GROOP: An object-oriented toolkit for animated 3D graphics

Larry Koved

IBM Research
T. J. Watson Research Center
Yorktown Heights, NY 10598

Wayne L. Wooten

Graphics Visualization and Usability Center
Georgia Institute of Technology
Atlanta, Georgia 30318

ABSTRACT

GROOP is an object-oriented toolkit for creating 3D computer graphics applications. It is designed for application developers who are not familiar with computer graphics, but are familiar with object-oriented programming. While application programmers are able to quickly create animated 3D graphical objects, the toolkit is also sophisticated enough for experienced programmers.

In addition to creating stationary 3D objects, the toolkit is used to construct animated objects. Sophisticated reusable articulated objects have been created for use in a variety of applications, similar to static 2D and 3D clip art available today.

1. *Introduction*

As 3-D computer generated graphics becomes more affordable, there is an increasing need to provide tools for application programmers that are easier to learn and use, yet retain flexibility for a wide range of modelling, animation, visualization and Virtual Reality applications. The challenge is to define the programming interface so that it is intuitive for programmers who have a rudimentary understanding of 3-D geometry, yet have no prior knowledge of computer generated graphics.

GROOP, GRaphics using Object-Oriented Programming, is a toolkit for animated 3D computer generated graphics. The basic metaphor is derived from theater or motion pictures -- a scene -- composed of a stage (display), actors (3D objects), lights and a camera. For the novice, simple animated models can be quickly constructed by creating objects, adding them to a scene, and applying transformations (scale, rotate and translate) to generate animation. To simplify learning and provide consistency, an object-oriented language with inheritance, operator overloading and polymorphism is employed, yielding code reuse and permitting the construction of reusable 3D objects with behavior.

GROOP is designed to be portable across graphics systems and operating environments. The system is divided into two major components - scene construction/animation, and rendering (displays). Scenes contain 3D objects, lights and a camera. Scenes are renderer independent. Therefore different renderers can be used to display the same scene description. The current implementation uses the Graphics Library (GL) for rendering [2]. However, a new Display object using a different graphics package, having the same functional interface as the GLwindow renderer object, will be able to display the same set of scenes. This separation of the scene from displays makes it possible to quickly port applications from one system to another, even when the rendering software is different.

The separation of scenes from displays and the simplified object-oriented design of GROOP all fit into the theme of making it easier for application programmers to write portable 3D graphics applications. Many of the aspects of GROOP that make it easier to learn and use by novices also make programming more productive for experienced graphics programmers. Libraries of 3D objects with behavior can be quickly constructed and included in a variety of applications and problem domains.

2. *Background*

A number of graphics systems support application programming interfaces (APIs) making it easier to develop 3D graphics applications portable to a number of different graphics hardware platforms and operating environments (c.f. [2, 6, 8, 13, 18, 22]). These APIs present a number of challenges to an application developer. To develop some seemingly straightforward programs, most of these systems require the application developer to cultivate a thorough understanding of a complex programming model that includes matrix operations, 3D geometry, and lighting models. These graphics systems can contain hundreds of functions, many of which generate system state changes that affect subsequent graphics function calls. Thus, creating interesting graphics applications requires significant investments of time and effort to become proficient at creating even relatively simple applications.

Some graphics tools have taken on a more object-oriented design [1, 5, 12]. Others employ object-oriented technology in the programming model (e.g. C++) [3, 21]. However, they retain significant aspects of their graphics programming heritage. The interface explicitly requires that a display list, or scene graph, be constructed with separate components for geometry, material properties (surface lighting) and geometric transformations. It is the responsibility of the programmer to build and traverse the display lists in the correct order so they can be fed into the graphics pipeline for rendering and other operations.

Researchers at Brown University use *delegation* rather than inheritance in their object-oriented system [25]. The approach allows instancing of objects and class methods. Objects communicate with each other through message objects. Interpretation of the messages is left up to the receiving objects, where the interpretation of the message may change over time. The work is oriented towards animation tools. However, delegation is not a widely used technique, and object-oriented programmers are currently more familiar with inheritance-based object-oriented languages.

Grams, recently reported work from the University of Illinois, employs an approach similar to ours [9, 10]. Grams uses an object-oriented language, C++, and also separates the specification of geometry from that of rendering, permitting an application to select from a number of different renderers. The primary differences between Grams and GROOP are in the class hierarchy and the approach we have taken to extensibility, 3D object construction, optimization and defining of objects that have behavior.

3. *GROOP*

GROOP is designed to be used by application programmers familiar with object-oriented programming and design. As shown in Figure 1, the system is layered. The top-most layer is the application that takes advantage of a number of graphics-oriented objects in the middle layer. Below these is the software interface to the 3D graphics rendering.

