Chapter 3

Scene Graphs and Renderers

At its lowest level, a graphics engine has the responsibility to draw the objects that are visible to an observer. An engine programmer typically uses a graphics API such as OpenGL or Direct3D to implement a renderer whose job it is to correctly draw the objects. On some platforms if no hardware acceleration exists or a standard API is unavailable, the programmer might even write the entire graphics system to run on the CPU; the result is called a software renderer. Although current consumer graphics hardware has a lot of power and obviates the need for writing software renderers on systems with this hardware, the ability to write software renderers is still important—for example, on embedded systems with graphics capabilities such as cell phones and handheld devices. That said, this book does not cover the topic of writing a fully featured software renderer. Rather, the focus is on writing a renderer using an existing graphics API, but hidden by an abstract rendering layer to allow applications not to worry about whether the API is standard or user-written. Wild Magic has renderers for OpenGL, for Direct3D, and even one to illustrate how you might write a software renderer. The examples in the book refer to the OpenGL renderer, but the ideas apply equally as well to Direct3D.

Building a renderer to draw primitives such as points, polylines, triangle meshes, and triangle strips using basic visual effects such as textures, materials, and lighting is a straightforward process that is well understood. The process is sometimes referred to as the *fixed-function pipeline*. The graphics API limits you to calling functions supported only by that API. Recent generations of graphics hardware, though, now provide the ability to program the hardware. The programs are called *shaders* and come in two flavors, *vertex shaders* and *pixel shaders*. Vertex shaders allow you to control drawing based on vertex attributes such as positions, normals, and colors. A simple vertex shader might draw a triangle mesh where the user supplies vertex positions and colors. Pixel shaders allow you to control drawing through image-based attributes. A simple pixel shader might draw a triangle mesh where the user supplies vertex positions, texture coordinates, and a texture image to be interpolated to fill in the final pixels that correspond to the drawn object. Writing shaders can be more challenging than using the fixed-function pipeline.

A large portion of Usenet postings to groups related to computer graphics and rendering are of the form "How do I do *X* with my graphics API?" The answers tend to be compact and concise with supporting code samples on the order of a few lines of API code. An abundant supply of Web sites may be found that provide tutorials and code samples to help novice programmers with their ventures into writing renderers for OpenGL or Direct3D. These are useful learning tools for understanding what it takes to do low-level drawing. But in my opinion they lack insight into how you architect a graphics system that supports complex applications such as games. In particular:

1. How do you provide data efficiently to the renderer to support applications that must run in real time?
2. How does an application interface with the renderer?
3. How do you make it easy for the application programmer to use the engine?
4. How can you help minimize the changes to the system when new features must be added to support the latest creations from your artists?

Although other questions may be asked, the four mentioned are the most relevant to a game application—my conclusion based on interacting with game companies that used NetImmerse as their game engine.

The first question is clear. The demands for a 3D game are that it run at real-time rates. Asking the renderer to draw every possible object in the game's world is clearly not going to support real time. The clipping and depth buffering mechanisms in the graphics API will eliminate those objects that are not visible, but these mechanisms use computation time. Moreover, they have no high-level knowledge of the game logic or data organization. As an engine programmer, you have that knowledge and can guide the renderer accordingly. The game's world is referred to as the *scene*. The objects in the world are part of the scene. When you toss in the interrelationships between the objects and their various attributes, visual or physical, you have what is called a *scene graph*. If you can limit the objects sent to the renderer to be only those that are visible or potentially visible, the workload of the renderer is greatly reduced. This type of data handling is called *scene graph management*. Visibility determination is one aspect of the management, but there are others, many of which are discussed later in the book.

Scene graph management is a higher-level system than the rendering system and may be viewed as a front end to the renderer, one constructed to efficiently feed it. The design of the interface between the two systems is important to get right, especially when the graphics engines evolve as rapidly as they do for game applications. This is the essence of the second question asked earlier. As new requirements are introduced during game development, the last thing you want to do is change the interface between data management and drawing. Such changes require maintenance of both the scene graph and rendering systems and can adversely affect a shipping schedule. Although some change is inevitable, a carefully thought-out abstract rendering layer will minimize the impact of those changes to other subsystems of the engine.

The third question is quite important, especially when your plan is to market your graphics engine as a middleware tool to other companies, or even to internal clients within your own company. A scene graph management system helps isolate the application programmer from the low-level details of rendering. However, it must expose the capabilities of the rendering system in a high-level and easy-to-use manner. I believe this aspect of Wild Magic to be the one that has attracted the majority of users. Application programmers can focus on the high-level details and semantics of how their objects must look and interact in the application. The low-level rendering details essentially become irrelevant at this stage!

The fourth question is, perhaps, the most important one. Anyone who has worked on a game project knows that the requirements change frequently—sometimes even on a daily or weekly basis. This aspect of frequent change is what makes software engineering for a game somewhat different than that for other areas of application. Knowing that change will occur as often as it does, you need to carefully architect the scene graph management system so that the impact of a change is minimal and confined to a small portion of the engine. In my experience, the worst type of requirement change is one of adding new visual effects or new geometric object types to the system. Yet these are exactly what you expect to occur most often during game development! Your favorite artist is hard at work creating a brand-new feature: environment-mapped, bump-mapped, iridescent (EMBMI) clouds. The cloud geometry is a mixture of points, polylines, and triangle meshes. The lead artist approves the feature, and the programming staff is asked to support it as soon as possible. After the usual fracas between the artists and programmers, with each side complaining about how the other side does not understand its side, the game producer intervenes and says, "Just do it."[1] Now you must create a new set of classes in the scene graph management system to support EMBMI clouds. The rendering system might (or might not) have to be modified to support the visual aspects of the clouds. The streaming system for persistent storage of the game assets must be modified to handle the new type. Finally, you must modify the exporter for the artist's modeling

1. Okay, I made this one up, but it is illustrative of what you might encounter. About the producer's decision: Let's face it. A good story, good game play, and fantastic artwork are essential. No consumer will notice that fancy hack you made to reduce an intersection test from 7 cycles to 6 cycles. Relish the fact that your name will be on the credits, hope that the consumer will actually read the credits, and look forward to the next Game Developer's Conference where your friends *will* congratulate you on that amazing hack!

package to export EMBMI clouds to the engine's file format. If any of these tasks requires you to significantly rewrite the scene graph manager or the renderer, there is a good chance that the original architectures were not designed carefully enough to anticipate such changes.[2]

This chapter is about the basic ideas that Wild Magic uses for scene graph management and for abstracting the renderer layer. I explain my design choices, but keep in mind that there may be other equally valid choices. My goal is not to compare with as many competing ideas as possible. Rather, it is to make it clear *what* motivated me to make my choices. The necessity to solve various problems that arise in data management might very well lead someone else to different choices, but the problems to solve are certainly the same.

Section 3.1 is a discussion of the subsystems I chose for the basic services provided by the scene graph management. These include the classes `Spatial`, `Node`, `Geometry`, and `Renderer`, which correspond to spatial decomposition, transformation, grouping of related data, representation of geometric data, and drawing of the data.

Sections 3.2 and 3.3 describe the geometric state and geometric types of the `Spatial` and `Geometry` classes. Topics include transformations and coordinate systems, bounding volumes, updating geometric state, and specialized geometric types.

Section 3.4 is about render state and effects, which is the information that controls how objects are drawn. I discuss an important distinction between the architecture of Wild Magic version 3 and older versions of the engine: *global state* and *local state*. Global state affects all objects in a specified portion of the scene (depth buffering, alpha blending, wire frame, etc.), whereas local state affects a single, specified object in the scene (texture coordinates, vertex colors, etc.).

Section 3.5 is a discussion about camera models and the renderer architecture. Also discussed are issues regarding caching of data on the graphics card and multipass rendering, not from a performance perspective, but from the perspective of how a scene graph management system can support them in a manner independent of the underlying graphics API.

3.1 The Core Classes

The most important subsystems of scene graph management are encapsulated in the classes `Spatial`, `Node`, `Geometry`, and the abstract renderer layer `Renderer`. The first three are designed to support feeding data to the last in an efficient manner. Figure 3.1 is the most important figure you will see in this book. The schematic diagram shows how the four classes are interrelated.

2. Be aware that major rewrites in the middle of a game development cycle can severely affect the value of your stock options!

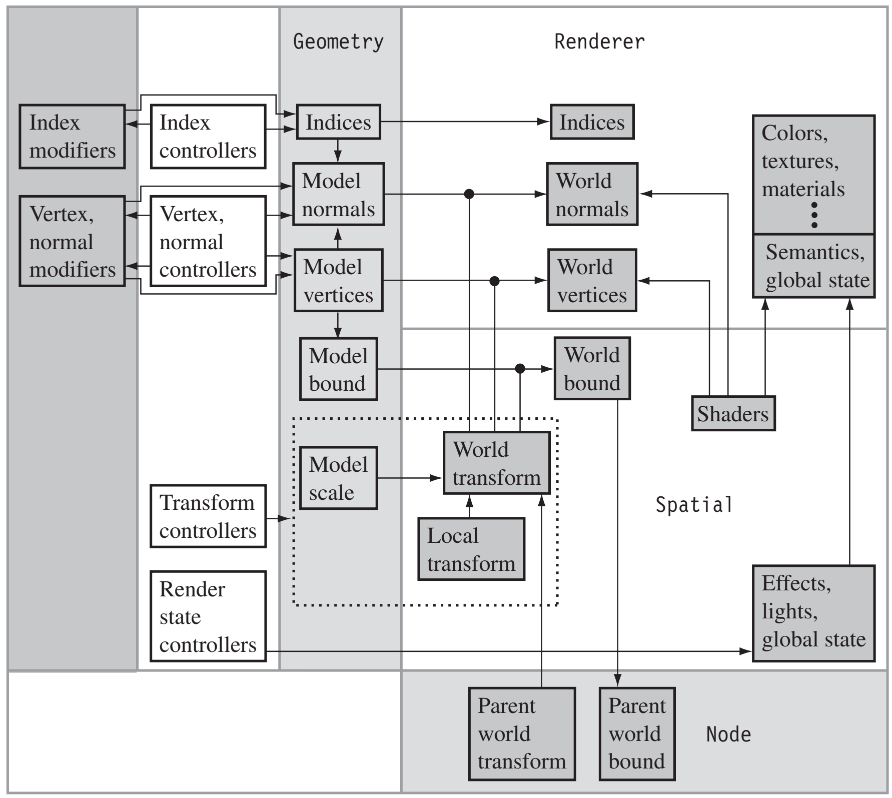


Figure 3.1 The interrelationships among classes `Spatial`, `Node`, `Geometry`, and `Renderer`.

The discussions in this section are all about why the various boxed items in the diagram are encapsulated as shown. The arrows in the diagram imply a loose form of dependency: An object at the arrowhead depends in some form on the object at the origin of the arrow.

3.1.1 Motivation for the Classes

Before you can draw objects using a renderer, you actually need objects to draw! Of course, this is the role of artists in the game development process. Using a modeling package, an artist will create geometric data, usually in the form of points, polylines, and triangle meshes, and assign various visual attributes, including textures, materials, and lighting. Additional information is also created by the artists. For example, keyframe data may be added to a biped structure for the purposes of animation of the character. Complicated models such as a biped are typically implemented by the modeling package using a scene graph hierarchy itself! For illustration, though, consider a simple, inanimate object such as a model of a wooden table.

Geometry

The table consists of geometric information in the form of a collection of *model vertices*. For convenience, suppose they are stored in an array `V[i]` for $0 \le i < n$. Most likely the table is modeled as a triangle mesh. The triangles are defined as triples of vertices, ordered in a consistent manner that allows you to say which side of the triangle is outward facing from a display perspective, and which side is inward facing. A classical choice for outward-facing triangles is to use counterclockwise ordering: If an observer is viewing the plane of the triangle and that plane has a normal vector pointing to the side of the plane on which the observer is located, the triangle vertices are seen in a counterclockwise order in that plane. The triangle information is usually stored as a collection of triples of *indices* into the vertex array. Thus, a triple `(i0,i1,i2)` refers to a triangle whose vertices are `(V[i0],V[i1],V[i2])`. If dynamic lighting of the table is desired, an artist might additionally create vertex *model normals*, although in many cases it is sufficient to generate the normals procedurally. Finally, the model units are possibly of a different size than the units used in the game's world, or the model is intended to be drawn in a different size than what the modeling package does. A *model scale* may be applied by the artist to accommodate these. This does allow for nonuniform scaling, where each spatial dimension may be scaled independently of the others. The region of space that the model occupies is represented by a *model bound*, typically a sphere that encloses all the vertices, but this information can always be generated procedurally and does not require the artist's input. The model bound is useful for identifying whether or not the model is currently visible to an observer. All the model information created by the artist, or procedurally generated from what the artist produces, is encapsulated by the class `Geometry`, as shown in Figure 3.1.

Spatial

Suppose that the artist was responsible for creating both a table and a room in which the table is placed. The table and room will most likely be created in separate modeling sessions. When working with the room model, it would be convenient to load the already-created table model and place it in the room. The technical problem is that the table and room were created in their own, independent *coordinate systems*. To place the table, it must be translated, oriented, and possibly scaled. The resulting *local transformation* is a necessary feature of the final scene for the game. I use the adjective *local* to indicate that the transformation is applied to the table relative to the coordinate system of the room. That is, the table is *located* in the room, and the relationship between the room and table may be thought of as a *parent-child* relationship. You start with the room (the parent) and place the table (the child) in the room using the coordinate system of the room. The room itself may be situated relative to another object—for example, a house—requiring a local transformation of the room into the coordinate system of the house. Assuming the coordinate system of the house is used for the game's world coordinate system, there is an implied *world transformation* from each object's model space to the world space. It is intuitive that the model bound for an object in model space has a counterpart in world space, a *world bound*, which is obtained by applying the world transformation to the model bound. The local and world transformations and the world bound are encapsulated by the class `Spatial`, as shown in Figure 3.1. The (nonuniform) model scale of the `Geometry` class and the transformations of the `Spatial` class are surrounded by a dotted-line box to indicate that both participate in transformations, even though the data is contained in their respective classes.

Node

The example of a house, room, and table has another issue that is partially related to the local and world transformations. The objects are ordered in a natural hierarchy. To make the example more illustrative, consider a house with two rooms, with a table and chair in one room, and a plate, fork, and knife placed on the table. The hierarchy for the objects is shown in Figure 3.2. Each object is represented by a node in the hierarchy.

The objects are all created separately. The hierarchy represents parent-child relationships regarding how a child object is placed relative to its parent object. The Plate,

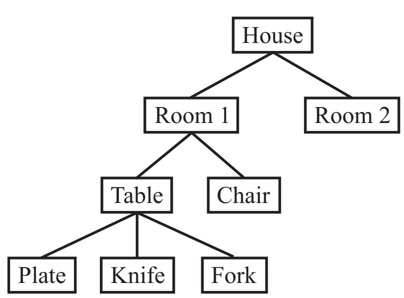


Figure 3.2 A hierarchy to represent a collection of related objects.

Knife, and Fork are assigned local transformations relative to the Table. The Table and Chair are assigned local transformations relative to Room 1. Room 1 and Room 2 are assigned local transformations relative to the House. Each object has world transformations to place it directly in the world. If L_{object} is the local transformation that places the object in the coordinate system of its parent and W_{object} is the world transformation of the object, the hierarchy implies the following matrix compositions. The order of application to vectors (the vertices) is from right to left according to the conventions used in Wild Magic

$$
\begin{aligned}
W_{\text{House}} &= L_{\text{House}} \\
W_{\text{Room1}} &= W_{\text{House}}\, L_{\text{Room1}} = L_{\text{House}}\, L_{\text{Room1}} \\
W_{\text{Room2}} &= W_{\text{House}}\, L_{\text{Room2}} = L_{\text{House}}\, L_{\text{Room2}} \\
W_{\text{Table}} &= W_{\text{Room1}}\, L_{\text{Table}} \;\; = L_{\text{House}}\, L_{\text{Room1}}\, L_{\text{Table}} \\
W_{\text{Chair}} &= W_{\text{Room1}}\, L_{\text{Chair}} \;\; = L_{\text{House}}\, L_{\text{Room1}}\, L_{\text{Chair}} \\
W_{\text{Plate}} &= W_{\text{Table}}\, L_{\text{Plate}} \quad = L_{\text{House}}\, L_{\text{Room1}}\, L_{\text{Table}}\, L_{\text{Plate}} \\
W_{\text{Knife}} &= W_{\text{Table}}\, L_{\text{Knife}} \quad = L_{\text{House}}\, L_{\text{Room1}}\, L_{\text{Table}}\, L_{\text{Knife}} \\
W_{\text{Fork}} &= W_{\text{Table}}\, L_{\text{Fork}} \quad = L_{\text{House}}\, L_{\text{Room1}}\, L_{\text{Table}}\, L_{\text{Fork}}.
\end{aligned}
$$

The first equation says that the house is placed in the world directly. The local and world transformations are the same. The second equation says that Room 1 is transformed first into the coordinate system of the House, then is transformed to the world by the House's world transformation. The other equations have similar interpretations. The last one says that the Fork is transformed into the coordinate system of the Table, then transformed to the coordinate system of Room 1, then transformed to the coordinate system of the House, then transformed to the world coordinates. A path through the tree of parent-child relationships has a corresponding sequence of local transformations that are composited. Although each local transformation may be applied one at a time, it is more efficient to use the world transformation of the parent (already calculated by the parent) and the local transformation of the child to perform a single matrix product that is the world transformation of the child.

The grouping together of objects in a hierarchy is the role of the `Node` class. The compositing of transformations is accomplished through a depth-first traversal of the parent-child tree. Each parent node provides its world transformation to its child nodes in order for the children to compute their world transformations, naturally a recursive process. The transformations are propagated *down the hierarchy* (from root node to leaf nodes).

Each geometry object has a model bound associated with it. A node does not have a model bound per se, given that it only groups together objects, but it can be assigned a world bound. The world bound indicates that portion of space containing the collection of objects represented by the node. Keep in mind that the bound is a coarse measurement of occupation, and that not all of the space contained in the bound is occupied by the object. A natural choice for the world bound of a node is any bound that contains the world bounds of its children. However, it is not necessary that the world bound contain the child bounds. All that matters is that the *objects* represented by the child nodes are contained in the world bound. Once a world bound is assigned to a node, it is possible to define a model bound—the one obtained by applying the inverse world transformation to the world bound. A model bound for a node is rarely used, so the `Node` class does not have a data member to store this information. If needed, it can be computed on the fly from other data.

Each time local transformations are modified at a node in the scene, the world transformations must be recalculated by a traversal of the subtree rooted at that node. But a change in world transformations also implies a change in the world bounds. After the transformations are propagated down the hierarchy, new world bounds must be recomputed at the child nodes and propagated *up the hierarchy* (from leaf nodes to root node) to parent nodes so that they may also recompute their world bounds.

Figure 3.1 shows the relationship between transformations and bounds. A connection is shown between the world transformation (in `Spatial`) and the link between the model bound (in `Geometry`) and the world bound (in `Spatial`). Together these indicate that the model bound is transformed to the world bound by the world transformation. The world transformation at a child node depends on its parent's world transformation. The relationship is shown in the figure by an arrow. The composition of the transformations occurs during the downward pass through the hierarchy. The parent's world bound depends on the child's world bound. The relationship is also shown in the figure by an arrow. The recalculation of the world bounds occurs during the upward passes through the hierarchy. The downward and upward passes together are referred to as a *geometric update*, whose implementation details will be discussed later.

Renderer

Figure 3.1 has a block representing the rendering layer in the engine. Naturally, the renderer needs to be fed the geometric data such as vertices and normals, and this data must be in its final position and orientation in the world. The renderer needs to know how the vertices are related to each other, say, as a triangle mesh, so the indices must also be provided. Notice that some connections are shown between the world transformations (in `Spatial`) and the links between the model vertices and normals and the world vertices and normals. These indicate that *someone* must be responsible for applying the transformations to the model data before the renderer draws them. Although the `Spatial` class can be given the responsibility, most likely performing the calculations on the central processing unit (CPU), the `Renderer` class instead takes on the responsibility. A software renderer most likely implements the transformations to be performed on the CPU, but renderers using current graphics hardware will allow the graphics processing unit (GPU) to do the calculations. Because the target processor is not always the CPU, it is natural to hide the transformation of model data inside the renderer layer.

The renderer must also be provided with any vertex attributes, texture maps, shader programs, and anything else needed to properly draw an object. On the renderer side, all this information is shown in the box in that portion of Figure 3.1 corresponding to the `Renderer` class. The provider of the information is class `Spatial`. Why `Spatial` and not `Geometry`? The choice is not immediately obvious. For simple objects consisting of triangle meshes and basic attributes such as vertex colors, materials, or textures, placing the data in `Geometry` makes sense. However, more complicated special effects (1) may be applied to the entire collection of geometric objects (at the leaf nodes) contained in a subtree of the hierarchy or (2) may require multiple drawing passes in a subtree. An example of (1) is projected textures, where a texture is projected from a postulated light source onto the surfaces of objects visible to the light. It is natural that a node store such a "global effect" rather than share the effect multiple times at all the geometric objects in the subtree. Shader programs are also stored by the `Spatial` class for the same reason. A shader can affect multiple objects, all in the same subtree of the hierarchy. An example of (2) is planar, projected shadows, where an object casts shadows onto multiple planes. Each casting of a shadow onto the plane requires its own drawing pass. The hierarchy support in Wild Magic is designed to handle both (1) and (2).

Controllers and Modifiers

The word *animation* tends to be used in the context of motion of characters or objects. I use the word in a more general sense to refer to any time-varying quantity in the scene. The engine has support for animation through *controllers*; the abstract base class is `Controller`. Figure 3.1 illustrates some standard quantities that are controlled.

The most common are transform controllers—for example, keyframe controllers or inverse kinematic controllers. For keyframe controllers, an artist provides a set of positions and orientations for objects (i.e., for the nodes in the hierarchy that represent the objects). A keyframe controller interpolates the keyframes to provide smooth motion over time. For inverse kinematic controllers, the positions and orientations for objects are determined by constraints that require the object to be in certain configurations. For example, a hand on a character must be translated and rotated to pick up a glass. The controller selects the translations and rotations for the hand according to where the glass is.

Vertex and normal controllers are used for morphing and mesh deformation. Render state controllers are used for animating just about any effect you like. For example, a controller could be used to vary the color of a light source. A texture may be animated by varying the texture coordinates associated with the texture and the object to which the texture applies. This type of effect is useful for giving the effect that a water surface is in motion.

Index controllers are less common, but are used to dynamically change the topology of a triangle mesh or strip. For example, continuous level of detail algorithms may be implemented using controllers.

Controllers are not limited to those shown in Figure 3.1. Use your creativity to implement as complex an animated effect as you can dream up.

I use the term *modifier* to indicate additional semantics applied to a collection of vertices, normals, and indices. The `Geometry` class is a container for these items, but is itself an abstract class. The main modifier is class `TriMesh`, which is derived from `Geometry`, and this class is used to provide indices to the base class. A similar example is class `TriStrip`, where the indices are implicitly created by the class and provided to the `Geometry` base class. In both cases, the derived classes may be viewed as index modifiers of the geometry base class.

Other geometric-based classes may also be viewed as modifiers of `Geometry`, including points (class `Polypoint`) and polylines (class `Polyline`). Both classes may be viewed as vertex modifiers. Particle systems (base class `Particles`) are derived from class `TriMesh`. The particles are drawn as rectangular billboards (the triangle mesh stores the rectangles as pairs of triangles), and so may be thought of as index modifiers. However, the physical aspects of particles are tied into only the point locations. In this sense, particle systems are vertex modifiers of the `Geometry` class.

How one adds the concept of modifiers to an engine is up for debate. The controller system allows you to attach a list of controllers to an object. Each controller manages the animation of some member (or members) of the object. As you add new `Controller`-derived classes, the basic controller system need not change. This is a good thing since you may extend the behavior of the engine without having to rearchitect the core. Preserving old behavior when adding new features is related to the object-oriented principle called the *open-closed principle*. After building a system that, over time, is demonstrated to function as designed and is robust, you want it to be *closed* to further changes in order to protect its integrity. Yet you also want the system to be *open* to extension with new features. Having a core system such as the controllers that allows you to create new features and support them in the (closed) core is one way in which you can have both open and closed.

The classical manner in which you obtain the open-closed principle, though, is through class derivation. The base class represents the closed portion of the system, whereas a derived class represents the open portion. Regarding modifiers, I decided to use class derivation to define the semantics. Such semantics can be arbitrarily complex—something not easily fitted by a system that allows a list of modifiers to be attached to an object. A derived class allows you to implement whatever interface is necessary to support the modifications. Controllers, on the other hand, have simple semantics. Each represents management of the animation of one or more object members, and each implements an update function that is called by the core system. The controller list-based system is natural for such simple objects.

3.1.2 Spatial Hierarchy Design

The main design goal for class Spatial is to represent a coordinate system in space. Naturally, the class members should include the local and world transformations and the world bounding volume, as discussed previously. The Geometry and Node classes themselves involve transformations and bounding volumes, so it is natural to derive these from Spatial. What is not immediately clear is the choice for having both classes Spatial and Node. In Figure 3.2, the objects Table, Plate, Knife, Fork, and Chair are Geometry objects. They all are built from model data, they all occupy a portion of space, and they are all transformable. The objects House, Room 1, and Room 2 are grouping nodes. We could easily make all these Spatial objects, but not Geometry objects. In this scenario, the Spatial class must contain information to represent the hierarchy of objects. Specifically, each object must have a link to its parent object (if any) and links to its child objects. The links shown in Figure 3.2 represent both the parent and child links.

The concepts of grouping and of representing geometric data are effectively disjoint. If Spatial objects were allowed child objects, then by derivation so would Geometry objects. Thus, Geometry objects would have double duty, as representations of geometric data and as nodes for grouping related objects. The interface for a Geometry class that supports grouping as well as geometric queries will be quite complicated, making it difficult to understand all the behavior that objects from the class can exhibit. I prefer instead a *separation of concerns* regarding these matters. The interfaces associated with Geometry and its derived classes should address only the semantics related to geometric objects, their visual appearances, and physical properties. The grouping responsibilities are delegated instead to a separate class, in this case the class Node. The interfaces associated with Node and its derived classes address only the semantics related to the subtrees associated with the nodes. By separating the responsibilities, it is easier for the engine designer and architect to maintain and extend the separate types of objects (geometry types or node types).

My choice for separation of concerns leads to class Spatial storing the parent link in the hierarchy and to class Node storing the child links in the hierarchy. Class Node derives from Spatial, so in fact the Node objects have both parent and child links. Class Geometry also derives from Spatial, but geometry objects can only occur as leaf nodes in the hierarchy. This is the main consequence of the separation of concerns. The price one pays for having the separation and a clean division of responsibilities is that the hierarchy as shown in Figure 3.2 is not realizable in this scheme. Instead the hierarchy may be structured as shown in Figure 3.3.

Two grouping nodes were added. The Table Group node was added because the Table is a geometric object and cannot be an interior node of the tree. The utensils (Plate, Knife, Fork) were children of the Table. To preserve this structure, the Utensil Group node was added to group the utensils together. To maintain the transformation structure of the original hierarchy, the Table Group is assigned the transformations the Table had, the Table is assigned the identity transformation, and the Utensil Group is assigned the identity transformation. This guarantees that the

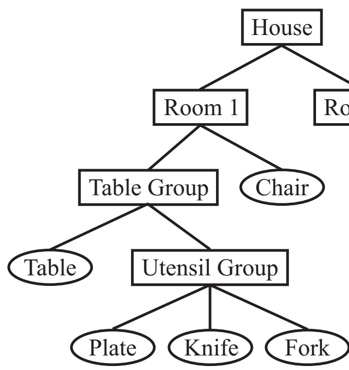


Figure 3.3 The new hierarchy corresponding to the one in Figure 3.2 when geometric objects can be only leaf nodes. Ellipses are used to denote geometric objects. Rectangles are used to denote grouping nodes.

Utensil Group is in the same coordinate system that the Table is in. Consequently, the utensils may be positioned and oriented using the same transformations that were used in the hierarchy of Figure 3.2.

Alternatively, you can avoid the Utensil Group node and just make the utensils siblings of the Table. If you do this, the coordinate system of the utensils is now that of the Table Group. The transformations of the utensils must be changed to ones relative to the coordinate system of the Table Group.

The portion of the interface for class `Spatial` relevant to the scene hierarchy connections is

```
class Spatial : public Object
{
public:
    virtual ~Spatial ();
    Spatial* GetParent ();

protected:
    Spatial ();
    Spatial* m_pkParent;

// internal use
public:
    void SetParent (Spatial* pkParent);
};
```

The default constructor is protected, making the class an abstract base class. The default constructor is implemented to support the streaming system. The class is derived from the root class `Object`, as are nearly all the classes in the engine. All of the root services are therefore available to `Spatial`, including run-time type information, sharing, streaming, and so on.

The parent pointer is protected, but read access is provided by the public interface function `GetParent`. Write access of the parent pointer is provided by the public interface function `SetParent`. That block of code is listed at the end of the class. My intention on the organization is that the public interface intended for the application writers is listed first in the class declaration. The public interface at the end of the class is tagged with the comment "internal use." The issue is that `SetParent` is called by the `Node` class when a `Spatial` object is attached as the child of a node. No other class (or application) should call `SetParent`. If the method were put in the protected section to prevent unintended use, then `Node` cannot call the function. To circumvent this problem, `Node` can be made a friend of `Spatial`, thus allowing it access to `SetParent`, but disallowing anyone else to access it. In some circumstances, a `Node`-derived class might also need access to a protected member of `Spatial`. In the C++ language, friendship is not inherited, so making `Node` a friend of `Spatial` will not make the `Node`-derived class a friend of `Spatial`. To avoid the somewhat frequent addition of friend declarations to classes to allow restricted access to protected members, I decided to use the system of placing the restricted access members in public scope, but tagging that block with the "internal use" comment to let programmers know that they should *not* use those functions.

The portion of the interface for class `Node` relevant to the scene hierarchy connections is

```
class Node : public Spatial
{
public:
    Node (int iQuantity = 1, int iGrowBy = 1);
    virtual ~Node ();

    int GetQuantity () const;
    int GetUsed () const;
    int AttachChild (Spatial* pkChild);
    int DetachChild (Spatial* pkChild);
    SpatialPtr DetachChildAt (int i);
    SpatialPtr SetChild (int i, Spatial* pkChild);
    SpatialPtr GetChild (int i);

protected:
    TArray<SpatialPtr> m_kChild;
    int m_iUsed;
};
```

The links to the child nodes are stored as an array of `Spatial` smart pointers. Clearly, the pointers cannot be `Node` pointers because the leaf nodes of the hierarchy are `Spatial`-derived objects (such as `Geometry`), but not `Node`-derived objects. The nonnull child pointers do not have to be contiguous in the array, so where the children are placed is up to the programmer. The data member `m_iUsed` indicates how many of the array slots are occupied by nonnull pointers.

The constructor allows you to specify the initial quantity of children the node will have. The array is dynamic; that is, even if you specify the node to have a certain number of children initially, you may attach more children than that number. The second parameter of the constructor indicates how much storage increase occurs when the array is full and an attempt to attach another child occurs.

The `AttachChild` function searches the pointer array for the first available empty slot and stores the child pointer in it. If no such slot exists, the child pointer is stored at the end of the array, dynamically resizing the array if necessary. This is an important feature to remember. For whatever reason, if you detach a child from a slot internal to the array and you do not want the next child to be stored in that slot, you must use the `SetChild` function because it lets you specify the exact location for the new child. The return value of `AttachChild` is the index into the array where the attached child is stored. The return value of `SetChild` is the child that was in the ith slot of the array before the new child was stored there. If you choose not to hang onto the return value, it is a smart pointer, in which case the reference count on the object is decremented. If the reference count goes to zero, the child is automatically destroyed.

Function `DetachChild` lets you specify the child, by pointer, to be detached. The return value is the index of the slot that stored the child. The vacated slot has its pointer set to `NULL`. Function `DetachChildAt` lets you specify the child, by index, to be detached. The return value is that child. As with `SetChild`, if you choose not to hang onto the return value, the reference count on the object is decremented and, if zero, the object is destroyed.

Function `GetChild` simply returns a smart pointer to the current child in the specified slot. This function is what you use when you iterate over an array of children and process them in some manner—typically something that occurs during a recursive traversal of a scene graph.

3.1.3 Instancing

The spatial hierarchy system is a *tree structure*; that is, each tree node has a single parent, except for a root node that has no parent. You may think of the spatial hierarchy as the skeleton for the scene graph. A scene graph really is an abstract graph because the object system supports sharing. If an object is shared by two other objects, effectively there are two *instances* of the first object. The act of sharing the objects is called *instancing*. I do not allow instancing of nodes in a spatial hierarchy, and this

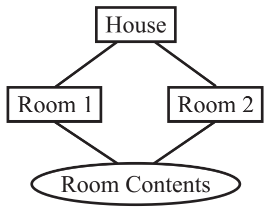


Figure 3.4 A scene graph corresponding to a house and two rooms. The rooms share the same geometric model data, called Room Contents.

is enforced by allowing a `Spatial` object to have only a single parent link. Multiple parents are not possible.[3] One of the questions I am occasionally asked is why I made this choice.

For the sake of argument, suppose that a hierarchy node is allowed to have multiple parents. A simple example is shown in Figure 3.4. The scene graph represents a house with two rooms. The rooms share the same geometric model data. The two rooms may be thought of as instances of the same model data. The implied structure is a directed acyclic graph (DAG). The house node has two directed arcs to the room nodes. Each room node has a directed arc to the room contents leaf node. The room contents are therefore shared. Reasons to share include reducing memory usage for the game application and reducing your artist's workload in having to create distinct models for everything you can imagine in the world. The hope is that the user is not terribly distracted by the repetition of like objects as he navigates through the game world.

What are some of the implications of Figure 3.4? The motivation for a spatial hierarchy was to allow for positioning and orienting of objects via local transformations. The locality is important so that generation of content can be done independently of the final coordinate system of the world (the coordinate system of the root node of the scene). A path through the hierarchy from root to leaf has a corresponding sequence of local transformations whose product is the world transformation for the leaf node. The problem in Figure 3.4 is that the leaf node may be reached via *two paths* through the hierarchy. Each path corresponds to an instance of the leaf object. Realize that the two rooms are placed in different parts of the house. The world transformations applied to the room contents are necessarily different. If you have any plans to make the world transformations persistent, they must be stored *somewhere*. In the tree-based hierarchy, the world transformations are stored directly at the node. To

3. *Predecessors* might be a better term to use here, but I will use the term *parents* and note that the links are directed from parent to child.

store the world transformations for the DAG of Figure 3.4, you can store them either at each node or in a separate location that the node has pointers to. In either case, a dynamic system is required since the number of parents can be any number and change at any time. World bounding volumes must also be maintained, one per instance.

