An Object-Oriented Graphics Kernel

Submitted to ACM Transactions on Graphics, February 1997

Gene S. Lee
Department of Computer Science
University of British Columbia
Vancouver, B.C. V6T 1Z1

Abstract

A graphics kernel serves as interface between an application program and an underlying graphics subsystem. Developers interact with kernel primitives while the primitives interact with a graphics subsystem. Although the two forms of interaction closely relate, their optimal designs conflict. The former interaction prefers a process that closely follows the mental (or object-based) model of application development while the latter prefers a process that parses *display-lists*.

This papers describes RDI, an object-oriented graphics kernel that resolves the differences between the two interactions with one design. Developers explicitly assign intuitive relationships between primitives while an underlying process interprets the primitives in an orderly manner. The kernel's extensible design decouples the processes of modeling and rendering. Primitives dynamically communicate with graphics subsystems to express their purpose and functions. The discussion of the kernel's design entails its optimizations, its benefits toward simulation, and its application toward parallel rendering.

General Terms: Object-Oriented, Graphics Kernel

1 Introduction

A graphics kernel serves as an interface between an application program and an underlying graphics subsystem. The kernel provides developers with the tools to transform a programmer's *mental model* of application development into constructs that are readily understood by a graphics subsystem. A kernel's quality derives from its ability to naturally express and translate a programmer's mental model and to efficiently interact with various graphics subsystems.

Traditional graphics kernels, such as GKS-3D[10] (the Graphical Kernel System) and PHIGS+[23] (Programmer's Hierarchical Interactive Graphics System), endorse the usage of *display lists*, a sequential arrangement of data structures. Developers organize predefined primitives, such as those representing geometric forms and display characteristics, into a linear format that the kernel parses from beginning to end. Although efficient and simple, traditional kernels suffer from two drawbacks. First, display list modeling violates the mental model of application development. It forces developers to assemble primitives into a format that favors the way (state-based) graphics subsystems interpret data, not the way developers arrange data. Developers favor structures that enable them to assign and query characteristics to and from objects, and to construct relationships between the objects. Second, the kernels tightly couple modeling to rendering. Primitives support only those data structure that the graphics subsystem can readily understand. Although this restriction increases the efficiency of the rendering process, it limits the extensibility of the kernel and forces developers to interact with the graphics subsystem at a low level of abstraction.

Modern graphics kernels, such as DORE[13], INVENTOR[21], HOOPS[11], and PIX[9], combine object-oriented techniques with an underlying display list architecture. Object-oriented methods facilitate the manipulation and arrangement of primitive objects within display list objects. As with the traditional ones, the modern kernels do not support methods to form direct relationships between primitives. Display list objects constrain primitive objects to derive their relationship from their relative placement within a linear grouping. It is not possible to directly link one primitive with another. Geometric objects receive, not possess attribute objects. This failure to support direct relationships violates the principle of locality, one of the main tenets of object-oriented design. The behavior (or nature) of an object should be locally controlled. Elements which affect an object's state should be assigned or queried directly from the object itself.

This paper describes RDI (RASP's Design Interface), an extensible graphics kernel that bridges the gap between display-list modeling and object-oriented design. The benefits of both design methods are combined to form a simple, yet robust kernel. As developers establish intuitive relationships among graphics primitives, an underlying process arranges the primitives into an orderly structure. Modeling does not violate the principle of locality and the traversal of primitives occurs in a linear manner. To achieve extensibility, the kernel decouples the modeling process from the rendering process. Primitives dynamically communicate with the graphics subsystem to optimize the exchange of information.

2 Issues

Merging display-list modeling with object-oriented design introduces four major difficulties. First, display-list modeling, as employed by [21, 23], imposes a strict ordering of elements (objects) while object-oriented design does not. Order, not methods or operators, form relationships between elements. Although ordering of elements benefits rendering, it creates meaningless relationships and answers few queries. For example, the list `(transform,redColor,sphere)` improperly implies that a relationship exists between `transform` and `redColor`. `transform` and `redColor` affect `sphere`, not each other. In addition, `sphere` can not return its color. To obtain `sphere`'s color, the display-list must be traversed backwards.

Second, display-list modeling, as employed by [21, 23, 2, 12, 7, 6, 24], prefers late binding of primitives while object-oriented design prefers early binding. Late binding supports flexible, easy modification of graphics data while early binding supports instantaneous notification of changes to graphics data. For simple applications, late binding of primitives is acceptable. Renderers traverse collections of display-lists to visualize the state of geometric forms. However, for complex simulations, late binding leads to errors or inaccurate results. Unless extensive preprocessing of data occurs, simulations are slow to recognize important state changes. For example, if a simulation waits for a figure to reach a certain location, the figure must be rendered frequently to determine its position. Each rendering pass temporarily binds the figure to its transformations. The simulation runs the risks of operating improperly if the figure's transformations update faster than the rendering rate. Rendering, unnecessarily, becomes an integral element of a simulation's design. In addition, late binding promotes overly complex operators. To ungroup or amalgamate existing relations between primitives, operators must manipulate complex structures to reconstruct error-free display-lists.

Third, display-list modeling disparages both extensibility and encapsulation whereas object-oriented design promotes them. Display-list modeling performs optimally when the rendering process easily recognizes the function and purpose of every element within a list. Increasing the variety of elements or encapsulating information within the elements complicates the rendering process. Display-list modeling, as employed by [23], often limits the variety of list elements or supports only one type of renderer. Numerous complications arise when display list elements interact with multiple renderers. Some renderers support a limited set of operations while others solicit information in uncommon formats.

Finally, display-list modeling incites the development of lightweight primitives while object-oriented design does not. Lightweight primitives are easier to create and to manipulate, but harder to interpret and to relate. Many object-oriented approaches resolve this dilemma by creating monolithic primitives. Created statically[7] or dynamically[3], the primitives moderate many operations. They manipulate shape, color, textures, etc. Although simple, this plan does not scale well. As the kernel expands, primitives are harder to augment and to manage. They grow to encapsulate too much information. It becomes difficult to create global attributes, to parse operations in parallel, and to regulate individual functions. Monolithic primitives are functionally inflexible.

3 Overview of Solution

RDI layers object-oriented principles upon a display-list architecture. Developers arrange display-list elements intuitively while graphics subsystems interact with display-list elements sequentially. Without great loss in performance, RDI supports encapsulation and encourages extensions.

RDI consists of two object sets, modeling primitives and rendering primitives. Modeling primitives facilitate the construction of geometric models with qualitative attributes. Relationships between primitives stem from couplings and direct associations. Modeling is intuitive and meaningful. Rendering primitives organize virtual scenes and generate synthetic images. They interact with modeling primitives to transform scene data from bytes to pixels. Contrary to traditional methods, rendering primitives are interpreted by modeling primitives. After they communicate their capabilities, rendering primitives receive information. To optimize the exchange of information, both primitive types communicate their preferences. The union of their preferences determines the format in which information flows.

To support a wide variety of features, modeling primitives bind information early while rendering primitives bind information late. By binding early, modeling primitives acknowledge their relationships. They know who influences them and how. This permits them to instantaneously react to changes and to effortlessly respond to queries. By binding late, rendering primitives disregard established relationships. They derive implicit relationships from the ordering of incoming data. Recent data always relates to or replaces older data.

