Lessons from Scene Graphs: Using Scene Graphs to Teach Hierarchical Modeling

Steve Cunningham
California State University Stanislaus

Michael J. Bailey
San Diego Supercomputer Center
University of California San Diego

Abstract

The scene graph, as defined in VRML and Java3D, is a powerful tool for modeling a scene. The ideas contained in the scene graph are fundamental principles in modeling. They give beginning computer graphics students the tools to understand and apply the techniques of hierarchical modeling in scene design and can be directly applied to graphics programming in several graphics APIs, including OpenGL and RenderMan™. This note outlines the approach to modeling with scene graphs and describes how students in a first computer graphics course can build their modeling designs with this approach.

Introduction

We define modeling as the process of defining and organizing a set of geometry that represents a particular scene. While modern graphics APIs can provide students with a great deal of assistance in rendering their images, modeling is usually supported less well and causes students difficulty in beginning computer graphics courses. Organizing a scene with transformations, particularly when that scene involves hierarchies of components and when some of those components are moving, involves relatively complex concepts that should be presented to students systematically to make them more understandable. Hierarchical modeling has long been done—and taught—by using trees or tree-like structures to organize the components of the model. This was done in some presentations of PHIGS [4] and is presented in textbooks such as [5] and [2]. However, the treatments in textbooks have been casual and often sketchy, leaving instructors or students to work out for themselves how to implement these ideas for their work.

More recent graphics systems, such as Java3D [7], [6] and VRML 2 [1], have extended the initial concept of scene graphs in Inventor [8] and other systems, and have formalized the scene graph as a powerful tool for both modeling scenes and organizing the rendering process for those scenes. By understanding and adapting the structure of this more sophisticated scene graph, we can organize a careful and formal tree approach to teaching both the design and the implementation of hierarchical models. This can give students tools to manage not only modeling the geometry of such models, but also animation and interactive control of these models and their components.

In this paper we will lay out a graph structure very much like the formal scene graph that students can use to design a scene and derive the three key transformations that go into creating a scene: the projection transformation, the viewing transformation, and the modeling transformation(s) for the scene's content. This structure is very general and lets the student manage all the fundamental principles in defining a scene and translating it into a graphics API.

A brief summary of scene graphs

We begin by reviewing the current state of the art in scene graphs based on the specifications of Java3D [7][6]. The scene graph has many different aspects and can be complex to understand fully, but it gives us an excellent model of thinking about scenes that we can use in teaching modeling. A brief outline of the Java3D scene graph in Figure 1 will give us a basis to discuss the general approach to graph-structured modeling as it can be applied to teaching.

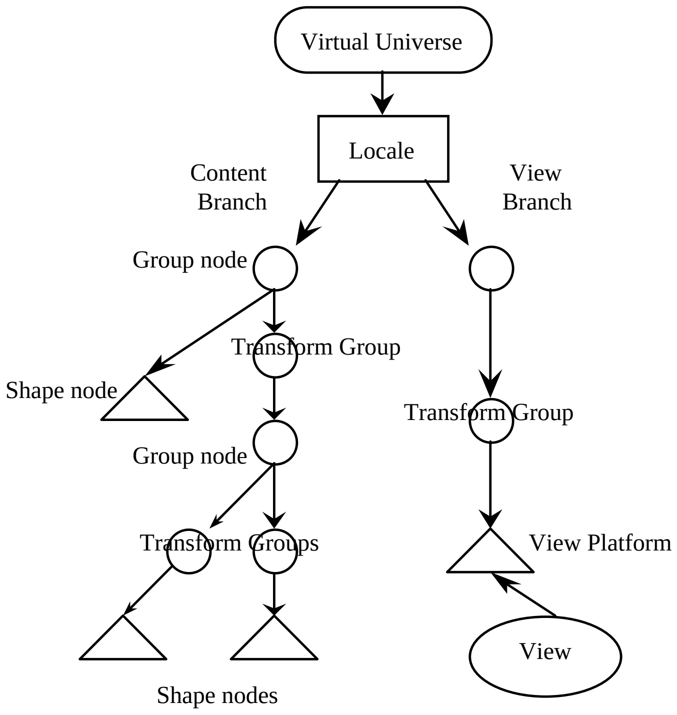

Figure 1: the structure of the scene graph as defined in Java3D

A virtual universe holds one or more (usually one) locales, positions in the universe to put scene graphs. Scene graphs have two kinds of branches: content branches, containing shapes, lights, and other content, and view branches, containing viewing information. This division is somewhat flexible, but we will focus on a standardized approach to give students a framework to build their work upon.

The *content branch* is organized as a collection of nodes that contains group nodes, transform groups, and shape nodes. A *group node* is a grouping structure that can have any number of children; besides simply organizing its children, a group can include a switch that selects which children to present in a scene. A *transform group* is a collection of standard transformations that define a new coordinate system relative to its parent. The transformations will be applied to any of the transform group's children with the convention that transforms "closer" to the geometry (as defined in shape nodes lower in the graph) are applied first. A *shape node* includes both geometry and appearance data for an individual graphic unit. The geometry data includes standard 3D coordinates, normals, and texture coordinates, and can include points, lines, triangles, and quadrilaterals, as well as triangle strips and fans and quadrilateral strips. The appearance data includes color, shading, or texture information. Lights and eye points are included in the content branch as a particular kind of geometry, having position, direction, and other appropriate parameters. Scene graphs also include shared groups, or groups that are included in more than one branch of the graph. In Java3D these are groups of shapes that are included indirectly through link leaf nodes, and any change to a shared group affects all references to that group. This allows scene graphs to include the kind of template-based modeling that is common in graphics applications.