Another implication is that if you want to change the data directly at the shared node, the room contents in our example, it is necessary for you to be able to specify *which instance is to be affected*. This alone creates a complex situation for an application programmer to manage. You may assign a set of names to the shared object, one name per path to the object. The path names can be arbitrarily long, making the use of them a bit overwhelming for the application programmer. Alternatively, you can require that a shared object not be directly accessible. The instances must be managed only through the parent nodes. In our example, to place Room 1 in the house, you set its local transformations accordingly. Room 2 is placed in the world with a different set of local transformations. The Room Contents always have the identity transformation, never to be changed. This decision has the consequence that if you only have a single instance (most likely the common case in a scene), a parent node should be used to indirectly access that instance. If you are not consistent in the manner of accessing the object, your engine logic must distinguish between a single instance of an object and multiple instances of an object, then handle the situations differently. Thus, every geometric object must be manipulated as a node-geometry pair. Worse is that if you plan on instancing a subgraph of nodes, that subgraph must have parent nodes through which you access the instances. Clearly this leads to "node bloat" (for lack of a better term), and the performance of updating such a system is not optimal for real-time needs.

Is this speculation or experience? The latter, for sure. One of the first tasks I was assigned when working on NetImmerse in its infancy was to support instancing in the manner described here. Each node stored a dynamic array of parent links and a dynamic array of child links. A corresponding dynamic array of geometric data was also maintained that stored transformations, bounding volumes, and other relevant information. Instances were manipulated through parent nodes, with some access allowed to the instances themselves. On a downward traversal of the scene by a recursive function, the parent pointer was passed to that function and used as a lookup in the child's parent array to determine which instance the function was to affect. This mechanism addresses the issue discussed earlier, unique names for the paths to the instance. Unfortunately, the system was complicated to build and complicated to maintain (adding new recursive functions for scene traversal was tedious), and the parent pointer lookup was a noticeable time sink, as shown by profiling any applications built on top of the engine. To eliminate the cost of parent pointer lookups, the node class was modified to include an array of instance pointers, one per child of the node. Those pointers were passed through recursive calls, thus avoiding the lookups, and used directly. Of course, this increased the per-node memory requirements and increased the complexity

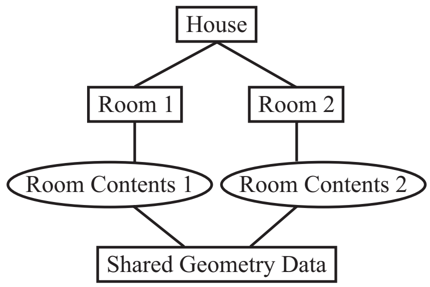


Figure 3.5 The scene graph of Figure 3.4, but with instancing at a low level (geometric data) rather than at a node level.

of the system. In the end we decided that supporting instancing by DAGs was not acceptable.

That said, instancing still needs to be supported in an engine. I mentioned this earlier and mention it again: What is important regarding instancing is that (1) you reduce memory usage and (2) you reduce the artist's workload. The majority of memory consumption has to do with models with *large* amounts of data. For example, a model with 10,000 vertices, multiple 32-bit texture images, each 512 × 512, and corresponding texture coordinates consumes a lot of memory. Instancing such a model will avoid duplication of the large data. The amount of memory that a node or geometry object requires to support core scene graph systems is quite small relative to the actual model data. If a subgraph of nodes is to be instanced, duplication of the nodes requires only a small amount of additional memory. The model data is shared, of course. Wild Magic 3 chooses to share in the manner described here. The sharing is *low level*; that is, instancing of models involves geometric data. If you want to instance an object of a `Geometry`-derived class, you create two unique `Geometry`-derived objects, but ask them to share their vertices, texture coordinates, texture images, and so on. The DAG of Figure 3.4 abstractly becomes the graph shown in Figure 3.5.

The work for creating an instance is more than what a DAG-style system requires, but the run-time performance is much improved and the system complexity is minimal.

3.2 Geometric State

Two basic objects involving geometric state are transformations and bounding volumes.

3.2.1 Transformations

Wild Magic version 2 supported transformations involving translations $\mathbf{T}$, rotations R, and uniform scaling $\sigma > 0$. A vector $\mathbf{X}$ is transformed to a vector $\mathbf{Y}$ by

$$\mathbf{Y} = R(\sigma\mathbf{X}) + \mathbf{T}. \tag{3.1}$$

The order of application is scale first, rotation second, and translation third. However, the order of uniform scaling and rotation is irrelevant. The inverse transformation is

$$\mathbf{X} = \frac{1}{\sigma}R^{\mathrm{T}}(\mathbf{Y} - \mathbf{T}). \tag{3.2}$$

Generally, a graphics API allows for any affine transformation, in particular, nonuniform scaling. The natural extension of Equation (3.1) to allow nonuniform scale $S = \mathrm{Diag}(\sigma_0, \sigma_1, \sigma_2)$, $\sigma_i > 0$, for all i, is

$$\mathbf{Y} = RS\mathbf{X} + \mathbf{T}. \tag{3.3}$$

The order of application is scale first, rotation second, and translation third. In this case the order of nonuniform scaling and rotation is relevant. Switching the order produces different results since, in most cases, $RS \neq SR$. The inverse transformation is

$$\mathbf{X} = S^{-1}R^{\mathrm{T}}(\mathbf{Y} - \mathbf{T}), \tag{3.4}$$

where $S^{-1} = \mathrm{Diag}(1/\sigma_0, 1/\sigma_1, 1/\sigma_2)$. The memory requirements to support nonuniform scaling are modest—only two additional floating-point numbers to store.

Wild Magic version 2 disallowed nonuniform scaling because of some undesirable consequences. First, a goal was to minimize the time spent on matrix and vector arithmetic. This was particularly important when an application has a physical simulation that makes heavy use of the transformation system. Using operation counts as a measure of execution time,[4] let μ represent the number of cycles for a multiplication, let α represent the number of cycles for an addition/subtraction, and let δ represent the number of cycles for a division. On an Intel Pentium class processor, μ and α are equal, both 3 cycles. The value δ is 39 cycles. Both Equations (3.1) and (3.3) use $12\mu + 9\alpha$ cycles to transform a single vector. Equation (3.2) uses $12\mu + 9\alpha + \delta$ cycles. The only difference between the inverse transform and the forward transform is the division required to compute the reciprocal of scale. The reciprocal is computed first and then multiplies the three components of the vector. Equation (3.4) uses $9\mu + 9\alpha + 3\delta$ cycles. Compared to the uniform scale inversion, the reciprocals are

4. A warning about operation counting: Current-day processors have other issues now that can make operation counting not an accurate measure of performance. You need to pay attention to memory fetches, cache misses, branch penalties, and other architectural aspects.

not computed first. The three vector components are divided directly by the nonuniform scales, leading to three less multiplications, but two more divisions. This is still a significant increase in cost because of the occurrence of the additional divisions. The divisions may be avoided by instead computing $p = \sigma_0\sigma_1\sigma_2$, $r = 1/p$, and observing that $S^{-1} = r\ \mathrm{Diag}(\sigma_1\sigma_2, \sigma_0\sigma_2, \sigma_0\sigma_1)$. Equation (3.3) then uses $19\mu + 9\alpha + \delta$ cycles, replacing two divisions by 10 multiplications. If the CPU supports a faster but lower-precision division, the increase is not as much of a factor, but you pay in terms of accuracy of the final result. With the advent of specialized hardware such as extended instructions for CPUs, game console hardware, and vector units generally, the performance for nonuniform scaling is not really a concern.

Second, an issue that is mathematical and that hardware cannot eliminate is the requirement to factor transformations to maintain the ability to store at each node the scales, the rotation matrix, and the translation vector. To be precise, if you have a path of nodes in a hierarchy and corresponding local transformations, the world transformation is a composition of the local ones. Let the local transformations be represented as homogeneous matrices in block-matrix form. The transformation $\mathbf{Y} = RS\mathbf{X} + \mathbf{T}$ is represented by

$$\left[\begin{array}{c} \mathbf{Y} \\ \hline 1 \end{array}\right] \left[\begin{array}{c|c} RS & \mathbf{T} \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right] \left[\begin{array}{c} \mathbf{X} \\ \hline 1 \end{array}\right].$$

The composition of two local transformations $\mathbf{Y} = R_1S_1\mathbf{X} + \mathbf{T}_1$ and $\mathbf{Z} = R_2S_2\mathbf{Y} + \mathbf{T}_2$ is represented by a homogeneous block matrix that is a product of the two homogeneous block matrices representing the individual transformations:

$$\left[\begin{array}{c|c} R_2S_2 & \mathbf{T}_2 \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right] \left[\begin{array}{c|c} R_1S_1 & \mathbf{T}_1 \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right] = \left[\begin{array}{c|c} R_2S_2R_1S_1 & R_2S_2\mathbf{T}_1 + \mathbf{T}_2 \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right] = \left[\begin{array}{c|c} M & \mathbf{T} \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right],$$

where $M = R_2S_2R_1S_1$ and $\mathbf{T} = R_2S_2\mathbf{T}_1 + \mathbf{T}_2$. A standard question that is asked somewhat regularly in the Usenet computer graphics newsgroups is how to factor

$$M = RS,$$

where R is a rotation and S is a diagonal nonuniform scaling matrix. The idea is to have a transformation class that always stores R, S, and $\mathbf{T}$ as individual components, thus allowing direct evaluation of Equations (3.3) and (3.4). Much to the posters' dismay, the unwanted answer is, You cannot always factor M in this way. In fact, it is not always possible to factor D_1R_1 into R_2D_2, where D_1 and D_2 are diagonal matrices and R_1 and R_2 are rotation matrices.

The best you can do is factor M using *polar decomposition* or *singular value decomposition* ([Hec94, Section III.4]). The polar decomposition is

$$M = UA,$$

where U is an orthogonal matrix and A is a symmetric matrix, but not necessarily diagonal. The singular value decomposition is closely related:

$$M = VDW^{\mathrm{T}},$$

where V and W are orthogonal matrices and D is a diagonal matrix. The two factorizations are related by appealing to the eigendecomposition of a symmetric matrix, $A = WDW^{\mathrm{T}}$, where W is orthogonal and D is diagonal. The columns of W are linearly independent eigenvectors of A, and the diagonal elements of D are the eigenvalues (ordered to correspond to the columns of W). It follows that $V = UW$. Implementing either factorization is challenging because the required mathematical machinery is more than what you might expect.

Had I chosen to support nonuniform scaling in Wild Magic *and* wanted a consistent representation of local and world transformations, the factorization issue prevents me from storing a transformation as a triple $(R, S, \mathbf{T})$, where R is a rotation, S is a diagonal matrix of scales, and $\mathbf{T}$ is a translation. One way out of the dilemma is to use a triple for local transformations, but a pair $(M, \mathbf{T})$ for world transformations. The 3×3 matrix M is the composition of rotations and nonuniform scales through a path in the hierarchy. The memory usage for a world transformation is smaller than for a local one, but only one floating-point number less. The cost for a forward transformation $\mathbf{Y} = M\mathbf{X} + \mathbf{T}$ is $9\mu + 9\alpha$, cheaper than for a local transformation. Less memory usage, faster transformation, but the cost is that you have no scaling or rotational information for the world transformation unless you factor into polar form or use the singular value decomposition. Both factorizations are very expensive to compute. The inverse tranformation $\mathbf{X} = M^{-1}(\mathbf{Y} - \mathbf{T})$ operation count is slightly more complicated to determine. Using a cofactor expansion to compute the inverse matrix,

$$M^{-1} = \frac{1}{\det(M)} M^{\mathrm{adj}},$$

where $\det(M)$ is the determinant of M and M^{adj} is the adjoint matrix—the transpose of the matrix of cofactors of M. The adjoint has nine entries, each requiring $2\mu + \alpha$ cycles to compute. The determinant is computed from a row of cofactors, using three more multiplications and two more additions, for a total of $3\mu + 2\alpha$ cycles. The reciprocal of the determinant uses δ cycles. Computing the inverse transformation as

$$\mathbf{X} = \frac{1}{\det(M)} \left(M^{\mathrm{adj}}(\mathbf{Y} - \mathbf{T}) \right)$$

requires $33\mu + 20\alpha + \delta$ cycles. This is a very significant increase in cost compared to the $19\mu + 9\alpha + \delta$ cycles used for computing $\mathbf{X} = S^{-1}R^{\mathrm{T}}(\mathbf{Y} - \mathbf{T})$.

To avoid the increase in cost for matrix inversion, you could alternatively choose a consistent representation where the transformations are stored as 4-tuples of the form $(L, S, R, \mathbf{T})$, where L and R are rotation matrices, S is a diagonal matrix of scales, and $\mathbf{T}$ is a translation. Once a world transformation is computed as a composition of local transformations to obtain M and $\mathbf{T}$, you have to factor $M = LDR$ using the singular value decomposition—yet another expensive proposition.

Given the discussion of nonuniform scaling and the performance issues arising from factorization and/or maintaining a consistent representation for transformations, in Wild Magic version 2 I decided to constrain the transformations to use only uniform scaling. I have relaxed the constraint slightly in Wild Magic version 3. The `Spatial` class stores three scale factors, but only the `Geometry` class may set these to be nonuniform. But doesn't this introduce all the problems that I just mentioned? Along a path of n nodes, the last node being a geometry leaf node, the world transformation is a composition of $n-1$ local transformations that have only uniform scale σ_i, $i \geq 2$, and a final local transformation that has nonuniform scales S_1:

$$
\begin{aligned}
\left[\begin{array}{c|c} R_n\sigma_n & \mathbf{T}_n \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right] \cdots \left[\begin{array}{c|c} R_2\sigma_2 & \mathbf{T}_2 \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right] \left[\begin{array}{c|c} R_1 S_1 & \mathbf{T}_1 \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right]
&= \left[\begin{array}{c|c} R'\sigma' & \mathbf{T}' \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right] \left[\begin{array}{c|c} R_1 S_1 & \mathbf{T}_1 \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right] \\
&= \left[\begin{array}{c|c} (R'R_1)(\sigma' S_1) & R'\sigma'\mathbf{T}_1 + \mathbf{T}' \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right] \\
&= \left[\begin{array}{c|c} R''S'' & \mathbf{T}'' \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right].
\end{aligned}
$$

Because of the commutativity of uniform scale and rotation, the product of the first $n-1$ matrices leads to another matrix of the same form, as shown. The product with the last matrix groups together the rotations and groups together the scales. The final form of the composition is one that does not require a general matrix inverse calculation. I consider the decision to support nonuniform scales only in the `Geometry` class an acceptable compromise between having only uniform scales or having nonuniform scales available at all nodes.

The class that encapsulates the transformations containing translations, rotations, and nonuniform scales is `Transformation`. The default constructor, destructor, and data members are shown next in a partial listing of the class:

```
class Transformation
{
public:
    Transformation ();
    ~Transformation ();

    static const Transformation IDENTITY;
```

```
private:
    Matrix3f m_kRotate;
    Vector3f m_kTranslate;
    Vector3f m_kScale;
    bool m_bIsIdentity, m_bIsUniformScale;
};
```

In a moment I will discuss the public interface to the data members. The rotation matrix is stored as a 3×3 matrix. The user is responsible for ensuring that the matrix really is a rotation. The three scale factors are stored as a 3-tuple, but they could just as easily have been stored as three separate floating-point numbers. The class has two additional data members, both Boolean variables. These are considered *hints* to allow for more efficient composition of transformations. The default constructor creates the identity transformation, where the rotation is the 3×3 identity matrix, the translation is the 3×1 zero vector, and the three scales are all one. The `m_bIsIdentity` and `m_bIsUniformScale` hints are both set to `true`. For an application's convenience, the static class member `IDENTITY` stores the identity transformation.

Part of the public interface to access the members is

```
class Transformation
{
public:
    void SetRotate (const Matrix3f& rkRotate);
    const Matrix3f& GetRotate () const;
    void SetTranslate (const Vector3f& rkTranslate);
    const Vector3f& GetTranslate () const;
    void SetScale (const Vector3f& rkScale);
    const Vector3f& GetScale () const;
    void SetUniformScale (float fScale);
    float GetUniformScale () const;
};
```

The `Set` functions have side effects in that each function sets the `m_bIsIdentity` hint to `false`. The hint is set, even if the final transformation is the identity. For example, calling `SetTranslate` with the zero vector as input will set the hint to `false`. I made this choice to avoid having to check if the transformation is really the identity after each component is set. The expected case is that the use of `Set` functions is to make the transformation something *other* than the identity. Even if we were to test for the identity transformation, the test is problematic when floating-point arithmetic is used. An exact comparison of floating-point values is not robust when some of the values were computed in expressions, the end results of which were produced after a small amount of floating-point round-off error. The `SetScale` function also has the side effect of setting the `m_bIsUniformScale` hint to `false`. As before, the hint is set even if the input scale vector corresponds to uniform scaling. The `Get` functions have no side effects and return the requested components. These functions are `const`, so the components are read-only.

Three other public member access functions are provided:

```
class Transformation
{
public:
    Matrix3f& Rotate ();
    Vector3f& Translate ();
    Vector3f& Scale ();
};
```

My convention is to omit the `Set` or `Get` prefixes on member accessors when I intend the accessor to provide read-write access. The displayed member functions are read-write, but also have the side effects of setting the `m_bIsIdentity` and/or the `m_bIsUniformScale` hints. Because the accessor cannot determine if it was called for read versus write, the hints are always set. You should avoid this style of accessor if your intent is only to read the member value, in which case you should use the `Get` version. A typical situation to use the read-write accessor is for updates that require both, for example,

```
Transformation kXFrm = <some transformation>;
kXFrm.Translate() += Vector3f(1.0f,2.0f,3.0f);
```

or for in-place calculations, for example,

```
Transformation kXFrm = <some transformation>;
kXFrm.Rotate().FromAxisAngle(Vector3f::UNIT_Z,Mathf::HALF_PI);
```

In both cases, the members are written, so setting the hints is an appropriate action to take.

Two remaining public accessors are for convenience:

```
class Transformation
{
public:
    float GetMinimumScale () const;
    float GetMaximumScale () const;
};
```

The names are clear. The first returns the smallest scale from the three scaling factors, and the second returns the largest. An example of where I use the maximum scale is in computing a world bounding sphere from a model bounding sphere and a transformation with nonuniform scaling. The exact transformation of the model bounding sphere is an ellipsoid, but since I really wanted a bounding sphere, I use the maximum scale as a uniform scale factor and apply a uniform scale transformation to the model bounding sphere.

Other convenience functions include the ability to tell a transformation to make itself the identity transformation or to make its scales all one:

```
class Transformation
{
public:
    void MakeIdentity ();
    void MakeUnitScale ();
    bool IsIdentity () const;
    bool IsUniformScale () const;
};
```

The last two functions just return the current values of the hints.

The basic algebraic operations for transformations include application of a transformation to points, application of an inverse transformation to points, and composition of two transformations. The member functions are

```
class Transformation
{
public:
    Vector3f ApplyForward (const Vector3f& rkInput) const;
    void ApplyForward (int iQuantity, const Vector3f* akInput,
        Vector3f* akOutput) const;

    Vector3f ApplyInverse (const Vector3f& rkInput) const;
    void ApplyInverse (int iQuantity, const Vector3f* akInput,
        Vector3f* akOutput) const;

    void Product (const Transformation& rkA,
        const Transformation& rkB,);

    void Inverse (Transformation& rkInverse);
};
```

The first `ApplyForward` and `ApplyInverse` functions apply to single vectors. The second pair of these functions apply to arrays of vectors. If the transformation is $\mathbf{Y} = RS\mathbf{X} + \mathbf{T}$, where R is a rotation matrix, S is a diagonal scale matrix, and $\mathbf{T}$ is a translation, function `ApplyForward` computes $\mathbf{Y}$ from the input vector(s) $\mathbf{X}$. Function `ApplyInverse` computes $\mathbf{X} = S^{-1}R^{\mathrm{T}}(\mathbf{Y} - \mathbf{T})$ from the input vector(s) $\mathbf{Y}$.

The composition of two transformations is performed by the member function `Product`. The name refers to a product of matrices when the transformations are viewed as 4×4 homogeneous matrices. For example,

```
Transformation kA = <some transformation>;
Transformation kB = <some transformation>;
Transformation kC;

// compute C = A*B
kC.Product(kA,kB);

// compute C = B*A, generally not the same as A*B
kC.Product(kB,kA);
```

We will also need to apply inverse transformations to vectors. Notice that I earlier used both the term *points* and the term *vectors*. The two are abstractly different, as discussed in the study of *affine algebra*. A point $\mathbf{P}$ is transformed as

$$\mathbf{P}' = RS\mathbf{P} + \mathbf{T},$$

whereas a vector $\mathbf{V}$ is transformed as

$$\mathbf{V}' = RS\mathbf{V}.$$

You can think of the latter equation as the difference of the equations for two transformed points $\mathbf{P}$ and $\mathbf{Q}$:

$$\begin{aligned}
\mathbf{V} &= \mathbf{P} - \mathbf{Q} \\
\mathbf{P}' &= RS\mathbf{P} + \mathbf{T} \\
\mathbf{Q}' &= RS\mathbf{Q} + \mathbf{T} \\
\mathbf{V}' &= \mathbf{P}' - \mathbf{Q}' = (RS\mathbf{P} + \mathbf{T}) - (RS\mathbf{Q} + \mathbf{T}) = RS(\mathbf{P} - \mathbf{Q}) = RS\mathbf{V}.
\end{aligned}$$

In terms of homogeneous vectors, the point $\mathbf{P}$ and vector $\mathbf{V}$ are represented by

$$\left[\frac{\mathbf{P}}{1}\right] \quad \text{and} \quad \left[\frac{\mathbf{V}}{0}\right].$$

The corresponding homogeneous transformations are

$$\left[\begin{array}{c|c} RS & \mathbf{T} \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right]\left[\begin{array}{c} \mathbf{P} \\ \hline 1 \end{array}\right] = \left[\begin{array}{c} RS\mathbf{P} + \mathbf{T} \\ \hline 1 \end{array}\right] = \left[\begin{array}{c} \mathbf{P}' \\ \hline 1 \end{array}\right]$$

and

$$\left[\begin{array}{c|c} RS & \mathbf{T} \\ \hline \mathbf{0}^{\mathrm{T}} & 1 \end{array}\right]\left[\begin{array}{c} \mathbf{V} \\ \hline 0 \end{array}\right] = \left[\begin{array}{c} RS\mathbf{V} \\ \hline 0 \end{array}\right] = \left[\begin{array}{c} \mathbf{V}' \\ \hline 0 \end{array}\right].$$

The inverse transformation of a vector $\mathbf{V}'$ is

$$\mathbf{V} = S^{-1}R^{\mathrm{T}}\mathbf{V}'.$$

The member function that supports this operation is

```
class Transformation
{
public:
    Vector3f InvertVector (const Vector3f& rkInput) const;
};
```

Finally, the inverse of the transformation is computed by

```
void Inverse (Transformation& rkInverse);
```

The translation, rotation, and scale components are computed. If $\mathbf{Y} = RS\mathbf{X} + \mathbf{T}$, the inverse is $\mathbf{X} = S^{-1}R^{\mathrm{T}}(\mathbf{Y} - \mathbf{T})$. The inverse transformation has scale S^{-1}, rotation R^{T}, and translation $-S^{-1}R^{\mathrm{T}}\mathbf{T}$. A warning is in order, though. The components are stored in the class data members, but the transformation you provide to the function *should not* be used as a regular `Transformation`. If you were to use it as such, it would represent

$$R^{\mathrm{T}}S^{-1}\mathbf{X} - S^{-1}R^{\mathrm{T}}\mathbf{T}.$$

Only call this function, access the individual components, and then discard the object.

The transformation of a plane from model space to world space is also sometimes necessary. Let the model space plane be

$$\mathbf{N}_0 \cdot \mathbf{X} = c_0,$$

where $\mathbf{N}_0$ is a unit-length normal vector, c_0 is a constant, and $\mathbf{X}$ is any point on the plane and is specified in model space coordinates. The inverse transformation of the point is $\mathbf{X} = S^{-1}R^{\mathrm{T}}(\mathbf{Y} - \mathbf{T})$, where $\mathbf{Y}$ is the point in world space coordinates. Substituting this in the plane equation leads to

$$\mathbf{N}_1 \cdot \mathbf{Y} = c_1, \qquad \mathbf{N}_1 = \frac{RS^{-1}\mathbf{N}_0}{|RS^{-1}\mathbf{N}_0|}, \qquad c_1 = \frac{c_0}{|RS^{-1}\mathbf{N}_0|} + \mathbf{N}_1 \cdot \mathbf{T}.$$

The member function that supports this operation is

```
class Transformation
{
public:
    Plane3f ApplyForward (const Plane3f& rkInput) const;
};
```

The input plane has normal $\mathbf{N}_0$ and constant c_0. The output plane has normal $\mathbf{N}_1$ and constant c_1.

In all the transformation code, I take advantage of the `m_bIsIdentity` and `m_bIsUniformScale` hints. Two prototypical cases are the implementation of `Apply-Forward` that maps $\mathbf{Y} = RS\mathbf{X} + \mathbf{T}$ and the implementation of `ApplyInverse` that maps $\mathbf{X} = S^{-1}R^{\mathrm{T}}(\mathbf{Y} - \mathbf{T})$. The forward transformation implementation is

```
Vector3f Transformation::ApplyForward (
    const Vector3f& rkInput) const
{
    if ( m_bIsIdentity )
        return rkInput;

    Vector3f kOutput = rkInput;
    kOutput.X() *= m_kScale.X();
    kOutput.Y() *= m_kScale.Y();
    kOutput.Z() *= m_kScale.Z();
    kOutput = m_kRotate*kOutput;
    kOutput += m_kTranslate;
    return kOutput;
}
```

If the transformation is the identity, then $\mathbf{Y} = \mathbf{X}$ and the output vector is simply the input vector. A generic implementation might do all the matrix and vector operations anyway, not noticing that the transformation is the identity. The hint flag helps avoid those unnecessary calculations. If the transformation is not the identity, it does not matter whether the scale is uniform or nonuniform since three multiplications by a scale parameter occur in either case.

The inverse transformation implementation is

```
Vector3f Transformation::ApplyInverse (
    const Vector3f& rkInput) const
{
    if ( m_bIsIdentity )
        return rkInput;
```

```
    if ( m_bIsUniformScale )
    {
        return ((rkInput - m_kTranslate)*m_kRotate) /
            GetUniformScale();
    }

    Vector3f kOutput = ((rkInput - m_kTranslate)*m_kRotate);
    float fSXY = m_kScale.X()*m_kScale.Y();
    float fSXZ = m_kScale.X()*m_kScale.Z();
    float fSYZ = m_kScale.Y()*m_kScale.Z();
    float fInvDet = 1.0f/(fSXY*m_kScale.Z());
    kOutput.X() *= fInvDet*fSYZ;
    kOutput.Y() *= fInvDet*fSXZ;
    kOutput.Z() *= fInvDet*fSXY;
    return kOutput;
}
```

If the transformation is the identity, then $\mathbf{X} = \mathbf{Y}$ and there is no reason to waste cycles by applying the transformation components. Unlike `ApplyForward`, if the transformation is not the identity, then there is a difference in performance between uniform and nonuniform scaling.

For uniform scale, $R^{\mathrm{T}}(\mathbf{Y} - \mathbf{T})$ has all three components divided by scale. The `Matrix3` class has an operator function such that a product of a vector (the left operand $\mathbf{V}$) and a matrix (the right operand M) corresponds to $M^{\mathrm{T}}\mathbf{V}$. The previous displayed code block uses this function. The `Vector3` class supports division of a vector by a scalar. Internally, the reciprocal of the divisor is computed and multiplies the three vector components. This avoids the division occurring three times, replacing the operation instead with a single division and three multiplications.

For nonuniform scale, I use the trick described earlier for avoiding three divisions. The displayed code replaces the three divisions by 10 multiplications and one division. For an Intel Pentium that uses 3 cycles per multiplication and 39 cycles per division, the three divisions would cost 78 cycles, but the 10 multiplications and one division costs 69 cycles.

3.2.2 Bounding Volumes

The term *bounding volume* is quite generic and refers to any object that contains some other object. The simplest bounding volumes that game programmers use tend to be spheres or axis-aligned bounding boxes. Slightly more complicated is an oriented bounding box. Yet more complicated is the convex hull of the contained object, a convex polyhedron. In all cases, the bounding volumes are convex. To be yet more complicated, a bounding volume might be constructed as a union of (convex) bounding volumes.

Culling

One major use for bounding volumes in an engine is for the purposes of *culling* objects. If an object is completely outside the view frustum, there is no reason to tell the renderer to try and draw it because if the renderer made the attempt, it would find that all triangles in the meshes that represent the object are outside the view frustum. Such a determination does take some time—better to avoid wasting cycles on this, if possible. The scene graph management system could itself determine if the mesh triangles are outside the view frustum, testing them one at a time for intersection with, or containment by, the view frustum, but this gains us nothing. In fact, this is potentially slower when the renderer has a specialized GPU to make the determination, but the scene graph system must rely on a general CPU.

A less aggressive approach is to use a convex bounding volume as an approximation to the region of space that the object occupies. If the bounding volume is outside the view frustum, then so is the object and we need not ask the renderer to draw it. The intersection/containment test between bounding volume and view frustum is hopefully a lot less expensive to compute than the intersection/containment tests for all the triangles of the object. If the bounding volume is a sphere, the test for the sphere being outside the view frustum is equivalent to computing the distance from the sphere center to the view frustum and showing that it is larger than the radius of the sphere.

Computing the distance from a point to a view frustum is more complicated than most game programmers care to deal with—hence the replacement of that test with an *inexact* query that is simpler to implement. Specifically, the sphere is tested against each of the six frustum planes. The frustum plane normals are designed to point into the view frustum; that is, the frustum is on the "positive side" of all the planes. If the sphere is outside any of these planes, say, on the "negative side" of a plane, then the sphere is outside the entire frustum and the object is not visible and therefore not sent to the renderer for drawing (it is culled). I call this *plane-at-a-time culling*. The geometry query I refer to as the *which-side-of-plane query.* There are situations when the sphere is not outside one of the planes, but is outside the view frustum; that is why I used earlier the adjective "inexact." Figure 3.6 shows the situation in two dimensions.

The sphere in the upper right of the image is not outside any of the frustum planes, but is outside the view frustum. The plane-at-a-time culling system determines that the sphere is not outside any plane, and the object associated with the bounding volume is sent to the renderer for drawing. The same idea works for convex bounding volumes other than spheres. Pseudocode for the general inexact culling is

```
bool IsCulled (ViewFrustum frustum, BoundingVolume bound)
{
    for each plane of frustum do
    {
        if bound is on the negative side of plane then
```

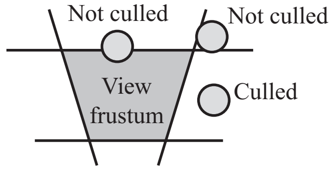


Figure 3.6 A two-dimensional view of various configurations between a bounding sphere and a view frustum.

```
            return true;
    }
    return false;
}
```

Hopefully the occurrence of false positives (bound outside frustum, but not outside all frustum planes) is infrequent.

Even though plane-at-a-time culling is inexact, it may be used to improve efficiency in visibility determination in a scene graph. Consider the scene graph of Figure 3.3, where each node in the tree has a bounding volume associated with it. Suppose that, when testing the bounding volume of the Table Group against the view frustum, you find that the bounding volume is on the positive side of one of the view frustum planes. The collective object represented by Table Group is necessarily on the positive side of that plane. Moreover, the objects represented by the children of Table Group must also be on the positive side of the plane. We may take advantage of this knowledge and pass enough information to the children (during a traversal of the tree for drawing purposes) to let the culling system know not to test the child bounding volumes against that same plane. In our example, the Table and Utensil Group nodes do not have to compare their bounding volumes to that plane of the frustum. The information to be stored is as simple as a bit array, each bit corresponding to a plane. In my implementation, discussed in more detail later in this chapter, the bits are set to 1 if the plane should be compared with the bounding volumes, and 0 otherwise.

An argument I read about somewhat regularly in some Usenet newsgroups is that complicated bounding volumes should be avoided because the which-side-of-plane query for the bounding volume is expensive. The recommendation is to use something as simple as a sphere because the query is very inexpensive to compute compared to, say, an oriented bounding box. Yes, a true statement, but it is taken out of the context of the bigger picture. There is a balance between the complexity of the bounding volume type and the cost of the which-side-of-plane query. As a rule of thumb, the more complex the bounding volume of the object, the better fitting it is to the object, but the query is more expensive to compute. Also as a rule of thumb,

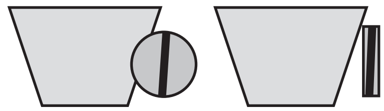


Figure 3.7 A situation where a better-fitting bounding volume leads to culling, but a worse-fitting one does not. (a) The bounding sphere is not tight enough to induce culling. (b) The bounding box is tight enough to induce culling.

the better fitting the bounding volume, the more likely it is to be culled compared to a worse-fitting bounding volume. Figure 3.7 shows a typical scenario.

Even though the cost for the which-side-of-plane query is more expensive for the box than for the sphere, the combined cost of the query for the sphere *and* the attempt to draw the object, only to find out it is not visible, is *larger* than the cost of the query for the box. The latter object has no rendering cost because it was culled.

On the other hand, if most of the objects are typically inside the frustum, in which case you get the combined cost of the query and drawing, the sphere bounding volumes look more attractive. Whether or not the better-fitting and more expensive bounding volumes are beneficial depends on your specific 3D environment. To be completely certain of which way to go, allow for different bounding volume types and profile your applications for each type to see if there is any savings in time for the better-fitting volumes. The default bounding volume type in Wild Magic is a bounding sphere; however, the system is designed to allow you to easily swap in another type without having to change the engine or the application code. This is accomplished by providing an abstract interface (base class) for bounding volumes. I discuss this a bit later in the section.

Collision Determination

Another major use for bounding volumes is *3D picking*. A picking ray in world coordinates is selected by some mechanism. A list of objects that are intersected by the ray can be assembled. As a coarse-level test, if the ray does not intersect the bounding volume of an object, then it does not intersect the object.