Although the two object sets bind information differently, they communicate without great difficulty. During the rendering process, modeling primitives carefully sequence their transmissions for rendering primitives to properly understand. From the ordered transmissions, rendering primitives infer relationships that are identical to those of the modeling primitives.

4 Object-Oriented Modeling

RDI partitions modeling primitives into three groups: *geometries*, *attributes*, and *transformations*. Geometries represent geometric shapes. They define the structure of a model. Attributes represent characteristic qualities. They specify the internal and external features of a model. Common attributes include those that regulate visual traits, and those that affect the interaction between RDI's primitives. Transformations represent "coordinate frame" modifiers. They alter the orientation, scale, and dimension of existing frames to produce new frames.

4.1 Geometries

Geometries consist of three types: *simple shapes*, *complex shapes*, and *mixed shapes*. Simple shapes are basic forms, such as spheres and cubes. Modifications affect their size, not their profiles. Complex shapes are elaborate forms, such as splines and indexed polyhedra. Unlike simple shapes, their appearance derives from the organization and interpretation of supplied data. Mixed shapes are compound forms. They intermix geometries, vertically and horizontally. Vertical construction produces hierarchical structures. Each level of the hierarchy represents a "part-of" the level above. Horizontal construction produces single-level structures. Each element of the single level "connects-to" the element before.

Initially, geometries are attribute-free. They acquire attributes to describe their characteristic qualities. However, all geometries possess one inherit and unmodifiable attribute, a *local coordinate frame*. Local coordinate frames specify modeling spaces that are independent of each other and most relevant to a geometry's needs. The final scale and orientation of geometries in world space depends on its local coordinate frame and its associated set of transformations.

4.2 Attributes

Attributes attach directly to the primitives they affect. Primitives accept attributes and refer to them as sources of information. Attributes that attach to geometries are called *local attributes* while those that attach to renderers are called *global attributes*. Local attributes possess limited scope. They affect only those geometries that reference them. For example, the texture map in Figure 1 affects only the geometry to its right. Global attributes possess unlimited scope. They affect the process in which rendering primitives interact with modeling primitives. Often, they act as default attributes and affect a large number of primitives. For example, if the texture map of Figure 1 had been attached to a rendering primitive, it would affect all the geometries within the scene. In essence, all geometries would possess the same texture. When in conflict, local attributes take precedence over global attributes. Global attributes take effect when local attributes are absent.

4.2.1 Attribute Association

Every geometry type maintains a list of attributes. As attributes attach and detach, the list updates automatically. Unlike similar structures found in other graphics kernels, the list never contains more than one copy of a particular type of attribute. Identical attributes replace each other as they join the list. A "copy-free" list liberates attributes from index values. Attributes always produce consistent results regardless of their list positions. Index values are not needed to add or detach attributes from geometries.

Geometries associate with identical attributes only when the attributes assemble within *mixed* attributes. Mixed attributes, of which there are many types, collate multiple attributes to produce single attributes. To create a single attribute, mixed attributes use various reductions techniques, such as demultiplexing and blending. Geometries interpret mixed attributes as composites. They ignore the many and recognize only the final result. Mixed attributes alter the way in which conflicting attributes are handled. Unlike other graphics kernels, resolution of multiple attributes is not simply a matter of order.

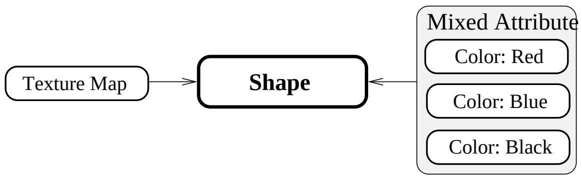

Figure 1: Simple and Mixed Attribute

For example, the geometric shape in Figure 1 possess four attributes, one texture map and three material colors. To avoid immediate replacement, the three material colors are stored within one mixed attribute. Of the four attributes, the geometric shape readily interprets only two, the texture map and one of the three colors. The shape interprets only the color that the mixed attribute selects.

4.2.2 Attribute Inheritance

Mixed shapes propagate their attributes to the geometries they intermix with. The intermixed geometries inherit the attributes as though they were their own. They interpret them and answer queries about them. However, they can not remove them. Intermixed geometries cease to inherit attributes that are removed indirectly or over-ridden. Indirect removal occurs when mixed shapes lose attributes and propagate the loss. Overriding occurs when the intermixed geometries gain attributes that conflict with inherited attributes. For example, the geometric shape in Figure 2 inherits two attributes, one display list and one complexity.

The shape interprets the display list, but ignores the complexity. The inherited complexity attribute is over-ridden by a local complexity attribute. Local attributes always take precedence over those that are inherited.

4.3 Transformations

Transformations transform coordinates frames. Independently or as members of composites, they scale, rotate, and translate coordinates frames about axes and points in space. Unlike attributes, they are not inherent properties of geometries. They modify, not describe space. Therefore, transformations may *attach* or *associate* with geometries. Attaching transformations alter the local coordinate frames of geometries whereas associating transformations introduce coordinate spaces onto which geometries map their local frames.

4.3.1 Attachment

Like attributes, attaching transformations link to geometries directly. Geometries, primarily simple shapes, accept the transformations and refer to them as local coordinate frame modifiers. They modify the basic shape and appearance of the geometries by altering the scale and orientation of local spaces. For example, anisotropic transformations change spheres to ellipsoids and squares to parallelograms.

4.3.2 Association

Transformations associate with geometries via mixed shapes. For every geometric shape, mixed shapes maintain a *translist*, an ordered list of transformations. Each translist directly affects the local frame of its associated geometry. Transformations are ordered to guarantee deterministic results and facilitate rapid editing. Initially, transformations are ordered upon entry into the translist. To retrieve or selectively alter the translist, transformation are referred to by their list index, type, or given name[1]. In addition to ordering transformations, the translist acts as a transformation. Its value is equal to the multiplicative accumulation of its contents[2]. The translist reacts to mathematical operators and responds to general queries. However, the translist is not directly modifiable. Its value changes only when its set of transformations changes.

As translists pair with geometries, they identify themselves to the geometries. This permits the geometries to obtain information directly from the translist that affects their space. Simple and complex shapes reference their translists to answer queries and to interact with renderers (see section 5.3). Mixed shapes reference their translists to create *hierarchical translists*. Hierarchical translists are translists that inherit the values of other translists. They form as geometries arrange hierarchically or vertically. For example, the mixed shapes in Figure 3 repeatedly embed inherited translists while the mixed shapes in Figure 4 repeatedly embed connective translists.

4.3.3 Hierarchical Update

Hierarchical translists consistently reflect the value of their accumulated set of transformations. As levels change value, they instantly

[1] All of RDI's primitives are identifiable by name or by type.

[2] An empty translist is equal to the identity matrix.