The top layer contains application objects. They are designed and implemented by the application

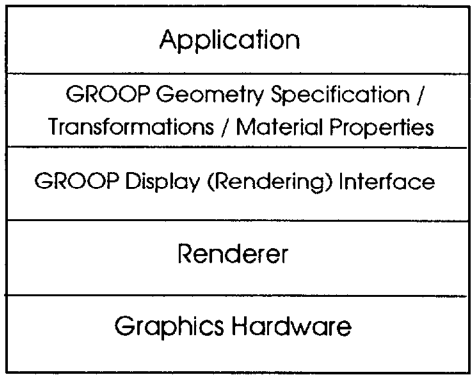
Application
GROOP Geometry Specification / Transformations / Material Properties
GROOP Display (Rendering) Interface
Renderer
Graphics Hardware

Figure 1. Graphics layers. At the top is the application. This includes, but is not limited to, animation systems, 3D modellers, visualization and Virtual Reality. This interfaces with GROOP by generating object descriptions (or reading them from a file) and defining object surface material properties such as color and texture. The next layer is the GROOP interface to the rendering software package. It performs data management, such as optimization and caching of data values from the layer above, and appropriately sequences all of the calls to the rendering software. The renderer and graphics hardware are provided by systems manufacturers and/or third party vendors (e.g. [1, 2, 6, 8, 18, 22]).

programmer. These include simulations, 3D object modellers, animation and data visualizers, among others. The application defines the geometry of and relationships between the objects to be displayed, along with the material properties. These properties include the color, reflection, and surface textures. The application determines the visual behavior (animation) of the 3D objects to be displayed through the use of *transformations*: *scaling* of the object size, *rotations* (about the origin) and *translations* (movement away from the origin). The scene description can be computed by the application, read from files, or imported from other external sources.

An example application is the spinning top simulator (Figure 2). In this program, a top is created by constructing an inverted cone and combining it with a cylinder to create the top. The geometry of the cone and cylinder are computed when they are created. Each time through the simulation, the top is tilted and rotated, giving it the appearance of wobbling[1] .

The middle layer is a set of classes used to describe a scene (see Figure 3). The underlying concept is that all elements of a scene are GeometricObjects, which can be transformed and given material properties.

3.1 GEOMETRICOBJECTS

All GeometricObjects contain three basic elements: 3D geometry, material (surface or lighting) properties, and lists of geometric transformations.

A number of classes provide flexibility in defining the geometry of 3D objects. Surfaces are often defined by polygons and strips of simple triangles. These objects along with polylines (multi-segment lines) are defined by a list of verticies, points represented in Cartesian coordinates by *(X, Y, Z)*. They inherit from the VertexList class,

[1] If the spinning top application were to be written as a GL program, the following 26 functions would be used: winopen(), prefposition(), winconstraints(), viewport() lookat(), RGBmode(), doublebuffer(), gconfig(), zbuffer(), czclear(), swapbuffers(), mode(), perspective(), loadmatrix(), popmatrix(), multmatrix(), rot(), scale(), lmdef(), lmbind(), bgnpolygon(), endpolygon(), bgntmesh(), endtmesh(), n3f(), v3f(). Many of these have complex options and change the system state. Programmers would have to understand how the state changes affect subsequent graphics system function calls.

```
main(void)
{
   GLwindow          mywindow("Spinning Top");        // create GL window
                                                      // with a title bar
   CappedCone*       conePtr     = new CappedCone;
   CappedCylinder*   cylinderPtr = new CappedCylinder;
   Composite*        topPtr      = new Composite;

   topPtr->Add(conePtr);       // add cone and cylinder to the top object
   topPtr->Add(cylinderPtr);
   mywindow.Add(topPtr);      // put the top in the window

   conePtr->material.Diffuse(rgb_RED);      // set object colors
   cylinderPtr->material.Diffuse(rgb_YELLOW);

   cylinderPtr->SetScale( 0.1, 1.0, 0.1 ); // make it tall & skinny
   conePtr->SetRotate( 0.0, 0.0, 180.0 );  // default cone is base down.
                                           // flip it over.

      // simple simulation -- the top wobbles as it spins
   for (float index = 0.0; index < 2000.0; index += 1.0) {

      topPtr->RotateZ( 45.0 * index / 2000.0 );    // make it wobble
      topPtr->RotateY( index );                    // spin the top

      mywindow.Display();                          // display the top
   }
}
```

Figure 2. Spinning Top. A simple animated object program. A Display object is created (mywindow). A number of GeometricObjects are created: two Composite objects are put into another Composite (topPtr) and given colors. Since their default shape and orientation need to be changed to create the desired model (a top), they are transformed. The "simulator" is a simple animation loop where the top is given tilt and rotated to give the appearance of wobbling. See color plate 2 located near the end of the conference proceedings.

which maintains a list of verticies and normals[2] that define the object. For example, a cube can be represented as six polygons, one for each of the faces. To create a face, a polygon object is created and the Add() member is called, inherited from VertexList, with the vertex and normal for each corner. Other primitive 3D objects include points, curved surfaces such as nurbs[3], and text.

Interesting objects are usually composed of many smaller parts. Composite objects are containers, either of heterogeneous or homogeneous objects. They are used to aggregate a number of GeometricObjects that can be manipulated as a group. A transformation applied to a Composite is applied to each of the sub-objects. In addition, default material properties can be defined that apply to the sub-objects, unless the sub-objects define their own material properties.

Heterogeneous Composite objects are collections of objects where the geometric descriptions are of different types. A candlestick may be composed of a vertical component defined by polygons and the base consisting of a nurbs surface. The components of the Composite object are individually created and added to the Composite. In addition, the individual GeometricObjects may each have surface material properties and geometric transformations.