The bounding volumes also support *collision determination*. More precisely, they may be used to determine if two objects are *not intersecting*, much in the same way they are used to determine if an object is not visible. Collision detection for two arbitrary triangle meshes is an expensive endeavor. We use a bounding volume as an approximation to the region of space that the object occupies. If the bounding volumes of two objects do not intersect, then the objects do not intersect. The hope is that the test for intersection of two bounding volumes is much less expensive than the test for intersection of two triangle meshes. Well, it is, unless the objects themselves are single triangles!

The discussion of how to proceed with picking after you find out that the ray intersects a bounding volume or how you proceed with collision detection after you find out that the bounding volumes intersect is deferred to Section 6.3.3.

The Abstract Bounding Volume Interface

My main goal in having an abstract interface was not to force the engine users to use my default, bounding spheres. I also wanted to make sure that it was very easy to make the change, one that did not require changes to the core engine components or the applications themselves. The abstraction forces one to think about the various geometric queries in object-independent ways. Although abstract interfaces tend not to have data associated with them, experience led me to conclude that a minimal amount of information is needed. At the lowest level, you need to know *where* a bounding volume is located and what its *size* is. The two data members that represent these are a center point and a radius. These values already define a sphere, so you may think of the base class as a representation of a bounding sphere for the bounding volume. The values for an oriented bounding box are naturally the box center and the maximum distance from the center to a vertex. The values for a convex polyhedron may be selected as the average of the vertices and the maximum distance from that average to any vertex. Other types of bounding volumes can define center and radius similarly.

The abstract class is `BoundingVolume` and has the following initial skeleton:

```
class BoundingVolume : public Object
public:
{
    virtual ~BoundingVolume ();

    Vector3f Center;
    float Radius;

    static BoundingVolume* Create ();

protected:
    BoundingVolume ();
    BoundingVolume (const Vector3f& rkCenter, float fRadius);
};
```

The constructors are protected, making the class abstract. However, most other member functions in the interface are pure virtual, so the class would have to be abstract anyway, despite the access level for the constructors. The center point and radius are in public scope since setting or getting them has no side effects. The static member function `Create` is used as a factory to produce objects without having to know what specific type (or types) exist in the engine. A derived class has the responsibility for implementing this function, and only one derived class may do so. In the engine, the `Create` call occurs during construction of a `Spatial` object (the world bounding volume) and a `Geometry` object (the model bounding volume). A couple of additional calls occur in `Geometry`-derived classes, but only because the construction of the model bounding volume is deferred until the actual model data is known by those classes.

Even though only a single derived class implements `Create`, you may have multiple `BoundingVolume`-derived classes in the engine. The ones not implementing `Create` must be constructed explicitly. Only the core engine components for geometric updates must be ignorant of the type of bounding volume.

Switching to a new `BoundingVolume` type for the core engine is quite easy. All you need to do is comment out the implementation of `BoundingVolume::Create` in the default bounding volume class, `SphereBV`, and implement it in your own derived class. The `SphereBV` class is

```
BoundingVolume* BoundingVolume::Create ()
{
    return new SphereBV;
}
```

If you were to switch to `BoxBV`, the oriented bound box volumes, then in `Wm3BoxBV.cpp` you would place

```
BoundingVolume* BoundingVolume::Create ()
{
    return new BoxBV;
}
```

The remaining interface for `BoundingVolume` is shown next. All member functions are pure virtual, so the derived classes must implement these.

```
class BoundingVolume : public Object
public:
{
    virtual void ComputeFromData (
        const Vector3fArray* pkVertices) = 0;

    virtual void TransformBy (const Transformation& rkTransform,
        BoundingVolume* pkResult) = 0;

    virtual int WhichSide (const Plane3f& rkPlane) const = 0;
```

```
    virtual bool TestIntersection (const Vector3f& rkOrigin,
        const Vector3f& rkDirection) const = 0;

    virtual bool TestIntersection (
        const BoundingVolume* pkInput) const = 0;

    virtual void CopyFrom (const BoundingVolume* pkInput) = 0;

    virtual void GrowToContain (const BoundingVolume* pkInput) = 0;
};
```

The bounding volume depends, of course, on the vertex data that defines the object. The `ComputeFromData` method provides the construction of the bounding volume from the vertices.

The transformation of a model space bounding volume to one in world space is supported by the method `TransformBy`. The first input is the model-to-world transformation, and the second input is the world space bounding volume. That volume is computed by the method and is valid on return from the function. The `Geometry` class makes use of this function.

The method `WhichSide` supports the which-side-of-plane query that was discussed for culling of nonvisible objects. The `Plane3` class stores unit-length normal vectors, so the `BoundingVolume`-derived classes may take advantage of that fact to implement the query. If the bounding volume is fully on the positive side of the plane (the side to which the normal points), the function returns $+1$. If it is fully on the negative side, the function returns -1. If it straddles the plane, the function returns 0.

The first `TestIntersection` method supports 3D picking. The input is the origin and direction vector for a ray that is in the same coordinate system as the bounding volume. The direction vector must be unit length. The return value is `true` if and only if the ray intersects the bounding volume. The second `TestIntersection` method supports collision determination. The input bounding volume must be the same type as the calling object, but the engine does not check this constraint, so you must. The bounding volumes are assumed to be stationary. The return value of the function is `true` if and only if the two bounding volumes are intersecting.

The last two member functions, `CopyFrom` and `GrowToContain`, support the upward pass through the scene graph that computes the bounding volume of a parent node from the bounding volumes of the child nodes. In Wild Magic, the parent bounding volume is constructed to contain all the child bounding volumes. The default bounding volume is a sphere, so the parent bounding volume is a sphere that contains all the spheres of the children. The function `CopyFrom` makes the calling object a copy of the input bounding volume. The function `GrowToContain` constructs the bounding volume of the calling bounding volume and the input bounding volume. For a node with multiple children, `CopyFrom` makes a copy of the first child, and `GrowToContain` creates a bounding volume that contains that copy and the bounding volume of the second child. The resulting bounding volume is grown further to contain each of the remaining children.

A brief warning about having a bounding volume stored in `Spatial` through an abstract base class (smart) pointer: Nothing prevents you from setting the bounding volume of one object to be a sphere and another to be a box. However, the `BoundingVolume` member functions that take a `BoundingVolume` object as input are designed to manipulate the input as if it is the same type as the calling object. Mixing bounding volume types is therefore an error, and the engine has no prevention mechanism for this. You, the programmer, must enforce the constraint. That said, it is possible to extend the bounding volume system to handle mixed types. The amount of code for n object types can be inordinate. For intersection queries between two bounding volumes, you need a function for each pair of types, a total of $n(n-1)/2$ functions. The semantics of the `CopyFrom` function must change. How do you copy a bounding sphere to an oriented bounding box? The semantics of `GrowToContain` must also change. What is the type of bounding volume to be used for a collection of mixed bounding volume types? If you have a sphere and a box, should the containing volume be a sphere or a box? Such a system can be built (NetImmerse had one), but I chose to limit the complexity of Wild Magic by disallowing mixing of bounding volume types.

3.2.3 The Core Classes and Geometric Updates

Recall that the scene graph management core classes are `Spatial`, `Geometry`, and `Node`. The `Spatial` class encapsulates the local and world transformations, the world bounding volume, and the parent pointer in support of the scene hierarchy. The `Geometry` class encapsulates the model data and the model bounding sphere and may exist only as leaf nodes in the scene hierarchy. The `Node` class encapsulates grouping and has a list of child pointers. All three classes participate in the *geometric update* of a scene hierarchy—the process of propagating transformations from parents to children (the downward pass) and then merging bounding volumes from children to parents (the upward pass).

The data members of the `Spatial` interface relevant to geometric updates are shown in the following partial interface listing:

```
class Spatial : public Object
{
public:
    Transformation Local;
    Transformation World;
    bool WorldIsCurrent;

    BoundingVolumePtr WorldBound;
    bool WorldBoundIsCurrent;
};
```

The data members are in public scope. This is a deviation from my choices for Wild Magic version 2, where the data members were protected or private and exposed only through public accessor functions, most of them implemented as inline functions. My choice for version 3 is to reduce the verbosity, so to speak, of the class interface. In earlier versions, you would have a protected or private data member, one or more accessors, and inline implementations of those accessors. For example,

```
// in OldSpatial.h
class OldSpatial : public Object
{
public:
    Transformation& Local ();                  // read-write access
    const Transformation& GetLocal () const;   // read-only access
    void SetLocal (const Transform& rkLocal);  // write-only access
protected:
    Transformation m_kLocal;
};

// in OldSpatial.inl
Transformation& OldSpatial::Local ()
    { return m_kLocal; }
const Transformation& OldSpatial::GetLocal () const
    { return m_kLocal; }
void OldSpatial::SetLocal (const Transformation& rkLocal)
    { m_kLocal = rkLocal; }
```

The object-oriented premise of such an interface is to allow the underlying implementation of the class to change without forcing clients of the class to have to change their code. This is an example of *modular continuity*; see [Mey88, Section 2.1.4], specifically the following paragraph:

> A design method satisfies Modular Continuity if a small change in a problem specification results in a change of just one module, or few modules, in the system obtained from the specification through the method. Such changes should not affect the *architecture* of the system, that is to say the relations between modules.

The interface for `OldSpatial` is a conservative way to achieve modular continuity. The experiences of two versions of Wild Magic have led me to conclude that exposing some data members in the public interface is acceptable as long as the subsystem involving those data members is stable; that is, the subsystem will not change as the engine evolves. This is a less conservative way to achieve modular continuity because it relies on you not to change the subsystem.

By exposing data members in the public interface, you have another issue of concern. The function interfaces to data members, as shown in `OldSpatial`, can hide side effects. For example, the function `SetLocal` has the responsibility of setting the `m_kLocal` data member of the class. But it could also perform operations on other data members or call other member functions, thus causing changes to state elsewhere in the system. If set/get function calls require side effects, it is not recommended that you expose the data member in the public interface. For if you were to do so, the engine user would have the responsibility for doing whatever is necessary to make those side effects occur.

In the case of the `Spatial` class in version 3 of the engine, the `Local` data member is in public scope. Setting or getting the value has no side effects. The new interface is

```
// in Spatial.h
class Spatial : public Object
{
public:
    Transformation Local;  // read-write access
};
```

and is clearly much reduced from that of `OldSpatial`. Observe that the prefix convention for variables is now used only for protected or private members. The convention for public data members is not to use prefixes and to capitalize the first letter of the name, just like function names are handled.

In class `Spatial` the world transformation is also in public scope. Recalling the previous discussion about transformations, the world transformations are compositions of local transformations. In this sense, a world transformation is computed as a (deferred) side effect of setting local transformations. I just mentioned that exposing data members in the public interface is not a good idea when side effects must occur, so why already violate that design goal? The problem has to do with the complexity of the controller system. Some controllers might naturally be constructed to *directly set the world transformations*. Indeed, the engine has a skin-and-bones controller that computes the world transformation for a triangle mesh. In a sense, the controller bypasses the standard mechanism that computes world transformations from local ones. The data members `World` and `WorldIsCurrent` are intended for read access by application writers, but may be used for write access by controllers. If a controller sets the `World` member directly, it should also set the `WorldIsCurrent` flag to let the geometric update system know that the world transformation for this node should not be computed as a composition of its parent's world transformation and its local transformation.

Similar arguments apply to the data members `WorldBound` and `WorldBoundIsCurrent`. In some situations you have a node (and subtree) whose behavior is known to you (by design), and whose world bounding volume may be assigned directly. For example, the node might be a room in a building that never moves. The child nodes correspond to objects in the room; those objects can move within the room, so their world bounding volumes change. However, the room's world bounding volume need not change. You may set the room's world bounding volume, but the geometric update system should be told not to recalculate that bounding volume from the child bounding volumes. The flag `WorldBoundIsCurrent` should be set to `true` in this case.

The member functions of `Spatial` relevant to geometric updates are shown in the following partial interface listing:

```
class Spatial : public Object
{
public:
    void UpdateGS (double dAppTime = -Mathd::MAX_REAL,
        bool bInitiator = true);
    void UpdateBS ();

protected:
    virtual void UpdateWorldData (double dAppTime);
    virtual void UpdateWorldBound () = 0;
    void PropagateBoundToRoot ();
};
```

The public functions `UpdateGS` ("update geometric state") and `UpdateBS` ("update bound state") are the entry points to the geometric update system. The function `UpdateGS` is for both propagation of transformations from parents to children and propagation of world bounding volumes from children to parents. The `dAppTime` ("application time") is passed so that any animated quantities needing the current time to update their state have access to it. The Boolean parameter will be explained later. The function `UpdateBS` is for propagation only of world bounding volumes. The protected function `UpdateWorldData` supports the propagation of transformations in the downward pass. It is virtual to allow derived classes to update any additional world data that is affected by the change in world transformations. The protected functions `UpdateWorldBound` and `PropagateToRoot` support the calculation of world bounding volumes in the upward pass. The `UpdateWorldBound` function is pure virtual to require `Geometry` and `Node` to implement it as needed.

The portion of the `Geometry` interface relevant to geometric updates is

```
class Geometry : public Spatial
{
public:
    Vector3fArrayPtr Vertices;
    Vector3fArrayPtr Normals;
    BoundingVolumePtr ModelBound;
    IntArrayPtr Indices;

    void UpdateMS ();
```

```
protected:
    virtual void UpdateModelBound ();
    virtual void UpdateModelNormals ();
    virtual void UpdateWorldBound ();
};
```

As with the `Spatial` class, the data members are in public scope because there are no immediate side effects from reading or writing them. But there are side effects that the programmer must ensure, namely, the geometric update itself.

The function `UpdateMS` ("update model state") is the entry point into the update of the model bound and model normals. The function should be called whenever you change the model vertices. All that `UpdateMS` does is call the protected functions `UpdateModelBound` and `UpdateModelNormals`. The function `UpdateModelBound` computes a model bounding volume from the collection of vertices. This is accomplished by a call to the `BoundingVolume` function `ComputeFromData`. I made the model bound update a virtual function just in case a derived class needs to compute the bound differently. For example, a derived class might have prior knowledge about the model bound and not even have to process the vertices.

The function `UpdateModelNormals` has an empty body in `Geometry` since the geometry class is just a container for vertices and normals. Derived classes need to implement `UpdateModelNormals` for their specific data representations. Not all derived classes have normals (for example, `Polypoint` and `Polyline`), so I decided to let them use the empty base class function rather than making the base function pure virtual and then requiring derived classes to implement it with empty functions.

The function `UpdateWorldBound` is an implementation of the pure virtual function in `Spatial`. All that it does is compute the world bounding volume from the model bounding volume by applying the current world transformation.

The member functions of `Node` relevant to geometric updates are shown in the following partial interface listing:

```
class Node : public Spatial
{
protected:
    virtual void UpdateWorldData (double dAppTime);
    virtual void UpdateWorldBound ();
};
```

The function `UpdateWorldData` is an implementation of the virtual function in the `Spatial` base class. It has the responsibility to propagate the geometric update to its children. The function `UpdateWorldBound` is an implementation of the pure virtual function in the `Spatial` base class. Whereas the `Geometry` class implements this to calculate a single world bounding volume for its data, the `Node` class implements this to compute a world bounding volume that contains the world bounding volume of all its children.

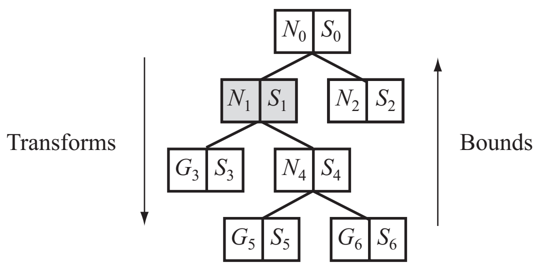


Figure 3.8 A geometric update of a simple scene graph. The light gray shaded node, N_1, is the one at which the `UpdateGS` call initiates.

Figure 3.8 illustrates the behavior of the update. The symbols are N for `Node`, S for `Spatial`, and G for `Geometry`. The rectangular boxes represent the nodes in the scene hierarchy. The occurrence of both an N and an S at a node stresses the fact the `Node` is derived from `Spatial`, so both classes' public and protected interfaces are available to `Node`. A similar statement is made for `Geometry` and `Spatial`.

If the model bounding volumes or the model normals for a `Geometry` object are not current, that object must call `Geometry::UpdateMS()` to make them current. In most cases, the model data is current—for example, in rigid triangle meshes; you will not call `UpdateMS` often for such objects. The other extreme is something like a morph controller that changes the vertex data frequently, and the `UpdateMS` call occurs after each change.

Assuming the model data is current at all leaf nodes, the shaded gray box in the figure indicates that node N_1 is the one initiating a geometric update because its local transformation was changed (translation, rotation, and/or uniform scale). Its world transformation must be recomputed from its parent's (N_0) world transformation and its newly changed local transformation. The new world transformation is passed to its two children, G_3 and N_4, so that they also may recompute their world transformations. The world bounding volume for G_3 must be recomputed from its model bounding volume. The process is repeated at node N_4. Its world transformation is recomputed from the world transformation of N_1 and its local transformation. The new world transformation is passed to its two children, G_5 and G_6, so that they may recompute their world transformations. Those leaf nodes also recompute their world bounding volumes from their current model bounding volumes and their new world transformations. On return to the parent N_4, that node must recompute its world bounding volume to contain the new world bounding volumes of its children. On return to the node N_1, that node must recompute its world bounding volume to contain the new world bounding volumes for G_3 and N_4. You might think the geometric update terminates at this time, but not yet. The change in world bounding volume at N_1 can cause the world bounding volume of its parent, N_0, to be out of date. N_0 must be told to update itself. Generally, the change in world bounding volume at the initiator of the update must propagate all the way to the root of the scene hierarchy. Now the geometric update is complete. The sequence of operations is listed as pseudocode in the following. The indentation denotes the level of the recursive call of `UpdateGS`.

```
double dAppTime = <current application time>;
N1.UpdateGS(appTime,true);
  N1.World = compose(N0.World,N1.Local);
  G3.UpdateGS(appTime,false);
    G3.World = Compose(N1.World,G3.Local);
    G3.WorldBound = Transform(G3.World,G3.ModelBound);
  N4.UpdateGS(appTime,false);
    N4.World = Compose(N1.World,N4.Local);
    G5.UpdateGS(appTime,false);
      G5.World = Compose(N4.World,G5.Local);
      G5.WorldBound = Transform(G5.World,G5.ModelBound);
    G6.UpdateGS(appTime,false);
      G6.World = Compose(N4.World,G6.Local);
      G6.WorldBound = Transform(G6.World,G6.ModelBound);
    N4.WorldBound = BoundContaining(G5.WorldBound,G6.WorldBound);
  N1.WorldBound = BoundContaining(G3.WorldBound,N4.WorldBound);
  N0.WorldBound = BoundContaining(N1.WorldBound,N2.WorldBound);
```

The Boolean parameter `bInitiator` in the function `UpdateGS` is quite important. In the example, the `UpdateGS` call initiated at N_1. A depth-first traversal of the subtree rooted at N_4 is performed, and the transformations are propagated downward. Once you reach a leaf node, the new world bounding volume is propagated upward. When the last child of N_1 has been visited, we found we needed to propagate its world bounding volume to its predecessors all the way to the root of the scene, in the example to N_0. The propagation of a world bounding volume from G_5 to N_4 is slightly different than the propagation of a world bounding volume from N_1 to N_0. The depth-first traversal at N_1 guarantees that the world bounding volumes are processed on the upward return. You certainly would not want each node to propagate its world bounding volume all the way to the root whenever that node is visited in the traversal because only the initiator has that responsibility. If you were to have missed that subtlety and not had a Boolean parameter, the previous pseudocode would become

```
double dAppTime = <current application time>;
N1.UpdateGS(appTime);
  N1.World = compose(N0.World,N1.Local);
  G3.UpdateGS(appTime);
    G3.World = Compose(N1.World,G3.Local);
    G3.WorldBound = Transform(G3.World,G3.ModelBound);
```

```
    N1.WorldBound = BoundContaining(G3.WorldBound,N4.WorldBound);
    N0.WorldBound = BoundContaining(N1.WorldBound,N2.WorldBound);
  N4.UpdateGS(appTime);
    N4.World = Compose(N1.World,N4.Local);
    G5.UpdateGS(appTime);
      G5.World = Compose(N4.World,G5.Local);
      G5.WorldBound = Transform(G5.World,G5.ModelBound);
      N4.WorldBound = BoundContaining(G5.WorldBound,G6.WorldBound);
      N1.WorldBound = BoundContaining(G3.WorldBound,N4.WorldBound);
      N0.WorldBound = BoundContaining(N1.WorldBound,N2.WorldBound);
    G6.UpdateGS(appTime);
      G6.World = Compose(N4.World,G6.Local);
      G6.WorldBound = Transform(G6.World,G6.ModelBound);
      N4.WorldBound = BoundContaining(G5.WorldBound,G6.WorldBound);
      N1.WorldBound = BoundContaining(G3.WorldBound,N4.WorldBound);
      N0.WorldBound = BoundContaining(N1.WorldBound,N2.WorldBound);
    N4.WorldBound = BoundContaining(G5.WorldBound,G6.WorldBound);
    N1.WorldBound = BoundContaining(G3.WorldBound,N4.WorldBound);
    N0.WorldBound = BoundContaining(N1.WorldBound,N2.WorldBound);
  N1.WorldBound = BoundContaining(G3.WorldBound,N4.WorldBound);
  N0.WorldBound = BoundContaining(N1.WorldBound,N2.WorldBound);
```

Clearly, this is an inefficient chunk of code. The Boolean parameter is used to prevent subtree nodes from propagating the world bounding volumes to the root.

The actual update code is shown next because I want to make a few comments about it. The entry point for the geometric update is

```
void Spatial::UpdateGS (double dAppTime, bool bInitiator)
{
    UpdateWorldData(dAppTime);
    UpdateWorldBound();
    if ( bInitiator )
        PropagateBoundToRoot();
}
```

If the object is a `Node` object, the function `UpdateWorldData` propagates the transformations in the downward pass. If the object is a `Geometry` object, the function is not implemented in that class, and the `Spatial` version is used. The two different functions are

```
void Node::UpdateWorldData (double dAppTime)
{
    Spatial::UpdateWorldData(dAppTime);
```

```
    for (int i = 0; i < m_kChild.GetQuantity(); i++)
    {
        Spatial* pkChild = m_kChild[i];
        if ( pkChild )
            pkChild->UpdateGS(dAppTime,false);
    }
}

void Spatial::UpdateWorldData (double dAppTime)
{
    UpdateControllers(dAppTime);

    // NOTE:  Updates on controllers for global state and lights
    // go here.  To be discussed later.

    if ( !WorldIsCurrent )
    {
        if ( m_pkParent )
            World.Product(m_pkParent->World,Local);
        else
            World = Local;
    }
}
```

The `Spatial` version of the function has the responsibility for computing the composition of the parent's world transformation and the object's local transformation, producing the object's world transformation. At the root of the scene (`m_pkParent` is `NULL`), the local and world transformations are the same. If a controller is used to compute the world transformation, then the Boolean flag `WorldIsCurrent` is `true` and the composition block is skipped. The `Node` version of the function allows the base class to compute the world transformation, and then it propagates the call (recursively) to its children. Observe that the `bInitiator` flag is set to `false` for the child calls to prevent them from propagating the world bounding volumes to the root node.

The controller updates might or might not affect the transformation system. For example, the point, particles, and morph controllers all modify the model space vertices (and possibly the model space normals). Each of these calls `UpdateMS` to guarantee the model bounding volume is current. Fortunately this step occurs before our `UpdateGS` gets to the stage of updating world bounding volumes. Keyframe and inverse kinematics controllers modify local transformations, but they do not set the `WorldIsCurrent` flag to `true` because the world transformations must still be updated. The skin controllers modify the world transformations directly and do set the `WorldIsCurrent` flag to `true`.

In `UpdateGS`, on return from `UpdateWorldData` the world bounding volume is updated by `UpdateWorldBound`. If the object is a `Node` object, a bound of bounds is computed. If the object is a Geometry object, the newly computed world transformation is used to transform the model bounding volume to the world bounding volume.

```
void Node::UpdateWorldBound ()
{
    if ( !WorldBoundIsCurrent )
    {
        bool bFoundFirstBound = false;
        for (int i = 0; i < m_kChild.GetQuantity(); i++)
        {
            Spatial* pkChild = m_kChild[i];
            if ( pkChild )
            {
                if ( bFoundFirstBound )
                {
                    // Merge current world bound with child
                    // world bound.
                    WorldBound->GrowToContain(pkChild->WorldBound);
                }
                else
                {
                    // Set world bound to first nonnull child
                    // world bound.
                    bFoundFirstBound = true;
                    WorldBound->CopyFrom(pkChild->WorldBound);
                }
            }
        }
    }
}

void Geometry::UpdateWorldBound ()
{
    ModelBound->TransformBy(World,WorldBound);
}
```

If the application has explicitly set the world bounding volume for the node, it should have also set WorldBoundIsCurrent to false, in which case Node::UpdateWorldBound has no work to do. However, if the node must update its world bounding volume, it does so by processing its child bounding volumes one at a time. The bounding volume of the first (nonnull) child is copied. If a second (nonnull) child exists, the current world bounding volume is modified to contain itself and the bound of the child. The growing algorithm continues until all children have been visited.

For bounding spheres, the iterative growing algorithm amounts to computing the smallest volume of two spheres, the current one and that of the next child. This is a greedy algorithm and does not generally produce the smallest volume bounding sphere that contains all the child bounding spheres. The algorithm to compute the smallest volume sphere containing a set of spheres is a very complicated beast [FG03]. The computation time is not amenable to real-time graphics, so instead we use a less exact bound, but one that can be computed quickly.

The last stage of `UpdateGS` is to propagate the world bounding volume from the initiator to the root. The function that does this is `PropagateBoundToRoot`. This, too, is a recursive function, just through a linear list of nodes:

```
void Spatial::PropagateBoundToRoot ()
{
    if ( m_pkParent )
    {
        m_pkParent->UpdateWorldBound();
        m_pkParent->PropagateBoundToRoot();
    }
}
```

As mentioned previously, if a local transformation has not changed at a node, but some geometric operations cause the world bounding volume to change, there is no reason to waste time propagating transformations in a downward traversal of the tree. Instead just call `UpdateBS` to propagate the world bounding volume to the root:

```
void Spatial::UpdateBS ()
{
    UpdateWorldBound();
    PropagateBoundToRoot();
}
```

Table 3.1 is a summary of the updates that must occur when various geometric quantities change in the system. All of the updates may be viewed as side effects to changes in the geometric state of the system. None of the side effects occur automatically because I want application writers to use as much of their knowledge as possible about their environment and not force an inefficient update mechanism to occur behind the scenes.

For example, Figure 3.9 shows a scene hierarchy that needs updating. The light gray shaded nodes in the scene have had their local transformations changed. You could blindly call

```
a.UpdateGS(appTime,true);
b.UpdateGS(appTime,true);
c.UpdateGS(appTime,true);
d.UpdateGS(appTime,true);
```

Table 3.1 Updates that must occur when geometric quantities change.

Changing quantity	*Required updates*	*Top-level function to call*
Model data	Model bound, model normals (if any)	`Geometry::UpdateMS`
Model bound	World bound	`Spatial::UpdateGS` or `Spatial::UpdateBS`
World bound	Parent world bound (if any)	`Spatial::UpdateGS` or `Spatial::UpdateBS`
Local transformation	World transformation, child transformations	`Spatial::UpdateGS`
World transformation	World bound	`Spatial::UpdateGS`

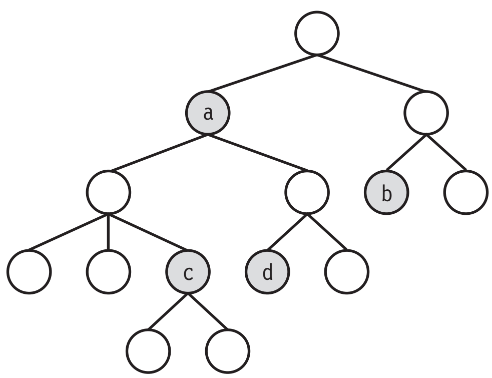


Figure 3.9 A scene hierarchy that needs updating. The light gray shaded nodes have had their local transformations changed.

to perform the updates, but this is not efficient. All that is needed is

```
a.UpdateGS(appTime,true);
b.UpdateGS(appTime,true);
```

Nodes `c` and `d` are updated as a side effect of the update at node `a`. In general, the minimum number of `UpdateGS` calls needed is the number of nodes requiring an update that have no predecessors who also require an update. Node `a` requires an update, but has no out-of-date predecessors. Node `c` requires an update, but it has a predecessor, node `a`, that does. Although it is possible to construct an automated system to determine the minimum number of `UpdateGS` calls, that system will consume too many cycles. I believe it is better to let the application writers take advantage of knowledge they have about what is out of date and specifically call `UpdateGS` themselves.

3.3 Geometric Types

The basic geometric types supported in the engine are collections of points, collections of line segments, triangle meshes, and particles. Various classes in the core engine implement these types. During the drawing pass through the scene graph, the renderer is provided with such objects and must draw them as their types dictate. Most graphics APIs require the type of object to be specified, usually via a set of enumerated values. To facilitate this, the `Geometry` class has enumerations for the basic types, as shown in the following code snippet:

```
class Geometry : public Spatial
{
    // internal use
    public:
        enum // GeometryType
        {
            GT_POLYPOINT,
            GT_POLYLINE_SEGMENTS,
            GT_POLYLINE_OPEN,
            GT_POLYLINE_CLOSED,
            GT_TRIMESH,
            GT_MAX_QUANTITY
        };

        int GeometryType;
};
```

The type itself is stored in the data member `GeometryType`. It is in public scope because there are no side effects in reading or writing it. However, the block is marked for internal use by the engine. There is no need for an application writer to manipulate the type.

The value `GT_POLYPOINT` indicates the object is a collection of points. The value `GT_TRIMESH` indicates the object is a triangle mesh. The three values with `POLYLINE` as part of their names are used for collections of line segments. `GT_POLYLINE_SEGMENTS` is for a set of line segments with no connections between them. `GT_POLYLINE_OPEN` is for a polyline, a set of line segments where each segment end point is shared by at most two lines. The initial and final segments each have an end point that is not shared by any other line segment; thus the polyline is said to be *open*. Another term for an open polyline is a *line strip*. If the two end points are actually the same point, then the polyline forms a loop and is said to be *closed*. Another term for a closed polyline is a *line loop*.

If you were to modify the engine to support other types that are native to the graphics APIs, you can add enumerated types to the list. You should add these after GT_TRIMESH, but before GT_MAX_QUANTITY, in order to preserve the numeric values of the current types.

3.3.1 Points

A collection of points is represented by the class Polypoint, which is derived from Geometry. The interface is very simple:

```
class Polypoint : public Geometry
{
public:
    Polypoint (Vector3fArrayPtr spkVertices);
    virtual ~Polypoint ();

    void SetActiveQuantity (int iActiveQuantity);
    int GetActiveQuantity () const;

protected:
    Polypoint ();

    int m_iActiveQuantity;
};
```

The points are provided to the constructor. From the application's perspective, the set of points is unordered. However, for the graphics APIs that use vertex arrays, I have chosen to assign indices to the points. The vertices and indices are both used for drawing. The public constructor is

```
Polypoint::Polypoint (Vector3fArrayPtr spkVertices)
    :
    Geometry(spkVertices)
{
    GeometryType = GT_POLYPOINT;

    int iVQuantity = Vertices->GetQuantity();
    m_iActiveQuantity = iVQuantity;

    int* aiIndex = new int[iVQuantity];
```

```
    for (int i = 0; i < iVQuantity; i++)
        aiIndex[i] = i;
    Indices = new IntArray(iVQuantity,aiIndex);
}
```

The assigned indices are the natural ones.

The use of an index array has a pleasant consequence. Normally, all of the points would be drawn by the renderer. In some applications you might want to have storage for a large collection of points, but only have a subset *active* at one time. The class has a data member, `m_iActiveQuantity`, that indicates how many are active. The active quantity may be zero, but cannot be larger than the total quantity of points. The active set is contiguous in the array, starting at index zero, but if need be, an application can move the points from one vertex array location to another.

The active quantity data member is not in the public interface. The function `SetActiveQuantity` has the side effect of validating the requested quantity. If the input quantity is invalid, the active quantity is set to the total quantity of points.

The index array `Indices` is a data member in the base class `Geometry`. Its type is `TSharedArray<int>`. This array is used by the renderer for drawing purposes. Part of that process involves querying the array for the number of elements. The shared array class has a member function, `GetQuantity`, that returns the *total* number of elements in the array. However, we want it to report the active quantity when the object to be drawn is of type `Polypoint`. To support this, the shared array class has a member function `SetActiveQuantity` that changes the internally stored total quantity to the requested quantity. The requested quantity must be no larger than the original total quantity. If it is not, no reallocation occurs in the shared array, and any attempt to write elements outside the original array is an access violation.

Rather than adding a new data member to `TSharedArray` to store an active quantity, allowing the total quantity to be stored at the same time, I made the decision that the caller of `SetActiveQuantity` must remember the original total quantity, in case the original value must be restored through another call to `SetActiveQuantity`. My decision is based on the observation that calls to `SetActiveQuantity` will be infrequent, so I wanted to minimize the memory usage for the data members of `TSharedArray`.

As in all `Object`-derived classes, a default constructor is provided for the purposes of streaming. The constructor is protected to prevent the application from creating default objects whose data members have not been initialized with real data.

3.3.2 Line Segments

A collection of line segments is represented by the class `Polyline`, which is derived from `Geometry`. The interface is

```
class Polyline : public Geometry
{
public:
    Polyline (Vector3fArrayPtr spkVertices, bool bClosed,
        bool bContiguous);
    virtual ~Polyline ();

    void SetActiveQuantity (int iActiveQuantity);
    int GetActiveQuantity () const;
    void SetClosed (bool bClosed);
    bool GetClosed () const;
    void SetContiguous (bool bContiguous);
    bool GetContiguous () const;

protected:
    Polyline ();
    void SetGeometryType ();

    int m_iActiveQuantity;
    bool m_bClosed, m_bContiguous;
};
```

The end points of the line segments are provided to the constructor. The three possible interpretations for the vertices are disjoint segments, open polyline, or closed polyline. The input parameters `bClosed` and `bContiguous` determine which interpretation is used. The inputs are stored as class members `m_bClosed` and `m_bContiguous`. The actual interpretation is implemented in `SetGeometryType`:

```
void Polyline::SetGeometryType ()
{
    if ( m_bContiguous )
    {
        if ( m_bClosed )
            GeometryType = GT_POLYLINE_CLOSED;
        else
            GeometryType = GT_POLYLINE_OPEN;
    }
    else
    {
        GeometryType = GT_POLYLINE_SEGMENTS;
    }
}
```

To be a polyline where end points are shared, the contiguous flag must be set to `true`. The closed flag has the obvious interpretation.