The *view branch* of the scene graph includes the specification of the display device, and thus the projection appropriate for that device. It also specifies the user's position and orientation in the scene and includes a wide range of abstractions of the different kinds of viewing devices that can be used by the viewer. It is intended to permit viewing the same scene on a traditional computer monitor, on a synchronized stereo screen, with a head-mounted display unit, or on multi-screen portals such as CAVEs, and to support a wide range of positional devices including head tracking. This is a much more sophisticated approach than we need for the simple modeling in a beginning computer graphics course, where we need only the viewpoint from which the user will view the scene. In our approach, we consider the eye point as part of the geometry of the scene, so we set the view by including the eye point in the content branch and extract the transformation information for the eye point to create the view transformations in the view branch.

In addition to the modeling aspect of the scene graph, it is also used by the Java3D runtime system to organize the processing as the scene is rendered. Recalling that the scene graph is processed from the bottom up, the content branch is processed first, followed by the viewing transformation and then the projection transformation. For our purpose below, it is productive to think of the viewing transformation as if it were placed at the top of the content branch with the actual view being simply the default. An explicit feature of grouping nodes is that the system does not guarantee any particular sequence in processing the node's branches. Instead, the system can optimize processing by selecting a processing order for efficiency, or can distribute the computations over a networked or multiprocessor system. Thus the programmer must be careful to make no assumptions about the state of the system when any shape node is processed.

The analogue of scene graphs for hierarchical design

We propose a graph structure for designing a scene that is organized very closely along the lines of the scene graph. The scene graph is a very strong structure and we will use it as a reference model for design rather than as a strong structure model, but this reference model will suffice for our teaching purposes. We describe our approach by developing a graph organization for an example scene. Consider the scene of Figure 2, where a helicopter is flying above a landscape and the scene is viewed from a fixed eye point.

This scene contains two principal objects: a helicopter and a ground plane. The helicopter is made up of a body and two rotors, and the ground plane is modeled as a single set of geometry with a texture map. In addition, the scene contains a light and an eye point, both at fixed locations. The first task in modeling such a scene is now complete: to identify all the parts of the scene, organize the parts into a hierarchical set of objects, and put this set of objects into a viewing context. We must next identify the relationship among the parts of the landscape way so we may create the tree that represents the scene. Here we note the relationship among the ground and the parts of the helicopter. Finally, we must put this information into a graph form.

The initial analysis of the scene in Figure 2, organized along the lines of view and content branches, leads to an initial (and partial) graph structure shown in Figure 3. The content branch of this graph captures the organization of the components for the modeling process. This describes how content is assembled to form the image, and the hierarchical structure of this branch helps us organize our modeling components. The view branch of this graph corresponds roughly to projection and viewing. It specifies the projection to be used and develops the projection transformation, as well as the eye position and orientation to develop the viewing transformation.

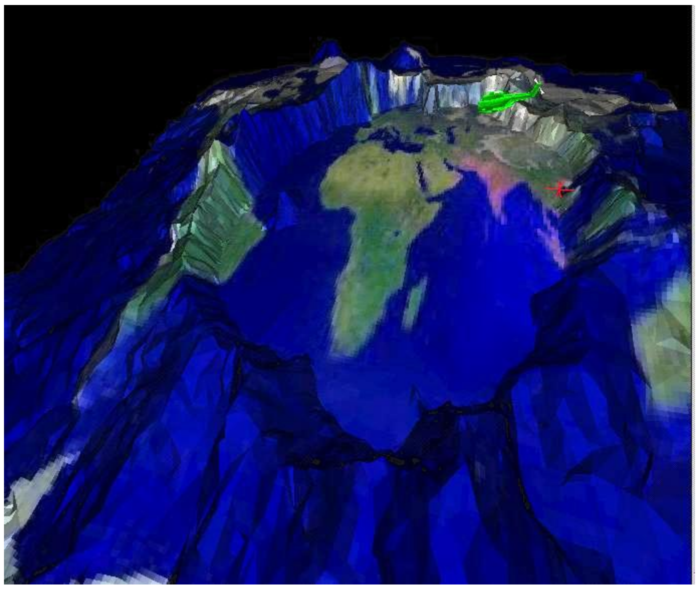

Figure 2: a scene that we will describe with a scene graph

This initial structure is compatible with the approach beginners usually learn in OpenGL, where the view is implemented by using the `gluLookAt(...)` function. This approach only takes you so far, however, because it can be difficult to compute the parameters of this function when the eye point is embedded in the scene and moves with the other content.