Composites can contain Composites. This leads to the construction of hierarchically composed complex objects. GROOP automatically handles the resulting nesting of the transformation hierar-

2 Unit vectors perpendicular to a surface at a point.

3 non-uniform rational B-splines

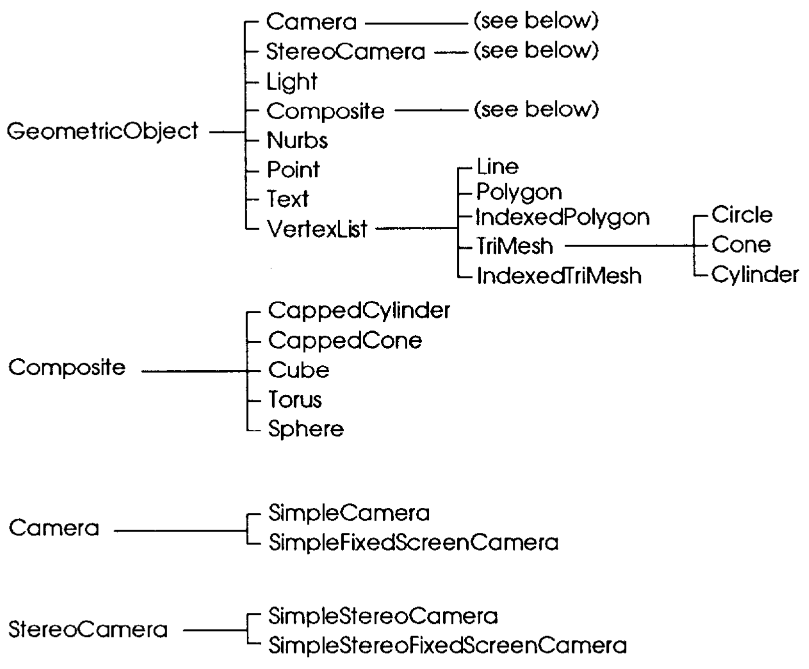

Figure 3. Geometric objects class hierarchy. Part of the class hierarchy that deals with GeometricObjects. Other primitive classes in GROOP include MaterialProperties, GeoTransforms, Vertex, Matrix, rgb, Scene, Display, GLwindow, and a number classes used for reading files of various formats. Domain and application specific classes are added to extend the system. (Currently IndexedPolygon, IndexedTriMesh and Nurbs are not implemented, but can easily be added.)

chies. This ability to nest Composites within Composites enables the construction of complex articulated objects, objects with behavior. A simple example is in the Spinning Top, where the cone and cylinder objects (CappedCone and CappedCylinder) are subclasses of Composite. These Composite objects are nested in another Composite object, topPtr, where they are manipulated as a group.

Special cases are automatically optimized. Homogeneous Composites are constructed from two or more objects of the same type. A cube is such an object, consisting of six Polygons, where each of the six faces is created and added to a Composite object. Often the individual sub-parts of the Composite object do not need to be independently transformed. Similarly, each of the subparts do not need to be given different material properties. GROOP transparently optimizes these different special cases.

3.2 LIGHTS

Scenes can contain multiple lights. Lights have properties in addition to those of GeometricObjects. By default, lights are omnidirectional. However, they can be redefined as spotlights by setting the spotlight direction, angle of focus (the cone of light spread) and light attenuation values. Generation of shadows is not provided by GROOP, but is left to the renderer being used. For example, a Gouraud or flat shading renderer may not necessarily provide automatic generation of shadows. Other renderers, such as RenderMan [22], will generate shadows.

3.3 CAMERAS

A number of full-featured and simplified camera classes are provided. The base camera class allows setting of the camera location and a number of screen parameters.[4] The settings include the width, height, a normal vector, near and far clipping planes, among other parameters. Understanding and setting these parameters can be a daunting task for the average programmer. A number of simplified camera models have been created that employ the basic camera.

Class SimpleCamera behaves like a video camera. Its location, a direction it is facing and a rotation can be set. The class member functions compute the new values for the underlying Camera object. This is an intuitive camera model usable in a wide variety of applications.

SimpleFixedScreenCamera creates the illusion that the observer is looking into a box in which the 3D objects reside. When the camera's location is coupled to the observer's location (e.g., using a 3D position sensor [4, 17, 20]), the objects appear to remain stationary as the observer moves about -- up, down, left, right, towards and away from -- the computer monitor or display unit.

Stereoscopic display units enhance the illusion that the computer generated objects are three dimensional [11, 14-16]. The most commonly used technique is for the graphics system to draw the scene twice, once for each eye. The stereoscopic display units make sure each eye sees only one of the images. A StereoCamera object, containing two Camera objects, can be inserted into a scene. To simplify the use of stereoscopic cameras, SimpleStereoCamera and SimpleStereoFixedScreenCamera classes were created. Similar to the monoscopic versions previously described, these two simplified StereoCamera subclasses also take eye separation distance and head orientation into consideration.

To reduce learning time and programming effort, all of the simple cameras (monoscopic and stereoscopic) have a common functional (polymorphic) interface. This allows rapid substitution of one camera type for another.

3.4 MATERIALPROPERTIES

All GeometricObjects contain an object, *material*, that is of type MaterialProperties. The base set of properties are diffuse, ambient, emissive and specular colors, and a shininess parameter. Other renderer specific attributes can be attached to objects, such as texture maps. The base and other attributes are used primarily for 3D surfaces and objects. For lights, a subset of these attributes are employed, and camera objects ignore them. A number of material values are predefined to simplify the process of selecting suitable values.

3.5 TRANSFORMATIONS - BASIC MODELLING AND ANIMATION

For a programmer not familiar with 3D graphics concepts, learning to apply transformations (scale, rotate, translate) can be quite confusing. Graphics packages typically require the user to understand matrix operations and figure out the order in which the transformations should be applied. In GL, for a given set of transformations to an object, the transformations need to be fed into the graphics pipeline in the reverse order (bottom-up). When hierarchically composing objects (nested Composite objects), the transforms for the Composites are fed into the pipeline in a top-down fashion. This is certainly not intuitive.