Let the points be $\mathbf{P}_i$ for $0 \le i < n$. If the contiguous flag is `false`, the object is a collection of disjoint segments. For a properly formed collection, the quantity of vertices n should be even. The $n/2$ segments are

$$\langle \mathbf{P}_0, \mathbf{P}_1 \rangle, \quad \langle \mathbf{P}_2, \mathbf{P}_3 \rangle, \quad \ldots, \quad \langle \mathbf{P}_{n-2}, \mathbf{P}_{n-1} \rangle.$$

If the contiguous flag is `true` and the closed flag is `false`, the points represent an open polyline with $n - 1$ segments:

$$\langle \mathbf{P}_0, \mathbf{P}_1 \rangle, \quad \langle \mathbf{P}_1, \mathbf{P}_2 \rangle, \quad \ldots, \quad \langle \mathbf{P}_{n-2}, \mathbf{P}_{n-1} \rangle.$$

The end point $\mathbf{P}_0$ of the initial segment and the end point $\mathbf{P}_{n-1}$ of the final segment are not shared by any other segments. If instead the closed flag is `true`, the points represent a closed polyline with n segments:

$$\langle \mathbf{P}_0, \mathbf{P}_1 \rangle, \quad \langle \mathbf{P}_1, \mathbf{P}_2 \rangle, \quad \ldots, \quad \langle \mathbf{P}_{n-2}, \mathbf{P}_{n-1} \rangle, \quad \langle \mathbf{P}_{n-1}, \mathbf{P}_0 \rangle.$$

Each point is shared by exactly two segments. Although you might imagine that a closed polyline in the plane is a single loop that is topologically equivalent to a circle, you can obtain more complicated topologies by duplicating points. For example, you can generate a bow tie (two closed loops) in the $z = 0$ plane with $\mathbf{P}_0 = (0, 0, 0)$, $\mathbf{P}_1 = (1, 0, 0)$, $\mathbf{P}_2 = (0, 1, 0)$, $\mathbf{P}_3 = (0, 0, 0)$, $\mathbf{P}_4 = (0, -1, 0)$, and $\mathbf{P}_5 = (-1, 0, 0)$. The contiguous and closed flags are both set to `true`.

The class has the ability to select an active quantity of end points that is smaller or equal to the total number, and the mechanism is exactly the one used in `Polypoint`. If your `Polyline` object represents a collection of disjoint segments, you should also make sure the active quantity is an even number.

3.3.3 Triangle Meshes

The simplest representation for a collection of triangles is as a list of m triples of $3m$ vertices:

$$\langle \mathbf{V}_0, \mathbf{V}_1, \mathbf{V}_2 \rangle, \quad \langle \mathbf{V}_3, \mathbf{V}_4, \mathbf{V}_5 \rangle, \quad \ldots, \quad \langle \mathbf{V}_{3m-3}, \mathbf{V}_{3m-2}, \mathbf{V}_{3m-1} \rangle.$$

The vertices of each triangle are listed in counterclockwise order; that is, the triangle is in a plane with a specified normal vector. An observer on the side of the plane to which the normal is directed sees the vertices of the triangle in a counterclockwise order on that plane. A collection like this is sometimes called a *triangle soup* (more generally, a *polygon soup*). Graphics APIs do support rendering where the triangles are provided this way, but most geometric models built from triangles are not built as a triangle soup. Vertices in the model tend to be part of more than one triangle.

Moreover, if the triangle soup is sent to the renderer, each vertex must be transformed from model space to world space, including running them through the clipping and lighting portions of the system. If a point occurs multiple times in the list of vertices, each one processed by the renderer, we are wasting a lot of cycles.

A more efficient representation for a collection of triangles is to have an array of unique vertices and represent the triangles as a collection of triples of indices into the vertex array. This is called a *triangle mesh*. If $\mathbf{V}_i$ for $0 \le i < n$ is the array of vertices, an index array I_j for $0 \le j < 3m$ represents the triangles

$$\langle \mathbf{V}_{I_0}, \mathbf{V}_{I_1}, \mathbf{V}_{I_2} \rangle, \quad \langle \mathbf{V}_{I_3}, \mathbf{V}_{I_4}, \mathbf{V}_{I_5} \rangle, \quad \ldots, \quad \langle \mathbf{V}_{I_{3m-3}}, \mathbf{V}_{I_{3m-2}}, \mathbf{V}_{I_{3m-1}} \rangle.$$

The goal, of course, is that n is a lot smaller than $3m$ because of the avoidance of duplicate vertices in the vertex array. Fewer vertices must be processed by the renderer, leading to faster drawing.

The class that represents triangle meshes is `TriMesh`. A portion of the interface is

```
class TriMesh : public Geometry
{
public:
    TriMesh (Vector3fArrayPtr spkVertices, IntArrayPtr spkIndices,
        bool bGenerateNormals);
    virtual ~TriMesh ();

    int GetTriangleQuantity () const;
    void GenerateNormals ();

protected:
    TriMesh ();
    virtual void UpdateModelNormals ();
};
```

I have omitted the interface that supports the picking system and will discuss that in Section 6.3.3.

The constructor requires you to provide the vertex and index arrays for the triangle mesh. The quantity of elements in the index array should be a multiple of three. The member function `GetTriangleQuantity` returns the quantity of indices divided by three. For the purposes of lighting, the renderer will need to use vertex normals. The third parameter of the constructor determines whether or not the normals should be generated.

The actual construction of the vertex normals is done in the method `UpdateModelNormals`. The method is protected, so you cannot call it directly. It is called indirectly through the public update function `Geometry::UpdateMS`. Multiple algorithms exist for the construction of vertex normals. The one I implemented is as follows. Let T_1

through T_m be those triangles that share vertex $\mathbf{V}$. Let $\mathbf{N}_1$ through $\mathbf{N}_m$ be normal vectors to the triangles, but not necessarily unit-length ones. For a triangle T with vertices $\mathbf{V}_0$, $\mathbf{V}_1$, and $\mathbf{V}_2$, the normal I use is $\mathbf{N} = (\mathbf{V}_1 - \mathbf{V}_0) \times (\mathbf{V}_2 - \mathbf{V}_0)$. The vertex normal is a unit-length vector,

$$\mathbf{N} = \frac{\sum_{i=1}^{m} \mathbf{N}_i}{\left|\sum_{i=1}^{m} \mathbf{N}_i\right|}.$$

The length $|\mathbf{N}_i|$ is twice the area of the triangle to which it is normal. Therefore, large triangles will have a greater effect on the vertex normal than small triangles. I consider this a more reasonable algorithm than one that computes the vertex normal as an average of unit-length normals for the sharing triangles, where all triangles have the same influence on the outcome regardless of their areas.

Should you decide to create a triangle mesh without normals, you can always force the generation by calling the method `GenerateNormals`. This function allocates the normals if they do not already exist and then calls `UpdateModelNormals`.

3.3.4 Particles

A *particle* is considered to be a geometric primitive with a *location* in space and a *size*. The size attribute distinguishes particles from points. A collection of particles is referred to as a *particle system*. Particle systems are quite useful, for interesting visual displays as well as for physical simulations. Both aspects are discussed later, the visual ones in Section 4.1.2 and the physical ones in Section 7.2. In this section I will discuss the geometric aspects and the class `Particles` that represents them.

The portion of the class interface for `Particles` that is relevant to data management is

```
class Particles : public TriMesh
{
public:
    Particles (Vector3fArrayPtr spkLocations,
        FloatArrayPtr spkSizes, bool bWantNormals);
    virtual ~Particles ();

    Vector3fArrayPtr Locations;
    FloatArrayPtr Sizes;
    float SizeAdjust;

    void SetActiveQuantity (int iActiveQuantity);
    int GetActiveQuantity () const;
```

```
protected:
    Particles ();
    void GenerateParticles (const Camera* pkCamera);

    int m_iActiveQuantity;
};
```

The first observation is that the class is derived from `TriMesh`. The particles are drawn as billboard squares (see Section 4.1.2) that always face the observer. Each square is built of two triangles, and all the triangles are stored in the base class as a triangle mesh. The triangle mesh has four times the number of vertices as it does particle locations, which is why the locations are stored as a separate array.

The constructor accepts inputs for the particle locations and sizes. The third parameter determines whether or not normal vectors are allocated. If they are, the normal vectors are in the opposite direction of view—they are directed toward the observer. Even though the particles are drawn as billboards, they may still be affected by lights in the scene, so the normal vectors are relevant.

The data members `Locations`, `Sizes`, and `SizeAdjust` are in public scope because no side effects must occur when they are read or written. The locations and sizes are as described previously. The data member `SizeAdjust` is used to uniformly scale the particle sizes, if so desired. The adjustment is a multiplier of the sizes stored in the member array `Sizes`, not a replacement for those values. The initial value for the size adjustment is one.

The class has the ability to select an active quantity of end points that is smaller or equal to the total number. The mechanism is exactly the one used in `Polypoint`.

3.4 Render State

I use the term *render state* to refer to all the information that is associated with the geometric data for the purposes of drawing the objects. Three main categories of render state are *global state*, *lights*, and *effects*.

3.4.1 Global State

Global state refers to information that is essentially independent of any information the objects might provide. The states I have included in the engine are alpha blending, triangle culling, dithering, fog, material, shading, wireframe, and depth buffering. For example, depth buffering does not care how many vertices or triangles an object has. A material has attributes that are applied to the vertices of an object, regardless of how many vertices it has. Alpha blending relies on the texture images of the `Geometry` object having an alpha channel, but it does not care about what the texture coordinates are for that object. A global state, when attached to an interior node in a scene hierarchy, affects all leaf nodes in the subtree rooted at the node. This property is why I used the adjective *global*.

The base class is `GlobalState` and has the following interface:

```
class GlobalState : public Object
{
public:
    virtual ~GlobalState ();

    enum // Type
    {
        ALPHA,
        CULL,
        DITHER,
        FOG,
        MATERIAL,
        SHADE,
        WIREFRAME,
        ZBUFFER,
        MAX_STATE
    };

    virtual int GetGlobalStateType () const = 0;

    static Pointer<GlobalState> Default[MAX_STATE];

protected:
    GlobalState ();
};
```

The base class is abstract since the constructor is protected (or since there is a pure virtual function declared). To support fast access of global states in arrays of smart pointers `GlobalStatePtr`, I chose to avoid using the `Object` run-time type information system. The enumerated type of `GlobalState` provides an alternate RTTI system. Each derived class returns its enumerated value through an implementation of `GetGlobalStateType`. Each derived class is also responsible for creating a default state, stored in the static array `GlobalState::Default[]`. Currently, the enumerated values are a list of all the global states I support. If you were to add another one, you would derive a class `MyNewGlobalState` from `GlobalState`. But you also have to add another enumerated value `MYNEWGLOBALSTATE` to the base class. This violates the open-closed principle of object-oriented programming, but the changes to `GlobalState` are so simple and so infrequent that I felt justified in the violation. None of the classes in Wild Magic version 3 ever write an array of global state pointers to disk, so adding a new state does not invalidate all of the scenes you had streamed before the change.

The global states are stored in class `Spatial`. A portion of the interface relative to global state storing and member accessing is

```
class Spatial : public Object
{
public:
    void SetGlobalState (GlobalState* pkState);
    GlobalState* GetGlobalState (int eType) const;
    void RemoveGlobalState (int eType);
    void RemoveAllGlobalStates ();

protected:
    TList<GlobalStatePtr>* m_pkGlobalList;
};
```

The states are stored in a singly linked list of smart pointers. The choice was made to use a list rather than an array, whose indices are the `GlobalState` enumerated values, to reduce memory usage. A typical scene will have only a small number of nodes with global states attached, so the array representation would generate a lot of wasted memory for all the other nodes. The names of the member functions make it clear how to use the functions. The `eType` input is intended to be one of the `GlobalState` enumerated values. For example, the code

```
MaterialState* pkMS = <some material state>;
Spatial* pkSpatial = <some Spatial-derived object>;
pkSpatial->SetGlobalState(pkMS);
pkSpatial->RemoveGlobalState(GlobalState::MATERIAL);
```

attaches a material state to an object, then removes it from the object.

The class `Geometry` also has storage for global states, but the storage is for all global states encountered along the path in the scene hierarchy from the root node to the geometry leaf node. The storage is assembled during a render state update, a topic discussed later in this section. The portion of the interface of `Geometry` relevant to storage is

```
class Geometry : public Spatial
{
// internal use
public:
    GlobalStatePtr States[GlobalState::MAX_STATE];
};
```

The array of states is in public scope, but is tagged for internal use only. An application should not manipulate the array or its members.

I will now discuss each of the derived global state classes. These classes have a couple of things in common. First, they must all implement the virtual function `GetGlobalStateType`. Second, they must all create default objects, something that is done at program initialization. At program termination, the classes should all destroy their default objects. The initialization-termination system discussed in Section 2.3.8 is used to perform these actions. You will see that each derived class uses the macros defined in `Wm3Main.mcr` and implements `void Initialize()` and `void Terminate()`. All the derived classes have a default constructor that is used to create the default objects.

Depth Buffering

In a correctly rendered scene, the pixels drawn in the frame buffer correspond to those visible points closest to the observer. But many points in the (3D) world can be projected to the same pixel on the (2D) screen. The graphics system needs to keep track of the actual depths in the world to determine which of those points is the visible one. This is accomplished through *depth buffering*. A frame buffer stores the pixel colors, and a depth buffer stores the corresponding depth in the world. The depth buffer is sometimes called a *z-buffer*, but in a perspective camera model, the depth is not measured along a direction perpendicular to the screen. It is depth along rays emanating from the camera location (the eye point) into the world.

The standard drawing pass for a system using a depth buffer is

```
FrameBuffer fbuffer = <current RGB values on the screen>;
DepthBuffer zbuffer = <current depth values for fbuffer pixels>;
ColorRGB sourceColor = <color of pixel to be drawn>;
float sourceDepth = <depth of the point in the world>;
int x, y = <location of projected point in the buffers>;

if ( sourceDepth <= zbuffer(x,y) )
{
    fbuffer(x,y) = sourceColor;
    zbuffer(x,y) = sourceDepth;
}
```

Three aspects of a depth buffer are apparent in the code. You need the ability to *read* from the depth buffer, *compare* a value against the read value, and *write* to the depth buffer. The comparison function in the pseudocode is “less than or equal to.” You could use “less than” if so desired, but allowing equality provides the ability to draw on top of something already visible when the new object is coincident with the already visible object (think “decal”). For generality, the comparison could be written as

```
if ( ComparesFavorably(sourceDepth,zbuffer(x,y)) )
{
    fbuffer(x,y) = sourceColor;
    zbuffer(x,y) = sourceDepth;
}
```

The function `ComparesFavorably(a,b)` can be any of the usual ones: $a < b$, $a \leq b$, $a > b$, $a \geq b$, $a = b$, or $a \neq b$. Two additional functions are allowed: always or never. In the former, the frame and depth buffers are always updated. In the latter, the frame and depth buffers are never updated.

The class that encapsulates the depth buffering is `ZBufferState`. Its interface is

```
class ZBufferState : public GlobalState
{
public:
    virtual int GetGlobalStateType () const { return ZBUFFER; }

    ZBufferState ();
    virtual ~ZBufferState ();

    enum // Compare values
    {
        CF_NEVER,
        CF_LESS,
        CF_EQUAL,
        CF_LEQUAL,
        CF_GREATER,
        CF_NOTEQUAL,
        CF_GEQUAL,
        CF_ALWAYS,
        CF_QUANTITY
    };

    bool Enabled;    // default: true
    bool Writable;   // default: true
    int Compare;     // default: CF_LEQUAL
};
```

The data members are in public scope because no side effects must occur when they are read or written. The member `Enabled` is set to `true` if you want depth buffering to be enabled. In this case, the buffer is automatically readable. To make the depth buffer writable, set the member `Writable` to `true`. The comparison function is controlled by the member `Compare`. The defaults are the standard ones used when depth buffering is desired (readable, writable, less than or equal to for the comparison). A simple code block for using standard depth buffering in an entire scene is

```
NodePtr m_spkScene = <the scene graph>;
m_spkScene->SetGlobalState(new ZBufferState);
```

There are situations where you want the depth buffer to be readable, but not writable. One of these occurs in conjunction with alpha blending and semitransparent objects, the topic of the next few paragraphs.

Alpha Blending

Given two RGBA colors, one called the *source color* and one called the *destination color*, the term *alpha blending* refers to the general process of combining the source and destination into yet another RGBA color. The source color is (r_s, g_s, b_s, a_s), and the destination color is (r_d, g_d, b_d, a_d). The blended result is the final color (r_f, g_f, b_f, a_f). All color channel values in this discussion are assumed to be in the interval [0, 1].

The classical method for blending is to use the alpha channel as an opacity factor. If the alpha value is 1, the color is completely opaque. If the alpha value is 0, the color is completely transparent. If the alpha value is strictly between 0 and 1, the colors are semitransparent. The formula for the blend of only the RGB channels is

$$\begin{aligned}(r_f, g_f, b_f) &= (1 - a_s)(r_s, g_s, b_s) + a_s(r_d, g_d, b_d) \\ &= ((1 - a_s)r_s + a_s r_d, (1 - a_s)g_s + a_s g_d, (1 - a_s)b_s + a_s b_d).\end{aligned}$$

The algebraic operations are performed component by component. The assumption is that you have already drawn the destination color into the frame buffer; that is, the frame buffer becomes the destination. The next color you draw is the source. The alpha value of the source color is used to blend the source color with the current contents of the frame buffer.

It is also possible to draw the destination color into an offscreen buffer, blend the source color with it, and then use the offscreen buffer for blending with the current contents of the frame buffer. In this sense we also want to keep track of the alpha value in the offscreen buffer. We need a blending equation for the alpha values themselves. Using the same operations as for the RGB channels, your choice will be

$$a_f = (1 - a_s)a_s + a_s a_d.$$

Combining the four channels into a single equation, the classic alpha blending equation is

$$(r_f, g_f, b_f, a_f) = ((1 - a_s)r_s + a_s r_d, (1 - a_s)g_s + a_s g_d, (1 - a_s)b_s + a_s b_d, (1 - a_s)a_s + a_s a_d). \quad (3.5)$$

If the final colors become the destination for another blending operation, then

$$(r_d, g_d, b_d, a_d) = (r_f, g_f, b_f, a_f)$$

sets the destination to the previous blending results.

Graphics APIs support more general combinations of colors, whether the colors come from vertex attributes or texture images. The general equation is

$$(r_f, g_f, b_f, a_f) = (\sigma_r r_s + \delta_r r_d, \sigma_g g_s + \delta_g g_d, \sigma_b b_s + \delta_b b_d, \sigma_a a_s + \delta_a a_d). \quad (3.6)$$

The blending coefficients are σ_i and δ_i, where the subscripts denote the color channels they affect. The coefficients are assumed to be in the interval [0, 1]. Wild Magic provides the ability for you to select the blending coefficients from a finite set of possibilities. The class that encapsulates this is `AlphaState`. Since the `AlphaState` class exists in the scene graph management system, it must provide a graphics-API-independent mechanism for selecting the coefficients. The names I use for the various possibilities are reminiscent of those OpenGL uses, but also map to what Direct3D supports.

The portion of the class interface for `AlphaState` relevant to the blending equation (3.6) is

```
class AlphaState : public GlobalState
{
public:
    enum // SrcBlend values
    {
        SBF_ZERO,
        SBF_ONE,
        SBF_DST_COLOR,
        SBF_ONE_MINUS_DST_COLOR,
        SBF_SRC_ALPHA,
        SBF_ONE_MINUS_SRC_ALPHA,
        SBF_DST_ALPHA,
        SBF_ONE_MINUS_DST_ALPHA,
        SBF_SRC_ALPHA_SATURATE,
        SBF_CONSTANT_COLOR,
        SBF_ONE_MINUS_CONSTANT_COLOR,
        SBF_CONSTANT_ALPHA,
        SBF_ONE_MINUS_CONSTANT_ALPHA,
        SBF_QUANTITY
    };
```

```
    enum // DstBlend values
    {
        DBF_ZERO,
        DBF_ONE,
        DBF_SRC_COLOR,
        DBF_ONE_MINUS_SRC_COLOR,
        DBF_SRC_ALPHA,
        DBF_ONE_MINUS_SRC_ALPHA,
        DBF_DST_ALPHA,
        DBF_ONE_MINUS_DST_ALPHA,
        DBF_CONSTANT_COLOR,
        DBF_ONE_MINUS_CONSTANT_COLOR,
        DBF_CONSTANT_ALPHA,
        DBF_ONE_MINUS_CONSTANT_ALPHA,
        DBF_QUANTITY
    };

    bool BlendEnabled;  // default: false
    int SrcBlend;       // default: SBF_SRC_ALPHA
    int DstBlend;       // default: DBF_ONE_MINUS_SRC_ALPHA
};
```

The data members are all public since no side effects must occur when reading or writing them. The data member `BlendEnabled` is set to `false` initially, indicating that the default alpha blending state is “no blending.” This member should be set to `true` when you do want blending to occur.

The data member `SrcBlend` controls what the source blending coefficients $(\sigma_r, \sigma_g, \sigma_b, \sigma_a)$ are. Similarly, the data member `DstBlend` controls what the destination blending coefficients $(\delta_r, \delta_g, \delta_b, \delta_a)$ are. Table 3.2 lists the possibilities for the source blending coefficients. The constant color, (r_c, g_c, b_c, a_c), is stored in the `Texture` class (member `BlendColor`), as is an RGBA image that is to be blended with a destination buffer.

Table 3.3 lists the possibilities for the destination blending coefficients. Table 3.2 has `DST_COLOR`, `ONE_MINUS_DST_COLOR`, and `SRC_ALPHA_SATURATE`, but Table 3.3 does not. Table 3.3 has `SRC_COLOR` and `ONE_MINUS_SRC_COLOR`, but Table 3.2 does not.

The classic alpha blending equation (3.5) is reproduced by the following code:

```
AlphaState* pkAS = new AlphaState;
pkAS->BlendEnabled = true;
pkAS->SrcBlend = AlphaState::SBF_SRC_ALPHA;
pkAS->DstBlend = AlphaState::DBF_ONE_MINUS_SRC_ALPHA;
Spatial* pkSpatial = <some Spatial-derived object>;
pkSpatial->SetGlobalState(pkAS);
```

Table 3.2 The possible source blending coefficients.

Enumerated value	$(\sigma_r, \sigma_g, \sigma_b, \sigma_a)$
`SBF_ZERO`	$(0, 0, 0, 0)$
`SBF_ONE`	$(1, 1, 1, 1)$
`SBF_DST_COLOR`	(r_d, g_d, b_d, a_d)
`SBF_ONE_MINUS_DST_COLOR`	$(1 - r_d, 1 - g_d, 1 - b_d, 1 - a_d)$
`SBF_SRC_ALPHA`	(a_s, a_s, a_s, a_s)
`SBF_ONE_MINUS_SRC_ALPHA`	$(1 - a_s, 1 - a_s, 1 - a_s, 1 - a_s)$
`SBF_DST_ALPHA`	(a_d, a_d, a_d, a_d)
`SBF_ONE_MINUS_DST_ALPHA`	$(1 - a_d, 1 - a_d, 1 - a_d, 1 - a_d)$
`SBF_SRC_ALPHA_SATURATE`	$(\sigma, \sigma, \sigma, 1), \sigma = \min\{a_s, 1 - a_d\}$
`SBF_CONSTANT_COLOR`	(r_c, g_c, b_c, a_c)
`SBF_ONE_MINUS_CONSTANT_COLOR`	$(1 - r_c, 1 - g_c, 1 - b_c, 1 - a_c)$
`SBF_CONSTANT_ALPHA`	(a_c, a_c, a_c, a_c)
`SBF_ONE_MINUS_CONSTANT_ALPHA`	$(1 - a_c, 1 - a_c, 1 - a_c, 1 - a_c)$

The default constructor for `AlphaState` sets `SrcBlend` and `DstBlend`, so only setting the `BlendEnabled` to `true` is necessary in an actual program. If the alpha state object is attached to a node in a subtree, it will affect the drawing of all the leaf node objects.

A more interesting example is one that does a *soft addition* of two textures. *Hard addition* refers to adding the two colors together and then clamping the result to [0, 1]. This may result in saturation of colors, causing the resulting image to look washed out. The soft addition combines the two colors to avoid the saturation, yet still look like an addition of colors. The formula is

$$(r_f, g_f, b_f, a_f) = ((1 - r_d)r_s, (1 - g_d)g_s, (1 - b_d)b_s, (1 - a_d)a_s) + (r_d, g_d, b_d, a_d). \tag{3.7}$$

The idea is that you start with the destination color and add a fraction of the source color to it. If the destination color is bright (values near 1), then the source blend coefficients are small, so the source color will not cause the result to wash out. Similarly, if the destination color is dark (values near 0), the destination color has little contribution to the result, and the source blend coefficients are large, so the source color dominates and the final result is a brightening of dark regions. The code block to obtain the blend is

Table 3.3 The possible destination blending coefficients.

Enumerated value	$(\delta_r, \delta_g, \delta_b, \delta_a)$
DBF_ZERO	$(0, 0, 0, 0)$
DBF_ONE	$(1, 1, 1, 1)$
DBF_SRC_COLOR	(r_s, g_s, b_s, a_s)
DBF_ONE_MINUS_SRC_COLOR	$(1 - r_s, 1 - g_s, 1 - b_s, 1 - a_s)$
DBF_SRC_ALPHA	(a_s, a_s, a_s, a_s)
DBF_ONE_MINUS_SRC_ALPHA	$(1 - a_s, 1 - a_s, 1 - a_s, 1 - a_s)$
DBF_DST_ALPHA	(a_d, a_d, a_d, a_d)
DBF_ONE_MINUS_DST_ALPHA	$(1 - a_d, 1 - a_d, 1 - a_d, 1 - a_d)$
DBF_CONSTANT_COLOR	(r_c, g_c, b_c, a_c)
DBF_ONE_MINUS_CONSTANT_COLOR	$(1 - r_c, 1 - g_c, 1 - b_c, 1 - a_c)$
DBF_CONSTANT_ALPHA	(a_c, a_c, a_c, a_c)
DBF_ONE_MINUS_CONSTANT_ALPHA	$(1 - a_c, 1 - a_c, 1 - a_c, 1 - a_c)$

```
AlphaState* pkAS = new AlphaState;
pkAS->BlendEnabled = true;
pkAS->SrcBlend = AlphaState::SBF_ONE_MINUS_DST_COLOR;
pkAS->DstBlend = AlphaState::DBF_ONE;
Spatial* pkSpatial = <some Spatial-derived object>;
pkSpatial->SetGlobalState(pkAS);
```

The `AlphaState` class also encapsulates what is referred to as *alpha testing*. The idea is that an RGBA source color will only be combined with the RGBA destination color as long as the source alpha value compares favorably with a specified *reference* value. Pseudocode for alpha testing is

```
source = (Rs,Gs,Bs,As);
destination = (Rd,Gd,Bd,Ad);
reference = Aref;
if ( ComparesFavorably(As,Aref) )
    result = BlendTogether(source,destination);
```

The `ComparesFavorably(x,y)` function is a standard comparison between two numbers: $x < y$, $x \leq y$, $x > y$, $x \geq y$, $x = y$, or $x \neq y$. Two additional functions are allowed: always or never. In the former, the blending always occurs. This is the default behavior of an alpha blending system. In the latter, blending never occurs.

The portion of the interface of `AlphaState` relevant to alpha testing is

```
class AlphaState : public GlobalState
{
public:
    enum // Test values
    {
        TF_NEVER,
        TF_LESS,
        TF_EQUAL,
        TF_LEQUAL,
        TF_GREATER,
        TF_NOTEQUAL,
        TF_GEQUAL,
        TF_ALWAYS,
        TF_QUANTITY
    };

    bool TestEnabled;   // default: false;
    int Test;           // default: TF_ALWAYS
    float Reference;    // default: 0, always in [0,1]
};
```

By default, alpha testing is turned off. To turn it on, set `TestEnabled` to `true`. The `Reference` value is a floating-point number in the interval [0, 1]. The `Test` function may be set to any of the first eight enumerated values prefixed with `TF_`. The value `TF_QUANTITY` is just a marker that stores the current number of enumerated values and is used by the renderers to declare arrays of that size.

In order to correctly draw a scene that has some semitransparent objects (alpha values smaller than 1), the rule is to draw your opaque objects first, then draw your semitransparent objects sorted from back to front in the view direction. The leaf nodes in a scene hierarchy can be organized so that those corresponding to opaque objects occur before those corresponding to semitransparent objects when doing a depth-first traversal of the tree. However, the leaf nodes for the semitransparent objects occur in a specific order that is not related to the view direction of the camera. A rendering system needs to have the capability for accumulating a list of semitransparent objects and then sorting the list based on the current view direction. The sorted list is then drawn an object at a time. Given an automated system for sorting, there is no need to worry about where the semitransparent objects occur in the scene. The scene organization can be based on geometric information, the main premise for using a scene hierarchy in the first place. I will discuss the sorting issue in Section 4.2.4.

Game programmers are always willing to take a shortcut to obtain a faster system, or to avoid having to implement some complicated system, and hope that the consequences are not visually distracting. In the context of correct sorting for scenes with semitransparency, the shortcut is to skip the sorting step. After all, what are the chances that someone will notice artifacts due to blending objects that are not sorted back to front? Alpha testing can help you with this shortcut. The scene is rendered twice, and on both passes, depth buffering is enabled in order to correctly sort the objects (on a per-pixel basis in screen space). Also on both passes, alpha testing is enabled. On the first pass, the test function is set to allow the blending for any colors with an alpha value equal to 1; that is, the opaque objects are drawn, but the semitransparent objects are not. On the second pass, the test function is set to allow the blending for any colors with an alpha value not equal to 1. This time the semitransparent objects are drawn, but the opaque objects are not.

As stated, this system is not quite right (ignoring the back-to-front sorting issue). Depth buffering is enabled, but recall that you have the capability to control whether reading or writing occurs. For the first pass through the scene, opaque objects are drawn. The depth buffering uses both reading and writing to guarantee that the final result is rendered correctly. Before drawing a pixel in the frame buffer, the depth buffer is read at the corresponding location. If the incoming depth passes the depth test, then the pixel is drawn in the frame buffer. Consequently, the depth buffer must be written to update the new depth for this pixel. If the incoming depth does not pass the test, the pixel is not drawn, and the depth buffer is not updated. For the second pass through the scene, semitransparent objects are drawn. These objects were not sorted from back to front. It is possible that two semitransparent objects are drawn front to back; that is, the first drawn object is closer to the observer than the second drawn object. You can see through the first drawn object because it is semitransparent, so you expect to see the second drawn object immediately behind it. To guarantee this happens, you have to disable depth buffer writes on the second pass. Consider if you did not do this. The first object is drawn, and the depth buffer is written with the depths corresponding to that object. When you try to draw the second object, its depths are larger than those of the first object, so the depth test fails and the second object is not drawn, even though it should be visible through the first. Disabling the depth buffer writing will prevent this error. Sample code to implement the process is

```
// in the application initialization phase
NodePtr m_spkScene = <the scene graph>;
Renderer* m_pkRenderer = <the renderer>;
AlphaState* pkAS = new AlphaState;
ZBufferState* pkZS = new ZBufferState;
m_spkScene->SetGlobalState(pkAS);
m_spkScene->SetGlobalState(pkZS);

pkAS->BlendEnabled = true;
pkAS->SrcBlend = AlphaState::SBF_SRC_ALPHA;
pkAS->DstBlend = AlphaState::DBF_ONE_MINUS_SRC_ALPHA;
```

```
pkAS->TestEnabled = true;
pkAS->Reference = 1.0f;

pkZS->Enabled = true;  // always readable
pkZS->Compare = ZBufferState::CF_LEQUAL;


// in the drawing phase (idle loop)
AlphaState* pkAS =
    m_spkScene->GetGlobalState(GlobalState::ALPHA);
ZBufferState* pkZS =
    m_spkScene->GetGlobalState(GlobalState::ZBUFFER);

// first pass
pkAS->Test = AlphaState::TF_EQUAL;
pkZS->Writable = true;
m_pkRenderer->DrawScene(m_spkScene);

// second pass
pkAS->Test = AlphaState::TF_NOTEQUAL;
pkZS->Writable = false;
m_pkRenderer->DrawScene(m_spkScene);
```

The alpha state and z-buffer state members that do not change in the drawing phase are all initialized once by the application. You could also store these states as members of the application object and avoid the `GetGlobalState` calls, but the lookups are not an expensive operation.

Another example of alpha testing is for drawing objects that have textures whose alpha values are either 0 or 1. The idea is that the texture in some sense defines what the object is. A classic example is for applying a decal to an object. The decal geometry is a rectangle that has a texture associated with it. The texture image has an artistically drawn object that does not cover all the image pixels. The pixels not covered are "see through"; that is, if the decal is drawn on top of another object, you see what the artist has drawn in the image, but you see the other object elsewhere. To accomplish this, the alpha values of the image are set to 1 wherever the artist has drawn, but to 0 everywhere else. Attach an `AlphaState` object to the decal geometry object (the rectangle). Set the reference value to be 0.5 (it just needs to be different from 0 and 1), and set the test function to be "greater than." When the decal texture is drawn on the object, only the portion is drawn with alpha values equal to 1 (greater than 0.5). The portion with alpha values equal to 0 (not greater than 0.5) is not drawn. Because they are not drawn, the depth buffer is not affected, so you do not have to use the two-pass technique discussed in the previous example.

Material

One of the simplest ways to color a geometric object is to provide it with a *material*. The material has various colors that affect how all the vertices of the object are colored. The class MaterialState represents the material and has the interface

```
class MaterialState : public GlobalState
{
public:
    virtual int GetGlobalStateType () const { return MATERIAL; }

    MaterialState ();
    virtual ~MaterialState ();

    ColorRGBA Emissive;  // default:  ColorRGBA(0,0,0,1)
    ColorRGBA Ambient;   // default:  ColorRGBA(0.2,0.2,0.2,1)
    ColorRGBA Diffuse;   // default:  ColorRGBA(0.8,0.8,0.8,1)
    ColorRGBA Specular;  // default:  ColorRGBA(0,0,0,1)
    float Shininess;     // default:  1
};
```

The data members are all public since no side effects are required when reading or writing them. If you want the geometric object to have the appearance that it is emitting light, you set the Emissive data member to the desired color. The member Ambient represents a portion of the object's color that is due to any ambient light in the scene. Other lights can shine on the object. How the object is lit depends on its material properties and on the normal vectors to the object's surface. For a matte appearance, set the Diffuse data member. Specular highlights are controlled by the Specular and Shininess parameters. Although all colors have an alpha channel, the only relevant one in the graphics API is the alpha channel in the diffuse color. Objects cannot really "emit" an alpha value. An alpha value for ambient lighting also does not make physical sense, and specular lighting says more about reflecting light relative to an observer. The diffuse color is more about the object itself, including having a material that is semitransparent.