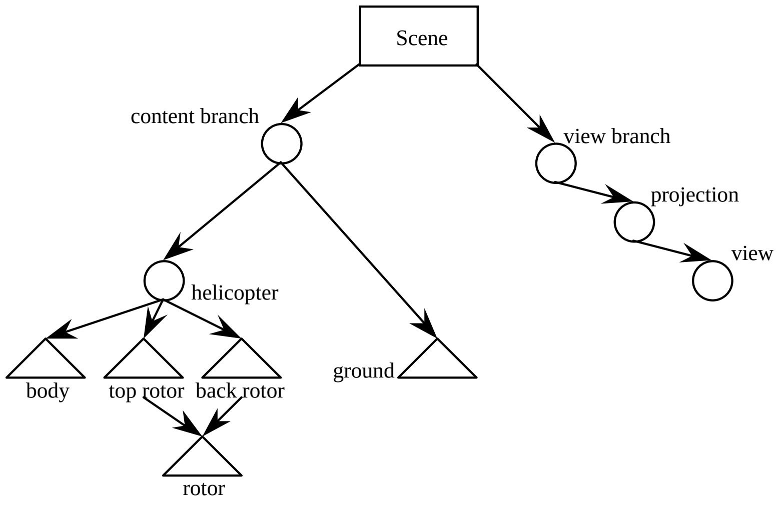

Figure 3: a scene graph that organizes the modeling of our simple scene

This initial approach to the scene graph is incomplete, however, because it merely includes the parts of the scene and describes which parts are associated with what other parts. To expand this first approximation to a more complete graph, we must add several things to the graph:

- the transformation information that describes the relationship among items in a group node, to be applied separately on each branch as indicated,
- the appearance information for each shape node, indicated by the shaded portion of those nodes,
- the light and eye position, either absolute (as shown in Figure 4) or relative to other components of the model, and
- the specification of the projection and view in the view branch.

These are all included in the expanded version of the scene graph with transformations, appearance, eye, and light shown in Figure 4.

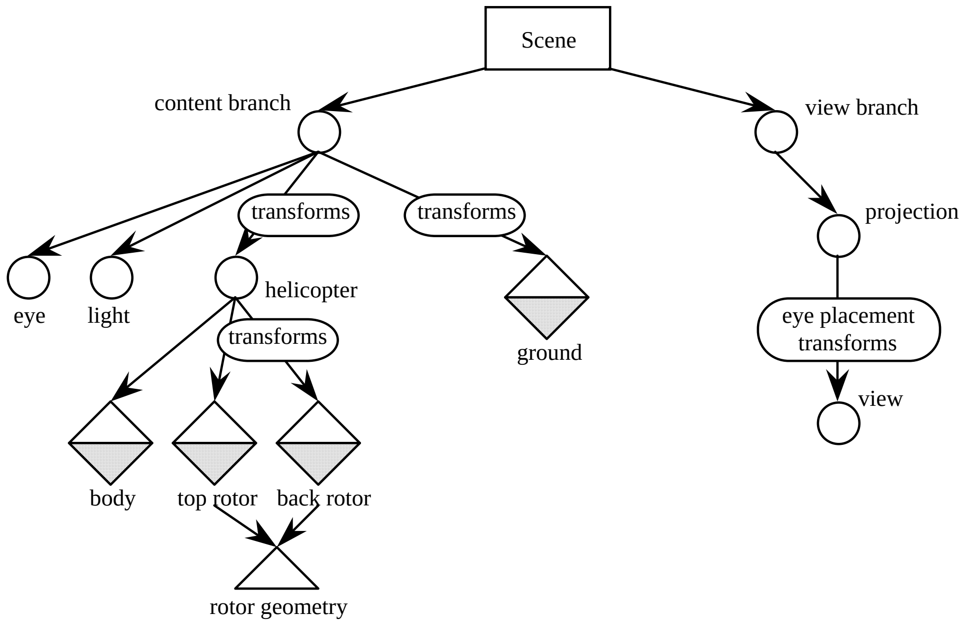

Figure 4: the more complete graph including transformations and appearance

The content branch of this graph handles all the scene modeling and is very much like the content branch of the scene graph. It includes all the geometry nodes of the graph in Figure 3 as well as appearance information; includes explicit transformation nodes to place the geometry into correct sizes, positions, and orientations; includes group nodes to assemble content into logical groupings; and includes lights and the eye point, shown here in fixed positions without excluding the possibility that a light or the eye might be attached to a group instead of being positioned independently. In the example above, it identifies the geometry of the shape nodes such as the rotors or individual trees as shared. This might be implemented, for example, by defining the geometry of the shared shape node in a function and calling that from each of the rotor or tree nodes that uses it.

The view branch of this graph is similar to the view branch of the scene graph but is treated much more simply, containing only projection and view components. The projection component includes the definition of the projection (orthogonal or perspective) for the scene and the definition of the window and viewport for the viewing. The view component includes the information needed to create the viewing transformation, which is simply a copy of the set of transformations that position the eye point in the content branch.

The scene graph for a particular image is not unique because there are many ways to organize a scene. When you have a well-defined transformation for the eye point, you can take advantage of that information to organize the scene graph in a way that can replace the `gluLookAt()` functionality. The real effect of `gluLookAt()` is to create a viewing transformation that is the inverse of the transformation that placed the eye. So as we noted when we discussed scene graphs initially, we can compute the inverse transformation ourselves and place that at the top of the scene graph. Thus we can restructure the scene graph of Figure 4 as shown below in Figure 5 taking any arbitrary eye position. This will be the key point below as we discuss managing the eyepoint as a dynamic part of a scene.