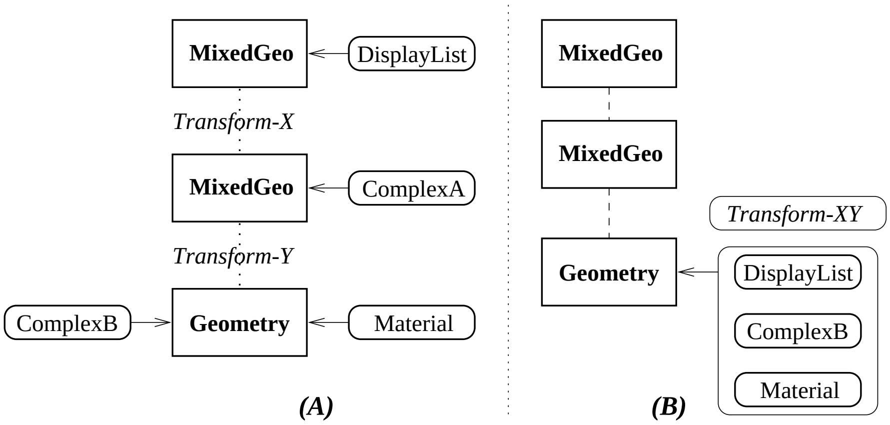

Figure 2: Attribute Inheritance

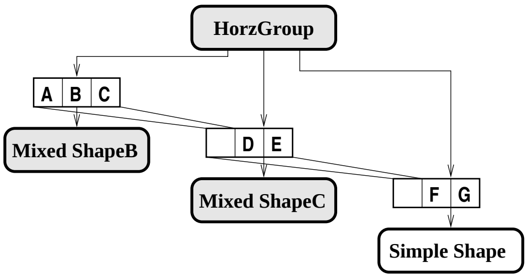

Figure 4: Connective Translists

propagate their updated values to lower levels. Propagation continues until all dependent translists update. Level changes occur when transformations alter state, compositions of translists change, or translists receive propagated update notices. To rapidly update hierarchical translists, transformations and translists maintain links to their dependents. Dependents rely upon the transformations and translists to compute an accumulative value. Immediately after transformations and translists change, they notify their dependents. For example, when transformation `B` in Figure 3 changes state, it informs the translist for `MixedShapeB` to recompute its value. Immediately afterwards, the new value propagates to the translist for `MixedShapeC`, and forces the eventual update of the translist for `SimpleShapeC`.

4.3.4 Design Benefits

Despite functional reasons, there are benefits to not link transformations to geometries. First, it emphasizes that transformations possess a variety of geometric interpretations. As noted by [5], transformations interpret as a change of coordinates, a transformation from one space onto another, or a transformation from a space onto itself. Transformations that link directly to geometries promote the latter of the three interpretations. Points within the geometries move while the coordinate system remains fixed. This interpretation works well to position objects, but fails miserably to relate objects, such as those found in articulated figures. Second, it simplifies the application and manipulation of transformations. Only mixed shapes accept, concatenate, and embed transformations. Simple and complex shapes use transformations while mixed shapes manage transformations.

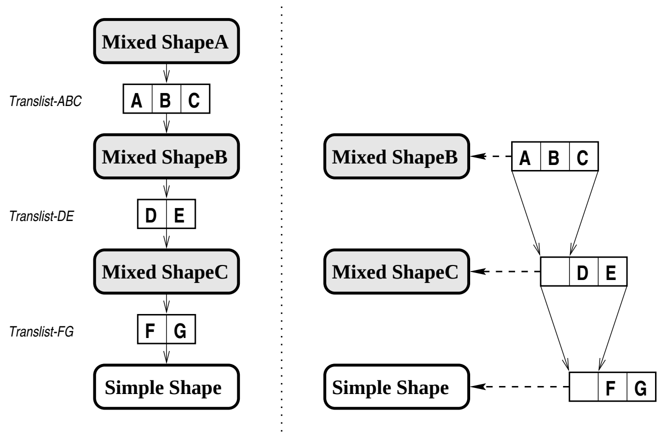

Figure 3: Hierarchical Translists

4.3.5 Implementation

RDI employs the handle/body idiom[4] to implement transformations. Transformations consist of two classes: an *outer class* (handle) and an *inner class* (body). The outer class manages interactions while the inner class reacts to interactions. Together, the two classes act as a *composite object*. The following class definition represents a transformation that employs two classes.

```
class MatrixRS: public MatrixBase {
     static void setMatrixRep( MtxRep );
  protected:
     static MtxRep  MtxRepr;
     MatrixRep     *mtxRep;
  public:
     Matrix();
     ...
};
```

The inner class `MatrixRep` implements the operations of outer class `MatrixRS`. Subclasses of `MatrixRep` interface with various matrix packages and hardware functions, such as those found in RogueWave[18] and OpenGL[17].

A simplified implementation employs inner classes as subclasses of outer classes. Outer classes act as base classes while inner classes act as specialized classes. Class inheritance replaces the run-time interaction between the two classes. Although this design is simpler, the handle/body idiom offers greater benefits. The handle/body idiom simplifies user interactions with packages of related classes. All transformations, irregardless of their implementation - hardware or software, are of the same type. The outer class remains the same while the inner class varies. As a single type, transformations are not subject to the pitfalls of inheritance. They easily interact with overloaded operators and endorse a simpler syntax. Transformations pass freely from one operation to another without excessive use of pointers. The handle/body idiom also reduces the impact of change on coding and recompilation. The idiom localizes the invocation of derived class constructors and establishes the representation of transformations at run-time. Altering applications to use new representations of transformations is simple and clear.

4.4 Reducing Growth

As the number of primitives within an application grows, memory requirements and model complexities soar. To reduce the severity of this growth, RDI permits primitives to duplicate and hierarchies to collapse. To duplicate, primitives support two copy methods, *deep* and *shallow*.

Deep copies the internal structure of primitives to produce exact duplicates. Duplicates act and react independently. Shallow references the internal structure of primitives to produce clones. Clones possess identity but share state. They act as one monolithic object. If any clone or the original changes state, everyone changes state. Clones are extremely useful to reuse geometries. For example, the following segment of code produces four copies of a sphere.

```
GeoSphere *sphr = new GeoSphere();
GeoHGroup *grp  = new GeoHGroup();
grp->insert( sphr->deep( NO_ATTRIB );
grp->insert( sphr->deep( SAME_ATTRIB );
grp->insert( sphr->shallow( SHALLOW_ATTRIB );
grp->insert( sphr->shallow( DEEP_ATTRIB );
```

The first two copies are duplicates while the last two copies are clones. The arguments to the copy methods dictate what the copies possess for attributes. The first copy is without them, the second copy references the originals, the third copy retains clones, and the last copy obtains duplicates.

To collapse hierarchies, mixed shapes support methods to eliminate intermediate levels. As intermediate levels disappear, hierarchies grow shorter and consume less memory. To eliminate a level, mixed shapes inherit the components of intermediate nodes. The intermediate nodes vanish and the components ungroup. For example, the following two lines of code collapse the hierarchy shown in Figure 5.

```
mxShpA->ungroup( mxShpB, CONCAT_TRANS, LOOSE_ATTRIB );
mxShpA->ungroup( mxShpC, MERGE_TRANS, CLONE_ATTRIB );
```

The first line eliminates `mixedShapeB` while the second line eliminates `mixedShapeC`. The second and third argument of each line determines the fate of the disappearing node's transformations and attributes. The first line purges the attributes and concatenates the transformations to the transformations of the components. The second line clones the attributes for the components and merges the transformations with the transformation of the components.

5 Object-Oriented Rendering

RDI partitions rendering primitives into two groups: *views* and *renderers*. Views manage the content of virtual scenes. They control lighting, timing, and display. Renderers interpret the content of virtual scenes. They inspect modeling primitives to generate synthetic images.