In the standard graphics APIs, it is not enough to assign a material to an object. The material properties take effect only when lights are present. Moreover, diffuse and specular lighting require the object to have vertex normals. If you choose to attach a MaterialState to an object, you will need at least one light in the scene. Specifically, the light must occur on the path from the root node to the node containing the material. If the material is to show off its diffuse and specular lighting, any leaf node geometry objects in the subtree rooted at the node containing the material must have vertex normals. Later in this section I will discuss lights, and at that time I will present the formal equations for how lights, materials, and normal vectors interact.

Fog

The portions of a rendered scene at locations far from the observer can look a lot sharper than what your vision expects. To remedy this, you can add *fog* to the system, with the amount proportional to the distance an object is from the observer. The density of the fog (the amount of increase in fog per unit distance in the view frustum) can be controlled. The fog can be calculated on a per-vertex basis for the purposes of speed, but you may request that the fog be calculated on a per-pixel basis. The final color is a blended combination of a fog color and the vertex/pixel color. Fog is applied after transformations, lighting, and texturing are performed, so such objects are affected by the inclusion of fog.

The class that encapsulates fog is `FogState`. The interface is

```
class FogState : public GlobalState
{
public:
    virtual int GetGlobalStateType () const { return FOG; }

    FogState ();
    virtual ~FogState ();

    enum // DensityFunction
    {
        DF_LINEAR,
        DF_EXP,
        DF_EXPSQR,
        DF_QUANTITY
    };

    enum // ApplyFunction
    {
        AF_PER_VERTEX,
        AF_PER_PIXEL,
        AF_QUANTITY
    };

    bool Enabled;         // default: false
    float Start;          // default: 0
    float End;            // default: 1
    float Density;        // default: 1
    ColorRGBA Color;      // default: ColorRGB(0,0,0)
    int DensityFunction;  // default: DF_LINEAR
    int ApplyFunction;    // default: AF_PER_VERTEX
};
```

The data members are all in public scope since no side effects must occur when reading or writing them. The default values are listed as comments after each member. The `Color` member is the fog color that is used to blend with the vertex/pixel colors. A *fog factor* $f \in [0, 1]$ is used to blend the fog color and vertex/pixel color. If (r_0, g_0, b_0, a_0) is the fog color and (r_1, g_1, b_1, a_1) is the vertex/pixel color, then the blended color is

$$(r_2, g_2, b_2, a_2) = f(r_0, g_0, b_0, a_0) + (1 - f)(r_1, g_1, b_1, a_1),$$

where the operations on the right-hand side are performed componentwise. If the fog factor is 1, the vertex/pixel color is unaffected by the fog. If the fog factor is 0, only the fog color appears. The `ApplyFunction` member selects whether you want per-vertex fog calculations (faster) or per-pixel fog calculations (slower).

The `DensityFunction` controls how the fog is calculated. If the data member is set to `DF_LINEAR`, then the `Start` and `End` data members must be set and indicate the range of depths to which the fog is applied. The fog factor is

$$f = \frac{\texttt{End} - z}{\texttt{End - Start}},$$

where z is the depth measured from the camera position to the vertex or pixel location. In practice, you may choose the start and end values to be linear combinations of the near and far plane values. Linear fog does not use the `Density` data member, so its value is irrelevant.

If the `DensityFunction` value is set to `DF_EXP`, then the fog factor has exponential decay. The amount of decay is controlled by the nonnegative `Density` member function. The fog itself appears accumulated at locations far from the camera position. The fog factor is

$$f = \exp(-\texttt{Density} * z),$$

where z is once again measured from the camera position to the vertex or pixel location. You may also obtain a tighter accumulation of fog at large distances by using `DF_EXPSQR`. The fog factor is

$$f = \exp(-(\texttt{Density} * z)^2).$$

The exponential and squared exponential fog equations do not use the `Start` and `End` data members, so their values are irrelevant. The terrain sample application uses squared exponential fog as an attempt to hide the entry of new terrain pages through the far plane of the view frustum.

Culling

Consider a triangle mesh that is a convex polyhedron. Some of the triangles are visible to the observer. These are referred to as *front-facing* triangles. The triangles not visible to the observer are referred to as *back-facing* triangles. The two subsets are dependent, of course, on the observer's location. If the mesh is closed, the triangles can still be partitioned into two subsets, one for the front-facing triangles and one for the back-facing triangles. The back-facing triangles are not visible to the observer, so there is no reason why they should be drawn. The rendering system should eliminate these—a process called *triangle culling*.

In Section 3.3.3, I mentioned that the triangles in a mesh have their vertices ordered in a counterclockwise manner when viewed by the observer. This means that the vertices of a front-facing triangle are seen as counterclockwise ordered. The vertices of a back-facing triangle are clockwise ordered from the observer's perspective. The rendering system may use these facts to classify the two types of triangles. Let the observer's eye point be $\mathbf{E}$ and let the observed triangle have counterclockwise-ordered vertices $\mathbf{V}_0$, $\mathbf{V}_1$, and $\mathbf{V}_2$. The vector $\mathbf{N} = (\mathbf{V}_1 - \mathbf{V}_0) \times (\mathbf{V}_2 - \mathbf{V}_0)$ is perpendicular to the plane of the triangle. The triangle is front facing if

$$\mathbf{N} \cdot (\mathbf{E} - \mathbf{V}_0) > 0,$$

and it is back facing if

$$\mathbf{N} \cdot (\mathbf{E} - \mathbf{V}_0) \le 0.$$

If the dot product is zero, the triangle is seen edge on and is considered not to be visible.

A rendering system will select a convention for the vertex ordering for front-facing triangles. This is necessary so that the dot product tests can be coded accordingly. A modeling package also selects a convention for the vertex ordering, but the problem is that the conventions might not be consistent. If not, you can always export the models to a format your engine supports, then reorder the triangle vertices to meet the requirements of your rendering system. The burden of enforcing the ordering constraint is yours. Alternatively, the rendering system can allow you to specify the convention, making the system more flexible. The renderer will use the correct equations for the dot product tests to identify the back-facing triangles. In fact, the standard graphics APIs allow for this. I have encapsulated this in class `CullState`.

The interface for `CullState` is

```
class CullState : public GlobalState
{
public:
    virtual int GetGlobalStateType () const { return CULL; }
```

```
    CullState ();
    virtual ~CullState ();

    enum // FrontType
    {
        FT_CCW,  // front faces are counterclockwise ordered
        FT_CW,   // front faces are clockwise ordered
        FT_QUANTITY
    };

    enum // CullType
    {
        CT_FRONT,  // cull front-facing triangles
        CT_BACK,   // cull back-facing triangles
        CT_QUANTITY
    };

    bool Enabled;   // default: true
    int FrontFace;  // default: FT_CCW
    int CullFace;   // default: CT_BACK
};
```

The data members are in public scope because no side effects must occur when reading or writing them. The default value for `Enabled` is `true`, indicating that triangle culling is enabled. The `FrontFace` member lets you specify the vertex ordering for the triangles that you wish to use. The default value is counterclockwise. The `CullFace` member lets you tell the renderer to cull front-facing or back-facing triangles. The default is to cull back-facing triangles.

When triangle culling is enabled, the triangles are said to be *single sided*. If an observer can see one side of the triangle, and if you were to place the observer on the other side of the plane of the triangle, the observer would not see the triangle from that location. If you are inside a model of a room, the triangles that form the walls, floor, and ceiling may as well be single sided since the intent is to only see them when you are inside the room. But if a wall separates the room from the outside environment and you want the observer to see the wall from the outside (maybe it is a stone wall in a castle), then the triangle culling system gets in your way when it is enabled. In this scenario you want the triangles to be *double sided*—both sides visible to an observer. This is accomplished by simply disabling triangle culling; set `Enabled` to `false`.

Disabling triangle culling is also useful when a triangle mesh is not closed. For example, you might have a model of a flag that flaps in the wind. The flag mesh is initially a triangle mesh in a plane. During program execution, the triangle vertices are dynamically modified to obtain the flapping. Because you want both sides of the flag to be visible, you would attach a `CullState` object to the mesh and set `Enabled` to `false`.

Wireframe

A *wireframe* is a rendering of a triangle mesh where only the edges of the triangles are drawn. Although such an effect might be interesting in a game, wireframes are typically used for debugging purposes. For example, in wireframe mode you might be able to spot problems with a geometric model that was not properly constructed. I used wireframe mode when implementing and testing the portal system. Portions of a scene are culled by the portal system, but you do not see the culling occur when in regular drawing mode. However, in wireframe you can see the objects appear or disappear, giving you an idea whether or not the culling is taking place as planned.

The class to support wireframe mode is `WireframeState`. The interface is

```
class WireframeState : public GlobalState
{
public:
    virtual int GetGlobalStateType () const { return WIREFRAME; }

    WireframeState ();
    virtual ~WireframeState ();

    bool Enabled;  // default: false
};
```

Very simple, as you can see. You either enable or disable the mode. In practice, I tend to attach a `WireframeState` object to the root of my scene graph. I make the wireframe object an application member and then allow toggling of the `Enabled` member:

```
// initialization code
NodePtr m_spkScene = <the scene graph>;
WireframePtr m_spkWireframe = new WireframeState;
m_spkScene->SetGlobalState(m_spkWireframe);

// key press handler
if ( ucKey == 'w' )
    m_spkWireframe->Enabled = !m_spkWireframe->Enabled;
```

Dithering

On a graphics system with a limited number of colors, say, one supporting only a 256-color palette, a method for increasing the apparent number of colors is *dithering*.

As an example, suppose your system supports only two colors, black and white. If you were to draw an image with a checkerboard pattern—alternate drawing pixels in black and white—your eye will perceive the image as gray, even though your graphics system cannot draw a gray pixel. Consider that a black-ink printer is such a graphics system. Color printers may also use dithering to increase the apparent number of colors. How to do this effectively is the topic of color science. The dithering pattern can be much more complicated than a checkerboard. Suffice it to say that a graphics API might support dithering.

For the 32-bit color support that current graphics hardware has, dithering is probably not useful, but I have support for it anyway. The class is `DitherState` and the interface is

```
class DitherState : public GlobalState
{
public:
    virtual int GetGlobalStateType () const { return DITHER; }

    DitherState ();
    virtual ~DitherState ();

    bool Enabled;  // default: false
};
```

The dithering is either enabled or disabled.

Shading

Some graphics APIs provide the ability to select a *shading model*. *Flat shading* refers to drawing a primitive such as a line segment or a triangle with a single color. *Gouraud shading* refers to drawing the primitive by interpolating the colors at the vertices of the primitive to fill in those pixels corresponding to the interior of the primitive. In theory, Gouraud shading is more expensive to compute because of the interpolation. However, with current graphics hardware, the performance is not an issue, so you tend not to use flat shading. I have provided support anyway, via class `ShadeState`. Its interface is

```
class ShadeState : public GlobalState
{
public:
    virtual int GetGlobalStateType () const { return SHADE; }

    ShadeState ();
    virtual ~ShadeState ();
```

```
    enum // ShadeMode
    {
        SM_FLAT,
        SM_SMOOTH,
        SM_QUANTITY
    };

    int Shade;  // default: SM_SMOOTH
};
```

The shading mode is either flat or smooth; the latter refers to Gouraud shading (the default).

3.4.2 Lights

Drawing objects using only textures results in renderings that lack the realism we are used to in the real world. Much of the richness our own visual systems provide is due to *lighting*. A graphics system must support the concept of lights, and of materials that the lights affect. The lighting models supported by standard graphics APIs are a simple approximation to true lighting, but are designed so that the lighting calculations can be performed quickly. More realistic lighting is found in systems that are almost never real time.

Materials were discussed earlier in this section. A material consists of various colors. The emissive color represents light that the material itself generates, which is usually none. *Ambient light* comes from light that has been scattered by the environment. A material reflects some of this light. The ambient color of the material indicates how much ambient light is reflected. Although referred to as a color, you may think of the ambient component as telling you the fraction of ambient light that is reflected. *Diffuse light* is light that strikes a surface. At each point on the surface the light arrives in some direction, but is then scattered equally in all directions at that point. A material also reflects some of this light. The diffuse color of the material indicates how much diffuse light is reflected. *Specular light* is also light that strikes a surface, but the reflected light has a preferred direction. The resulting appearance on the surface is referred to as *specular highlights*. A material reflects some of this light. The fractional amount is specified by the material's specular color.

The lights have physical attributes themselves, namely, colors (ambient, diffuse, specular), intensity, and attenuation (decrease in energy as the light travels over some distance). Lights also come in various types. I already mentioned ambient light due to scattering. The light has no single source and no specific direction. A source that provides light rays that are, for all practical purposes, parallel is referred to as a *directional light*. The motivation for this is sunlight. The Sun is far enough away from the Earth that the Sun's location is irrelevant. The sunlight is effectively unidirectional. A source that has a location, but emanates light in all directions is called a *point light*. The motivation is an incandescent light bulb. The filament acts as the light source, and the bulb emits light in all directions. A light that has a source location but emits lights in a restricted set of directions (typically a cone of directions) is called a *spot light*. The motivation is a flashlight or airport beacon that has a lightbulb as the source and a reflector surface that concentrates the light to emanate in a fixed set of directions.

The types of lights and their attributes are sufficiently numerous that many engines provide multiple classes. Usually an engine will provide an abstract base class for lights and then derived classes such as an ambient light class, a directional light class, a point light class, and a spot light class. I did so in Wild Magic version 2, but decided that the way the renderer accessed a derived-class light's information was more complicated than it needed to be. Also in Wild Magic version 2, the `Light` class was derived from `Object`. A number of users were critical of this choice and insisted that `Light` be derived from `Spatial`. By doing so, a light automatically has a location (the local translation) and an orientation (the local rotation). One of the orientation vectors can assume the role of the direction for a directional light. I chose not to derive `Light` from `Spatial` because ambient lights have no location or direction and directional lights have no location. In this sense they are not very spatial! The consequence, though, was that I had to add a class, `LightNode`, that was derived from `Node` and that had a `Light` data member. This allows point and spot lights to change location and directional lights to change orientation and then have the geometric update system automatically process them. Even these classes presented some problems to users. One problem had to do with importing LightWave objects into the engine because LightWave uses left-handed coordinates for everything. The design of `LightNode` (and `CameraNode`) prevented a correct import of lights (and cameras) when they were to be attached as nodes in a scene.

In the end, I decided to satisfy the users. In Wild Magic version 3, I changed my design and created a single class called `Light` that is derived from `Spatial`. Not all data members make sense for each light type, but so be it. When you manipulate a directional light, realize that setting the location has no effect. Also be aware that by deriving from `Spatial`, some subsystems are available to `Light` that are irrelevant. For example, attaching to a light a global state such as a depth buffer has no meaning, but the engine semantics allow the attachment. In fact, you can even attach lights to lights. You can attach a light as a leaf node in the scene hierarchy. For example, you might have a representation of a headlight in an automobile. A node is built with two children: One child is the `Geometry` object that represents the headlight's model data, and the other child is a `Light` to represent the light source for the headlight. The geometric data is intended to be drawn to visualize the headlight, but the light object itself is not renderable. The virtual functions for global state updates and for drawing are stubbed out in the `Light` class, so incorrect use of the lights should not be a problem. So be warned that you can manipulate a `Light` as a `Spatial` object in ways that the engine was not designed to handle.

The Light Class

The Light class has a quite complicated interface. I will look at portions of it at a time. The class supports the standard light types: ambient, directional, point, and spot.

```
class Light : public Spatial
{
public:
    enum // Type
    {
        LT_AMBIENT,
        LT_DIRECTIONAL,
        LT_POINT,
        LT_SPOT,
        LT_QUANTITY
    };

    Light (int iType = LT_AMBIENT);
    virtual ~Light ();

    int Type;             // default: LT_AMBIENT
    ColorRGBA Ambient;    // default: ColorRGBA(0,0,0,1)
    ColorRGBA Diffuse;    // default: ColorRGBA(0,0,0,1)
    ColorRGBA Specular;   // default: ColorRGBA(0,0,0,1)
    float Intensity;      // default: 1
    float Constant;       // default: 1
    float Linear;         // default: 0
    float Quadratic;      // default: 0
    bool Attenuate;       // default: false
    bool On;              // default: true

    // spot light parameters (valid only when Type = LT_SPOT)
    float Exponent;
    float Angle;
};
```

When you create a light, you specify the type you want. Each light has ambient, diffuse, and specular colors and an intensity factor that multiplies the colors. The member On is used to quickly turn a light on or off. This is preferred over attaching and detaching a light in the scene.

The data members Constant, Linear, Quadratic, and Attenuate are used for attenuation of point and spot lights over some distance. To allow attenuation, you set

`Attenuate` to `true`. The attenuation factor multiplies the light colors, just as the intensity factor does. The attenuation factor is

$$\alpha = \frac{1}{C + Ld + Qd^2}, \tag{3.8}$$

where C is the `Constant` value, L is the `Linear` value, and Q is the `Quadratic` value. The variable d is the distance from the light's position to a vertex on the geometric object to be lit.

The actual lighting model is somewhat complicated, but here is a summary of it. The assumption is that the object to be lit has a material with various colors. I will write these as vector-valued quantities (RGBA) for simplicity of the notation. Additions and multiplications are performed componentwise. The material emissive color is $\mathbf{M}_{\text{ems}}$, the ambient color is $\mathbf{M}_{\text{amb}}$, the diffuse color is $\mathbf{M}_{\text{dif}}$, and the specular color is $\mathbf{M}_{\text{spc}}$. The shininess is M_s, a nonnegative scalar quantity. A global ambient light is assumed (perhaps representing the Sun). This light is referred to by $\mathbf{L}^{(0)}$ and has subscripts for the colors just like the material colors use. In the engine, this light automatically exists, so you need not create one and attach it to the scene. The object to be lit may also be affected by n lights, indexed by $\mathbf{L}^{(i)}$ for $1 \le i \le n$. Once again these lights have subscripts for the colors. The ith light has a contribution to the rendering of

$$\alpha_i \sigma_i \left(\mathbf{A}^{(i)} + \mathbf{D}^{(i)} + \mathbf{S}^{(i)}\right).$$

The term α_i is the attenuation. It is calculated for point and spot lights using Equation (3.8). It is 1 for ambient and directional lights—neither of these is attenuated. The term σ_i is also an attenuator that is 1 for ambient, diffuse, and point lights, but potentially less than 1 for spot lights. The light has an ambient contribution,

$$\mathbf{A}^{(i)} = \mathbf{M}_{\text{amb}} \mathbf{L}^{(i)}_{\text{amb}},$$

a diffuse contribution,

$$\mathbf{D}^{(i)} = \mu^{(i)}_D \mathbf{M}_{\text{dif}} \mathbf{L}^{(i)}_{\text{dif}},$$

and a specular contribution,

$$\mathbf{S}^{(i)} = \mu^{(i)}_S \mathbf{M}_{\text{spc}} \mathbf{L}^{(i)}_{\text{spc}}.$$

The color assigned to a vertex on the object to be lit is

$$\mathbf{M}_{\text{ems}} + \mathbf{L}^{(0)}_{\text{amb}} \mathbf{M}_{\text{amb}} + \sum_{i=1}^{n} \alpha_i \sigma_i \left(\mathbf{A}^{(i)} + \mathbf{D}^{(i)} + \mathbf{S}^{(i)}\right). \tag{3.9}$$

The modulators $\mu_D^{(i)}$, $\mu_S^{(i)}$, and σ_i depend on the light type and geometry of the object. For an ambient light, the diffuse modulator is 1. For a directional light with unit-length world direction vector **U** and at a vertex with outer-pointing unit-length normal **N**,

$$\mu_D^{(i)} = \max\{-\mathbf{U} \cdot \mathbf{N}, 0\}.$$

The diffuse modulator is 1 when the light shines directly downward on the vertex. It is 0 when the light direction is tangent to the surface. For a point or spot light with position **P** and a vertex with position **V**, define the unit-length vector

$$\mathbf{U} = \frac{\mathbf{V} - \mathbf{P}}{|\mathbf{V} - \mathbf{P}|}. \tag{3.10}$$

This does assume the light is not located at the vertex itself. The diffuse modulator equation for a directional light also applies here, but with the vector **U** as defined.

The specular modulator is 1 for an ambient light. For the other light types, the specular modulator is computed based on the following quantities. Let **V** be the vertex location and **N** be a unit-length vertex normal. If the light is directional, let **U** be the unit-length direction. If the light is a point light or spot light, let **U** be the vector defined by Equation (3.10). The specular modulator is 0 if $-\mathbf{U} \cdot \mathbf{N} \leq 0$. Otherwise, define the unit-length vector[5]

$$\mathbf{H} = \frac{\mathbf{U} + (0, 0, -1)}{|\mathbf{U} + (0, 0, -1)|},$$

and the specular modulator is

$$\mu_S^{(i)} = (\max\{-\mathbf{H} \cdot \mathbf{N}, 0\})^{M_s}.$$

The spot light modulator σ_i is 1 for a light that is not a spot light. When the light is a spot light, the modulator is 0 if the vertex is not contained in the cone of the light. Otherwise, define **U** by Equation (3.10), where **P** is the spot light position. The modulator is

$$\sigma_i = (\max\{\mathbf{U} \cdot \mathbf{D}, 0\})^{e_i},$$

where **D** is the spot light direction vector (unit length) and e_i is the exponent associated with the spot light. The spot light angle $\theta_i \in [0, \pi)$ determines the cone of the light. The `Light` data members that correspond to these parameters are `Exponent` and `Angle`.

5. In OpenGL terminology, I do not use a local viewer for the light model. If you were to use a local viewer, then $(0, 0, -1)$ in the equation for **H** is replaced by $(\mathbf{P} - \mathbf{E})/|\mathbf{P} - \mathbf{E}|$, where **P** is the light position and **E** is the eye point (camera position).

The remaining portion of the Light interface is related to the class being derived from Spatial:

```
class Light : public Spatial
{
public:
    // light frame (local coordinates)
    //   default location  E = (0,0,0)
    //   default direction D = (0,0,-1)
    //   default up        U = (0,1,0)
    //   default right     R = (1,0,0)
    void SetFrame (const Vector3f& rkLocation,
        const Vector3f& rkDVector, const Vector3f& rkUVector,
        const Vector3f& rkRVector);
    void SetFrame (const Vector3f& rkLocation,
        const Matrix3f& rkAxes);
    void SetLocation (const Vector3f& rkLocation);
    void SetAxes (const Vector3f& rkDVector,
        const Vector3f& rkUVector, const Vector3f& rkRVector);
    void SetAxes (const Matrix3f& rkAxes);
    Vector3f GetLocation () const;  // Local.Translate
    Vector3f GetDVector () const;   // Local.Rotate column 0
    Vector3f GetUVector () const;   // Local.Rotate column 1
    Vector3f GetRVector () const;   // Local.Rotate column 2

    // For directional lights.  The direction vector must be
    // unit length.  The up vector and left vector are generated
    // automatically.
    void SetDirection (const Vector3f& rkDirection);

    // light frame (world coordinates)
    Vector3f GetWorldLocation () const;  // World.Translate
    Vector3f GetWorldDVector () const;   // World.Rotate column 0
    Vector3f GetWorldUVector () const;   // World.Rotate column 1
    Vector3f GetWorldRVector () const;   // World.Rotate column 2

private:
    // updates
    virtual void UpdateWorldBound ();
    void OnFrameChange ();

    // base class functions not supported
```

```
    virtual void UpdateState (TStack<GlobalState*>*,
        TStack<Light*>*) { /**/ }
    virtual void Draw (Renderer&, bool) { /**/ }

};
```

Normally, the local transformation variables (translation, rotation, scale) are for exactly that—transformation. In the `Light` class, the local translation is interpreted as the origin for a coordinate system of the light. The columns of the local rotation matrix are interpreted as the coordinate axis directions for the light's coordinate system. The choice for the default coordinate system is akin to the standard camera coordinate system relative to the screen: The center of the screen is the origin. The up vector is toward the top of the screen (the direction of the positive y-axis), the right vector is toward the right of the screen (the direction of the positive x-axis), and the z-axis points out of the screen. The light is positioned at the origin, has direction into the screen (the direction of the negative z-axis), has up vector to the top of the screen, and has right vector to the right of the screen. Because the light's coordinate system is stored in the local translation vector and local rotation matrix, you should use the interface provided and avoid setting the data member `Local` explicitly to something that is not consistent with the interpretation as a coordinate system.

The first block of code in the interface is for set/get of the coordinate system parameters. The `SetDirection` function is offset by itself just to draw attention to the fact that you are required to pass in a unit-length vector. As the comment indicates, the up and left vectors are automatically generated. Their values are irrelevant since the direction vector only pertains to a directional light, and the up and left vectors have no influence on the lighting model. The last block of code in the public interface allows you to retrieve the world coordinates for the light's (local) coordinate system.

The `Light` class has no model bound; however, the light's position acts as the center of a model bound of radius zero. The virtual function `UpdateWorldBound` computes the center of a world bound of radius zero. The function `OnFrameChange` is a simple wrapper around a call to `UpdateGS` and is executed whenever you set the coordinate system components. Therefore, you do not need to explicitly call `UpdateGS` whenever the coordinate system components are modified.

The two virtual functions in the private section are stubs to implement pure virtual functions in `Spatial` (as required by C++). None of these make sense for lights anyway, as I stated earlier, but they exist just so that `Light` inherits other properties of `Spatial` that are useful.

Support for Lights in Spatial and Geometry

The `Spatial` class stores a list of lights. If a light is added to this list, and the object really is of type `Node`, then my design choice is that the light illuminates all leaf geometry in the subtree rooted at the node. The portion of the interface for `Spatial` relevant to adding and removing lights from the list is

```
class Spatial : public Object
{
public:
    void SetLight (Light* pkLight);
    int GetLightQuantity () const;
    Light* GetLight (int i) const;
    void RemoveLight (Light* pkLight);
    void RemoveAllLights ();

protected:
    TList<Pointer<Light> >* m_pkLightList;
};
```

The list is considered to be unordered since Equation (3.9) does not require the lights to be ordered. Notice that the list contains smart pointers to lights. I use `typedef` to create aliases for the smart pointers. For `Light` it is `LightPtr`. The `TList` declaration cannot use `LightPtr`. The `typedef` for `LightPtr` is contained in `Wm3Light.h`. Class `Light` is derived from `Spatial`, so `Wm3Light.h` includes `Wm3Spatial.h`. If we were to include `Wm3Light.h` in `Wm3Spatial.h` to access the definition of `LightPtr`, we would create a circular header dependency, in which case the compiler has a complaint. To avoid this, the class `Light` is forward-declared and the `typedef` is not used.

Function `SetLight` checks to see if the input light is already in the list. If so, no action is taken. If not, the light is added to the front of the list. The function `GetLightQuantity` just iterates over the list, counts how many items it has, and returns that number. The function `GetLight` returns the ith light in the list. Together, `GetLightQuantity` and `GetLight` allow you to access the list as if it were an array. This is convenient, especially in the renderer code. The function `RemoveLight` searches the list for the input light. If it exists, it is removed. The list is singly linked, so the search uses two pointers, one in front of the other, in order to facilitate the removal. The function `RemoveAllLights` destroys the list by removing the front item repeatedly until the list is empty.

The `Geometry` class stores an array of lights, which is separate from the list. It stores smart pointers to all the lights encountered in a traversal from the root node to the geometry leaf node. These lights are used by the renderer to light the geometric object. The render state update, discussed later in this section, shows how the lights are propagated to the leaf nodes.

3.4.3 TEXTURES

The `Texture` class is designed to encapsulate most of the information needed to set up a texture unit on a graphics card. Minimally, the class should store the texture image.

The texture coordinates assigned to vertices of geometry objects are not stored in Texture, which allows sharing of Texture objects across multiple geometry objects.

In Wild Magic version 2, the texture coordinates were stored in the Geometry object itself. Having support for multitexturing meant that Geometry needed to store as many texture coordinate arrays as there are textures attached to the object, which caused the interface to Geometry to become unruly. On the release of each new generation of graphics cards that supported more texture units than the previous generation, I had to modify Geometry to include more array pointers for the texture coordinates. Naturally, the streaming code for Geometry had to be modified to store the new data and to load old files knowing that they were saved using a smaller number of texture units. Most Geometry objects used only one or two arrays of texture coordinates, but the class had to be prepared for the worst case that all arrays were used. The class even had a separate array to handle textures associated with bump maps. The bulkiness of Geometry regarding texture coordinates and the fact that its code evolved on a somewhat regular basis made me realize I needed a different design.

Wild Magic version 3 introduces a new class, Effect. The class, discussed in detail later in this section, now encapsulates the textures and corresponding texture coordinates that are to be attached to a Geometry object. An increase in the number of texture units for the next-generation hardware requires *no changes* to either the Geometry class or the Effect class. The Geometry class is responsible now only for vertex positions and normals and the indices that pertain to connectivity of the vertices. During the development of advanced features for Wild Magic version 3, the redesign of Spatial, Geometry, TriMesh, and the introduction of Effect has paid off. The core classes are generic enough and isolated sufficiently well that changes to other parts of the engine have not forced rewrites of those classes. This is an important aspect of any large library design—make certain the core components are robust and modular, protecting them from the evolution of the rest of the library that is built on top of them.

Now back to the Texture class. The texture image is certainly at the heart of the class. The relevant interface items for images are

```
class Texture : public Object
{
public:
    Texture (Image* pkImage = NULL);
    virtual ~Texture ();

    void SetImage (ImagePtr spkImage);
    ImagePtr GetImage () const;

protected:
    ImagePtr m_spkImage;
};
```

The only constructor acts as the default constructor, but also allows you to specify the texture image immediately. The data member `m_spkImage` is a smart pointer to the texture image. You may set/get the image via the accessors `SetImage` and `GetImage`.

A graphics system provides a lot of control over how the image is drawn on an object. I will discuss a portion of the interface at a time.

Projection Type

The first control is over the type of projection used for drawing the image:

```
class Texture : public Object
{
public:
    enum // CorrectionMode
    {
        CM_AFFINE,
        CM_PERSPECTIVE,
        CM_QUANTITY
    };

    int Correction;  // default: CM_PERSPECTIVE
};
```

The two possibilities are affine or perspective. The standard is to use perspective-correct drawing. Affine drawing was a popular choice for software rendering because it is a lot faster than using perspective-correct drawing. However, affine drawing looks awful! I tried to generate some images to show the difference using the Wild Magic OpenGL renderer, but apparently the current-generation hardware and OpenGL drivers ignore the hint to use affine texturing, so I could only obtain perspective-correct drawing. The option should remain, though, because on less powerful platforms, affine texturing is quite possibly necessary for speed—especially if you plan on implementing a software renderer using the Wild Magic API.

Filtering within a Single Image

A texture image is mapped onto a triangle by assigning texture coordinates to the vertices of the triangle. Once the triangle is mapped to screen space, the pixels inside the triangle must be assigned colors. This is done by linearly interpolating the texture coordinates at the vertices to generate texture coordinates at the pixels. It is possible (and highly likely) that the interpolated texture coordinates do not correspond exactly to an image pixel. For example, suppose you have a 4×4 texture image and a triangle with texture coordinates $(0, 0)$, $(1, 0)$, and $(0, 1)$ at its vertices. The pixel

(i, j) in the image, where $0 \le i < 4$ and $0 \le j < 4$, has the associated texture coordinates $(u, v) = (i/3, j/3)$. If a pixel's interpolated texture coordinates are (0.2, 0.8), the real-valued image indices are $i' = 3(0.2) = 0.6$ and $j' = 3(0.8) = 2.4$. I used prime symbols to remind you that these are not integers. You have to decide how to create a color from this pair of numbers and the image. The portion of the `Texture` interface related to the creation is

```
class Texture : public Object
{
public:
    enum // FilterMode
    {
        FM_NEAREST,
        FM_LINEAR,
        FM_QUANTITY
    };

    int Filter;  // default: FM_LINEAR
};
```

Two choices are available. By setting `Filter` to `FM_NEAREST`, the texturing system rounds the real-valued indices to the nearest integers. In this case, $i = 1$ since $i' = 0.6$ is closer to 1 than it is to 0, and $j = 2$ since $j' = 2.4$ is closer to 2 than it is to 3. The image value at location $(i, j) = (1, 2)$ is chosen as the color for the pixel.

The second choice for `Filter` is `FM_LINEAR`. The color for the pixel is computed using bilinear interpolation. The real-valued indices (i', j') fall inside a square whose four corners are integer-valued indices. Let $\lfloor v \rfloor$ denote the floor of v, the largest integer smaller than or equal to v. Define $i_0 = \lfloor i' \rfloor$ and $j_0 = \lfloor j' \rfloor$. The four corners of the square are (i_0, j_0), $(i_0 + 1, j_0)$, $(i_0, j_0 + 1)$, and $(i_0 + 1, j_0 + 1)$. The bilinear interpolation formula generates a color C' from the image colors $C_{i,j}$ at the four corners:

$$C' = (1 - \Delta_i)(1 - \Delta_j)C_{i_0, j_0} + (1 - \Delta_i)\Delta_j C_{i_0, j_0+1} + \Delta_i(1 - \Delta_j)C_{i_0+1, j_0} + \Delta_i \Delta_j C_{i_0+1, j_0+1},$$

where $\Delta_i = i' - i_0$ and $\Delta_j = j' - j_0$. Some attention must be given when $i_0 = n - 1$ when the image has n columns. In this case, $i_0 + 1$ is outside the image domain. Special handling must also occur when the image has m rows and $j_0 = m - 1$.

Figure 3.10 shows a rectangle with a checkerboard texture. The object is drawn using nearest-neighbor interpolation. Notice the jagged edges separating gray and black squares.

Figure 3.11 shows the same rectangle and checkerboard texture. The edges are smoothed using bilinear interpolation. For reference later, notice that the edges near the top of the image still have a small amount of jaggedness.