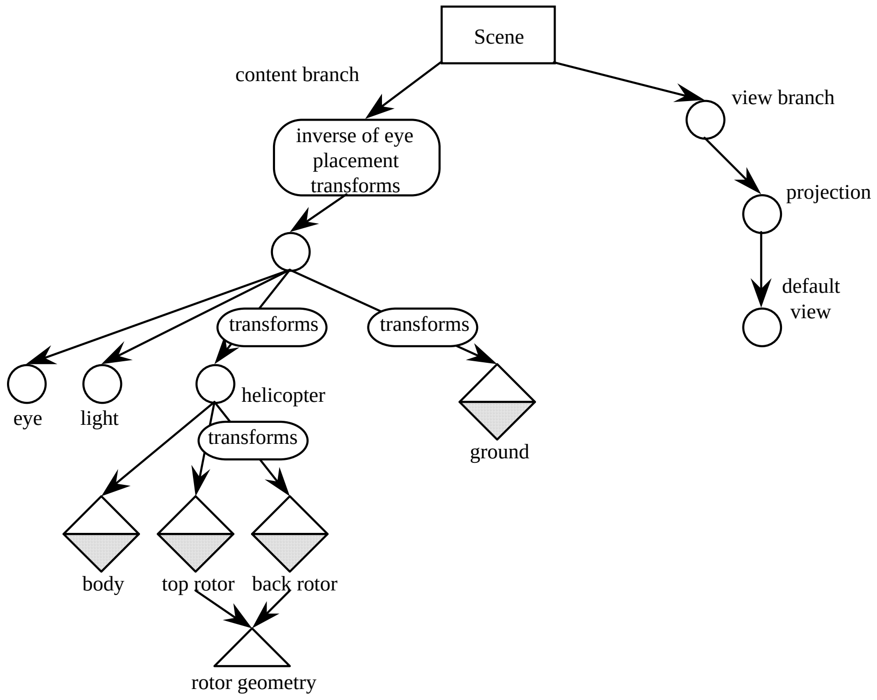

Figure 5: the scene graph with the viewing transformation figured in

It is very important to note that the scene graph need not describe a static geometry. Callbacks for user interaction and other events can affect the graph by controlling parameters of its components, as noted in the re-write guidelines in the next section. This can permit a single graph to describe an animated scene or even alternate views of the scene. The graph may thus be seen as having some components with external controllers, and the controllers are the event callback functions.

As we said in the introduction, we need to extract the three key transformations from this graph. The projection transformation is straightforward and is built from the projection information in the view branch. The viewing transformation is created from the transformation information in the view, and the modeling transformations for the various components are built by working with the various transformations in the content branch as the components are drawn. These operations are all straightforward; we begin with the viewing transformation and move on to coding the modeling transformations.

<u>The viewing transformation</u>

In a scene graph with no view specified, we assume that the default view puts the eye at the origin looking in the negative z-direction with the y-axis upward. If we use a set of transformations to position the eye differently, then the viewing transformation is built by inverting those transformations to restore the eye to the default position. This inversion takes the sequence of transformations that positioned the eye and inverts the primitive transformations in reverse order, so if $T_1T_2T_3...T_K$ is the original transformation sequence, the inverse is $T_K^u...T_3^uT_2^uT_1^u$ where the superscript u indicates inversion, or “undo” as we might say to the beginning student. Because each of the primitive scaling, rotation, and translation transformations is easily inverted, the student will have no difficulty writing the set of inverse transformations.

Deriving the eye transformations from the tree is straightforward. Because we suggest that the eye be considered one of the content components of the scene, we can place the eye at any position relative to other components of the scene. When we do so, we can follow the path from the root of the content branch to the eye to obtain the sequence of transformations that lead to the eye point. That sequence is the eye transformation that we may record in the view branch.

In Figure 6 we show the change that results in the view of Figure 2 when we define the eye to be immediately behind the helicopter, and in Figure 7 we show the change in the scene graph of Figure 4 that implements the changed eye point. The eye transform consists of the transforms that places the helicopter in the scene, followed by the transforms that place the eye relative to the helicopter. Then as we noted earlier, the viewing transformation is the inverse of the eye positioning transformation, which in this case is the inverse of the transformations that placed the eye relative to the helicopter, followed by the inverse of the transformations that placed the helicopter in the scene.