5.1 Views

Views consists of four types: *windows*, *lights*, *cameras*, and *settings*. Windows present images of scenes. They visualize the output of renderers. Windows provide users with methods to open and close user-interface displays, to buffer data, and to delimit the viewing sizes of rendered images. Lights illuminates scenes. They regulate the distribution and characteristic qualities of light sources. Cameras specify viewing frustums. They circumscribe portions of world space for renderers to observe. Although they are not modeling primitives, lights and cameras are manipulatable objects. They possess location, orientation, and direction. Settings produce and render scenes. They determine its geometric content, and control its global parameters and temporal state. To render a scene, settings simply pair renderers with geometries.

5.2 Renderers

Renderers interact with modeling primitives to produce synthetic images. Unlike traditional methods, renderers do **not** interpret primitives; the primitives interpret them. Geometries communicate with renderers to specify the geometric contents of a scene while attributes communicate with renderers to specify the characteristic qualities of a scene. Renderers receive, not retrieve information. Common retrieval processes, such as the traversal of hierarchical structures and the transformation of data, are executed by geometries. Shifting the responsibility of interpretation promotes several benefits: it augments the flexibility of the RDI's design; it supports the con-

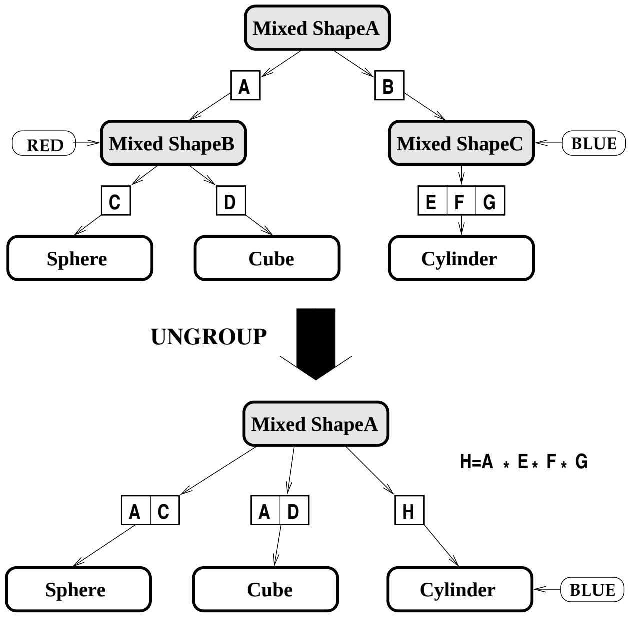

Figure 5: Collapsing an hierarchy

struction of new primitives; and facilitates parallel rendering.

5.2.1 Geometries

A loose coupling exists between geometries and renderers. The format and quantity of information that passes from one to the other is not dependent upon a strict protocol. It depends on the intrinsic capabilities of each. Not all shapes and renderers manage information alike. Each prefers to supply or receive information in formats that suit their needs. Therefore, to maximize the needs of both, shapes and renderers compare capabilities. Those capabilities that match determine the formats in which information flows.

To compare capabilities, shapes pass their preferences to renderers. Renderers compare the preferences to their own and store the results in a table[3]. Shapes remain oblivious to the results until interaction occurs. When shapes are ready to supply information, they receive the results and act accordingly. Generally, this entails caching data, performing transformations, and interacting with attributes. Shapes never cache the results they receive from renderers. It is not uncommon for renderers to temporarily alter their preferences and modify their tables as an application progresses. Often, this occurs when renderers acquire global attributes or receive information from local attributes (see 5.2.2).

Mixed shapes interact with renderers differently. They retrieve, not supply, sources of information. They traverse collections of geometries, often arranged hierarchically, to associate renderers with renderable shapes. For example, the mixed shape in Figure 7 notifies its shapes to render in the following order: `sphere`, `cube`, and `cylinder`.

[3] The size of a renderer's table is function of the number of geometric types, not the number of geometric instances. For example, one hundred spheres occupy only one table, while one sphere and one cube occupy two entries.

5.2.2 Attributes

Like geometries, attributes maintain a loose coupling with renderers. They interact to compare preferences and to exchange information in compatible formats. However, unlike geometries, attributes can simply pass themselves to a renderer. If a renderer indicates that it prefers to interpret an attribute, the attribute temporarily attaches itself to the renderer and acts as a source of information. For example, almost all renderers accept color attributes. Therefore, when color attributes are encountered, the renderers refer to them directly and invoke their member functions.

When attributes, global or local, attach themselves to renderers, they nullify *default attributes*. Maintained by renderers, default attributes establish the global state. They apply to geometries that lack a complete set of local attributes. For example, if renderers encounter shapes that lack color attributes, they refer to default attributes to establish coloring information. Global attributes completely replace default attributes whereas local attributes simply override them. As noted in section 4.2, local attributes prevail over global attributes. However, renderers can prevent this if they wish. They accept instructions to disregard the effects of local attributes. This permits renderers to alter quickly the characteristics of entire scenes. Local attributes need not be replaced, or deactivated, to induce a variety of effects, such as displaying scenes in wireframe mode or presenting scenes without textures.

Some attributes, such as `Complexity` and `DisplayList`, interact with geometries, not renderers. During the rendering process, they alter the way geometries operate and supply information. For example, `Complexity` attributes dictate how much information geometries produce. To create smooth surfaces, the attributes force geometries to produce and forward large quantities of surface detail.

	Initialization
1	Geometries inform renderers of preferences.
2	Attributes inform renderers of preferences.
	Visualization
1	Render non-inherited attributes.
2	Initialize translist.
3	Pass identity to renderer and accept format.
4	Compute information in required format.
5	Transform information and pass to renderer.
6	Deinitialize translist.
7	UnRender non-inherited attributes.

Figure 6: Rendering Process

5.3 The Rendering Process

The rendering process consists of two phases: *initialization* and *visualization*. During initialization, view primitives introduce modeling primitives to rendering primitives. The modeling primitives identify themselves and communicate their preferences, as described in section 5.2. Communication between primitives does not occur on an instance to instance basis. Instances of rendering primitives interact with classes of modeling primitives. Because modeling primitives do not differ from instance to instance, only one representative from each class need communicate.

During visualization, view primitives initiate interactions between geometries and renderers. They instruct geometries to communicate their visual state and their characteristic qualities. To do so, geometries follow seven steps, as shown in Figure 5.3. First, they instruct to their attributes to interact. Attributes interact with renderers in any order and in any way they deem best. Order is not important because geometries never possess or inherit multiple attributes of the same type, as described in section 4.2.1. Next, they initialize their translist. Often, this operation produces no results. However, there are times when this step is essential. Some attributes and some implementations of matrices require it. In the next two steps, they identify themselves, accept a format, and compute geometric information. Afterwards, they transform the information and pass it forward. In the last two steps, they deinitialize their translists and withdraw their non-inherited attributes.

Withdrawing attributes retract their previous interactions. They restore the states of altered primitives by replacing current values with old ones. For example, before color attributes take effect, they retain pointers to working color attributes. When the color attributes withdraw, they simply reinstate the working attributes. The process of withdrawing attributes is similar to the process of popping-state, as employed by traditional display-list traversal algorithms. Both undo the effects of previous actions.

