occurs when a node above and to the left of a separator changes, where that node affects elements that are used by some node under the separator. To detect these cases, it is necessary for each separator to know which elements are used by nodes under it; if any of those elements is "different" when the separator is traversed again, the cache must be invalidated. Determining what is meant by "different" is up to each element class, but is usually a simple value or identifier comparison.

Alert readers may note that render caching is equivalent to storing two versions of the application data, which, as mentioned in the introduction, is usually something to avoid. However, there are two important differences here. One is that only part of the scene is being duplicated, and the application writer has complete control over what that part is. The second difference is that the cached version of the scene is created automatically by the toolkit, so the application is not at all responsible for maintaining links between the two representations.

Caching can also be used to speed up operations besides rendering. For example, cached 3D bounding boxes are used to speed up picking.

Extensibility

Developers can derive new classes from most of the public classes in Inventor, including nodes, fields, engines, actions, manipulators, components, and viewers. Most of these base classes provide a set of C preprocessor macros to facilitate subclassing.

Nodes

When deriving a new node class, a programmer has to implement virtual functions for whichever actions that node supports. Property nodes typically need only a subset of these functions; for example, the `Material` node has no effect during traversal of the `GetBoundingBoxAction`. Group nodes typically implement traversal for all actions, sharing a common method.

Shape classes are required to implement at least two methods: generating primitives (triangles, line segments, and points) representing the shape and bounding box computation. Shape classes may also provide rendering and picking methods for efficiency. If they fail to do so, the primitive generation method will be used to do the rendering or picking.

Of course, a node class that uses fields to store instance data does not have to implement file reading or writing methods.

Actions

Implementing an action class is somewhat complicated, requiring a method for each node class that affects or is affected by the action traversal. The dispatch table must be loaded with pointers to these functions.

However, an action class that is derived from an existing class inherits the methods from that class. Therefore, it is easy to define an action that differs from an existing action in the way one node is affected. For example, it is easy to derive a class from `WriteAction` that never writes out cameras, creating and registering a method for the `Camera` node that does nothing.

Manipulators

The surrogate geometries and feedback appearance of existing manipulator classes can be customized by editing resource files, so it is not always necessary to create a new class to create a new manipulator. Also, compound manipulator classes that use existing primitive manipulators can be created fairly easily. For example, the `Trackball` class illustrated earlier uses several simpler instances for the spherical and cylindrical rotation effects.

Creating a new manipulator class to edit a different type of node is similarly easy. For example, many manipulators are derived from the `Transform` node, which it replaces when active. Programmers that create a new

types of transformation nodes can define easily new manipulator classes derived from those nodes.

Summary

By using object-oriented techniques, IRIS Inventor is able to provide a higher level of 3D graphics programming to application developers, compared to conventional 3D libraries. Encapsulation of rendering, picking, and other behaviors in 3D objects means that users of those objects do not need to know how they are implemented and can take advantage of the built-in performance features. The object-oriented design of the library also makes extensibility much easier.

Most users of the toolkit have reported significant increases in productivity, compared to developing applications with a lower-level graphics library. In practice, reasonably structured Inventor scene graphs exhibit little or no additional performance costs in typical applications.

Inventor has been used to create a variety of applications, ranging from industrial design to motion picture special effects. A snapshot of a sample application is shown in Color Plate 3.4; this is a document presentation program that uses Inventor to incorporate 3D graphics.

Acknowledgments

The other members of the Inventor team at Silicon Graphics are Gavin Bell, Rikk Carey, Alain Dumesny, Dave Immel, Paul Isaacs, Howard Look, David Mott, and Josie Wernecke. Thanks to Alan Borning for his many helpful suggestions and his support.

References

[1] Paul Calder and Mark Linton, "The Object-Oriented Implementation of a Document Editor," *ACM SIGPLAN Notices (OOPSLA '92 Conference Proceedings)* **27**(10) pp. 154-165 (October, 1992).

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

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

[4] 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.

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

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

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

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

[9] Paul S. Strauss and Rikk Carey, "An Object-Oriented 3D Graphics Toolkit," *Computer Graphics (SIGGRAPH '92 Proceedings)* **26**(2) pp. 341-349 (July, 1992).

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

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

[12] 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).