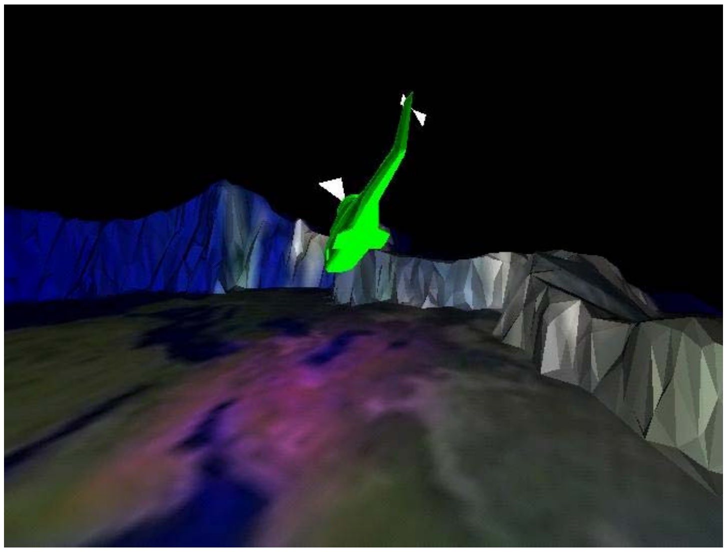

Figure 6: the same scene as in Figure 2 but with the eye point following directly behind the helicopter

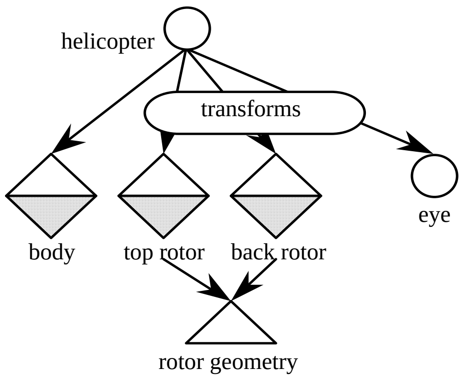

Figure 7: the change in the scene graph of Figure 2 to implement the view in Figure 6

This change in the position of the eye means that the set of transformations that lead to the eye point in the view branch must be changed, but the mechanism of writing the inverse of these transformations in the `display()` function before beginning to write the definition of the scene graph still applies; only the actual transformations to be inverted will change.

This process can readily be generalized. If you want to design a scene with several possible eye points and allow a user to choose among them, you can design the view branch by creating one view for each eye point and using the set of transformations leading to each eye point as the transformation for the corresponding view. The choice of eye point will then create a choice of view, and the transformation for that view will be inverted to create the viewing transformation being used at that time.

Because the viewing transformation is performed before the modeling transformations, the student should be shown that the inverse transformations for the eye are to be applied before the content branch is analyzed and its operations are placed in the code. This has the effect of moving the eye to the top of the content branch and placing the inverse of the eye path at the front of each set of transformations for each shape node, and this explanation may help some students understand the viewing transformation.

Inverting a Mechanism

A common technique in multi-body animation is to "ground" or freeze one of the moving bodies and then let the other bodies continue their relative motion with respect to the frozen body. In this way, the relative relationships among all the bodies are maintained, but the chosen part is seen as being stationary. This is a useful technique if a user wants to zoom in on one of the bodies and examine its relationship to the universe around it in more detail. It is difficult to zoom in on something that is moving. In Mechanical CAD, this process is known as "inverting the mechanism" and it is often desired to allow a user to select and freeze a part as the CAD model is being animated. At the moment of the selection, the chosen part is now "grounded," and the part that was initially grounded is now moving.

In MCAD applications, machine parts are usually hierarchically connected; they are pinned, guided, or connected in some other way to adjacent bodies. In scene graph terms, the relationship among the parts could be shown as in Figure 8. The boxes represent both the geometry of and the relationships between the moving parts. It is understood that there is a transformation between each box that positions a part with respect to its parent. It is also understood that these transformations change with each frame of the animation.

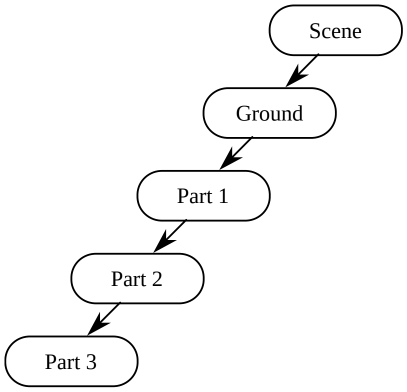

Figure 8: a hierarchy of parts in a mechanism

Note that the hierarchical relationships are not necessary for this discussion. If each body undergoes an independently-computed motion, the scene graph could simply appear as shown in Figure 9.

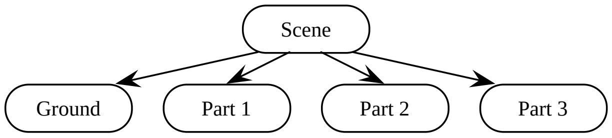

Figure 9: the same parts without a hierarchy

Inverting the mechanism is very similar to the eye placement we discussed in the previous section. We must identify the transformations that place the part to be fixed, and then must invert them so the rest of the mechanism will be seen to move relative to the new now-fixed part. In order to identify the transformations that placed the part to be fixed, we must capture the actual transformations at the moment the part is frozen and then create a duplicate of the hierarchical tree at that moment. The full duplicate branch is not needed, of course; we need create only the part of the tree as deeply as the part that is being frozen. So, for example, if Part #2 was selected to be frozen at a given time, the entire tree would now look like Figure 10, where the * superscript on a part name indicates that it captures the transformations at the moment that the part is frozen.