Even for simple scaling, rotating and translating, transformation usage is not always intuitive. Transformations are *order dependent.* For

[4] Informally, a screen is a rectangular 2D area onto which images of the 3D objects are projected.

example, this means that performing a rotate before a translate will most likely produce a result different then if the order were reversed. Also, the means for applying transformations to lights and cameras may be different than for other GeometricObjects.

GROOP hides the details of feeding the graphics pipeline with the list of transformations and their composition. The programmer adds the transformations in the sequence that is intuitive - in the forward (top-down) direction, and through hierarchical nesting of GeometricObjects (the Composite class).

Many times primitive objects or composite objects are not oriented, located or scaled appropriately for a given scene. Transformations need to be applied to these objects at the time they are displayed to create the necessary model. In addition, these models need to be animated. GROOP addresses the modelling and animation issues in two ways (see the Spinning Top example in Figure 2).

The first is to provide a simple interface that performs transformations -- SetScale(), SetRotate() and SetTranslate() -- in a fixed order every time an object is displayed. Once defined for an object, these transformations remain with the object unless subsequently changed by the application.

The second method permits transformations on GeometricObjects to be specified in any order when they are displayed. The transformation functions are Scale(), RotateX(), RotateY(), RotateZ(), Translate(), GenericTrans(), where the latter function accepts any 4x4 transformation matrix. These functions generate a list of operations (GeoTransform objects) to be performed on the next display update (rendering). These transforms are frequently used for animating objects. Each time through the animation loop, the appropriate transformations are specified to define the new scale, location and orientation of the objects. By default, the display object's Display() function (which performs the rendering) will discard the list of transformations every time the object is displayed. This simplifies the animation loop by not having to add extra code just to reset the objects' transformations.

To correctly model some objects, the simple Set*() functions described above may be insufficient, so the general transformation functions need to be used. To prevent Display() from discarding the transformations, a SaveTransforms() is added to the object's list of transformations. This causes the list of transformations prior to the SaveTransforms() to be computed and saved with the object, and used by subsequent Display() calls. Also, transformations can still be added after the SaveTransforms() call to perform object animation, as previously described.[5]

3.6 COPYING INSTANCES

Once an object has been modelled, it is sometimes desirable to make copies. It is intuitive to use the assignment operator to make that copy. GROOP supports this by providing an assignment operator for GeometricObject classes. In Figure 4, the code fragment shows how four wheels for a wagon are created from the original wheel. Each of these wheels can be independently transformed into their desired location to be part of the wagon.

GROOP can handle the assignment operation in two ways. The simplest is copy-by-value (deep copy). However, for large complex objects, the memory allocation overhead may be severe. In this case, copy-by-reference is possible (shallow copy). (A compromise between these extremes would be to have copy-on-change.)

[5] A different approach would be to provide a set of modelling functions to transform the original data points rather than transforming the data every time through the animation loop.

```
 ...

Composite     Wheel_1, Wheel_2, Wheel_3, Wheel_4;

 ...                   // code to construct the Wheel_1 object

Wheel_2 = Wheel_1;               // copy the wheel
Wheel_3 = Wheel_1;
Wheel_4 = Wheel_1;

Wheel_1->Translate( 3, 0, -1);    // move wheels to new locations
Wheel_2->Translate( 3, 0,  1);
Wheel_3->Translate(-3, 0, -1);
Wheel_4->Translate(-3, 0,  1);

 ...                   // code to construct the rest of the wagon
```

Figure 4. Wheel objects. Multiple wheels are created from a single wheel by using the assignment operator. They are then transformed into their final location for use in a wagon. See color plate 2 located near the end of the conference proceedings.

3.7 MEMORY MANAGEMENT

GROOP provides automatic memory management for GeometricObjects. That is, when a Scene or a Composite object is deleted, then all of the objects contained in these objects and their attributes are also deleted. This is important especially for programmers who want to focus on constructing models and animating them. They do not want to worry about keeping track of hundreds, thousands or millions of objects. These objects may be created by modeling applications and imported into GROOP through one of the file reader classes. Keeping track of the thousands of objects would be impractical if not handled by GROOP. However, there are circumstances where automatic memory management may be undesirable. So, functions are provided to turn off this feature.

4. *Scenes and Displays*

Scenes are a collection of GeometricObjects: 3D objects, lights and a camera.[6] Scene's Add() and Delete() functions are used to insert and remove GeometricObjects. The only constraint is that a scene can have just one Camera or StereoCamera at a time.

Displays interpret scene descriptions. The Display object sends a message to each of the GeometricObjects in the scene to obtain the GeometricObjects' geometry, set of transformations, MaterialProperties and other attributes. Display classes interfacing to renderers, such as GLwindow, use the information to make the appropriate graphics system calls. While GLwindow uses the GL graphics library for rendering, a Display does not have to be a renderer. For example, different Display subclasses can be defined to write scene descriptions to files in any desired format.

To achieve renderer independence, Display is defined as a C++ virtual base class. The class defines a set of member functions that must be implemented by all subclasses. Most of these functions are the routines that will interpret the objects' geometry data (e.g., triangles, polygons, nurbs, etc.) and other object attributes. By implementing the minimal set of functions defined in Display, any scene description can be rendered or written to a file.