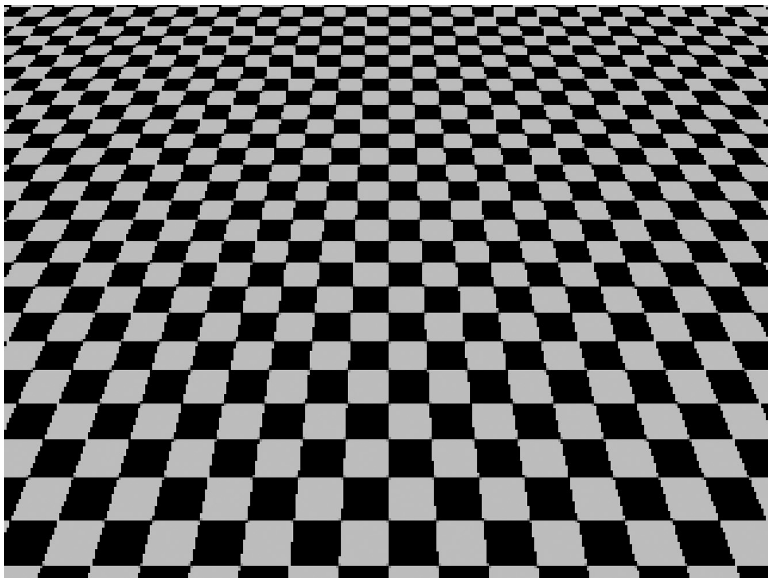


Figure 3.10 Illustration of nearest-neighbor interpolation using `Texture::FM_NEAREST`.

Mipmapping: Filtering within Multiple Images

Bilinear filtering produces better-quality texturing than choosing the nearest neighbor, but it comes at greater expense in computational time. Fortunately with graphics hardware support, this is not an issue. Even with bilinear filtering, texturing can still have some artifacts. When a textured object with bilinear interpolation is close to the eye point, most of the pixels obtain their colors from the interpolation. That is, the texture coordinates of the pixels are strictly inside the square formed by four texture image samples, so the pixel colors are always influenced by four samples. The texture image samples are referred to as *texels*.[6] For the close-up object, the texels are in a sense a lot larger than the pixels. This effect is sometimes referred to as *magnification*. The texture image is magnified to fill in the pixels.

When that same object is far from the eye point, aliasing artifacts show up. Two adjacent pixels in the close-up object tend to be in the same four-texel square. In the

6. The word *pixel* is an abbreviation of the words "picture element." Similarly, the word *texel* represents the words "texture element." The names people choose are always interesting! It certainly is easier to say pixel and texel than the original phrases.

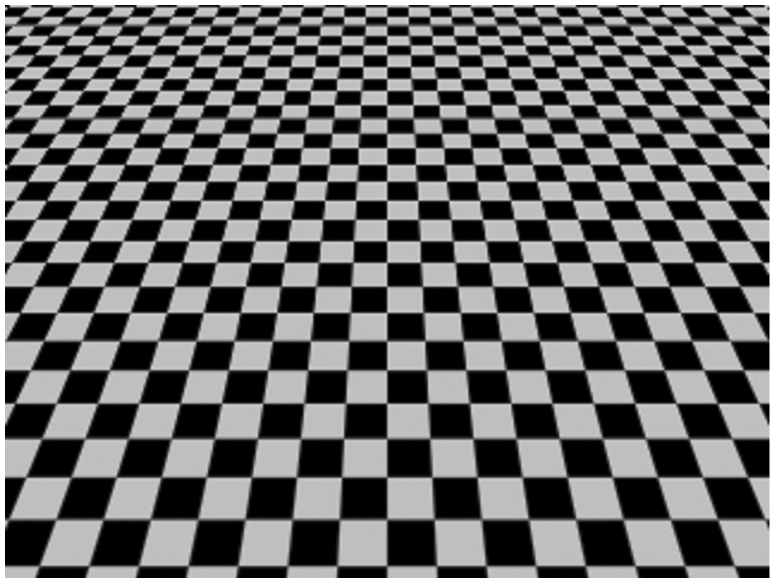


Figure 3.11 Illustration of bilinear interpolation using `Texture::FM_LINEAR`.

faraway object, two adjacent pixels tend to be in different four-texel squares. In this situation, the texels are in a sense a lot smaller than the pixels.

To eliminate the aliasing artifacts, an alternative is needed that is the reverse of magnification, *minification*. The idea is to generate a set of texture images from the original. A pixel in the new image is an average of a 2×2 block of pixels in the old image. Thus, each new image is half the size per dimension of the previous image. A full pyramid of images starts with the original, a $2^n \times 2^m$ image. The next image has dimensions $2^{n-1} \times 2^{m-1}$. The averaging process is repeated until one of the dimensions is 1. For a square image $n = m = 1$, the final image is 1×1 (a single texel). Pixels corresponding to a portion of the object close to the eye point are selected from the original image. For pixels corresponding to a portion of the object further away from the eye point, a selection must be made about which of the pyramid images to use. The alternatives are even more varied because you can choose to use nearest-neighbor interpolation or bilinear interpolation within a single image *and* you can choose to use the nearest image slice or linearly interpolate between slices. The process of texturing with a pyramid of images is called *mipmapping* [Wil83]. The prefix *mip* is an acronym for the Latin *multum in parvo*, which means "many things in a small place." The pyramid itself is referred to as the *mipmap*.

As mentioned, there are a few choices for how mipmapping is applied. The interface of `Texture` supporting these is

```
class Texture : public Object
{
public:
    enum // MipmapMode
    {
        MM_NEAREST,
        MM_LINEAR,
        MM_NEAREST_NEAREST,
        MM_NEAREST_LINEAR,
        MM_LINEAR_NEAREST,
        MM_LINEAR_LINEAR,
        MM_QUANTITY
    };

    int Mipmap;  // default: MM_LINEAR_LINEAR
};
```

The enumerated values that are assigned to the data member `Mipmap` refer to the following algorithms:

- `MM_NEAREST`: Only the original texture image is used, so the pyramid of images is not constructed. Nearest-neighbor interpolation is used to select the texel that is nearest to the target pixel.
- `MM_LINEAR`: Only the original texture image is used, so the pyramid of images is not constructed. Bilinear interpolation is used to generate the color for the target pixel.

The next four options do require building the mipmap. The graphics drivers provide some mechanism for selecting which image in the mipmap to use. That mechanism can vary with graphics API and/or manufacturer's graphics cards, so I do not discuss it here, but see [Ebe00] for details.

- `MM_NEAREST_NEAREST`: The mipmap image nearest to the pixel is selected. In that image, the texel nearest to the pixel is selected and assigned to the pixel.
- `MM_NEAREST_LINEAR`: The two mipmap images that bound the pixel are selected. In each image, the texel nearest to the pixel is selected. The two texels are linearly interpolated and assigned to the pixel.
- `MM_LINEAR_NEAREST`: The mipmap image nearest to the pixel is selected. In that image, bilinear interpolation of the texels is used to produce the pixel value.
- `MM_LINEAR_LINEAR`: The two mipmap images that bound the pixel are selected. In each image, bilinear interpolation of the texels is used to generate two colors. Those colors are linearly interpolated to produce the pixel value. This is sometimes call *trilinear interpolation*.

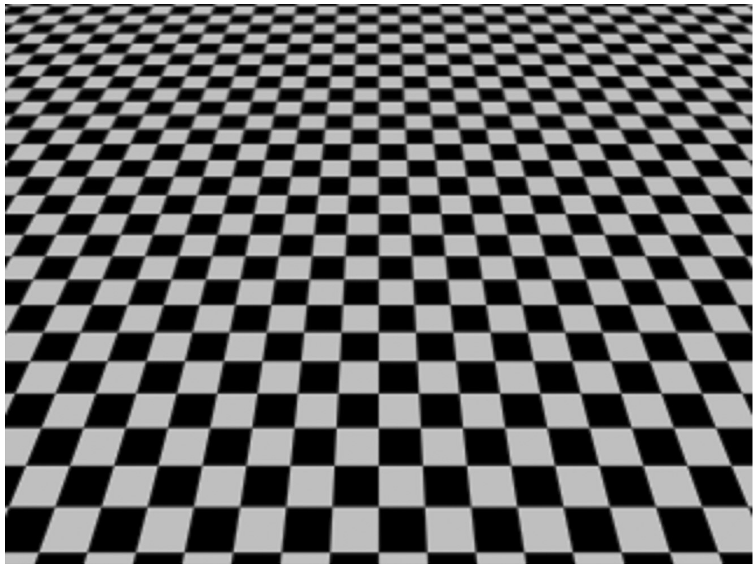


Figure 3.12 Illustration of trilinear interpolation using `Texture::MM_LINEAR_LINEAR`.

Note that for each enumerated value, the first name refers to the interpolation type within an image. The second name refers to the interpolation type across two images.

In theory, `MM_NEAREST` and `MM_LINEAR` use only the original texture image, so mipmaps need not (and should not) be built. In fact, the choices are equivalent to using single-image filtering along. The OpenGL renderer indeed does not generate the mipmaps; the result is that standard filtering is used (as specified by the `Filter` data member).[7]

Figure 3.12 shows the rectangle and checkerboard texture using trilinear interpolation for mipmapping. The slightly jagged edges that appear in the top half of Figure 3.11 do not appear in the top half of Figure 3.12.

7. That said, if you have had much experience with graphics drivers for different brands of graphics cards, you will find that the drivers do not always adhere to the theory. In a test program for `MM_LINEAR`, one of my graphics cards rendered an image that should have been identical to Figure 3.11, but instead rendered an image that showed a small, triangular shaped, bilinearly interpolated region near the bottom of the image. The remainder of the image showed that nearest- neighbor interpolation was used. A graphics card from a different manufacturer correctly rendered the image entirely using bilinear interpolation.

Out-of-Range Texture Coordinates

In the discussion of bilinear interpolation for texture image filtering, I mentioned that special attention must be paid to interpolation at texels that are on the boundary of the image. The natural inclination is to *clamp* values outside the domain of the image indices. If a texture coordinate is (0.7, 1.1), the clamped value is (0.7, 1.0). The texture coordinate (−0.4, 0.2) is clamped to (0.0, 0.2). Other choices are possible. The coordinates may be *repeated* by using modular arithmetic. Any value larger than 1 has its integer part removed, and any value smaller than 0 has an integer added to it until the result is 0 or larger. For example, (0.7, 1.1) is wrapped to (0.7, 0.1), and (−0.4, 0.2) is wrapped to (0.6, 0.2). In the latter example, we only needed to add 1 to −0.4 to obtain a number in the interval [0, 1]. The texture coordinate (−7.3, 0.2) is wrapped to (0.7, 0.2). In this example, we had to add 8 to −7.3 to obtain a number in the interval [0, 1].

The handling of texture coordinates at the image boundaries is supported by the interface

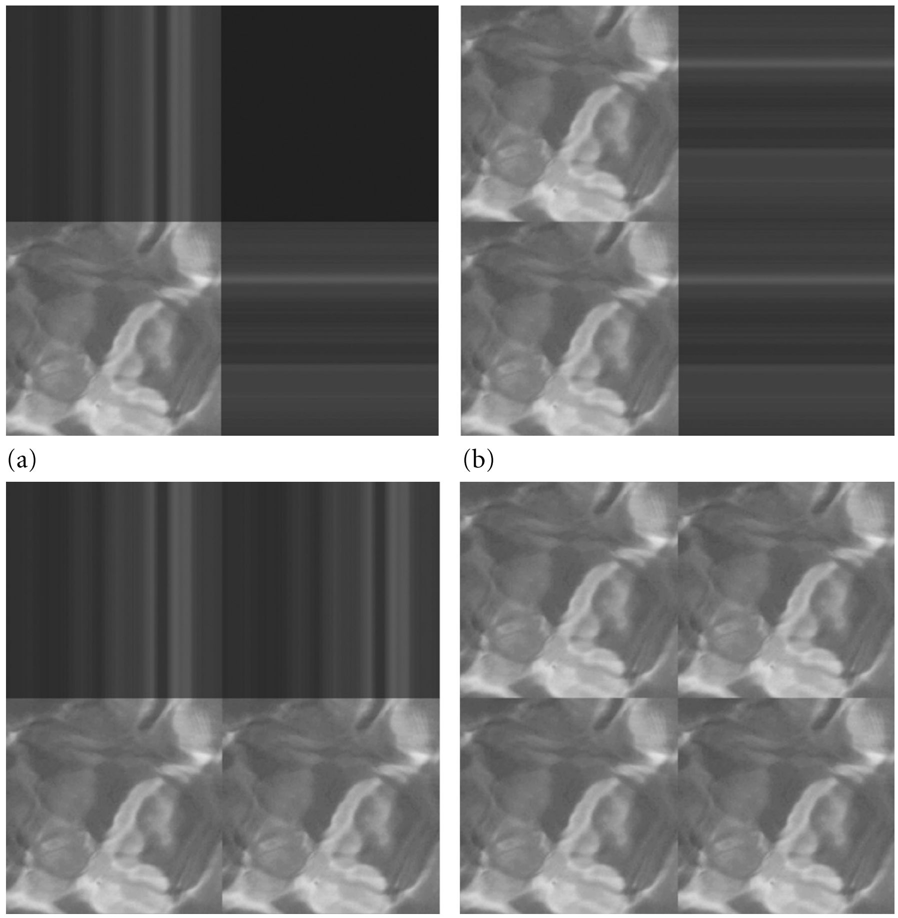
```
class Texture : public Object
{
public:
    enum // CoordinateMode
    {
        WM_CLAMP,
        WM_REPEAT,
        WM_CLAMP_BORDER,
        WM_CLAMP_EDGE,
        WM_QUANTITY
    };

    ColorRGBA BorderColor;  // default: ColorRGBA(0,0,0,0)

    int CoordU;  // default: WM_CLAMP_EDGE
    int CoordV;  // default: WM_CLAMP_EDGE
};
```

Given a texture coordinate (u, v), each component can be clamped or repeated. Figure 3.13 illustrates the four possibilities.

Unfortunately, the `WM_CLAMP` mode has issues when the filter mode is not `FM_NEAREST` and/or the mipmap mode is not `MM_NEAREST`. When bilinear interpolation is used and the texture coordinates are on the image boundary, the interpolation uses the *border color*, which is stored in the data member `Texture::BorderColor`. The OpenGL renderer is set up to tell the graphics API about the border color only if that color is valid. When it is invalid, a black border color is used by default. Figure 3.14 shows the effect when two textures on two objects have a common boundary. In

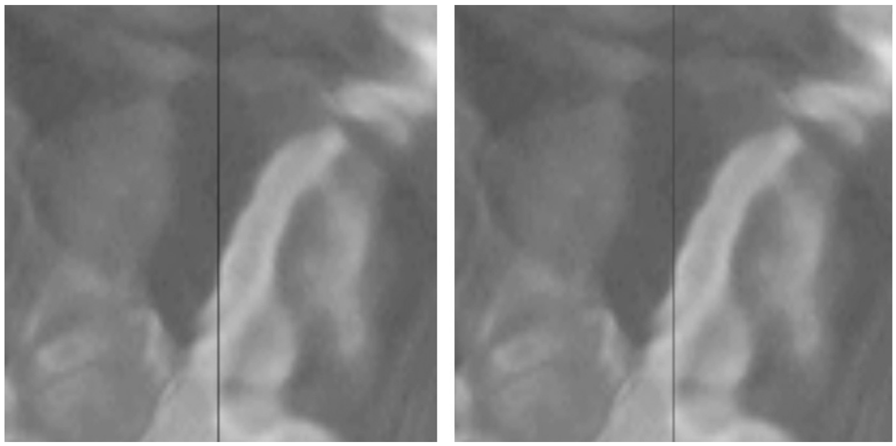


(a) (b) (c) (d)

Figure 3.13 A square with vertices $(-1, -1)$, $(1, -1)$, $(1, 1)$, and $(-1, 1)$ is drawn with a texture image. The texture coordinates at the square's corners are $(0, 0)$, $(2, 0)$, $(2, 2)$, and $(0, 2)$. (a) Clamp u and clamp v. (b) Clamp u and repeat v. (c) Repeat u and clamp v. (d) Repeat u and repeat v. (See also Color Plate 3.13.)

either case, if you have a tiled environment such as terrain, the clamping to the border color is not the effect you want.

Instead, use clamping to the edge of the texture. Figure 3.15 illustrates with the same squares and texture coordinates as in Figure 3.14. In Figure 3.15(a), notice that the dark line that appeared with border clamping no longer occurs. However, you will



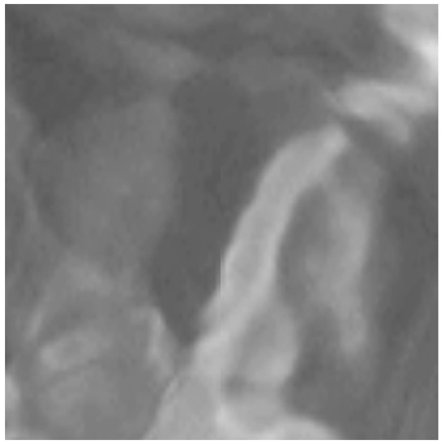
(a) (b)

Figure 3.14 Two squares, one with vertices $(-1, -1)$, $(0, -1)$, $(0, 1)$, and $(-1, 1)$, and one with vertices $(0, -1)$, $(1, -1)$, $(1, 1)$, and $(0, 1)$, are drawn with texture images. The images were obtained by taking a 128×128 bitmap and splitting it into 64×128 images. The texture coordinates at the squares' corners are $(0, 0)$, $(1, 0)$, $(1, 1)$, and $(0, 1)$. (a) Clamp u and v to border, no border color assigned. (b) Clamp u and v to border, red border color assigned. (See also Color Plate 3.14).

notice in the middle of the image about one-third of the distance from the bottom a discontinuity in the image intensities. The bilinear interpolation and handling of texture coordinates is causing this. Figure 3.15(b) shows how to get around this problem. The discontinuity is much less noticeable. The left edge of the texture on the left duplicates the right edge of the texture on the right. When tiling terrain, you want to generate your textures to have color duplication on shared boundaries.

Automatic Generation of Texture Coordinates

Some rendering effects require the texture coordinates to change dynamically. Two of these are environment mapping, where an object is made to appear as if it is reflecting the environment around it, and projected textures, where an object has a texture applied to it as if the texture were projected from a light source. A graphics API can provide the services for updating the texture coordinates instead of requiring the application to explicitly do this. When creating textures of these types, you may use the following interface for `Texture`:


(a)

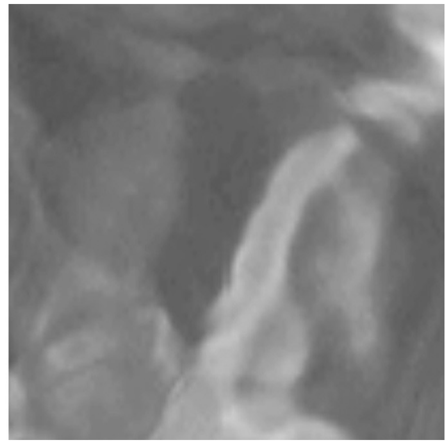

(b)

Figure 3.15 (a) Clamp u and v to edge, border color is always ignored. (b) Clamp u and v to edge, but textures were created differently. (See also Color Plate 3.15.)

```
class Texture : public Object
{
public:
    enum // TexGenMode
    {
        TG_NONE,
        TG_ENVIRONMENT_MAP,
        TG_PROJECTED_TEXTURE,
        TG_QUANTITY
    };

    int Texgen;  // default: TG_NONE
};
```

I only support the two aforementioned effects, but if you add new ones, you will need to add more enumerated values to the class. You should add these after the `TG_PROJECTED_TEXTURE` item, but before the `TG_QUANTITY` item, to guarantee that the streaming system loads already saved files correctly. In other words, if you insert a new enumerated value elsewhere, you will cause a change in the implicit numbering of the values, thereby invalidating the numbers that were saved in previous streaming operations to disk.

The `Effect` system that I describe later already has derived classes to support environment mapping and projected textures, so there is no need for you to explicitly manipulate the `Texgen` data member. You can just provide the texture image to `EnvironmentMapEffect` and `ProjectedTextureEffect` and attach the effect to a node.

Application Mode

The `Texture` class has enumerated values that specify how a texture is to be applied to an object. This is called the *apply mode*:

```
class Texture : public Object
{
public:
    enum // ApplyMode
    {
        AM_REPLACE,
        AM_DECAL,
        AM_MODULATE,
        AM_BLEND,
        AM_ADD,
        AM_COMBINE,
        AM_QUANTITY
    };

    int Apply;  // default: AM_REPLACE
};
```

For a single texture, the mode you want is `AM_REPLACE`. This tells the graphics system to just draw the texture image on the object. Any colors that were drawn to the object pixels previously are *replaced* by those from the texture image.

The other enumerated values have to do with the blending of multiple textures onto an object, the topic of the next subsection.

3.4.4 Multitexturing

The term *multitexturing* refers to drawing an object with two or more texture images. A classic application is light maps. A primary texture is drawn on the object, and a secondary texture representing variations in light intensity is then applied to the object. The colors from the primary and secondary texture images must be combined somehow, much like the blending that occurs with `AlphaBlending`.

The `Texture` class has a variable for storing an apply mode. The relevant interface is

Table 3.4 Blending equations for the apply mode values. The alpha channel is handled separately from the red, green, and blue channels.

Apply mode/ image type	*RGB*		*RGBA*	
AM_REPLACE	$\mathbf{C}_t$	A_f	$\mathbf{C}_t$	A_t
AM_DECAL	$\mathbf{C}_t$	A_f	$(1 - A_t)\mathbf{C}_f + A_t\mathbf{C}_t$	A_f
AM_MODULATE	$\mathbf{C}_t\mathbf{C}_f$	A_f	$\mathbf{C}_t\mathbf{C}_f$	A_tA_f
AM_BLEND	$(\mathbf{1} - \mathbf{C}_t)\mathbf{C}_f + \mathbf{C}_t\mathbf{C}_c$	A_f	$(\mathbf{1} - \mathbf{C}_t)\mathbf{C}_f + \mathbf{C}_t\mathbf{C}_c$	A_tA_f
AM_ADD	$\mathbf{C}_t + \mathbf{C}_f$	A_f	$\mathbf{C}_t + \mathbf{C}_f$	A_tA_f

```
class Texture : public Object
{
public:
    enum // ApplyMode
    {
        AM_REPLACE,
        AM_DECAL,
        AM_MODULATE,
        AM_BLEND,
        AM_ADD,
        AM_COMBINE,
        AM_QUANTITY
    };

    ColorRGBA BlendColor;   // default: ColorRGBA(0,0,0,1)

    int Apply;  // default: AM_REPLACE
};
```

I already discussed that objects to be drawn with a single texture, and no vertex colors or material colors, should use `AM_REPLACE`. The remaining enumerated values have to do with blending the texture image colors with other quantities. Wild Magic supports 24-bit RGB and 32-bit RGBA images. The apply modes perform blending according to Table 3.4.

The vector arguments are RGB colors, $\mathbf{C}_s = (r_s, g_s, b_s)$ for source index $s \in \{t, c, f\}$. The arguments A_s are alpha values. The *texture color* $(\mathbf{C}_t, A_t)$ comes from the RGBA image associated with the texture unit. The color itself may be filtered based on the mode assigned to the data member `Texture::Filter`. The *primary color*

$(\mathbf{C}_f, A_f)$ comes from vertex colors or material colors (interpolated across the triangles, of course). The primary colors are computed before any texturing is applied. The *constant color* $(\mathbf{C}_c, A_c)$ is the color assigned to `Texture::BlendColor`.

The mode that gives you full control over the blending is `AM_COMBINE`. When the renderer encounters a texture object, it passes its information along to the graphics API. If the combine mode is in effect, the graphics API must be told what the blending equation should be. The equations are not in themselves complicated, but the colors to be blended can come from a variety of sources. The `Texture` class has additional information that you must set in order to control the blending equation, the sources, and the operations among the sources. The relevant portion of the interface when `Apply` is set to `Texture::AM_COMBINE` is

```
class Texture : public Object
{
public:
    enum // ApplyCombineFunction
    {
        ACF_REPLACE,
        ACF_MODULATE,
        ACF_ADD,
        ACF_ADD_SIGNED,
        ACF_SUBTRACT,
        ACF_INTERPOLATE,
        ACF_DOT3_RGB,
        ACF_DOT3_RGBA,
        ACF_QUANTITY
    };

    enum // ApplyCombineSrc
    {
        ACS_TEXTURE,
        ACS_PRIMARY_COLOR,
        ACS_CONSTANT,
        ACS_PREVIOUS,
        ACS_QUANTITY
    };

    enum // ApplyCombineOperand
    {
        ACO_SRC_COLOR,
        ACO_ONE_MINUS_SRC_COLOR,
        ACO_SRC_ALPHA,
        ACO_ONE_MINUS_SRC_ALPHA,
        ACO_QUANTITY
    };
```

```
    enum // ApplyCombineScale
    {
        ACSC_ONE,
        ACSC_TWO,
        ACSC_FOUR,
        ACSC_QUANTITY
    };

    int CombineFuncRGB;     // default: ACF_REPLACE
    int CombineFuncAlpha;   // default: ACF_REPLACE
    int CombineSrc0RGB;     // default: ACS_TEXTURE
    int CombineSrc1RGB;     // default: ACS_TEXTURE
    int CombineSrc2RGB;     // default: ACS_TEXTURE
    int CombineSrc0Alpha;   // default: ACS_TEXTURE
    int CombineSrc1Alpha;   // default: ACS_TEXTURE
    int CombineSrc2Alpha;   // default: ACS_TEXTURE
    int CombineOp0RGB;      // default: ACO_SRC_COLOR
    int CombineOp1RGB;      // default: ACO_SRC_COLOR
    int CombineOp2RGB;      // default: ACO_SRC_COLOR
    int CombineOp0Alpha;    // default: ACO_SRC_COLOR
    int CombineOp1Alpha;    // default: ACO_SRC_COLOR
    int CombineOp2Alpha;    // default: ACO_SRC_COLOR
    int CombineScaleRGB;    // default: ACSC_ONE
    int CombineScaleAlpha;  // default: ACSC_ONE
};
```

The parameters and names are quite daunting, but once you understand how these are used to generate a blending equation, you should find these useful for advanced multitexturing effects.

The parameter `CombineFuncRGB` lets you specify the blending of the red, green, and blue colors. The alpha channel is handled by a separate function specified by `CombineFuncAlpha`. Table 3.5 lists the possible blending equations based on the selection of `CombineFuncRGB` and `CombineFuncAlpha`. The table omits the entries `ACF_DOT3_RGB` and `ACF_DOT3_RGBA`; these are used for bump mapping, the topic of Section 5.1.6. The arguments can be scalars (alpha values) or 3-tuples (RGB values). Any operations between two 3-tuples are performed componentwise.

The `CombineSrc[i]` and `CombineOp[i]` parameters determine what the V_i values are for $i \in \{0, 1, 2\}$. Table 3.6 lists the possible V_i values. The vector arguments are RGB colors, $\mathbf{C}_s = (r_s, g_s, b_s)$ for source index $s \in \{t, c, f, p\}$. The arguments A_s are alpha values. The *texture color* $(\mathbf{C}_t, A_t)$ comes from the RGBA image associated with the texture unit. The color itself may be filtered based on the mode assigned to the data member `Texture::Filter`. The *primary color* $(\mathbf{C}_f, A_f)$ comes from vertex colors or material colors (interpolated across the triangles, of course). The primary colors are computed before any texturing is applied. The *constant color* $(\mathbf{C}_c, A_c)$ is the color assigned to `Texture::BlendColor`. The *previous color* $(\mathbf{C}_p, A_p)$ is the output from the

Table 3.5 Combine functions and their corresponding blending equations.

Combine function	*Blending equation*
ACF_REPLACE	V_0
ACF_MODULATE	$V_0 * V_1$
ACF_ADD	$V_0 + V_1$
ACF_ADD_SIGNED	$V_0 + V_1 - \frac{1}{2}$
ACF_SUBTRACT	$V_0 - V_1$
ACF_INTERPOLATE	$V_0 * V_2 + V_1 * (1 - V_2)$

Table 3.6 The pair `CombineSrc[i]` and `CombineOp[i]` determine the argument V_i.

Src/op	ACO_SRC_COLOR	ACO_ONE_MINUS_ SRC_COLOR	ACO_SRC_ALPHA	ACO_ONE_MINUS_ SRC_ALPHA
ACS_TEXTURE	$\mathbf{C}_t$	$\mathbf{1} - \mathbf{C}_t$	A_t	$1 - A_t$
ACS_PRIMARY_COLOR	$\mathbf{C}_f$	$\mathbf{1} - \mathbf{C}_f$	A_f	$1 - A_f$
ACS_CONSTANT	$\mathbf{C}_c$	$\mathbf{1} - \mathbf{C}_c$	A_c	$1 - A_c$
ACS_PREVIOUS	$\mathbf{C}_p$	$\mathbf{1} - \mathbf{C}_p$	A_p	$1 - A_p$

texture unit previous to the current one. If the current texture unit is unit 0, then the previous color is the same as the primary color; that is, the inputs to texture unit 0 are the vertex colors or material colors.

After the blending equation is computed, it is possible to magnify the resulting color by a scaling factor of 1 (keep the resulting color), 2, or 4. If any color channel of the scaled color is greater than 1, it is clamped to 1. You may select the scaling parameter by setting `CombineScaleRGB` and `CombineScaleAlpha`. The valid parameters to assign are `ACSC_ONE`, `ACSC_TWO`, or `ACSC_FOUR`.

As an example, consider Equation (3.7), which blends a light map with a base texture, but avoids the oversaturation that a simple addition tends to produce. That equation is rewritten as

$$V_0 * V_2 + V_1 * (1 - V_2) = \mathbf{1} * \mathbf{C}_d + \mathbf{C}_s * (\mathbf{1} - \mathbf{C}_d),$$

where $\mathbf{C}_d$ is a base texture color and $\mathbf{C}_s$ is a light map color. The vector $\mathbf{1}$ represents the color white. The `Texture::ACF_INTERPOLATE` function is the one to use. The following code block sets up the combine function, sources, and operands to obtain the desired effect:

```
Texture* pkBase = <goes in texture unit 0>;
Texture* pkLightMap = <goes in texture unit 1>;

// draw the base texture onto the triangle first
pkBase->Apply = Texture::AM_REPLACE;

// use the interpolate combine function
pkLightMap->Apply = Texture::AM_COMBINE;
pkLightMap->CombineFuncRGB = Texture::ACF_INTERPOLATE;

// V0 = (1,1,1)
pkLightMap->BlendColor = ColorRGBA::WHITE;
pkLightMap->CombineSrc0RGB = Texture::ACS_CONSTANT;
pkLightMap->CombineOp0RGB = Texture::ACO_SRC_COLOR;

// V1 = base texture (previous texture unit values)
pkLightMap->CombineSrc1RGB = Texture::ACS_PREVIOUS;
pkLightMap->CombineOp1RGB = Texture::ACO_SRC_COLOR;

// V2 = light map (current texture unit values)
pkLightMap->CombineSrc2RGB = Texture::ACS_TEXTURE;
pkLightMap->CombineOp2RGB = Texture::ACO_SRC_COLOR;
```

The simple addition $V_0 + V_1$ can be controlled by a combine function:

```
// draw the base texture onto the triangle first
pkBase->Apply = Texture::AM_REPLACE;

// use the add function
pkLightMap->Apply = Texture::AM_COMBINE;
pkLightMap->CombineFuncRGB = Texture::ACF_ADD;

// V0 = base texture
pkLightMap->CombineSrc0RGB = Texture::ACS_PREVIOUS;
pkLightMap->CombineOp0RGB = Texture::ACO_SRC_COLOR;

// V1 = light map
pkLightMap->CombineSrc1RGB = Texture::ACS_TEXTURE;
pkLightMap->CombineOp1RGB = Texture::ACO_SRC_COLOR;
```

However, the apply mode AM_ADD works as well:

```
// draw the base texture onto the triangle first
pkBase->Apply = Texture::AM_REPLACE;

// add the light map to the base texture
pkLightMap->Apply = Texture::AM_ADD;
```

As you can see, there are many ways you can obtain the same effect.

3.4.5 Effects

The *effects system* in Wild Magic version 3 is a new invention to the engine. Wild Magic version 2 had a base class RenderState that encapsulated what I now call global states, lights, and texture information. In both versions of the engine, the Texture class stores information to configure the texture units on the graphics hardware and also stores a smart pointer to the texture image. In Wild Magic version 2, I had a class TextureState that stored an array of Texture objects, supporting multitexturing in a sense. A TextureState could be attached to a Node. All geometry leaf objects in the subtree rooted at the node used the Texture objects of the TextureState. On the other hand, the Geometry objects stored their own texture coordinates. To configure the texture units, the renderer needed to obtain the texture image and setup information from the TextureState object and texture coordinates from the Geometry object.

In a multitexturing situation, a further complication was that one TextureState could store the base texture in slot 0 of the array and be attached to one node in the scene hierarchy. Another TextureState could store the secondary texture and be attached to another node. The idea is that the accumulation of the texture states along a path from the root node to a leaf would lead to an array of Texture objects to be used in the multitexturing. The accumulation maintained an array whose slot 0 stored the Texture from slot 0 of any TextureState encountered along the path. In the current example, that means the TextureState storing the secondary texture cannot store it in slot 0; otherwise one of the texture objects hides the other. That means storing the secondary texture in, say, slot 1. The consequence of this design is that the application writer has to be very careful (and has the responsibility) about how to fill the slots in the TextureState. As some contractors added special effects to Wild Magic, the problems with my design became apparent.

In particular, projected textures were a problem. A projected texture is intended to be the last texture applied to a geometric object. Wild Magic version 2 has a ProjectedTexture class that is derived from Node. The class has a TextureState data member to store the Texture object corresponding to the projected texture. The intent was to create a specialized node to be an interior node of the scene hierarchy, and to have its texture be the projected texture for all geometry objects at the leafs of the subtree of the node. The dilemma was which slot in the array of TextureState to place the projected texture. Not knowing the slots used for the textures for the geometry leaf nodes, the only reasonable slot was the last one so as not to hide the textures used by the geometry leaf nodes. But this choice led to yet another problem. If there were four slots, the projected texture was placed in slot 3 (zero-based indexing). Now if a geometry object has only a single texture, placed in slot 0, then the renderer is given an object to draw using two textures. The renderer implementation was set up in a very general manner to iterate through the final array of textures and configure each texture unit accordingly. The texture unit 0 (for the base texture in slot 0) is set up, but texture units 1 and 2 are not used. Texture unit 1 had to be told to pass the output from texture unit 0 without changing it. Similarly, texture unit 2 had to pass the output from texture unit 1 without changing it. Texture unit 3 used the output from texture unit 2 and blended it with the projected texture that was assigned to texture unit 3. Clearly, this is an inefficiency that resulted from a substandard design in the scene graph management front end.