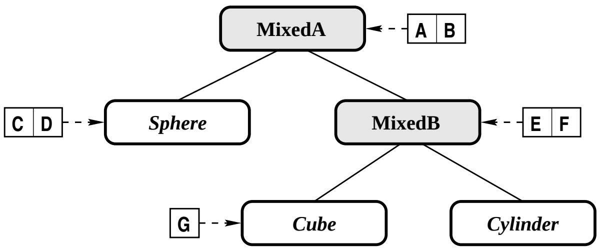

Figure 7: Attributes D, E, H must withdraw

The diagram in Figure 7 illustrates the necessity to withdraw attributes. As the rendering process traverses `MixedA`, it sequentially induces attributes `A` though `G` to interact. Unless attributes `C`, `D`, and `G` withdraw, they will improperly improperly affect `cylinder`. Only attributes `A`, `B`, `E`, and `F` apply to `cylinder`. These attributes need not withdraw until `MixedA` ceases to interact.

5.3.1 Accelerations

RDI readily accepts two techniques to accelerate the rendering process: *subclassing* and *parallelization*. Subclassing extends existing primitives to create new primitives that operate quickly. New primitives redefine general methods to accept fewer parameters or to pass fewer arguments. In addition, they limit their associations to reduce their communications. For example, subclassing can create specialized attributes that operate only with OpenGL-based renderers. These types of attributes neither identify themselves nor communicate their preferences.

Parallelization produces faster images by increasing the communication rate between RDI's primitives. Normally, rendering primitives interact with modeling primitives one by one. They receive information sequentially. With parallelization, information flows concurrently. Viewing primitives simultaneously associate geometries with renderers while geometries simultaneously associate attributes with renderers. Simultaneous association succeeds for two major reasons. First, geometries retain links to all their attributes and transformations. Hierarchical traversal of complex structures is unnecessary. Although some attributes and transformations interact more often than before, the gain in overall efficiency is much greater. Second, geometries never retain multiple attributes of the same type. Attributes never conflict; therefore, they may interact at the same time with the same renderer.

6 Example

This section demonstrates the utility of RDI with a small application. Consisting of three phases, the application furnishes a scene with windows, cameras, models, and renderers. The application uses local and global attributes, hierarchical and vertical groupings, and clones.

The first phase, shown in Figure 8, adds a camera and light source to a scene. The camera defines a perspective view while the source emits directional light. The first two lines initialize the toolkit and define the internal representation of a matrix. Hidden from the user for the remainder of the application, matrices will use the Roguewave library to perform mathematical operations.

The second phase, shown in Figure 9, adds models to the scene. Initially, it creates one

```
                  initialize the toolkit & create a scene
(1)   RASP::init();
(2)   MatrixSlot::setMatrixmRep( MatrixBase::ROGUEWAVE );
(3)   Scene world;
                  create a window
(4)   fRect windRect( 0, 0, 600, 600 );
(5)   GLWindow *wind = new GLWindow( windRect );
                  create a perspective camera
(6)   PerspCamera *kamera = new PerspCamera;
(7)   kamera→setFView( AngleRS(45), AngleRS(45) );
(8)   kamera→setView( Point3(20,35,110), Point3(0,0,0));
(9)   kamera→setWindow( wind );
(10)  world.addObject( kamera );
                  create a directional light
(11)  DirectLight *light = new DirectLight(dVector(0,1,0));
(12)  world.addObject( Point3( 100, 100, 100 ), light );
```

Figure 8: Phase 1: Creating the World

torus and two spheres - one of which is a clone of the other. The first sphere accepts two attributes, one material and one drawing style. The material colors the sphere sea-green while the drawing style displays the sphere as lines. The clone accepts a material and texture. The texture attribute applies a checkerboard texture to clone. Following their creation, the torus and spheres join `comp`, a mixed shape that joins the geometries vertically (lines 12-15). `comp` associates transformation with the spheres to move one to the left and the other to the right.

Afterwards, the second phase creates `comp2`, a shallow copy of `comp`. The torus within `comp2` receives a material attribute that colors it dark-yellow. To obtain the identity of the torus, the phase simply queries `comp2` for a torus. `comp2` returns the first torus within its collection of geometries. Finally, phase two adds `comp`, `comp2`, and a new cylinder to `compH`, a mixed shape that joins primitives horizontally (lines 19-22). `comph` concatenate the transformation of geometries as they join the group. For example, the cylinder, linked last, associates with `mtxL`, `mtxR`'s, and `mtxU`.

The last two lines of the second phase link `compH` with a "model" (object holder, see [14]) and defines a local default color. The default color, blue, colors those geometries of the model without a material attribute, such as the cylinder and the shallow sphere. Had the default color not been specified, the sphere and cylinder would have inherited its color from the renderer that renders them.

The third phase, shown in Figure 10, renders the scene twice. It creates two renderers and applies global attributes to each. The first renderer, `glRend`, renders the scene with GL (SGI's graphics library)[15]. The resultant image, shown in Figure 11, appears in a `GLwindow` (line 5, first phase). `glRend`'s global attributes regulate the scene's ambient light and specify the scene's texture mapping environment.

The second renderer, `opRend`, produces an Optik[1] scene description. Optik parses the scene description to produce a high-quality image. `opRend`'s global attribute applies a "checkerboard" texture to all the elements within the scene. The altered scene is seen in Figure 12.

For a quick comparison between RDI and Inventor, see Appendix A. Appendix A generates phases two and three of the scene with lines of Inventor.

```
create shapes & assign attributes
GeoTorus *torus = new GeoTorus( 10, 20 );
GeoSphere *sph1 = new GeoSphere( 25. );
GeoSphere *sph2 = (GeoSphere*) sph1→shallow( Qualities::NO_ATTRIB );
sph1→setAttrib( new Material( ColorBase::SEA_GREEN ));
sph1→setAttrib( new DrawingStyle(DrawingStyle::LINES ));
sph2→setAttrib( new Material( ColorBase::ENGLISH_RED ));
sph2→setAttrib( new Texture( Texture::CHECKERBOARD ));
GeoCylinder *cyl = new GeoCylinder(20,10);
create transformations
MatrixRS *mtxL = new MatrixRS( MatrixRS::TRANS, Point3(0,0,-25) );
MatrixRS *mtxR = new MatrixRS( MatrixRS::TRANS, Point3(0,0,25) );
MatrixRS *mtxU = new MatrixRS( MatrixRS::TRANS, Point3(0,35,0) );
create mixed shape - vertical grouping
GeoComp *comp = new GeoComp;
comp→addGeom( mtxL, sph1 );
comp→addGeom( mtxR, sph2 );
comp→addGeom( torus );
copy comp & assign material to its torus
GeoComp *comp2 = (GeoComp*) comp→shallow( Qualities::NO_ATTRIB );
GeoBase *torus2 = comp2→getGeom( GeoTorus::getClassTypeId() );
torus2→setAttribute( new Material( ColorBase::DARK_YELLOW ));
create mixed shape - horizontal grouping
GeoHorzComp *compH = new GeoHorzComp;
compH→addGeom( mtxL, comp );
compH→addGeom( mtxR, mtxR, mtxR, comp2 );
compH→addGeom( mtxU, cyl ));
assign geometry to model
Model *obj1 = new Model( Point3(0,0,0), compH );
world.addObject( obj1, ColorBase::BASIC_BLUE );
```

Figure 9: Phase 2: Adding Models to the World

7 Related Work

As previously mentioned, there have been numerous attempts to extend and to simplify the architecture of the basic graphics kernel. New extensions introduce object-oriented principles, multi-layered designs, and interactive structures. The following two sections compare and contrast RDI's architecture to recent proposals and popular commercial packages. The first section discusses modeling techniques while the second discusses rendering interfaces.