As a convenience, Display is a subclass of Scene. This alleviates the extra step of creating a

[6] Displays contain a default light and camera when none are otherwise specified.

```
enum LeftRight { LeftHand, RightHand };

class Hand : public Composite {
private:
    ...       // private declarations
public:
    Hand(LeftRight handedness);    // create the hand
    BendFingers(
                float, float,         // thumb joints
                float, float, float, // index joints
                float, float, float, // middle joints
                float, float, float, // ring joints
                float, float, float  // pinky joints
               );
};
```

Figure 5. Hand object. A simple interface to an object defining a 3D hand with behavior. Hand is a subclass of Composite, which is a subclass of GeometricObject. This means that the palm and fingers can be transformed (scaled, rotated, translated) as a single unit rather than having to perform the transformations on the individual components. The details of managing the complex geometric relationships are hidden within the class.

Scene object and adding it to a Display object. Also, multiple scenes can be added to a Display. Scene objects can be used to contain scene segments that are added and deleted from the display as needed.

5. *Reusable objects with behavior*

One goal of GROOP is to afford the construction or importation of reusable 3D graphical objects that have behavior. Many 3D modelling products are commercially available. However, for animation tasks, visualization and Virtual Reality applications, among others, there is a need to give these objects programmable behavior. For example, it is straightforward to build a graphical model of a hand. However, it is cumbersome to take the geometry from a modelling program and animate it. More desirable would be the ability to define the palm and fingers as a Composite object with behavior. This object would have an interface for creating instances as either left or right hands, and bending the fingers (e.g. Figure 5). The Bend() member function takes care of the details of how the finger bending is modelled. The application developer can now focus on where the hand should be rotated and translated, and the hand gesture (bending of the fingers). Another example is the Lathe (see Figure 6). An object is inserted into the lathe for turning. Member functions define the lathe's rotation, plus the location and orientation of the cutting tool.

By defining 3D objects with behavior, it becomes much easier for application developers to write applications containing 3D graphics. Much of the laborious work entailed in building the models, specifying the geometric relationships and their dynamics (motions relative to each other) can be hidden inside the classes.

GROOP is effective in constructing objects with behavior. They are usually defined as a Composite so they can contain any number of other GeometricObjects. Since it is also a GeometricObject, it can be transformed to create the desired model for the scene in which it is to be used. The constructor can create the necessary GeometricObjects and add them to itself, defining the geometric relationships needed between the sub-objects. For example, the Hand object contains a number of Finger objects. The finger is modelled in a separate class that defines the geometry of the parts of the fingers and defines a member function for finger bending. The Hand

```
class Lathe : public Composite {
private:
    ...        // private declarations
public:
    Lathe(GeometricObject object);  // object to be turned in the lathe
    Turn(float angle);              // angle to rotate object
    ToolOrientation(float angle1, float angle2); // cutting tool angle
    ToolLocation(float x, float y, float z);     // location of tool
};
```

Figure 6. Lathe object declaration.

class creates five fingers, transforms (scales and translates), and adds them to itself. When Hand's Bend() function is called, the Finger's Bend() function is called for each finger with the appropriate parameters. All of these low-level details are hidden from the end-user of the Hand class. It is the combination of hierarchical object construction and object-oriented programming that makes reusable objects easy to create.

6. *Using GROOP in applications*

GROOP was designed to be embedded in a variety of applications rather than being a domain specific tool. Hence, it does not have a specific user interfaces, such as a 3D modeller, animation classes, Virtual Reality or scientific visualization functions. Instead, these can be added to suit the application needs. For example, at IBM we have created a simple interactive 3D modeller by using standard X Windows widgets. We routinely use GROOP in our Virtual Reality toolkit.

6.1 ANIMATION

GROOP is a library that is flexible enough to be used in the creation of a generic animation system. Hierarchical composition of objects through the use of the Composite class makes it is easier to create articulated figures such as humans and animals. Code which uses inverse-kinematic techniques can then be used to drive the joints of the articulated figures (c.f. [19, 23]). In addition, actual dynamics simulations can be used to drive the joint angles of the exact same GROOP model, with no modifications to the articulated figure code. Only the simulation changes, the actual GROOP model can be reused.

6.2 VR

GROOP has been used extensively in the construction of Virtual Reality applications. While GROOP can be used in stand-alone applications, it has also been embedded into a distributed runtime environment that supports a variety of input and output devices [7]. A number of reusable objects have been created, including an object similar to the Hand object previously described. Objects modelled in systems such as CATIA and WaveFront have been imported into GROOP applications. File import is accomplished by writing file reader classes, subclassed from the virtual base class FileReader, which are written to read data of different formats.

6.3 SIMULATIONS

Both discrete and continuous systems can be easily accommodated (Figure 7). The two simulation classes (Discrete and Continuous) are the objects to be included in the simulation. The actual simulation objects are subclassed from either the Discrete or Continuous class, and are also a subclass of one of the GeometricObject classes. By using multiple inheritance, the object can be both a simulation object and a 3D graphics object. The Simulate class is a container, holding a list of objects to

```
class Discrete {
private:
    ...        // private declarations
public:
    ...
    virtual void TimeStep(int step) = 0;    // called every time step
    ...
};

class Continuous {
private:
    ...        // private declarations
public:
    ...
    virtual float TimeStep(float time) = 0; // accept current clock
                                            // return next time to be called
    ...
};

class Simulate {
private:
    ...        // private declarations
public:
    ...
    void Add(Discrete*   object);       // add objects to be simulated
    void Add(Continuous* object);

    void Delete(Discrete*   object);    // delete objects from simulation
    void Delete(Continuous* object);
    ...
};
```

Figure 7. Simulations. The two simulation classes (Discrete and Continuous) are virtual base classes, which are subclassed by applications. The subclasses implement the function of the simulation by creating the actual TimeStep() functions. The Simulate class manages the clock used in the simulation.

be simulated. Simulate objects increment the time step or clock and call the TimeStep() member function for each of the simulation objects. The application would create these simulation objects and insert them into a Simulate object, which drives the simulation, and a Display object (e.g., GLwindow) for display at an appropriate time determined by the application.