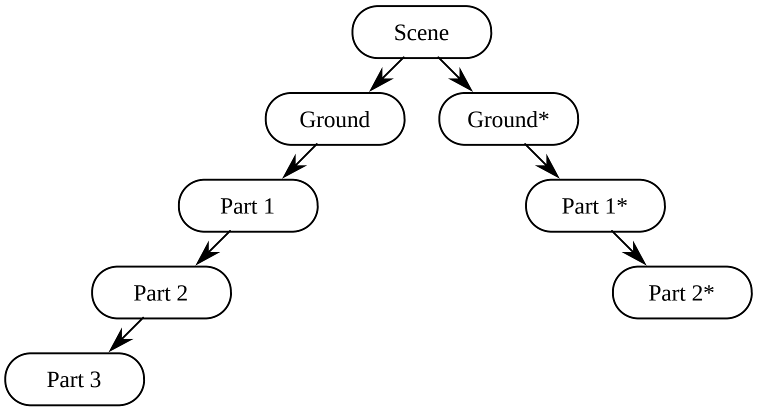

Figure 10: the scene graph with the transformations for Part 2* captured

Grounding a moving part essentially means that an observer sitting on that part now provides the overall scene eye position. So, using the techniques we developed earlier in order to establish an eye position in a scene graph, and denoting the inverse transformations by "Part-1," we slide the right branch of the tree up and over so that Part 2* is at the top of the tree as shown in Figure 11 and thus does not move as the animation proceeds. We could call this mechanism an "AimAt" mechanism, because we aim the view at the part being grounded.

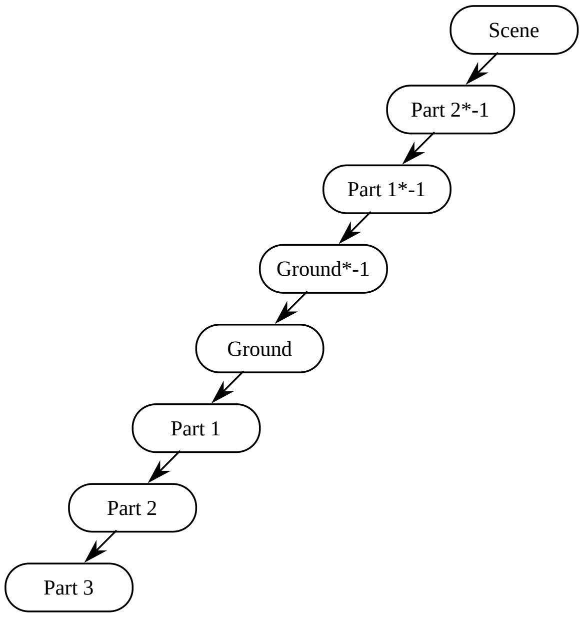

Figure 11: the scene graph with the transformation for Part 2* inverted

The two parts of Figure 12 show time-exposures of a mechanical four-bar linkage. The left-hand image image of the figure shows how the mechanism was originally intended to function, with the bottom piece being ground. The right-hand image in the figure shows the same mechanism in motion with the top piece grounded.

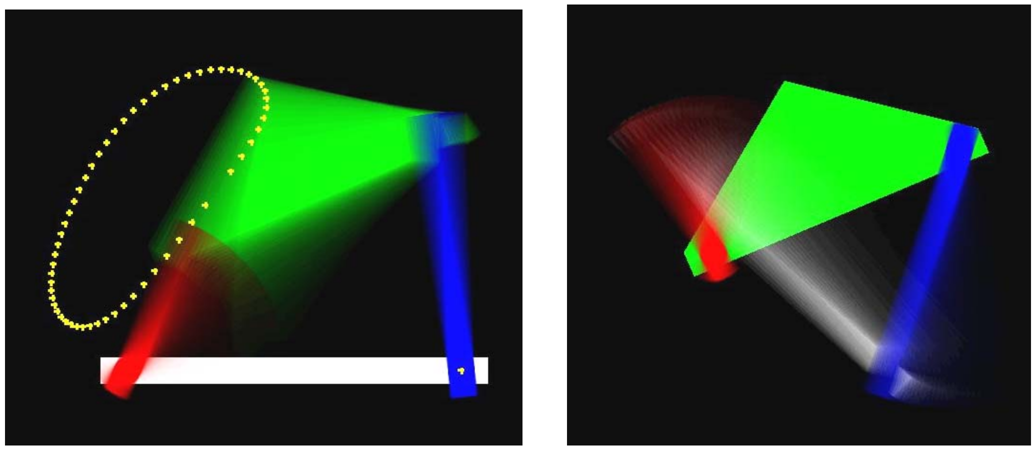

Figure 12: animated mechanisms with different parts fixed

<u>RenderMan™</u>

In the PhotoRealistic RenderMan ™ system[3], the shadow-producing light source, *shadowdistant*, has no explicit notion of an eye position or a light source position. The eye must be explicitly positioned similarly to the OpenGL `gluLookAt()` construct. Each light in the scene must be positioned with the `AimAt()` construct discussed earlier. This makes RenderMan a particularly good example of using the scene graphs described here as a way to organize the 3D scene and position the eye.