To remedy this for Wild Magic version 3, I scrapped the idea of having lights managed by a LightState object and textures managed by a TextureState object. Regarding textured objects, the renderer should be provided with the geometric information (vertices, normals, indices, transformations), texture information (texture images and texture coordinates), color information (vertex colors), lighting information (lights and material), and any semantics on how to combine these. The scene graph management system has to decide how to package these quantities to send them to the renderer. The packaging should require as little work as possible from the application writer, yet allow the renderer to efficiently gather the information and configure the texture units. The semantics for the configuration should not be exposed to the application writer, as was the projected texture example in Wild Magic version 2 (i.e., having to decide in which array slots to place the texture objects).

The effort to achieve these goals led to a redesign of the core classes Spatial, Geometry, and Node, and to the creation of a base class Effect. Information such as texture objects (images and configuration information), texture coordinates, and vertex colors are stored in Effect. The initial design change was to allow global states to be applied at interior nodes of a scene hierarchy, but allow only "local effects" to be applied at the leaf nodes. The Effect should encapsulate all the relevant information and semantics for producing a desired visual result. Many of the special effects that were added to Wild Magic version 2 as Node-derived classes were replaced by Effect objects that apply only to the geometry objects to which they are attached. However, projected textures were still problematic with regard to the new system. The projected textures usually are applied to a collection of geometric objects in the scene, not just to one. Having a projected texture affect an entire subtree of a scene is still desirable. A small redesign midstream led to allowing "global effects" (such as projected textures, projected shadows, and planar reflections) to occur at interior nodes, yet still based on the premise of Effect encapsulating the drawing attributes, and still leading to a general but efficient renderer implementation.

The discussion of global effects is postponed until Section 3.5.6, at which time I will discuss *multipass* operations. Such operations involve traversing through portions of the scene multiple times. Be aware that *multitexturing* refers to the use of multiple textures on an object. Many of the rendering effects can use single-pass multitexturing. Multipass can involve a single texture, or it can involve multiple textures. In the remainder of this section, the mechanism for local effects is described.

The interface for class `Effect` is

```
class Effect : public Object
{
public:
    Effect ();
    virtual ~Effect ();

    // Create a clone of the effect.  Colors and textures are
    // shared.  Each derived class can override this behavior and
    // decide what is copied and what is shared.
    virtual Effect* Clone ();

    // data common to many effects
    ColorRGBArrayPtr ColorRGBs;
    ColorRGBAArrayPtr ColorRGBAs;
    TArray<TexturePtr> Textures;
    TArray<Vector2fArrayPtr> UVs;

// internal use
public:
    // function required to draw the effect
    Renderer::DrawFunction Draw;
};
```

The class has storage for vertex colors, either RGB or RGBA, but not both. If you happen to set both, the RGBA colors will be used. Storage for an array of `Texture` objects is provided. Storage for an array of corresponding texture coordinates is also provided. Usually the arrays should have the same quantity of elements, but that is not necessary if the graphics system is asked to automatically generate the texture coordinates that are associated with a texture object.

The class has a function pointer data member—a pointer to some drawing function in the class `Renderer` interface. Many of the standard drawing operations are handled by `Renderer::DrawPrimitive`, but others require special handling. For example, environment mapping is implemented in the `Renderer` function `DrawEnvironmentMap`. A derived class will implement its constructors to assign the correct function pointer. The application writer should not manipulate this member.

The base class is not abstract. This allows you to create an `Effect` object and set the colors and textures as desired. In particular, if you have a special effect that involves a fancy combination of textures, you can do this without having to derive a class from `Effect` to manage the new feature. However, if the desired effect requires specialized handling by the renderer via a new drawing function in the `Renderer` interface, then you will need to derive a class from `Effect` and implement the drawing function in the derived renderer classes.

The `Spatial` class provides the storage for the effect, including the ability to set/get one:

```
class Spatial : public Object
{
public:
    virtual void SetEffect (Effect* pkEffect);
    Effect* GetEffect () const;

protected:
    EffectPtr m_spkEffect;
};
```

Use of the set/get functions is clear. If you set an effect and the object already had one attached, the old one is removed in the sense that its reference count is decremented. If the count becomes zero, the object is automatically destroyed.

3.4.6 The Core Classes and Render State Updates

The core classes `Spatial`, `Geometry`, and `Node` all have some form of support for storing render state and making sure that the renderer has the complete state for each object it draws. The class `Geometry` has the storage capabilities for the render state that affects it. My decision to do this in Wild Magic version 3 was to provide a single object type (`Geometry`) to the renderer. Wild Magic version 2 had an abstract rendering that required the object to be passed as the specific types they were, but the interface was cumbersome. The redesign for version 3 has made the rendering interface much more streamlined. The process of assembling the rendering information in the `Geometry` object is referred to as *updating the render state*.

The portions of the interfaces for classes `Spatial`, `Node`, and `Geometry` that are relevant to updating the render state are

```
class Spatial : public Object
{
public:
    virtual void UpdateRS (TStack<GlobalState*>* akGStack = NULL,
        TStack<Light*>* pkLStack = NULL);
```

```
protected:
    void PropagateStateFromRoot (TStack<GlobalState*>* akGStack,
        TStack<Light*>* pkLStack);
    void PushState (TStack<GlobalState*>* akGStack,
        TStack<Light*>* pkLStack);
    void PopState (TStack<GlobalState*>* akGStack,
        TStack<Light*>* pkLStack);
    virtual void UpdateState (TStack<GlobalState*>* akGStack,
        TStack<Light*>* pkLStack) = 0;
};

class Node : public Object
{
protected:
    virtual void UpdateState (TStack<GlobalState*>* akGStack,
        TStack<Light*>* pkLStack);
};

class Geometry : public Object
{
protected:
    virtual void UpdateState (TStack<GlobalState*>* akGStack,
        TStack<Light*>* pkLStack);
};
```

The entry point into the system is method `UpdateRS` ("update render state"). The input parameters are containers to assemble the global state and lights during a depth-first traversal of the scene hierarchy. The parameters have default values. The caller of `UpdateRS` should *not* set these and make a call: `object.UpdateRS()`. The containers are allocated and managed internally by the update system.

The protected member functions are helper functions for the depth-first traversal. The function `PushState` pushes any global state and lights that the `Spatial` object has attached to it onto stacks. The function `PopState` pops those stacks. The intent is that the stacks are used by all the nodes in the scene hierarhcy as they are visited. Function `Node::UpdateState` has the responsibility for propagating the update in a recursive traversal of the scene hierarchy. Function `Geometry::UpdateState` is called at leaf nodes of the hierarchy. It has the reponsibility for copying the contents of the stacks into its appropriate data members. The stacks store smart pointers to global states and lights, so the copy is really a smart pointer copy and the objects are shared.

The render state at a leaf node represents all the global states and lights that occur on the path from the root node to the leaf node. However, the `UpdateRS` call need only be called at a node whose subtree needs a render state update. Figure 3.16 illustrates a common situation.

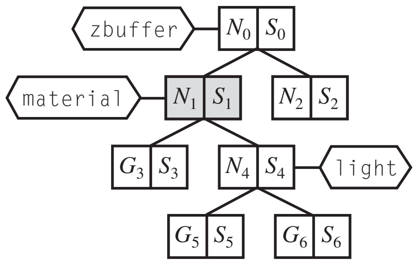


Figure 3.16 A common situation for updating render state.

The z-buffer state is already attached to node N_0, and the light is already attached to node N_4. A material state is attached to node N_1. The render state update is initiated at N_1. The result of the depth-first traversal of the subtree at N_1 is the following: G_3 has links to the z-buffer and material states; G_5 has links to the z-buffer state, the material state, and the light; and G_6 has links to the z-buffer state, the material state, and the light. The z-buffer state is, however, not in the subtree of N_1, so we in fact have to start collecting the states from the root node and along paths that lead to the leaf nodes that are in the subtree of N_1. The function `PropagateStateFromRoot` has the responsibility of starting the render state update at N_1 by first traversing to the root N_0, collecting the render state of the path from N_0 to N_1, and then passing this state to the leaf nodes of the subtree at N_1 together with any additional render state that is in that subtree. Pseudocode for the sequence of operations is listed next. The indentation denotes the level of the calling stack.

```
N1.UpdateRS();
N1: create global state stack GS;       // GS = {}
N1: create light stack LS;              // LS = {}
N1.PropagateStateFromRoot(GS,LS);
    N0.PropagateStateFromRoot(GS,LS);
        N0.PushState(GS,LS);            // GS = {zbuffer}, LS = {}
    N1.PushState(GS,LS);                // GS = {zbuffer,material},
                                        // LS = {}
N1.UpdateState(GS,LS);
    G3.UpdateRS(GS,LS);
        G3.PushState(GS,LS);            // GS = {zbuffer,material},
                                        // LS = {}
        G3.UpdateState(GS,LS);          // share: zbuffer,material
        G3.PopState(GS,LS);
    N4.UpdateRS(GS,LS);
        N4.PushState(GS,LS);            // GS = {zbuffer,material},
                                        // LS = {light}
```

```
        N4.UpdateState(GS,LS);
            G5.UpdateRS(GS,LS);
                G5.PushState(GS,LS);    // GS = {zbuffer,material},
                                        // LS = {light}
                G5.UpdateStore(GS,LS);  // share: zbuffer,material,
                                        //        light
                G5.PopState(GS,LS);     // GS = {zbuffer,material},
                                        // LS = {light}
            G6.UpdateRS(GS,LS);
                G6.PushState(GS,LS);    // GS = {zbuffer,material},
                                        // LS = {light}
                G6.UpdateStore(GS,LS);  // share: zbuffer,material,
                                        //        light
                G6.PopState(GS,LS);     // GS = {zbuffer,material},
                                        // LS = {light}
        N4.PopState(GS,LS);             // GS = {zbuffer,material},
                                        // LS = {}
N1: destroy global state stack GS;      // GS = {}
N1: destroy light stack LS;             // LS = {}
```

The pseudocode is slightly deceptive in that it indicates the global state stack is initially empty, but in fact it is not. After the stack is allocated, smart pointers to the default global state objects are pushed onto it. The copy of smart pointers from the global state stack to the local storage of `Geometry` results in a full set of global states to be used when drawing the geometry object, and the global states are the ones that are at the top of the stack. Nothing prevents you from having multiple states of the same type in a single path from root node to leaf node. For example, the root node can have a z-buffer state that enables depth buffering, but a subtree of objects at node N that can be correctly drawn without depth buffering enabled can also have a z-buffer state that disables depth buffering.

At first glance you might be tempted not to have `PropagateStateFromRoot` in the update system. Consider the current example. Before the material state was attached to node N_1, and assuming the scene hierarchy was current regarding render state, G_3 should have in its local storage the z-buffer. G_5 and G_6 should each have local storage containing the z-buffer and light. When you attach the material to node N_1 and call `UpdateRS` whose implementation does only the depth-first traversal, it appears the correct states will occur at the geometry leaf nodes. In my implementation this is not the case. The global state stack is initialized to contain all the default global states, including the default z-buffer state. The copy of smart pointers in the `Geometry::UpdateState` will overwrite the z-buffer state pointer of N_0 with the default z-buffer state pointer, thus changing the behavior at the leaf nodes.

Now you might consider changing the render state update semantics so that the global state stack is initially empty, accumulate only the render states visited in the depth-first traversal, and then have `Geometry::UpdateState` copy only those pointers into its local storage. To throw a wrench into the works, suppose that the subtree at N_4 is detached from the scene and a new subtree added as the second child of N_1. The leaf nodes of the new subtree are unaware of the render state that N_1 and its predecessors have. A call to the depth-first-only `UpdateRS` at N_1 will propagate the render states from N_1 downward, but now the z-buffer state of N_0 is missing from the leaf nodes. To remedy this problem, you should have called `UpdateRS` at the root node N_0. The leaf nodes will get all the render state they deserve, but unfortunately other subtrees of the scene hierarchy are updated even though they have current render state information. My decision to include `PropagateStateFromRoot` is based on having as efficient a render state update as possible. In a situation such as the current example, the application writer does not have to call `UpdateRS` at N_0 when all that has changed is a subtree modification at N_4. In my update system, after the subtree is replaced by a new one, you only need to call `UpdateRS` at N_4.

The previous discussion does point out that there are various circumstances when you have to call `UpdateRS`. Clearly, if you attach a new global state or light to a node, you should call `UpdateRS` to propagate that information to the leaf nodes. Similarly, if you detach a global state or light from a node, the leaf nodes still have smart pointers to those. You must call `UpdateRS` to eliminate those smart pointers, replacing the global state pointers with ones to the default global states. The light pointers are just removed from the storage. A change in the topology of the scene, such as attaching new children or replacing children at a node N, also requires you to call `UpdateRS`. This is the only way to inform the leaf nodes of the new subtree about their render state.

If you change the data members in a global state object or in a light object, you do *not* have to call `UpdateRS`. The local storage of smart pointers in `Geometry` to the global states and lights guarantees that you are sharing those objects. The changes to the data members are immediately known to the `Geometry` object, so when the renderer goes to draw the object, it has access to the new values of the data members.

To finish up, here is a brief discussion of the implementations of the render state update functions. The entry point is

```
void Spatial::UpdateRS (TStack<GlobalState*>* akGStack,
    TStack<Light*>* pkLStack)
{
    bool bInitiator = (akGStack == NULL);

    if ( bInitiator )
    {
        // stack initialized to contain the default global states
        akGStack = new TStack<GlobalState*>[GlobalState::MAX_STATE];
        for (int i = 0; i < GlobalState::MAX_STATE; i++)
            akGStack[i].Push(GlobalState::Default[i]);
```

```
        // stack has no lights initially
        pkLStack = new TStack<Light*>;

        // traverse to root and push states from root to this node
        PropagateStateFromRoot(akGStack,pkLStack);
    }
    else
    {
        // push states at this node
        PushState(akGStack,pkLStack);
    }

    // propagate the new state to the subtree rooted here
    UpdateState(akGStack,pkLStack);

    if ( bInitiator )
    {
        delete[] akGStack;
        delete pkLStack;
    }
    else
    {
        // pop states at this node
        PopState(akGStack,pkLStack);
    }
}
```

The initiator of the update calls `UpdateRS()` with no parameters. The default parameters are null pointers. This lets the function determine that the initiator is the one who is responsible for allocating and deallocating the stacks. Notice that the global state "stack" is really an array of stacks, one stack per global state type. The initiator is also responsible for calling `PropagateStateFromRoot`. The `UpdateState` call propagates the update to child nodes for a `Node` object, but copies the smart pointers in the stacks to local storage for a `Geometry` object. For the noninitiators, the sequence of calls is effectively

```
PushState(akGStack,pkLStack);
UpdateState(akGStack,pkLStack);
PopState(akGStack,pkLStack);
```

In words: push my state onto the stacks, propagate it to my children, and then pop my state from the stacks.

The propagation of state from the root is

```
void Spatial::PropagateStateFromRoot (
    TStack<GlobalState*>* akGStack, TStack<Light*>* pkLStack)
{
    // traverse to root to allow downward state propagation
    if ( m_pkParent )
        m_pkParent->PropagateStateFromRoot(akGStack,pkLStack);

    // push states onto current render state stack
    PushState(akGStack,pkLStack);
}
```

This is a recursive call that traverses a linear list of nodes. The traversal takes you up the tree to the root, and then you push the states of the nodes as you return to the initiator.

The pushing and popping of state is straightforward:

```
void Spatial::PushState (TStack<GlobalState*>* akGStack,
    TStack<Light*>* pkLStack)
{
    TList<GlobalStatePtr>* pkGList = m_pkGlobalList;
    for (/**/; pkGList; pkGList = pkGList->Next())
    {
        int eType = pkGList->Item()->GetGlobalStateType();
        akGStack[eType].Push(pkGList->Item());
    }

    TList<LightPtr>* pkLList = m_pkLightList;
    for (/**/; pkLList; pkLList = pkLList->Next())
        pkLStack->Push(pkLList->Item());
}

void Spatial::PopState (TStack<GlobalState*>* akGStack,
    TStack<Light*>* pkLStack)
{
    TList<GlobalStatePtr>* pkGList = m_pkGlobalList;
    for (/**/; pkGList; pkGList = pkGList->Next())
    {
        int eType = pkGList->Item()->GetGlobalStateType();
        GlobalState* pkDummy;
        akGStack[eType].Pop(pkDummy);
    }
```

```
    TList<LightPtr>* pkLList = m_pkLightList;
    for (/**/; pkLList; pkLList = pkLList->Next())
    {
        Light* pkDummy;
        pkLStack->Pop(pkDummy);
    }
}
```

The code iterates over a list of global states attached to the object and pushes them on the stack (pops them from the stack) corresponding to the type of the state. The code also iterates over a list of lights attached to the object and pushes them on the stack (pops them from the stack).

The propagation of the update down the tree is

```
void Node::UpdateState (TStack<GlobalState*>* akGStack,
    TStack<Light*>* pkLStack)
{
    for (int i = 0; i < m_kChild.GetQuantity(); i++)
    {
        Spatial* pkChild = m_kChild[i];
        if ( pkChild )
            pkChild->UpdateRS(akGStack,pkLStack);
    }
}
```

This, too, is a straightforward operation. Just as with the geometric update functions `UpdateGS` and `UpdateWorldData`, the pair `UpdateRS` and `UpdateState` form a recursive chain (A calls B, B calls A, etc.).

Finally, the copy of smart pointers from the stacks to local storage is

```
void Geometry::UpdateState (TStack<GlobalState*>* akGStack,
    TStack<Light*>* pkLStack)
{
    // update global state
    int i;
    for (i = 0; i < GlobalState::MAX_STATE; i++)
    {
        GlobalState* pkGState = NULL;
        akGStack[i].GetTop(pkGState);
        assert( pkGState );
        States[i] = pkGState;
    }
```

```
    // update lights
    Light* const* akLight = pkLStack->GetData();
    int iQuantity = pkLStack->GetQuantity();
    for (i = 0; i < iQuantity; i++)
        Lights.Append(akLight[i]);
}
```

No surprises here, either. The `Geometry` class has an array of smart pointers to `GlobalState` for global state storage, and it maintains a list of lights. Although the light list may be arbitrarily long, in practice the graphics APIs limit you to a fixed number, typically eight. The rendering system is designed to process only those lights up to the predetermined maximum.

3.5 Renderers and Cameras

This section describes the two basic objects that are necessary to draw a scene—renderers and cameras. A camera model is simpler to describe than a renderer, so I will discuss cameras first.

3.5.1 Camera Models

Only a portion of the world is displayed at any one time; this region is called the *view volume*. Objects outside the view volume are not visible and therefore not drawn. The process of determining which objects are not visible is called *culling*. Objects that intersect the boundaries of the view volume are only partially visible. The visible portion of an object is determined by intersecting it with the view volume, a process called *clipping*.

The display of visible data is accomplished by projecting it onto a *view plane*. Wild Magic uses perspective projection. Our assumption is that the view volume is a bounded region in space, so the projected data lies in a bounded region in the view plane. A rectangular region in the view plane that contains the projected data is called a *viewport*. The viewport is what is drawn on the rectangular computer screen. The standard view volume used is called the *view frustum*. It is constructed by selecting an eye point and forming an infinite pyramid with four planar sides. Each plane contains the eye point and an edge of the viewport. The infinite pyramid is truncated by two additional planes called the *near plane* and the *far plane*. Figure 3.17 shows a view frustum. The perspective projection is computed by intersecting a ray with the view plane. The ray has origin $\mathbf{E}$, the eye point, and passes through the world point $\mathbf{X}$. The intersection point is $\mathbf{X}_p$.

The combination of an eye point, a view plane, a viewport, and a view frustum is called a *camera model*. The model has a coordinate system associated with

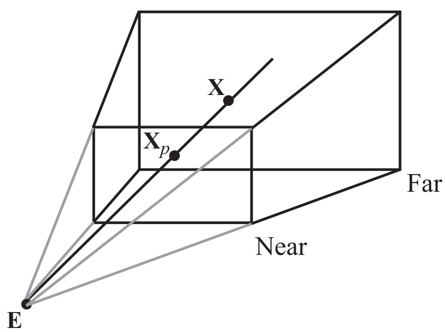


Figure 3.17 An eye point **E** and a view frustum. The point **X** in the view frustum is projected to the point $\mathbf{X}_p$ in the viewport.

it. The *camera origin* is the eye point **E**. The *camera direction vector* (the view vector) is the unit-length vector **D** that is perpendicular to the view plane. The eye point is considered to be on the negative side of the plane. The *camera up vector* is the unit-length **U** vector chosen to be parallel to two opposing edges of the viewport. The *camera right vector*[8] is the unit-length vector **R** chosen to be perpendicular to the camera direction and camera up vector with $\mathbf{R} = \mathbf{D} \times \mathbf{U}$. The set of vectors $\{\mathbf{D}, \mathbf{U}, \mathbf{R}\}$ is a right-handed system and may be stored as columns of a rotation matrix $R = [\mathbf{D}\ \mathbf{U}\ \mathbf{R}]$. The right vector is parallel to two opposing edges of the viewport.

Figure 3.18 shows the camera model, including the camera coordinate system and the view frustum. The six frustum planes are labeled with their names: near, far, left, right, bottom, top. The camera location **E** and the camera axis directions **D**, **U**, and **R** are shown. The view frustum has eight vertices. The near plane vertices are $\mathbf{V}_{t\ell}$, $\mathbf{V}_{b\ell}$, $\mathbf{V}_{tr}$, and $\mathbf{V}_{br}$. Each subscript consists of two letters, the first letters of the frustum planes that share that vertex. The far plane vertices have the name **W** and use the same subscript convention. The equations for the vertices are

8. And there was much rejoicing! Wild Magic version 2 had a *left vector* $\mathbf{L} = \mathbf{U} \times \mathbf{D}$. My choice was based on storing the camera axis vectors in the local rotation matrices as $R = [\mathbf{L}\ \mathbf{U}\ \mathbf{D}]$; that is, the axis vectors are the columns of the matrix. The default values were chosen so that $R = I$, the identity matrix. This had been a source of so much confusion that I changed my default camera model to resemble the OpenGL default camera model.

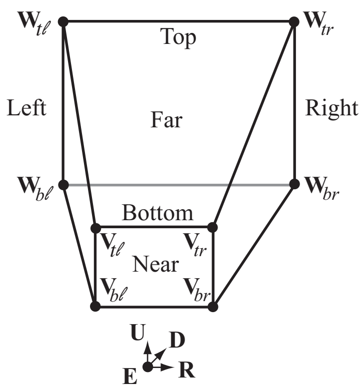


Figure 3.18 A camera model and view frustum.

$$
\begin{aligned}
\mathbf{V}_{b\ell} &= \mathbf{E} + d_{\min}\mathbf{D} + u_{\min}\mathbf{U} + r_{\min}\mathbf{R} \\
\mathbf{V}_{t\ell} &= \mathbf{E} + d_{\min}\mathbf{D} + u_{\max}\mathbf{U} + r_{\min}\mathbf{R} \\
\mathbf{V}_{br} &= \mathbf{E} + d_{\min}\mathbf{D} + u_{\min}\mathbf{U} + r_{\max}\mathbf{R} \\
\mathbf{V}_{tr} &= \mathbf{E} + d_{\min}\mathbf{D} + u_{\max}\mathbf{U} + r_{\max}\mathbf{R} \\
\mathbf{W}_{b\ell} &= \mathbf{E} + \frac{d_{\max}}{d_{\min}}\left(d_{\min}\mathbf{D} + u_{\min}\mathbf{U} + r_{\min}\mathbf{R}\right) \\
\mathbf{W}_{t\ell} &= \mathbf{E} + \frac{d_{\max}}{d_{\min}}\left(d_{\min}\mathbf{D} + u_{\max}\mathbf{U} + r_{\min}\mathbf{R}\right) \\
\mathbf{W}_{br} &= \mathbf{E} + \frac{d_{\max}}{d_{\min}}\left(d_{\min}\mathbf{D} + u_{\min}\mathbf{U} + r_{\max}\mathbf{R}\right) \\
\mathbf{W}_{tr} &= \mathbf{E} + \frac{d_{\max}}{d_{\min}}\left(d_{\min}\mathbf{D} + u_{\max}\mathbf{U} + r_{\max}\mathbf{R}\right) .
\end{aligned}
\tag{3.11}
$$

The near plane is at a distance $d_{\min}$ from the camera location and the far plane is at a distance $d_{\max}$. These distances are the extreme values in the **D** direction. The extreme values in the **U** direction are $u_{\min}$ and $u_{\max}$. The extreme values in the **R** direction are $r_{\min}$ and $r_{\max}$.

Object culling is implemented to use plane-at-a-time culling. The frustum planes are assigned unit-length normals that point inside the frustum. A bounding volume is tested against each frustum plane. If the bounding volume is fully outside one of the planes, the object is not visible and is culled from the display system. To support culling we need to know the equations of the six frustum planes.

The near plane has inner-pointing, unit-length normal $\mathbf{D}$. A point on the plane is $\mathbf{E} + d_{\min}\mathbf{D}$. An equation of the near plane is

$$\mathbf{D} \cdot \mathbf{X} = \mathbf{D} \cdot (\mathbf{E} + d_{\min}\mathbf{D}) = \mathbf{D} \cdot \mathbf{E} + d_{\min}. \tag{3.12}$$

The far plane has inner-pointing, unit-length normal $-\mathbf{D}$. A point on the plane is $\mathbf{E} + d_{\max}\mathbf{D}$. An equation of the far plane is

$$-\mathbf{D} \cdot \mathbf{X} = -\mathbf{D} \cdot (\mathbf{E} + d_{\max}\mathbf{D}) = -(\mathbf{D} \cdot \mathbf{E} + d_{\max}). \tag{3.13}$$

The left plane contains the three points $\mathbf{E}$, $\mathbf{V}_{t\ell}$, and $\mathbf{V}_{b\ell}$. A normal vector that points inside the frustum is

$$\begin{aligned}
(\mathbf{V}_{b\ell} - \mathbf{E}) \times (\mathbf{V}_{t\ell} - \mathbf{E}) &= (d_{\min}\mathbf{D} + u_{\min}\mathbf{U} + r_{\min}\mathbf{R}) \times (d_{\min}\mathbf{D} + u_{\max}\mathbf{U} + r_{\min}\mathbf{R}) \\
&= (d_{\min}\mathbf{D} + r_{\min}\mathbf{R}) \times (u_{\max}\mathbf{U}) + (u_{\min}\mathbf{U}) \times (d_{\min}\mathbf{D} + r_{\min}\mathbf{R}) \\
&= (d_{\min}\mathbf{D} + r_{\min}\mathbf{R}) \times ((u_{\max} - u_{\min})\mathbf{U}) \\
&= (u_{\max} - u_{\min})(d_{\min}\mathbf{D} \times \mathbf{U} + r_{\min}\mathbf{R} \times \mathbf{U}) \\
&= (u_{\max} - u_{\min})(d_{\min}\mathbf{R} - r_{\min}\mathbf{D}).
\end{aligned}$$

An inner-pointing, unit-length normal and the left plane are

$$\mathbf{N}_\ell = \frac{d_{\min}\mathbf{R} - r_{\min}\mathbf{D}}{\sqrt{d_{\min}^2 + r_{\min}^2}}, \quad \mathbf{N}_\ell \cdot (\mathbf{X} - \mathbf{E}) = 0. \tag{3.14}$$

An inner-pointing normal to the right plane is $(\mathbf{V}_{tr} - \mathbf{E}) \times (\mathbf{V}_{br} - \mathbf{E})$. A similar set of calculations as before will lead to an inner-pointing, unit-length normal and the right plane:

$$\mathbf{N}_r = \frac{-d_{\min}\mathbf{R} + r_{\max}\mathbf{D}}{\sqrt{d_{\min}^2 + r_{\max}^2}}, \quad \mathbf{N}_r \cdot (\mathbf{X} - \mathbf{E}) = 0. \tag{3.15}$$

Similarly, an inner-pointing, unit-length normal and the bottom plane are

$$\mathbf{N}_b = \frac{d_{\min}\mathbf{U} - u_{\min}\mathbf{D}}{\sqrt{d_{\min}^2 + u_{\min}^2}}, \quad \mathbf{N}_b \cdot (\mathbf{X} - \mathbf{E}) = 0. \tag{3.16}$$

An inner-pointing, unit-length normal and the top plane are

$$\mathbf{N}_t = \frac{-d_{\min}\mathbf{U} + u_{\max}\mathbf{D}}{\sqrt{d_{\min}^2 + u_{\max}^2}}, \quad \mathbf{N}_t \cdot (\mathbf{X} - \mathbf{E}) = 0. \tag{3.17}$$

The Camera Class

Time for a few comments about the Camera class, similar to those for the Light class. In Wild Magic version 2, the Camera class was derived from Object. I considered a Camera a special type of object that had some spatial information, but also a lot of other information that did not warrant it being derived from Spatial. A number of users were critical of this choice and insisted that Camera be derived from Spatial. For example, if you were to build a model of a room with a security camera mounted in a corner, the camera orientation could be modified using a controller (rotate camera back and forth for coverage of the area of the room). The camera itself can be used for rendering what it sees and then displaying that rendering on a television monitor that is also part of the room model. To support this, I added a class CameraNode that is derived from Node and that had a Camera data member. I had a similar class to encapsulate lights, namely, LightNode. But these classes presented some problems to users; one problem had to do with importing LightWave objects into the engine. Because LightWave uses left-handed coordinates for everything, the design of CameraNode and LightNode prevented a correct import of lights (and cameras) when they were to be attached as nodes in a scene.

In Wild Magic version 3, I changed my design and derived Camera from Spatial, thus eliminating the need for CameraNode. The warnings I issued about deriving Light from Spatial apply here as well. Some subsystems of Spatial are available to Camera that are irrelevant. For example, attaching to a camera a global state such as a depth buffer has no meaning, but the engine semantics allow the attachment. You can attach lights to cameras, but this makes no sense. The camera object itself is not renderable. The virtual functions for global state updates and for drawing are stubbed out in the Camera class, so incorrect use of the cameras should not be a problem. So be warned that you can manipulate a Camera as a Spatial object in ways that the engine was not designed to handle.

The portion of the interface for Camera that relates to the camera coordinate system is

```
class Camera : public Spatial
{
public:
    Camera ();

    // Camera frame (local coordinates)
    //   default location  E = (0,0,0)
    //   default direction D = (0,0,-1)
    //   default up        U = (0,1,0)
    //   default right     R = (1,0,0)
    // If a rotation matrix is used for the axis directions, the
    // columns of the matrix are [D U R].  That is, the view
    // direction is in column 0, the up direction is in column 1,
```

```
    // and the right direction is in column 2.
    void SetFrame (const Vector3f& rkLocation,
        const Vector3f& rkDVector, const Vector3f& rkUVector,
        const Vector3f& rkRVector);
    void SetFrame (const Vector3f& rkLocation,
        const Matrix3f& rkAxes);
    void SetLocation (const Vector3f& rkLocation);
    void SetAxes (const Vector3f& rkDVector,
        const Vector3f& rkUVector, const Vector3f& rkRVector);
    void SetAxes (const Matrix3f& rkAxes);
    Vector3f GetLocation () const;  // Local.Translate
    Vector3f GetDVector () const;   // Local.Rotate column 0
    Vector3f GetUVector () const;   // Local.Rotate column 1
    Vector3f GetRVector () const;   // Local.Rotate column 2

    // camera frame (world coordinates)
    Vector3f GetWorldLocation () const;  // World.Translate
    Vector3f GetWorldDVector () const;   // World.Rotate column 0
    Vector3f GetWorldUVector () const;   // World.Rotate column 1
    Vector3f GetWorldRVector () const;   // World.Rotate column 2

protected:
    virtual void UpdateWorldBound ();
    void OnFrameChange ();
};
```

Normally, the local transformation variables (translation, rotation, scale) are for exactly that—transformation. In the Camera class, the local translation is interpreted as the origin for a coordinate system of the camera; that is, the local translation is the eye point. The columns of the local rotation matrix are interpreted as the coordinate axis directions for the camera's coordinate system. Think of the camera's right and up vectors as the positive x- and positive y-axes for the display screen. The view direction is into the screen, the negative z-axis. The eye point is not the center of the screen, but is positioned in front of the screen. Because the camera's coordinate system is stored in the local translation vector and local rotation matrix, you should use the interface provided and avoid setting the data member Local explicitly to something that is not consistent with the interpretation as a coordinate system.

The first block of code in the interface is for set/get of the coordinate system parameters. The second block of code in the public interface allows you to retrieve the world coordinates for the camera's (local) coordinate system.

The Camera class has no model bound. However, the camera's position acts as the center of a model bound of radius zero. The virtual function UpdateWorldBound computes the center of a world bound of radius zero. The function OnFrameChange is a wrapper around a call to UpdateGS and is executed whenever you set the coordinate system components. Therefore, you do not need to explicitly call UpdateGS whenever the coordinate system components are modified. Unlike the Light class, the Camera version of OnFrameChange has the job of computing the world coordinate representations for the frustum planes to be used for culling. It also has the job of informing the renderer associated with it that the camera coordinate system has changed. The renderer takes the appropriate action to update any of its state, such as making specific camera-related calls to the graphics API that it encapsulates.

The two virtual functions in the private section are stubs to implement pure virtual functions in Spatial (as required by C++). None of these make sense for cameras anyway. They exist just so that Camera inherits other properties of Spatial that are useful.

View Frustum Parameters

The view frustum parameters $r_{\min}$ (left), $r_{\max}$ (right), $u_{\min}$ (bottom), $u_{\max}$ (top), $d_{\min}$ (near), and $d_{\max}$ (far) are set/get by the following interface:

```
class Camera : public Spatial
{
public:
    void SetFrustum (float fRMin, float fRMax, float fUMin,
        float fUMax, float fDMin, float fDMax);

    void SetFrustum (float fUpFovDegrees, float fAspectRatio,
        float fDMin, float fDMax);

    void GetFrustum (float& rfRMin, float& rfRMax, float& rfUMin,
        float& rfUMax, float& rfDMin, float& rfDMax) const;

    float GetDMin () const;
    float GetDMax () const;
    float GetUMin () const;
    float GetUMax () const;
    float GetRMin () const;
    float GetRMax () const;

protected:
    void OnFrustumChange ();

    float m_fDMin, m_fDMax, m_fUMin, m_fUMax, m_fRMin, m_fRMax;

    // Values computed in OnFrustumChange that are needed in
    // OnFrameChange.
```

```
    float m_afCoeffL[2], m_afCoeffR[2];
    float m_afCoeffB[2], m_afCoeffT[2];
};
```

For those of you familiar with Wild Magic version 2, notice that the order of the parameters to the first `SetFrustum` method has changed. The new ordering is the same used by OpenGL's `glFrustum` function. The second `SetFrustum` method is equivalent to OpenGL's `gluPerspective` function. This method creates a symmetric view frustum ($u_{\min} = -u_{\max}$ and $r_{\min} = -r_{\max}$) using a field of view specified in the up direction and an aspect ratio corresponding to width divided by height. The field of view is an angle specified in degrees and must be in the interval (0, 180). The angle is that between the top and bottom view frustum planes. The typical aspect ratio is 4/3, but for wide-screen displays is 16/9.