7. *Comparisons with other object-oriented systems*

7.1 IRIS INVENTOR

One of the main advantages GROOP has over Iris Inventor [3, 21] is that it is not dependent on the GL graphics library [2]. The GROOP system was designed so that drivers for other renderers or 3D display libraries could be written. High performance can be maintained by having the new display class performing optimizations that are specific to the renderer. Applications using one renderer could be used with another renderer by simply changing declaration of the display (renderer) object.

Another main difference is in the class structure. In Inventor the scene database is designed as a tree of nodes which contain objects that are used to draw the scene. The objects include shapes, lights, cameras, transformations, and materials. To build a scene in Inventor one has to remember to insert all of the objects into the tree *in the correct order*. For example, if a material node is inserted after a shape node, then the material will not apply to that shape. Inventor allows *node-kits* to be built that provide materials, transforms, and geometry to exist together in a branch of the scene tree.

```
// This function will be called by the Timer sensor to spin the top.
static void
myCallback( void *dataP, SoSensor * )
{
   static float index = 0;

   // Modify the transformation a little
   SoTransform *transformP = (SoTransform *) dataP;
   SbRotation r1( SbVec3f( 0, 0, 1 ), (5.0*index/2000.0) * M_PI/180.0);
   SbRotation r2( SbVec3f( 0, 1, 0 ), index*M_PI/180.0 );
   transformP->rotation.setValue( r2 * r1 );
   index += 3.0;
}

main( int argc, char **argv )
{
   Widget myWindow = SoXt::init(argv 0");  // Initialize Inventor
   if ( myWindow == NULL ) exit( 1 );      // and X Windows

   SoSeparator *rootP = new SoSeparator;
   rootP->ref();
   SoPerspectiveCamera *cameraP = new SoPerspectiveCamera;
   rootP->addChild( cameraP );
   rootP->addChild( new SoDirectionalLight );

   // Rotate the whole world so it is upright
   SoRotationXYZ *rotateP = new SoRotationXYZ;
   rotateP->angle = -M_PI;
   rotateP->axis = SoRotationXYZ::Z;
   rootP->addChild( rotateP );

   // This transformation is modified to rotate the cone
   SoTransform *transformP = new SoTransform;
   rootP->addChild( transformP );

   // This is transform is used to make the cylinder skinny
   SoScale *scaleP = new SoScale;
   scaleP->scaleFactor.setValue ( .1, 1.0, .1 );
   SoTranslation *translateP = new SoTranslation;
   translateP->translation.setValue ( 0, -.5, 0 );

   // These are the shape nodes with colors
   SoMaterial *materialP = new SoMaterial;
   materialP->diffuseColor.setValue( 1.0, 0.0, 0.0 );   // Red
   rootP->addChild( materialP );
   rootP->addChild( new SoCone );
   SoMaterial *material2P = new SoMaterial;
   material2P->diffuseColor.setValue( 1.0, 1.0, 0.0 );   // Yellow
   rootP->addChild( material2P );
   rootP->addChild( scaleP );
   rootP->addChild( translateP );
   rootP->addChild( new SoCylinder );
   cameraP->viewAll( rootP );

   // A TimerSensor rotates the object
   SoTimerSensor *mySensorP =
      new SoTimerSensor( myCallback, transformP );
   mySensorP->schedule( SbTime( 0, 2 ) );

   SoXtRenderArea *renderAreaP = new SoXtRenderArea;
   (void) renderAreaP->build( myWindow );
   renderAreaP->setSceneGraph( rootP );
   renderAreaP->show();

   SoXt::show( myWindow );
   SoXt::mainLoop();
}
```

Figure 8. Iris Inventor. This SpinningTop program is similar in function to the GROOP version (see Figure 2), yet is much longer and more complex.

However, it is up to the user to build node-kits not provided by Inventor.

The approach taken by Inventor is to build display lists where attributes need to be placed into the list. As noted by [24] in his critique of PHIGS, this style violates the *principle of locality* because visible-object attributes need not be placed near the object a programmer wants to modify. If the programmer wants to change a color of an object, it is necessary to know where in the tree the color attribute and/or object is located. In addition, Wisskirchen also notes that when implemented on a parallel processor, it is harder to parallelize this code. In GROOP the programming model is more intuitive. Materials, transformations and geometric data are all attributes of a GeometricObject, and are maintained as a single unit, thus maintaining the principle of locality.

Since Inventor is based on a tree, in which only certain types of nodes can be placed into the tree, it is difficult to build higher level objects using the inheritance features of the C++ language. GROOP on the other hand easily allows higher-level complex objects to be designed using inheritance from the GeometricObject or Composite classes, as was described in section 5 (*Reusable objects with behavior*). These higher-level objects can have their own member functions while also maintaining all of the capabilities of a GeometricObject.