7.1 Modeling

RDI employs modeling techniques similar to those found in Geo++[24], Yart[3], QuickDraw-3D[2], Inventor[21], Groop[12], Grams[7], Tbag[8], and VRS[6]. All nine kernels employ object-oriented principles to encapsulate or to simplify the interaction between users and graphics subsystems. The top sheet in Table 1 compares the nine kernels in twelve categories. For each category, kernels receive bullets for strong designs, blanks for weak designs, and circles for minimal designs. Minimal designs support a limited range of functions and burden users to use awkward constructs or to invoke greater number of operations.

```
create renderers
GLRenderer3D *glRend = new GLRenderer3D;
OptikRenderer *opRend = new OptikRenderer;
assign global attribute
glRend→setAttrib( new LightingModel( new RGBColor(.2,.2,.2) ));
glRend→setAttrib( new TextureEnv(TextureEnv::MODULATE) );
opRend→setAttrib( new Texture(Texture::CHECKERBOARD) );
render scene with GL
kamera→setRenderer( glRend );
world.initSetting();
world.renderAll();
render scene with Optik
kamera→setRenderer( opRend );
world.initSetting();
world.renderAll();
```

Figure 10: Phase 3: Rendering the World

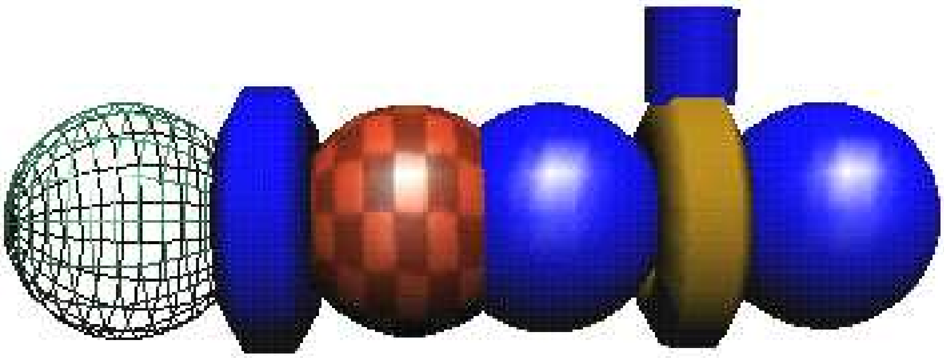

Figure 11: Model rendered with GL

All but two kernels implement their designs with C++. C++ is popular and supports object-oriented constructs. The two remaining kernels, Geo++ and QuickDraw-3D, use Smalltalk, which is not popular, and C, which is not object-oriented. The learning curves for these two kernels is longer than those based upon common object-oriented languages. One forces users to learn unfamiliar keywords while the other forces users to decipher pseudo-object-oriented structures.

Some kernels, like RDI and QuickDraw-3D, support lightweight primitives while others, like Yart and Grams, support heavyweight primitives. Lightweight primitives manage few operations and form relationships at run-time. Popular lightweight primitives include shapes and surface characteristics. Conversely, heavyweight primitives manage numerous operations and form relationships at compile time. For example, in Grams, shapes manage everything. They inherit operations from superclasses to control their geometry, color, and material properties. Although this scheme is simple, it consumes great resources and restricts itself to few optimizations. As the attribute set grows, shapes

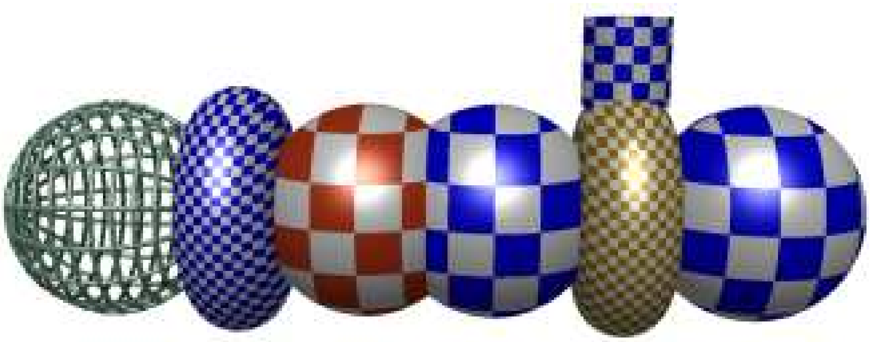

Figure 12: Model rendered with Optik

Modeling	RDI	Yart	Geo++	Qui3D	Inven	Groop	Grams	Tbag	Vrs
language	C++	C++	Smltk	C	C++	C++	C++	C++	C++
lightweight primitives	●		●	●	●			●	●
extensible attributes	●	○	●	●	●	○	○	●	●
principle of locality	●	●	○	○		○	○		
early binding	●	○							
orderless attributes	●	●	○	○		●	●	○	○
separ transformations	●		○	○	●	○		○	●
connectivity	●	●	○	●	●	○	○		●
ungroup/flatten	●		●	●	●				●
constraints								●	
interaction		○	○	●	●	○		●	○
animation					○	○		○	

Rendering	RDI	Yart	Geo++	Qui3D	Inven	Groop	Grams	Tbag	Vrs
loose coupling	●	○		○		○	○	●	●
global attributes	●	○	○	●	○				○
parallelizable	●	○	○	○	○			○	

Table 1: ●: full support, ○: partial support

inherit greater number of operations and consume greater amounts of memory.

Six of the nine kernels adhere to the principle of locality. Attributes apply directly to objects and objects acknowledge their attributes. However, only two kernels, RDI and Yart, permit objects to acknowledge their inherited attributes and their inherited transformations. Most kernels disregard these important relationships. Objects are permitted to acknowledge only a subset of the primitives that affect them. Apart from limiting queries, this drawback prevents objects from binding early. Early binding is fruitless if inherited attributes are ignored.

Of all the kernels, only Inventor forces users to order the application of attributes. The remaining kernels employ a variety of techniques to eliminate order. The simplest technique, applied by Yart, Grams and Groop, denies objects from possessing more than one attribute of the same type. Attributes never conflict; therefore, order is insignificant. Although this technique is simple, it complicates the process of rapidly altering an object's attributes. Techniques of greater complexity issue priorities, such as those found in Vrs and RDI. Vrs applies priorities directly to attributes while RDI apply priorities directly to (attribute) containers.

Most kernels regard transformations as attributes. They apply transformations in the same way they apply attributes. Only RDI and INVENTOR resist applying transformations directly to shapes. Transformations link shapes, not describe them. QUICKDRAW-3D promotes both approaches. Transformations apply directly to and directly between shapes.

Most kernels support connectivity. Some, like INVENTOR, accept it as a default method, while others, like RDI and GEO++, recognize it as an alternative method. Of the two methods, only the latter applies connectivity to transformations. The former connects attributes as well as transformation. Although both methods are useful, the semantics of connecting attributes is not clear. While connected attributes save time and space, they obscure meaning and violate principles of locality.