Using light sources to cast shadows in RenderMan also drops out nicely as well. RenderMan's approach to shadows requires the scene to be rendered first from the point of view of each light source. These preliminary renderings do not record an RGB image, but instead record a "shadow image" in which the distance from the light to the closest object in the scene is recorded pixel-by-pixel. This is known as a "shadow file". For the final pass, the shadow files are ready in and used to determine what is in a shadow and what is not. Thus, the preliminary renderings use the `LookAt()` construct for each light:

```
RiFormat( (RtInt) 256, (RtInt) 256, 1.0);
RiScreenWindow(-4.0, 4.0, -4.0, 4.0);
RiProjection("orthographic", RI_NULL);
RiDisplay( ZFILE, "zfile", RI_Z, RI_NULL );
LookAtv( LightFrom, LightTo, ZUp );
RiWorldBegin();
   . . .
```

and the final rendering uses the `AimAt()` construct.

```
RiTransformBegin();
   samples = 4;
   AimAtv( LightFrom, LightTo, ZUp );
   RiLightSource( "shadowdistant",
      (RtToken)"shadowname", (RtPointer)&shadowfile,
      (RtToken)"samples", (RtPointer)&samples,
      (RtToken)"lightcolor", (RtPointer)&LightColor,
      (RtToken)"intensity", (RtPointer)&LightInten,
      RI_NULL );
RiTransformEnd();
   . . .
```

Note that the lights must be correctly positioned in the final rendering. Even though that information is no longer needed for shadows, it is still needed to make the light source shading look correct.

In Figure 13, the left-hand image shows a grayscale display of a RenderMan shadow file image, with brighter shades implying a distance that is closer to the eye than the distances shown by darker shades. The right-hand image shows a final rendering of this scene in which the shadow file is read back in and the light source is properly positioned to shade the scene correctly.

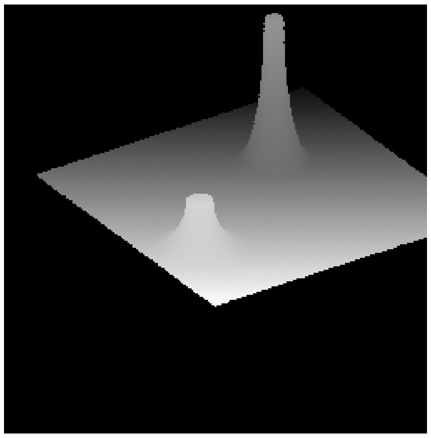

Figure 13: a RenderMan™ shadow file and scene rendering

Using the scene graph analogue for coding

Let us use the name "modeling graph" for the analogue of the scene graph we illustrated in the previous section. Because the modeling graph is intended as a learning tool, we will resist the temptation to formalize its definition beyond the terms we used there:

- shape node containing two components
  - geometry content
  - appearance content
- transformation node
- group node
- projection node
- view node

In this paper, we do not want to look at any kind of automatic parsing of the modeling graph to create the scene. We intend our approach to be used for beginning students who may not be computer science specialists and, even if they are, may not yet have the experience to build a graph traverser. So instead of looking at automatic graph parsing, we want to use the graph to help organize the structure and the relationships in the model to help programmers organize their code to implement the hierarchical modeling.

Once the student knows how to organize all the components of the model in the modeling graph, he or she next needs to write the code to implement the model. This turns out to be straightforward, and we can provide a set of re-write guidelines that allow a student to re-write the graph as code. In this set of rules, we assume that transformations are applied in the reverse of the order they are declared, as they are in OpenGL, for example. This is consistent with most students' first experience with tree handling, which is usually an expression tree which is parsed in leaf-first order. It is also consistent with the Java3D convention that transforms that are "closer" to the geometry (nested more deeply in the scene graph) are applied first.

The informal re-write guidelines are as follows, including the re-writes for the view branch as well as the content branch:

- Nodes in the view branch involve only the window, viewport, projection, and viewing transformations. The window, viewport, and projection are handled by simple functions in the API and should be at the top of the display function.

- The viewing transformation is built from the transformations of the eye point within the content branch by copying those transformations and undoing them to place the eye effectively at the top of the content branch. This sequence should be next in the display function.
- The content branch of the modeling graph is usually maintained fully within the display function, but parts of it may be handled by other functions called from within the display, depending on the design of the scene. A function that defines the geometry of an object may be used by one or more shape nodes. The modeling may be affected by parameters set by event callbacks, including selections of the eye point, lights, or objects to be displayed in the view.
- Group nodes are points where several elements are assembled into a single object. Each separate object is a different branch from the group node. Before writing the code for a branch that includes a transformation group, the student should push the modelview matrix; when returning from the branch, the student should pop the modelview matrix.
- Transformation nodes include the familiar translations, rotations, and scaling that are used in the normal ways, including any transformations that are part of animation or user control. In writing code from the modeling graph, students can write the transformations in the same sequence as they appear in the tree, because the bottom-up nature of the design work corresponds to the last-defined, first-used order of transformations. Because of the simple nature of each transformation primitive, it is straightforward to undo each as needed to create the viewing transformation.
- Shape nodes involve both geometry and appearance, and the appearance must be done first because the current appearance is applied when geometry is defined.
    - An appearance node can contain texture, color, blending, or material information that will make control how the geometry is rendered and thus how it will appear in the scene.
    - A geometry node will contain vertex information, normal information, and geometry structure information such as strip or fan organization.
- Most of the nodes in the content branch can be affected by any interaction or other event-driven activity. This can be done by defining the content by parameters that are modified by the event callbacks. These parameters can control location (by parametrizing rotations or translations), size (by parametrizing scaling), appearance (by parametrizing appearance details), or even content (by parametrizing switches in the group nodes).