Inventor is an event driven system and provides a selection of interactors. These allow users to choose objects in a scene and manipulate them interactively. While GROOP does not currently provide this level of interaction, it can easily be added. The fact that GROOP is not event driven makes it easier to write code to generate animation sequences. In Inventor a timer sensor must be used to edit transformations in the scene tree over time in order to generate animations.

In conclusion GROOP provides a simpler, more intuitive, and extensible class structure based on a camera/stage/actor paradigm. The code in Figure 8 is the Inventor code to create a spinning top similar to the one shown in Figure 2. The GROOP code is much simpler and easier to create and understand.

7.2 GRAMS

Of all the object-oriented 3D graphics systems, GRAMS [9, 10] is the most similar to GROOP. Many of the basic design issues, such as separation of geometry and lighting/material specifications from the rendering process. This permits a scene to be rendered by more than one renderer. Many of the differences between GRAMS and GROOP are in the class hierarchy and some design differences (c.f. [10], page 86).

GRAMS specifies the renderer as an argument when passing a geometric (Application) object to the world object. When the world object is told to display a scene, the world object passes the geometric object to the renderer.

GROOP works with a simpler model. The programmer builds a Scene, typically a renderer (e.g., GLwindow is a subclass of Scene). When appropriate, the Scene is told to display itself. GROOP is not concerned with mixing renderers in a single window. In addition, "application" classes are left up to the programmer, whereas GRAMS requires displayable objects to be subclasses of an Application class. Also, GROOP offers a richer set of primitives for specifying surface material properties, and has a richer set of camera objects. GROOP makes extensive use of inheritance and virtual base classes to simplify the design of the system as well as allow for future expansion (e.g., new renderers, file readers, file writers).

7.3 GEO++

Many of the object-oriented concepts described by [24] are employed by GROOP. GEO++ is a Smalltalk-based system that provides an object-oriented interface to a 3D graphics system. In the case of GEO++, the underlying graphics system is PHIGS. Wisskirchen provides an in depth review of GKS and PHIGS and describes their shortcomings with respect to object-oriented approach as embodied by GEO++.

As in GROOP, GEO++ makes extensive use of part hierarchies to construct complex multi-part objects. The construction of part hierarchies in both systems is is somewhat similar. GEO++ makes extensive use of paths and subpaths for the construction and traversal of the parts hierarchy. While an interesting and perhaps useful concept, it can be confusing to some programmers. For that reason, GROOP does not expose the hierarchy paths/subpaths.

"GEO++ supports part hierarchies quite extensively, but lacks in supporting connectivity" ([24], page 210). For example, in the case of the hand object (Figure 5), it is not possible to specify the transformation relationships between each of the parts of the fingers. That is, GEO++ does no support concatenation (or composition) of transformations. GROOP supports the composition of the transformations in order to create articulated and animated objects.

8. *Conclusions*

GROOP set out to achieve a number of goals. It is an object-oriented toolkit for application programmers who are not familiar with 3D graphics and graphics programming concepts, yet are familiar with object-oriented design and programming. It is designed to allow them to quickly build 3D graphics applications without having to learn and understand the intricacies of graphics geometry and pipelines. The toolkit is also for experienced graphics programmers who would like to focus on the high-level application development issues rather than devoting effort to write to the low-level graphics interface. To succeed, the toolkit must be flexible and maintain sufficient performance. It must be possible to read geometric object descriptions generated by other programs such as 3D modellers. One highly desirable benefit is the ability to create reusable 3D objects with behavior. A toolkit should also be able to be part of a larger application, such as a modeller, animation, visualization or Virtual Reality system.

3D Objects with behavior are straightforward to create when using GROOP. Libraries of these objects can be developed and made available to application developers, similar to 2D and 3D clip art that is commercially available today. These libraries will make 3D graphics programming accessible to increasing range of application developers.

GROOP has been successful in its goals of quickly allowing non-graphics programmers to develop 3D graphics applications for Virtual Reality applications. They were able to do so without having to understand low-level graphics primitives. The toolkit has sufficient function that experienced graphics programmers did not have to resort to low-level graphics systems calls. It allowed them to be more productive by focusing their effort on the high-level design tasks rather than coding to the graphics system interface. GROOP performs a variety of optimizations that boost overall graphics system throughput and can reduce memory utilization.

Object-oriented programming simplified a number of important aspects of GROOP's design. The class GeometricObject is the basis of objects in scenes. This permits a number of uniform operations to be performed, such as transformations and assignment of material properties. VertexList allows the creation of objects whose geometry is defined by a series of Cartesian coordinates. GROOP is able to remain independent of renderers and file formats by using a virtual base class, Display, and virtual file reader classes. Objects can be hierarchically nested to create complex objects by using the Composite class, which allows the aggregation of GeometricObjects, including other Composite objects. A variety of simplified camera classes were subclassed from the base camera classes, Camera and StereoCamera.

The regularity of GROOP's design, through the extensive use of polymorphism, operator overloading, and virtual base classes GeometricObject, Camera, Display and FileReader simplifies its implementation. It also eases the learning curve for novices and improves the productivity of experienced graphics programmers. This is true because fewer programming concepts need to be presented in order to use the system. Many of the complex details of graphics programming and optimization techniques are transparently handled inside the classes.

GROOP is extensible in a number of ways. Additional graphics primitives and geometric objects can be built on top of the base provided by the toolkit. In addition, GROOP is not tied to any specific file formats, renderers or application domains. For example, a number of the 3D objects in the Virtual Reality system at IBM were designed by 3D modellers and imported into GROOP. New domain or application specific classes can be created and can inherit directly from GROOP classes, or multiple inheritance can be used to create composite behavior, such as in the case of the simulation classes previously described.