Only kernels with lightweight primitives ungroup hierarchies. To ungroup properly, primitives must be able to release all of their attributes. Of all kernels that support lightweights primitives, only TBAG is unable to ungroup. TBAG's functional approach hides the internal representation of immutable objects. Users never see hierarchies; therefore, they can never selectively ungroup hierarchies.

Except for RDI, many of the kernels support constraints, interaction, and animation. Constraints establish bi-directional relationships, interaction introduces external influences, and animation produces time-varying behaviors. Future versions of RDI hope to incorporate constraints and interaction, but not animation. Animation is simpler to describe if the tools for animation and modeling are distinct. Unlike some kernels and animation systems[16, 25] that combine the two processes, RDI manages only one. Temporal elements, such as those found in RASP, interact with RDI to produce dynamic systems.

7.2 Rendering

Of the six kernels that support a loose coupling between modeling elements and graphics subsystems, all but TBAG employs a variation of the same technique. Most kernels introduce communication protocols to optimize the exchange of information. GROOP enforces a strict protocol while RDI and VRS enforce a flexible protocol - elements are free to support a wider variety of functions and structures. In place of a protocol, TBAG employs multiple dispatching[19]. Run-time interactions between types evoke specialized operations. Although this plan promotes greater extensibility, it operates slower and undermines the encapsulation of information.

Six of the nine kernels support global attributes. However, only RDI links global attributes with rendering elements. Most kernels, like INVENTOR, position global attributes at the apex of modeling hierarchies. Global attributes describe the state of models, not the default behavior of renderers.

Of all the kernels, only RDI readily supports parallel rendering. Renderers may concurrently interact with all geometries. Unlike other proposed techniques, such as those by GEO++, parallel rendering is not dependent upon the branching factor of hierarchical arrangements. Concurrency occurs immediately, not when parsing of branches occurs. Beside being slower, parallel branching techniques address only nonconnective arrangements. Branches of connective arrangements must be parsed sequentially.

8 Conclusion

This paper has presented RDI, an extensible graphics kernel that applies object-oriented principles to display-list modeling. Developers relate primitives directly while graphics subsystems interpret primitives sequentially. Developers and graphics subsystems interface with primitives in the manner that suits them the best. An underlying process organizes the primitives in an arrangement that caters to both interfaces.

RDI's modeling primitives permit developers to express naturally their mental model of application development. Disregarding order, developers query and assign attributes directly to geometries. Once assigned, attributes bind instanteously. Geometries recognize their inherited and non-inherited relationships. To promote multiple interpretations and support connective arrangements, developers associate, not link transformations to geometries. Like attributes, transformations bind early too. They premultiply and concatenate instantly. The modeling process need not wait for the rendering process to create bindings. To reduce growth and simplify syntax, modeling primitives employ letter classes, copy methods, and editing operations.

RDI's rendering primitives permit graphics subsystems to efficiently manage and visualize information. Views control the viewing of scenes while renderers control the interpretation of scenes. Graphics subsystems employ renderers to interface with modeling primitives and present their preferences. During the rendering process, modeling primitives cross-reference their preferences with those of renderers to optimize the exchange of information. Modeling primitives carefully sequence their interactions (with renderers) to transmit early binding information as late-binding data. Renderers accept the data and derive relations that are identical to those of the original model. Unlike traditional kernels, renderers simply accept information. Views and modeling primitives manage the traversal of hierarchical structures and the transfer of information. This permits the rendering process to support parallel algorithms and specialized primitives.

8.1 Future Directions

In the near future, RDI will be extended to incoporate *coordinate-free geometry*, *filters*, and *constraints*. As noted by [5], coordinate-free geometry clarifies the role of transformations. RDI's current design encourages multiple interpretations, but advances no means to specify a particular interpretation. Coordinate-free geometry provides such structures. It removes ambiguities and improve understanding. Filters, as found in PHIGS[20], change the visibility or detectability of related sets of primitives. They provide the means to rapidly disable or highlight primitives according to name, class, state, or association. RDI's filters will apply locally and globally. Local filters apply to specific models while global filter apply to all the models within a scene. Constraints, like those in TBAG and [22], create dynamic relationships between primitives. They provide structures to link primitives without explicitly applying them to each other. Constraints are necessary to form semantic relations and logical associations.

Acknowledgments

The author would like to thank Dave Forsey and Dinesh Pai for contributions to the ideas and implementation of RDI. Tien Truong and Alain Fournier for valuable comments on early drafts, and Katherine Witowich for critical comments on grammar.

References

[1] AMANATIDES, J., BUCHANAN, J., POULIN, P., AND WOO, A. Optik Users' Manual — Version 2.6. Technical Report Imager 1992–1, University of British Columbia, August 1992.

[2] APPLE COMPUTER INC. *3D Graphics Programming With QuickDraw 3D.* Addison-Wesley, Reading, MA, 1995.

[3] BEIER, E. *Objektorientierte 3D-Grafik.* International Thomson Publishing, September 1994.

[4] COPLIEN, J. O. *Advanced C++: Programming Styles and Idioms.* Addison-Wesley, Reading, MA, 1992.

[5] DEROSE, T. D. Coordinate-Free Geometric Programming. In *Siggraph 'XX Course Notes*

(XX). Association for Computing Machinery, 1991, pp. 0–73.

[6] Doellner, J., and Hinrichs, K. The Virtual Rendering System - A Toolkit for Object-Oriented 3D Graphics. Tech. Rep. Tech Report 19/95, University of Münster, 1995.

[7] Egbert, P. K. An Object-Oriented Approach to Graphical Application Support. Tech. Rep. UIUCDCS-R-92-1755, University of Illinois at Urbana-Champaign, Urbana, IL, 1992.

[8] Elliott, C., Schechter, G., Yeung, R., and Abi-Ezzi, S. TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications. In *Computer Graphics* (Orlando, FL, August 1994), SIGGRAPH, Association of Computing Machinery, Inc., pp. 421–434.

[9] Huynh, D. L., Jensen, M., Larsen, R., Southard, J., Wang, Y.-F., Wang, Y., and Mangaser, A. PIX: An Object-Oriented Network Graphics Environment. In *Visual Computing: Integrating Computer Graphics with Computer Vision*, T. Kunii, Ed. Springer Verlag, 1992, pp. 917–936.

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

[11] Kliewer, B. HOOPS: Powerful Portable 3D Graphics. *BYTE 14*, 7 (1989).

[12] Koved, L., and Wooten, W. L. GROOP: An Object-Oriented Toolkit for Animated 3D Graphics. In *OOPSLA '93* (1993), ACM, pp. 309–325.

[13] Kubota Pacific Computer, Inc. *Dore Programmer's Guide*, 5 ed., Sept 1993. Dore Graphics Library.

[14] Lee, G. S. RASP: Robotics and Animation Simulation Platform. Master's thesis, University of British Columbia, Vancouver, British Columbia, January 1994.

[15] McLendon, P. *Graphics Library Programming Guide*. Silicon Graphics, Inc., Mountain View, CA, 1991.

[16] Najork, M. A., and Brown, M. H. Obliq-3D: A High-Level, Fast-Turnaround 3D Animation System. *IEEE Transactions on Visualization and Computer Graphics 1*, 2 (June 1995), 175–193.

[17] Neiger, J., Davis, T., and Woo, M. *OpenGL Programming Guide*. Addison-Wesley, Reading, MA, 1993.