In the example above, we would use the tree to write code as shown in skeleton form in Figure 14. Most of the details, such as the parameters for the transformations and the details of the appearance of individual objects, have been omitted, but we have used indentation to show the push/pop pairs for the modelview matrix and to be able to see the operations between these pairs easily. This is straightforward for a student to understand and to learn to organize for himself or herself.

Animation is simple to add to this example. The rotors can be animated by adding an extra rotation in their definition plane immediately after they are scaled and before the transformations that orient them to be placed on the helicopter body, and by updating angle of the extra rotation each time the idle event callback executes. The helicopter's behavior itself can be animated by updating the parameters of transformations that are used to position it, again with the updates coming from the idle callback. The helicopter's behavior may be controlled by the user if the positioning transformation parameters are updated by callbacks of user interaction events. So there are ample opportunities to have this graph represent a dynamic environment and to include the dynamics in creating the model from the beginning.

Other variations in this scene could by developed by changing the position of the light from its current absolute position to a position relative to the ground (by placing the light as a part of the branch group containing the ground) or to a position relative to the helicopter (by placing the light as a part of the branch group containing the helicopter). The eye point could similarly be placed relative to another part of the scene, or either or both could be placed with transformations that are controlled by user interaction with the interaction event callbacks setting the transformation parameters.

We emphasize that the student should include appearance content with each shape node. Many of the appearance parameters involve a saved state in APIs such as OpenGL [9] and so parameters set for one shape will be retained unless they are re-set for the new shape. It is possible to design your scene so that shared appearances will be generated consecutively in order to increase the efficiency of rendering the scene, but this is a specialized organization that is inconsistent with more advanced APIs such as Java3D. Thus in order to give students the best general background, re-setting the appearance with each shape is better than relying on saved state, but students can be given information on state so they can focus on efficiency if they wish.

```
display()
   set the viewport and projection as needed
   initialize modelview matrix to identity
   define viewing transformation by undoing eye location
   set eye through gluLookAt with default values
   define light position         // note absolute location
   pushMatrix()      // ground
      translate
      rotate
      scale
      define ground appearance (texture)
      draw ground
   popMatrix()
   pushMatrix()      // helicopter
      translate
      rotate
      scale
      pushMatrix()   // top rotor
         translate
         rotate
         scale
         define top rotor appearance
         draw top rotor
      popMatrix()
      pushMatrix()   // back rotor
         translate
         rotate
         scale
         define back rotor appearance
         draw back rotor
      pushMatrix()
      // assume no transformation for the body
      define body appearance
      draw body
   popMatrix()
   swap buffers
```

Figure 14: code sketch to implement the modeling in Figure 4

Conclusions

The modeling graph presented in this paper is a natural analogue of the VRML or Java3D scene graph that is itself a development of previous graph-oriented modeling, and we have shown that it has many applications in modeling outside those original systems. We believe that the modeling graph can give students assistance in structuring their modeling that allows them to grasp modeling concepts more easily and to create models that include both animation and interactive control of their components. This kind of modeling and control is becoming particularly important as the increasing power of low-cost computers and the ease of developing graphics applications with APIs such as OpenGL lead to an increasing study of computer graphics among students outside computer science.

References

[1] Andrea L. Ames, David R. Nadeau, and John L. Moreland, *VRML 2.0 Sourcebook*, Wiley, 1997

[2] Ed Angel, *Interactive Computer Graphics with OpenGL*, second edition, Addison-Wesley, 2000

[3] Anthony A. Apodaca and Larry Gritz, *Advanced Renderman*, Morgan Kaufmann, 2000

[4] Maxine D. Brown, *Understanding PHIGS: The Hierarchical Computer Graphics Standard*, Template Software Division of Megatek Corporation, 1985

[5] James D. Foley et al, *Computer Graphics Principles and Practice*, 2nd edition, Addison-Wesley, 1990

[6] Henry Sowrizal, Kevin Rushforth, and Michael Deering, *The Java3D™ 3D API Specification*, Addison-Wesley, 1995

[7] Henry A. Sowizral and David R. Nadeau, *Introduction to Programming with Java 3D*, SIGGRAPH 99 Course Notes, Course 40

[8] Josie Wernecke, *The Inventor Mentor*, Addison-Wesley, 1994

[9] Mason Woo et al., *OpenGL Programmers Guide*, 3rd edition (version 1.2), Addison-Wesley, 1999

Author contact information:

Steve Cunningham
Department of Computer Science
California State University Stanislaus
801 W. Monte Vista Avenue
Turlock, CA 95382
+1.209.667.3176; fax +1.209.667.3848
cunningham@siggraph.org

Michael J. Bailey
Senior Principal Scientist
San Diego Supercomputer Center
PO Box 85608
San Diego, CA 92186
+1.858.534.5142; fax +1.858.534.5152
mjb@sdsc.edu