9. *Future work*

The Virtual Reality and animation applications are on-going activities. As such, GROOP will continue to be extended to meet the needs of these projects. Many of the extensions will most likely be add-ons to the base system, such as animation and simulation classes. We also intend to port GROOP to platforms other than the IBM RISC System 6000 and Silicon Graphics Iris™ computers. As personal computers become faster, it becomes desirable to run 3D graphics applications on these low-end systems. GROOP is being ported to PC-DOS-based systems.

One notable omission from GROOP at this time is its lack of interaction classes (e.g., keyboard, mouse, dials and window management). For the applications we have developed to date, the interaction elements were provide by X Window widgets or the Virtual Reality system. While not immediately needed, interaction classes will be added in the future.

10. *Acknowledgements*

We would like to acknowledge the support and contributions of our colleagues in the Virtual Worlds group at the IBM T. J. Watson Research Center. Robert H. Wolfe provided invaluable contributions to the camera models and testing the system by building applications. James S. Lipscomb provided general guidance on 3D graphics issues. Christopher Codella was a wonderful sounding board for C++ programming issues. Reza Jalili wrote sample applications and asked insightful questions. A former member of the group, C. P. Wang helped in the initial stages of learning GL. And, special thanks to the manager of the Virtual Worlds Group, J. Bryan Lewis.

11. *Color figures*

Four figures generated using GROOP are in color plate 2 located near the end of the conference proceedings.

12. *References*

1. -----. *Dore Programmer's Guide, Release 5.0.* Kubota Pacific Computer, Incorporated, Santa Clara, CA, 1991.

2. -----. *Graphics Library Programming Guide.* Silicon Graphics Computer Systems, MountainView, CA, 1991.

3. -----. *Iris Inventor Programming Guide.* Silicon Graphics Computer Systems, MountainView, CA, 1992.

4. Ascension Technology Corporation, The Bird Position and Orientation Measurement System Installation and Operation Guide, POB 527 Burlington, Vermont, 1990.

5. E.H. Blake and P. Wisskirchen. *Advances in Object-Oriented Graphics I.* Springer-Verlag, Berlin, 1991.

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

7. Christopher F. Codella, Reza Jalili, Larry Koved and J. Bryan Lewis. A Toolkit for Developing Multi-User, Distributed Virtual Environments. *Submitted to VRAIS'93*, IEEE, September 1993.

8. Linda Kosko, ed. *PHIGS Reference Manual: 3D Programming in X.* O'Reilly & Associates, Sebastopol, CA, 1992.

9. Parris K. Egbert. *An Object-Oriented Approach To Graphical Application Support*, PhD thesis. University of Illinois at Urbana-Champaign, June 1992.

10. Parris K. Egbert and William J. Kubitz. Application Graphics Modeling Support Through Object Orientation. *Computer*, 88-90, IEEE, October 1992.

11. Phil Johnson. Hardware for stereoscopic computer graphics and imaging. *Information Display*, 5(7 & 8), Society for Information Display, July/August 1989.

12. Michael R. Kaplan. The Design of the Dore Graphics System. In E. H. Blake and P. Wisskirchen, editors, *Advances in Object-Oriented Graphics I*, 177-198, Springer-Verlag, Berlin, 1991.

13. B. D. Kliewer. HOOPS: Powerful Portable 3D Graphics. *Byte*, 14(7):193-194, July 1989.

14. James S. Lipscomb. Experience with stereoscopic display devices and output algorithms. *Three-dimensional Visualization and Display Tech.*, 1083, Society of Photo-Optical Instrumentation Engineers (SPIE), February 1989.

15. Lenny Lipton. Displays Gain Depth. *Computer Graphics World*, PennWell Publishing Company, March 1988.

16. Lenny Lipton. *The Crystal Eyes Handbook.* StereoGraphics Corporation, San Rafael, CA, 1991.

17. Logitech, Inc., 2D / 6D Mouse Technical Reference Manual, Fremont, CA, 1991.

18. International Standards Organization. *International Standard Information Processing Systems - Computer Graphics - Graphics Kernel System for Three Dimensions (GKS - 3D) Functional Description.* American National Standards Institute, New York, 1988.

19. Cary B. Phillips and Norman I. Badler. Interactive Behaviors for Bipedal Articulated Figures. *Computer Graphics (SIGGRAPH'92 Conference Proceedings)*, 25(4):359-362, July 1991.

20. Polhemus Navigation Sciences Division of McDonnell Douglass Electronics Company, 3SPACE® USER'S MANUAL, Colchester, Vermont, 1987.

21. Paul S. Strauss and Rikk Carey. An Object-Oriented 3D Graphics Toolkit. *Computer Graphics (Proceedings of SIGGRAPH'92)*, 341-349, ACM, July 1992.

22. Steve Upstill. *The RenderMan Companion.* Addison-Wesley, Reading, MA, 1990.

23. Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques: Theory and Practice.* Addison-Wesley, Reading, MA, 1992.

24. P. Wisskirchen. *Object-Oriented Graphics: from GKS and PHIGS to Object-Oriented Systems.* Springer-Verlag, Berlin, 1990.

25. Robert C. Zeleznik, D. Brookshire Conner, Matthias M. Wloka, Daniel G. Aliaga, Nathan T. Huang, Philip M. Hubbart, 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'92 Conference Proceedings),* 25(4):105-111, ACM, July 1991.