[18] RogueWave. *Roguewave*. RogueWave Associates, Inc., XXX, 1900.

[19] Schechter, G., Elliott, C., Yeung, R., and Abi-Ezzi, S. Functional 3D Graphics in C++ - with an Object-Oriented, Multiple Dispatching Implementation. In *1994 Eurographics Object-Oriented Graphics Workshop* (1994), Eurographics.

[20] Shuey, D., Bailey, D., and Morrissey, T. PHIGS: A Standardm Dynamic, Interactive Graphics Interface. *IEE Computer Graphics and Applications 0*, 0 (August 1986), 50–86.

[21] Strauss, P. S., and Carey, R. An Object-Oriented 3D Graphics Toolkit. In *Computer Graphics* (Chicago, IL, July 1992), SIGGRAPH, Association of Computing Machinery, Inc., pp. 341–349. SGI Inventor Toolkit.

[22] Teixeira, J. C., and Sakas, V. *Towards an Object-Oriented Kernel for Geometric Modeling*. Springer-Verlag, 1993, pp. 111–127.

[23] van Dam, A. PHIGS+ Functional Description, Revision 3.0. *Computer Graphics 22*, 3 (July 1988), 125–218.

[24] Wisskirchen, P. *Object-oriented graphics : from GKS and PHIGS to object-oriented systems*. Springer-Verlag, Berlin, 1990.

[25] Zeleznik, R. C., Conner, D. B., Wloka, M. M., Aliaga, D. G., Huang, N. T., Hubbard, P. M., Knep, B., Kaufman, H., Hughes, J. F., and van Dam, A. An Object-Oriented Framework for the Integration of Interactive Animation Techniques. In *Computer Graphics* (Las Vegas, Nevada, July 28 - August 2 1991), SIGGRAPH, Association of Computing Machinery, Inc., pp. 105–111.

A Inventor Comparison

To illustrate the difference between Inventor and RDI, this appendix describes the same scene in section 6 with lines of Inventor. Programming notes in both specifications are consistent to accentuate the correspondence between blocks of code that perform analogous functions.

The first part of phase two, shown in Figure 13, creates shapes, attributes, and transformations. Primitive states are set via operations that access public data members. Although this approach differs from that of RDI, it does not augment the quality or quantity of the primitive set. The purpose and design of most primitives in both toolkits are the same. Although they function similarly, Inventor primitives do not interact similarly. Unlike RDI, Inventor primitives do not link directly. As seen in Figure 15, they must explicitly group to work together. Although simple, the act of grouping primitives violates the principle of locality and burdens developers to search for links between primitives.

The latter half of phase two, shown in Figure 14, creates groups to link attributes and associate transformations (lines 26-41). The dual purpose of groups restricts the use of direct operators, such as shallow copying, and creates deep hierarchies. To create a shallow copy of a geometric primitive without attributes, the attributes must be explicitly removed[4] from the cloned group (lines 42-48), or a reference to the geometric primitive must be placed into another group. Deep hierarchies create complex structures that are troublesome to parse. To relate attributes to a deeply embedded primitive, a recursive task must search for the primitive's location within a hierarchy (lines 49-52). If the attributes are misplaced in the hierarchy, unwanted results may occur.

Phase three, shown in Figure 15, applies a global attribute (line 67) and renders the scene. Unlike RDI, the global attribute relates to the modeling primitives, not the rendering primitives. Inventor lacks rendering primitives which act as image synthesizers. Inventor's reliance on GL (and OpenGL) hampers its ability to create distinct rendering primitives and to provide multiple renderers.

[4]The user-defined function `removeChild` deletes specific types of primitives from groups.

create shapes

```
SoTorus *torus = new SoTorus;
torus→width = 50;
torus→height = 50;
SoSphere *sph1 = new SoSphere;
sph1→radius = 25;
SoCylinder *cyl = new SoCylinder;
cyl→radius = 10;
cyl→height = 20;
```

create attributes

```
SoMaterial *sea_grn = new SoMaterial;
sea_grn→diffuseColor.setValue( .18, .55, .34);
SoDrawStyle *lines = new SoDrawStyle;
lines→style.setValue( SoDrawStyle::LINES );
SoMaterial *eng_red = new SoMaterial;
eng_red→diffuseColor.setValue( .83, .24, .1 );
SoTexture2 *checkTex = new SoTexture2;
checkTex→filename.setValue( ”checkboard” );
checkTex→model.setValue( ”SoTexture2::MODULATE” );
SoMaterial *yellow = new SoMaterial;
yellow→diffuseColor.setValue( 1, 1, .87 );
```

create transformations

```
SoTranslation *mtxL = new SoTranslation;
mtxL→translation.setValue( 0,0,-25 );
SoTranslation *mtxR = new SoTranslation;
mtxR→translation.setValue( 0,0, 25 );
SoTranslation *mtxU = new SoTranslation;
mtxU→translation.setValue( 0,35, 0 );
```

Figure 13: Phase 2, Part I: Adding Models to the World

```
          create mixed shape - vertical grouping
SoSeparator *grp1 = new SoSeparator;
grp1→addChild( sea_grn );
grp1→addChild( lines );
grp1→addChild( mtxL );
grp1→addChild( sph1 );
SoSeparator *grp2 = new SoSeparator;
grp2→addChild( checkTex );
grp2→addChild( eng_red );
grp2→addChild( mtxR );
grp2→addChild( sph1 );
SoSeparator *grp3 = new SoSeparator;
grp3→addChild( torus );
SoSeparator *comp = new SoSeparator;
comp→addChild( grp1 );
comp→addChild( grp2 );
comp→addChild( grp3 );
          copy comp & assign material to its torus
SoSeparator *comp2 = (SoSeparator*) comp→copy();
SoSeparator *grp1_c = (SoSeparator*) comp2→getChild( 0 );
removeChild( grp1_c, sea_grn→getTypeId() );
removeChild( grp1_c, lines→getTypeId() );
SoSeparator *grp2_c = (SoSeparator*) comp2→getChild( 1 );
removeChild( grp2_c, eng_red→getTypeId() );
removeChild( grp2_c, checkTex→getTypeId() );
SoSeparator *grp3_c = (SoSeparator*) comp2→getChild( 2 );
for(int i=0; i¡grp3_c→getNumChildren(); i++)
if (grp3_c→getChild(i)→isOfType( SoCube::getClassTypeId() ))
    grp3_c→insertChild(yellow,i);
          create mixed shape - horizontal grouping
SoSeparator *compH = new SoSeparator;
compH→addChild( mtxL );
compH→addChild( comp );
compH→addChild( mtxR );
compH→addChild( mtxR );
compH→addChild( mtxR );
compH→addChild( comp2 );
compH→addChild( mtxU );
compH→addChild( cyl );
```

Figure 14: Phase 2, Part II: Adding Models to the World

assign global attribute

```
SoMaterial *blue = new SoMaterial;
blue→diffuseColor.setValue( 0, 0, 1.0 );
SoSeparator *root = new SoSeparator;
root→addChild( new SoPerspectiveCamera );
root→addChild( new SoDirectionalLight );
root→addChild( blue );
root→addChild( compH );
SoXtRenderArea *myRenderArea = new SoXtRenderArea(¡window-ptr¿);
myRenderArea→setSceneGraph( root );
```

Figure 15: Phase 3: Rendering the World