The function `OnFrustumChange` is a callback that is executed whenever `SetFrustum` is called. The callback informs the renderer to which the camera is attached that the frustum has changed. The renderer makes the appropriate changes to its state (informing the graphics API of the new frustum parameters). The callback also computes some quantities related to culling—specifically, the coefficients of the coordinate axis vectors in Equations (3.14) through (3.17). The coefficients from Equation (3.14) are stored in `m_afCoeffL`. The coefficients from Equation (3.15) are stored in `m_afCoeffR`. The coefficients from Equation (3.16) are stored in `m_afCoeffB`. The coefficients from Equation (3.17) are stored in `m_afCoeffT`. The function `OnFrameChange` is called very frequently and uses these coefficients for computing the world representations of the frustum planes.

You will see in most of the applications that I set the frustum to a symmetric one with the first `SetFrustum` method. The typical call is

```
// order:  left, right, bottom, top, near, far
m_spkCamera->SetFrustum(-0.55f,0.55f,-0.4125f,0.4125f,1.0f,100.0f);
```

The ratio of right divided by top is 4/3. The near plane distance from the eye point is 1, and the far plane distance is 100. If you decide to modify the near plane distance in an application using this call to `SetFrustum`, you must modify the left, right, bottom, and top values accordingly. Specifically,

```
float fNear = <some positive value>;
float fFar = <whatever>;
float fLeft = -0.55f*fNear;
float fRight = 0.55f*fNear;
float fBottom = -0.4125f*fNear;
float fTop = 0.4125f*fNear;
m_spkCamera->SetFrustum(fLeft,fRight,fBottom,fTop,fNear,fFar);
```

The second SetFrustum method is probably more intuitive for the user.

A question that arises periodically on the Usenet computer graphics newsgroups is how to do *tiled rendering*. The idea is that you want to have a high-resolution drawing of an object, but the width and/or height of the final result is larger than your computer monitor can display. You can accomplish this by selecting various view frustum parameters and rendering the object as many times as it takes to generate the final image. For example, suppose your computer monitor can display at 1600×1200; that is, the monitor has 1200 scan lines, and each scan line has 1600 columns. To generate an image that is 3200×2400, you can render the scene four times, each rendering to a window that is 1600×1200. I have not yet described the renderer interface, but the use of it is clear in this example. The view frustum is symmetric in this example.

```
// initialization code
NodePtr m_spkScene = <the scene graph>;
Renderer* m_pkRenderer = <the renderer>;
CameraPtr m_spkCamera = <the camera assigned to the renderer>;
float m_fDMin = <near plane distance>;
float m_fDMax = <far plane distance>;
float m_fRMax = <some value>;
float m_fUMax = <some value>;
m_spkCamera->SetFrustum(-m_fRMax,m_fRMax,-m_fUMax,m_fUMax,
    m_fDMin,m_fDMax);

// keyboard handler code (ucKey is the input keystroke)
switch (ucKey)
{
case 0:  // draw all four quadrants
    m_spkCamera->SetFrustum(-m_fRMax,m_fRMax,-m_fUMax,m_fUMax,
        m_fDMin,m_fDMax);
    break;
case 1:  // upper-right quadrant
    m_spkCamera->SetFrustum(0.0f,m_fRMax,0.0f,m_fUMax,
        m_fDMin,m_fDMax);
    break;
case 2:  // upper-left quadrant
    m_spkCamera->SetFrustum(-m_fRMax,0.0f,0.0f,m_fUMax,
        m_fDMin,m_fDMax);
    break;
case 3:  // lower-left quadrant
    m_spkCamera->SetFrustum(-m_fRMax,0.0f,-m_fUMax,0.0f,
        m_fDMin,m_fDMax);
    break;
```

```
case 4:  // lower-right quadrant
    m_spkCamera->SetFrustum(0.0f,m_fRMax,-m_fUMax,0.0f,
        m_fDMin,m_fDMax);
    break;
}

// idle-loop callback or on-display callback
m_pkRenderer->DrawScene(m_spkScene);
```

I use the keyboard handler approach on a Microsoft Windows machine so that I can use ALT+PRINTSCREEN to capture the window contents, edit it in Windows Paint to keep only the contents of the client window, and then copy that into the appropriate quadrant in a bitmap file of size 3200×2400. You can certainly automate this task by rendering each tile one at a time, and then reading the frame buffer contents after each rendering and copying it to the appropriate location in a memory block that will eventually be saved as a bitmap file.

Viewport Parameters

The viewport parameters are used to represent the computer screen in normalized display coordinates $(\bar{x}, \bar{y}) \in [0, 1]^2$. The left edge of the screen is $\bar{x} = 0$, and the right edge is $\bar{x} = 1$. The bottom edge is $\bar{y} = 0$, and the top edge is $\bar{y} = 1$. The Camera interface is

```
class Camera : public Spatial
{
public:
    void SetViewPort (float fLeft, float fRight, float fTop,
        float fBottom);
    void GetViewPort (float& rfLeft, float& rfRight, float& rfTop,
        float& rfBottom);

protected:
    void OnViewPortChange ();

    float m_fPortL, m_fPortR, m_fPortT, m_fPortB;
};
```

The function OnViewPortChange is a callback that is executed whenever SetViewPort is called. The callback informs the renderer to which the camera is attached that the viewport has changed. The renderer makes the appropriate changes to its state (informing the graphics API of the new viewport parameters).

In most cases the viewport is chosen to be the entire screen. However, some applications might want to display an offset window with a rendering that is separate from what occurs in the main window. For example, you might have an application that draws a scene based on a camera at an arbitrary location and with an arbitrary orientation. A front view, top view, and side view might also be desired using fixed cameras. The four desired renderings may be placed in four quadrants of the screen. The sample code shows how to do this. Once again, I have not discussed the renderer interface, but the use of it is clear.

```
// initialization (all camera frames assumed to be set properly)
NodePtr m_spkScene = <the scene graph>;
Renderer* m_pkRenderer = <the renderer>;
CameraPtr m_spkACamera = <the camera for arbitrary drawing>;
CameraPtr m_spkFCamera = <the camera for front view>;
CameraPtr m_spkTCamera = <the camera for top view>;
CameraPtr m_spkSCamera = <the camera for side view>;
m_spkACamera->SetViewport(0.0f,0.5f,1.0f,0.5f);  // upper left
m_spkFCamera->SetViewport(0.5f,1.0f,1.0f,0.5f);  // upper right
m_spkTCamera->SetViewport(0.0f,0.5f,0.5f,0.0f);  // lower left
m_spkSCamera->SetViewport(0.5f,1.0f,0.5f,0.0f);  // lower right

// on-idle callback
m_pkRenderer->SetCamera(m_spkACamera);
m_pkRenderer->DrawScene(m_spkScene);
m_pkRenderer->SetCamera(m_spkFCamera);
m_pkRenderer->DrawScene(m_spkScene);
m_pkRenderer->SetCamera(m_spkTCamera);
m_pkRenderer->DrawScene(m_spkScene);
m_pkRenderer->SetCamera(m_spkSCamera);
m_pkRenderer->DrawScene(m_spkScene);
```

I used four separate cameras in this example. It is also possible to use a single camera, but change its position, orientation, and viewport before each rendering:

```
// initialization code
NodePtr m_spkScene = <the scene graph>;
Renderer* m_pkRenderer = <the renderer>;
CameraPtr m_spkCamera = <the camera assigned to the renderer>;
Vector3f kACLoc = <camera location for arbitrary view>;
Matrix3f kACAxes = <camera orientation for arbitrary view>;
Vector3f kFCLoc = <camera location for front view>;
Matrix3f kFCAxes = <camera orientation for front view>;
Vector3f kTCLoc = <camera location for top view>;
Matrix3f kTCAxes = <camera orientation for top view>;
```

```
Vector3f kSCLoc = <camera location for side view>;
Matrix3f kSCAxes = <camera orientation for side view>;

// on-idle callback
m_spkCamera->SetFrame(kACLoc,kACAxes);
m_spkCamera->SetViewport(0.0f,0.5f,1.0f,0.5f);
m_pkRenderer->DrawScene(m_spkScene);
m_spkCamera->SetFrame(kFCLoc,kFCAxes);
m_spkCamera->SetViewport(0.5f,1.0f,1.0f,0.5f);
m_pkRenderer->DrawScene(m_spkScene);
m_spkCamera->SetFrame(kTCLoc,kTCAxes);
m_spkCamera->SetViewport(0.0f,0.5f,0.5f,0.0f);
m_pkRenderer->DrawScene(m_spkScene);
m_spkCamera->SetFrame(kSCLoc,kSCAxes);
m_spkCamera->SetViewport(0.5f,1.0f,0.5f,0.0f);
m_pkRenderer->DrawScene(m_spkScene);
```

Object Culling

The object culling support in the Camera class is the most sophisticated subsystem for the camera. This system interacts with the Spatial class during the drawing pass of the scene graph. I will talk about the drawing pass later, but for now it suffices to say that the Spatial class has the following interface for drawing:

```
class Spatial : public Object
{
public:
    BoundingVolumePtr WorldBound;
    bool ForceCull;

// internal use
public:
    void OnDraw (Renderer& rkRenderer, bool bNoCull = false);
    virtual void Draw (Renderer& rkRenderer,
        bool bNoCull = false) = 0;
};
```

We have already seen the WorldBound data member. It is used for culling purposes. The Boolean flag ForceCull allows a user to force the object not to be drawn, which is especially convenient for a complicated system that partitions the world into cells. Each cell maintains two lists: One list is for the visible objects; the other for the invisible objects, whenever the camera is in the cell. At the moment the camera enters the cell, the list of visible objects is traversed. Each object has its ForceCull flag set to false. The other list is traversed, and each object has its ForceCull flag set to true.

Notice that the second public block is flagged for internal use. An application should never call these functions. A call to the function OnDraw is a request that the object draw itself. The drawing itself is performed by Draw. If the input parameter bNoCull is set to true, if the object is not force-culled, then the culling tests that compare the world bound to the view frustum planes are skipped.

Before drawing itself, the object must check to see if it is potentially visible. If not, it culls itself; that is, it does not call the Draw function. The code for OnDraw is

```
void Spatial::OnDraw (Renderer& rkRenderer, bool bNoCull)
{
    if ( ForceCull )
        return;

    CameraPtr spkCamera = rkRenderer.GetCamera();
    unsigned int uiState = spkCamera->GetPlaneState();

    if ( bNoCull || !spkCamera->Culled(WorldBound) )
        Draw(rkRenderer,bNoCull);

    spkCamera->SetPlaneState(uiState);
}
```

If ForceCull is set to true, the request to be drawn is denied. Otherwise, the object gets access to the camera attached to the renderer. Before attempting the culling, some camera state information is saved (on the calling stack) in the local variable uiState. Before exiting OnDraw, that state is restored. More about this in a moment. Assuming the object allows the culling tests (bNoCull set to false), a call is made to Camera::Culled. This function compares the world bound to the view frustum planes (in world coordinates). If the world bound is outside any of the planes, the function returns true, indicating that the object is culled. If the object is not culled, finally the drawing occurs via the function Draw. As we will see, Node::Draw propagates the drawing request to its children, so OnDraw and Draw form a recursive chain.

Now about the camera state that is saved and restored. I mentioned earlier that in a scene hierarchy, if a bounding volume of a node is *inside* a view frustum plane, the object contained by the bounding volume is also inside the plane. The objects represented by child nodes must necessarily be inside the plane, so there is no reason to compare a child's bounding volume to this same frustum plane. The Camera class maintains a bit flag (as an unsigned int) where each bit corresponds to a frustum plane. A bit value of 1 says that the bounding volume should be compared to the plane corresponding to that bit, and a bit value of 0 says to skip the comparison. The bits in the flag are all initialized to 1 in the Camera constructor. A drawing pass will set and restore these bits, so at the end of a drawing pass, the bits are all 1 again. The determination that a bounding volume is inside a frustum plane is made during the Camera::Culled call. If the bounding volume is inside, the corresponding bit is set to 0. On a recursive call to Draw, the Camera::Culled function will be called for the child nodes. When a zero bit is encountered, the camera knows not to compare the child's bounding volume to the corresponding frustum plane because the parent's bounding volume is already inside that plane. The goal of maintaining the bit flags is to reduce the computational time spent in comparing bounding volumes to frustum planes—particularly important when the comparison is an expensive calculation (convex hull versus plane, for example).

The portion of the Camera interface relevant to the culling discussion to this point is

```
class Camera : public Spatial
{
protected:
    unsigned int m_uiPlaneState;

    // world planes:
    //   left = 0, right = 1, bottom = 2,
    //   top = 3, near = 4, far = 5,
    //   extra culling planes >= 6
    enum
    {
        CAM_FRUSTUM_PLANES = 6,
        CAM_MAX_WORLD_PLANES = 32
    };
    int m_iPlaneQuantity;
    Plane3f m_akWPlane[CAM_MAX_WORLD_PLANES];

// internal use
public:
    // culling support in Spatial::OnDraw
    void SetPlaneState (unsigned int uiPlaneState);
    unsigned int GetPlaneState () const;
    bool Culled (const BoundingVolume* pkWBound);
};
```

The data member m_uiPlaneState is the set of bits corresponding to the frustum planes. Bit 0 is for the left plane, bit 1 is for the right plane, bit 2 is for the bottom plane, bit 3 is for the top plane, bit 4 is for the near plane, and bit 5 is for the far plane. The data member m_iPlaneQuantity specifies how many planes are in the system. This number is at least six, but can be larger! The world representations for the planes are stored in m_akWPlane. We already saw in Spatial::OnDraw the use of SetPlaneState and GetPlaneState for the bit flag management.

The final function to consider is `Culled`:

```
bool Camera::Culled (const BoundingVolume* pkWBound)
{
    // Start with last pushed plane (potentially the most
    // restrictive plane).
    int iP = m_iPlaneQuantity - 1;
    unsigned int uiMask = 1 << iP;

    for (int i = 0; i < m_iPlaneQuantity; i++, iP-, uiMask >>= 1)
    {
        if ( m_uiPlaneState & uiMask )
        {
            int iSide = pkWBound->WhichSide(m_akWPlane[iP]);

            if ( iSide < 0 )
            {
                // Object is on negative side.  Cull it.
                return true;
            }

            if ( iSide > 0 )
            {
                // Object is on positive side of plane.  There is
                // no need to compare subobjects against this
                // plane, so mark it as inactive.
                m_uiPlaneState &= ~uiMask;
            }
        }
    }

    return false;
}
```

The function iterates over the array of world planes. If a plane is active (its bit is 1), the world bounding volume of the object is compared to the plane. If the bounding volume is on the positive side, then the bit for the plane is set to 0 so that the bounding volumes of descendants are never compared to that plane. If the bounding volume is on the negative side, it is outside the plane and is culled. If the bounding volume straddles the plane (part of it inside, part of it outside), then the object is not culled and you cannot disable the plane from comparisons with descendants.

I had mentioned that the number of planes can be larger than six. The public interface for `Camera` also contains the following functions:

```
class Camera : public Spatial
{
public:
    int GetPlaneQuantity () const;
    const Plane3f* GetPlanes () const;
    void PushPlane (const Plane3f& rkPlane);
    void PopPlane ();
};
```

An application writer may tell the camera to use additional culling planes by calling `PushPlane` for each such plane. The manual addition is useful in environments where you have knowledge of the placement of objects and you can safely say that objects behind a particular plane are not visible. For example, an application that has a fixed camera position and orientation could have a building in view of the observer. Objects behind the building are not visible. The plane of the back side of the building can be added to the camera system for culling purposes. But you need to be careful in using this support. In the current example, if a character is behind the building, the culling works fine. But if the character moves to the side of the building and is visible to the observer, but is still behind the plane of the back side of the building, you would cull the character when in fact he is visible.

The ability to push and pop planes is also useful in an automatic portaling system. Indeed, Wild Magic has support for portals for indoor occlusion culling. That system, described later, pushes and pops planes as necessary depending on the camera location and nearby portals. The `Camera` class has a public function flagged for internal use:

```
bool Culled (int iVertexQuantity, const Vector3f* akVertex,
    bool bIgnoreNearPlane);
```

This function is designed specifically for the portal system and will be described later.

Whether planes are pushed manually or automatically, the data member `m_uiPlaneState` has 6 bits reserved for the frustum planes. The remaining bits are used for the additional culling planes. On a 32-bit system, this means you can push up to 26 additional culling planes. I suspect that 26 is more than enough for practical applications. You should also be aware the planes are only used for object culling. The objects are *not clipped* against any of these planes. On current graphics hardware, selecting additional clipping planes can have some interesting and surprising side effects. For example, if you select an additional clipping plane, you might lose the use of one of your texture units. My recommendation is not to worry about the clipping, only the culling.

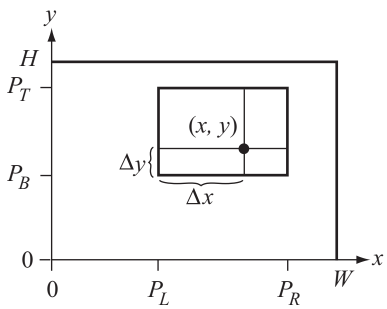


Figure 3.19 A pixel (x, y) selected in a viewport that is not the entire screen.

Picking

The engine supports *picking* operations. Generally these determine whether or not a linear component (line, ray, segment) intersects an object. The classical application is to select an object that is displayed on the screen. The user clicks on a screen pixel that is part of the object. A ray is generated in world coordinates: The origin of the ray is the camera location in world coordinates, and the direction of the ray is the vector from the camera location to the world point that corresponds to the screen pixel that was selected.

The construction of the world point is slightly complicated by having an active viewport that is not the full window. Figure 3.19 shows a window with a viewport and a selected pixel (x, y).

The current viewport settings are P_L (left), P_R (right), P_B (bottom), and P_T (top). Although these are placed at tick marks on the axes, all the numbers are normalized (in [0, 1]). The screen coordinates satisfy the conditions $0 \le x < W$ and $0 \le y < H$, where W is the width of the screen and H is the height of the screen. The screen coordinates must be converted to normalized coordinates:

$$x' = \frac{x}{W-1}, \quad y' = \frac{H-1-y}{H-1}.$$

The screen coordinates are left-handed: $y = 0$ is the top row, $y = H - 1$ is the bottom row, $x = 0$ is the left column, and $x = W - 1$ is the right column. The normalized coordinates (x', y') are right-handed due to the inversion of the y value. The relative distances within the viewport are

$$\Delta_x = \frac{x' - P_L}{P_R - P_L}, \quad \Delta_y = \frac{y' - P_B}{P_T - P_B}.$$

The picking ray is $\mathbf{E} + t\mathbf{U}$, where $\mathbf{E}$ is the eye point in world coordinates and $\mathbf{U}$ is a unit-length direction vector in world coordinates:

$$\mathbf{U} = d_{\min}\mathbf{D} + ((1 - \Delta_x)r_{\min} + \Delta_x r_{\max})\mathbf{R} + ((1 - \Delta_y)u_{\min} + \Delta_y u_{\max})\mathbf{U},$$

where $\mathbf{D}$, $\mathbf{R}$, and $\mathbf{U}$ are the axis directions for the camera coordinate system in world coordinates. The equation is derived from the fact that the viewport contains the full extent of the rendering to the frustum rectangle $[r_{\min}, r_{\max}] \times [u_{\min}, u_{\max}]$.

The portion of the `Camera` interface to support construction of the pick ray is

```
class Camera : public Spatial
{
public:
    bool GetPickRay (int iX, int iY, int iWidth, int iHeight,
        Vector3f& rkOrigin, Vector3f& rkDirection) const;
};
```

The (x, y) input point is in left-handed screen coordinates. The function returns `true` if and only if the input point is located in the current viewport. When `true`, the origin and direction values are valid and are in world coordinates. The direction vector is unit length. If the returned function value is `false`, the origin and direction are invalid.

Some graphics APIs support picking in an alternate manner. In addition to the frame buffer and depth buffer values at a pixel (x, y) on the screen, the renderer maintains a buffer of names. The visible object that led to the frame buffer value at pixel (x, y) has its name stored in the name buffer. When the user clicks on the screen pixel (x, y), the graphics API returns the name of the corresponding object. The depth buffer value at the pixel may be used to gauge how far away the object is (at least how far away the world point is that generated the pixel). My concept of picking is more general. For example, if a character has a laser gun and shoots at another character, you have to determine if the target was hit. The laser beam itself is a ray whose origin is the gun and whose direction is determined by the gun barrel. A picking operation is initiated to determine if that ray intersects the target. In this case, the camera is not the originator of the picking ray.

3.5.2 Basic Architecture for Rendering

The class `Renderer` is an abstract class that defines an API that the scene graph management system uses for drawing. The intent is to provide a portable layer that hides platform-dependent constructs such as operating system calls, windowing systems, and graphics APIs. Derived classes that do handle the platform-dependent issues are built on top of `Renderer`. Wild Magic version 2 had a few derived classes. The class `OpenGLRenderer` encapsulated the OpenGL API. This class is itself portable to those platforms that support OpenGL. However, window creation and memory allocation on the graphics card are dependent on the platform. I had constructed three classes derived from OpenGLRenderer. The class GlutRenderer encapsulates GLUT, which is itself intended to be a portable wrapper around OpenGL. The GlutRenderer runs on Microsoft Windows, on Macintosh OS X, and on PCs with Linux. Unfortunately, GLUT does not expose much in the way of creating subwindows, menus, and other controls. The class WglRenderer is derived from OpenGLRenderer, but makes no attempt to hide the fact that it runs on Microsoft Windows. A programmer may create a Windows-specific application with all the desired bells and whistles and then add to the application a WglRenderer. On the Macintosh, the class AglRenderer is derived from OpenGLRenderer and does not attempt to hide the fact that it runs using Apple's OpenGL.

For folks who prefer working only on Microsoft Windows using Direct3D, Wild Magic version 2 also had a class DxRenderer derived from Renderer. Naturally, applications using this are not portable to other platforms.

Wild Magic version 3 has the same philosophy about a portable rendering layer. As of the time of writing, I only have support for OpenGL. Hopefully by the time the book is in print, a Direct3D renderer will be posted at my Web site.

Regarding construction, destruction, and information relevant to the window in which the renderer will draw, the interface for Renderer is

```
class Renderer
{
public:
    // abstract base class
    virtual ~Renderer ();

    // window parameters
    int GetWidth () const;
    int GetHeight () const;

    // background color access
    virtual void SetBackgroundColor (const ColorRGB& rkColor);
    const ColorRGB& GetBackgroundColor () const;

    // text drawing
    virtual int LoadFont (const char* acFace, int iSize,
        bool bBold, bool bItalic) = 0;
    virtual void UnloadFont (int iFontID) = 0;
    virtual bool SelectFont (int iFontID) = 0;
    virtual void Draw (int iX, int iY, const ColorRGBA& rkColor,
        const char* acText) = 0;
```

```
protected:
    // abstract base class
    Renderer (int iWidth, int iHeight);

    // window parameters
    int m_iWidth, m_iHeight;
    ColorRGB m_kBackgroundColor;

    // current font for text drawing
    int m_iFontID;
};
```

First, notice that `Renderer` is *not* derived from `Object`. You only need one renderer in an application, so the sharing subsystem of `Object` is not necessary. Searching for a renderer by name or ID is also not necessary since there is only one. Derived-class renderers are dependent on platform, so you do not want to stream them. Renderers have nothing animated, and making copies is not an issue. Consequently, there is no reason to derive the class from `Object`.

A derived class must construct the base class through the protected constructor. The width and height of the window's client region to which the renderer must draw are provided to the base class. Notice that the window location is *not* given to the renderer. The application has the responsibility for window positioning and resizing, but the renderer only needs to know the dimensions of the drawing region. The background color for the window is stored in the renderer so that it can clear (if necessary) the window to that color before drawing.

The renderer API has pure virtual functions for drawing text on the screen and for font selection. The text drawing and font selection are usually done in a platform-specific manner, so implementations for the API must occur in the derived classes. The data member `m_iFontID` acts as a handle for the derived-class renderer. Multiple fonts can be loaded and managed by the application. The `LoadFont` member lets you create a font. The return value is a font ID that the application should store. The ID is passed to `SelectFont` to let the renderer know that text should be drawn using the corresponding font. The ID is also passed to `UnloadFont` when the font is to be destroyed. The actual text drawing occurs via the member function `Draw`. You specify where the text should occur on the screen and what its color should be. The color has an alpha channel, so the text may be drawn with some transparency. The environment mapping sample application illustrates font selection:

```
// select a font for text drawing
int iFontID = m_pkRenderer->LoadFont("Verdana",24,false,false);
m_pkRenderer->SelectFont(iFontID);
```

As you can see, it is simple enough to load a font and tell the renderer to use it.

A renderer must have a camera assigned to it in order to define the region of space that is rendered. The relevant interface is

```
class Renderer
{
public:
    void SetCamera (Camera* pkCamera);
    Camera* GetCamera () const;

protected:
    Camera* m_pkCamera;
};
```

The use of the interface is quite clear. To establish a two-way communication between the camera and the renderer, the `Camera` class has a data member `m_pkRenderer` that is set by `Renderer` during a call to `SetCamera`. The two-way communication is necessary: The renderer queries the camera for relevant information such as the view frustum parameters and coordinate frame, and, if the camera parameters are modified at run time, the camera must notify the renderer about the changes so that the renderer updates the graphics system (via graphics API calls).

Various resources are associated with a renderer, including a frame buffer (the front buffer) that stores the pixel colors, a back buffer for double-buffered drawing, a depth buffer for storing depths corresponding to the pixels, and a stencil buffer for advancing effects. The back buffer, depth buffer, and stencil buffer may need to be cleared before drawing a scene. The interface supporting these buffers is

```
class Renderer
{
public:
    // full window buffer operations
    virtual void ClearBackBuffer () = 0;
    virtual void ClearZBuffer () = 0;
    virtual void ClearStencilBuffer () = 0;
    virtual void ClearBuffers () = 0;
    virtual void DisplayBackBuffer () = 0;

    // clear the buffer in the specified subwindow
    virtual void ClearBackBuffer (int iXPos, int iYPos,
        int iWidth, int iHeight) = 0;
    virtual void ClearZBuffer (int iXPos, int iYPos,
        int iWidth, int iHeight) = 0;
    virtual void ClearStencilBuffer (int iXPos, int iYPos,
        int iWidth, int iHeight) = 0;
```

```
    virtual void ClearBuffers (int iXPos, int iYPos,
        int iWidth, int iHeight) = 0;
};
```

All the clearing functions are pure virtual, but the derived-class implementations are simple wrappers around standard graphics API calls. The function DisplayBackBuffer is the request to the graphics system to copy the back buffer into the front buffer.

The graphics hardware also has a fixed number of texture units, and the graphics system supports at most a certain number of lights. You may query the renderer for these via

```
class Renderer
{
public:
    int GetMaxLights () const;
    int GetMaxTextures () const;

protected:
    int m_iMaxLights;
    int m_iMaxTextures;
};
```

The data members are initialized to zero in the Renderer constructor. The derived classes are required to set these to whatever limits exist for the user's environment. Wild Magic version 2 had a hard-coded number of texture units (4) and lights (8); both numbers were class-static data members. The number of texture units was chosen at a time when consumer graphics cards had just evolved to contain 4 texture units. Some engine users decided that it was safe to change that number. Unfortunately, the streaming system had a problem with this. If a scene graph was saved to disk when the texture units number was 4, all TextureState objects streamed exactly 4 TexturePtr smart pointers. When the texture units number is then changed to 6 or 8 and the disk copy of the scene is loaded, the loader attempts to read more than 4 TexturePtr links, leading to a serious error. The file pointer is out of synchronization with the file contents. Wild Magic version 3 fixes that because there is no more TextureState class. Generally, any resource limits are not saved during streaming. A scene graph may contain a Geometry object that has more textures attached to it than a graphics card can support. The rendering system makes sure that the additional textures just are not processed. But that does mean you must think about the target platform for your applications. If you use 8 texture units for a single object, you should put on the software packaging that the minimum requirement is a graphics card that has 8 texture units!

3.5.3 Single-Pass Drawing

In a sense, this system is the culmination of all the work you have done regarding scene management. At some point, you have set up your scene graph, and you want to draw the objects in it. The geometry leaf nodes of the scene have been properly updated, `Spatial::UpdateGS` for the geometric information and `Spatial::UpdateRS` for the render state. The leaf nodes contain everything needed to correctly draw the object. The interface support for the entry point into the drawing system is

```
class Renderer
{
public:
    // pre- and postdraw semantics
    virtual bool BeginScene ();
    virtual void EndScene ();

    // object drawing
    void DrawScene (Node* pkScene);

protected:
    Geometry* m_pkGeometry;
    Effect* m_pkLocalEffect;

// internal use
public:
    void Draw (Geometry* pkGeometry);

    typedef void (Renderer::*DrawFunction)();
    void DrawPrimitive ();
};
```

The pair of functions `BeginScene` and `EndScene` give the graphics system a chance to perform any operations before and after drawing. The `Renderer` class stubs these to do nothing. The OpenGL renderer has no need for pre- and postdraw semantics, but the Direct3D renderer does. The function `DrawScene` is the top-level entry point into the drawing system. Wild Magic version 2 users take note: The top-level call was named `Draw`, but I changed this to make it clear that it is the entry point and used the name `Draw` internally for multipass drawing. If you forget to change the top-level calls in your version 2 applications, I have a comment waiting for you in one of the `Renderer` functions!

The typical block of rendering code in the idle-loop callback is

```
NodePtr m_spkScene = <the scene graph>;
Renderer* m_pkRenderer = <the renderer>;
```

```
// in the on-idle callback
m_pkRenderer->ClearBuffers();
if ( m_pkRenderer->BeginScene() )
{
    m_pkRenderer->DrawScene(m_spkScene);
    m_pkRenderer->EndScene();
}
m_pkRenderer->DisplayBackBuffer();
```

The `ClearBuffers` call clears the frame buffer, the depth buffer, and the stencil buffer. Predraw semantics are performed by the call to `BeginScene()`. If they were successful, `BeginScene` returns `true` and the drawing commences with `DrawScene`. The drawing is to the back buffer. On completion of drawing, postdraw semantics are performed by the call to `EndScene`. Finally, the call to `DisplayBackBuffer` requests a copy of the back buffer to the front buffer.

The `DrawScene` starts a depth-first traversal of the scene hierarchy. Subtrees are culled, if possible. When the traversal reaches a `Geometry` object that is not culled, the object tells the renderer to draw it using `Renderer::Draw(Geometry*)`. The core classes `Spatial`, `Geometry`, and `Node` all have support for the drawing pass. The relevant interfaces are

```
class Spatial : public Object
{
// internal use
public:
    void OnDraw (Renderer& rkRenderer, bool bNoCull = false);
    virtual void Draw (Renderer& rkRenderer,
        bool bNoCull = false) = 0;
};

class Node : public Spatial
{
// internal use
public:
    virtual void Draw (Renderer& rkRenderer, bool bNoCull = false);
};

class Geometry : public Spatial
{
protected:
    virtual void Draw (Renderer& rkRenderer, bool bNoCull = false);
};
```

The OnDraw and Draw functions form a recursive chain. The Draw function is pure virtual in Spatial, requiring derived classes to implement it. Node::Draw propagates the call through the scene, calling Spatial::OnDraw for each of its children. When a Geometry leaf node is encountered, Geometry::Draw is called; it is a simple wrapper for a call to Renderer::Draw(Geometry*).

The traversal for drawing is listed next. The function Renderer::DrawScene has some code for deferred drawing for the purposes of sorting, but I defer talking about this until Section 4.2.4.

```
void Renderer::DrawScene (Node* pkScene)
{
    if ( pkScene )
    {
        pkScene->OnDraw(*this);

        if ( DrawDeferred )
        {
            (this->*DrawDeferred)();
            m_iDeferredQuantity = 0;
        }
    }
}

void Node::Draw (Renderer& rkRenderer, bool bNoCull)
{
    if ( m_spkEffect == NULL )
    {
        for (int i = 0; i < m_kChild.GetQuantity(); i++)
        {
            Spatial* pkChild = m_kChild[i];
            if ( pkChild )
                pkChild->OnDraw(rkRenderer,bNoCull);
        }
    }
    else
    {
        // A "global" effect might require multipass rendering, so
        // the Node must be passed to the renderer for special
        // handling.
        rkRenderer.Draw(this);
    }
}
```

```
void Geometry::Draw (Renderer& rkRenderer, bool)
{
    rkRenderer.Draw(this);
}
```

We have already seen earlier that `Spatial::OnDraw` attempts to cull the object using its world bounding volume and the camera frustum. If the object is not culled, the `Draw` is called. In the `Node::Draw` function, the call is propagated to its children in the "then" clause. This is the typical behavior for single-pass drawing. Multipass drawing occurs when the node has a *global effect* attached to it. I discuss this later in Section 3.5.6.

This brings us to the question at hand: What does `Renderer::Draw` do with the `Geometry` object? The function is listed below. The code related to deferred drawing is discussed in Section 4.2.4.

```
void Renderer::Draw (Geometry* pkGeometry)
{
    if ( !DrawDeferred )
    {
        m_pkGeometry = pkGeometry;
        m_pkLocalEffect = pkGeometry->GetEffect();

        if ( m_pkLocalEffect )
            (this->*m_pkLocalEffect->Draw)();
        else
            DrawPrimitive();

        m_pkLocalEffect = NULL;
        m_pkGeometry = NULL;
    }
    else
    {
        m_kDeferredObject.SetElement(m_iDeferredQuantity,pkGeometry);
        m_kDeferredIsGeometry.SetElement(m_iDeferredQuantity,true);
        m_iDeferredQuantity++;
    }
}
```

The data members `m_pkGeometry` and `m_pkLocalEffect` are used to hang onto the geometric object and its effect object (if any) for use by the renderer when it does the actual drawing. Standard rendering effects use a `Renderer` function called `DrawPrimitive`. Some advanced effects require a specialized drawing function, a pointer to which the `Effect` object provides. I will discuss the advanced effects in Chapter 5.

3.5.4 The DrawPrimitive Function

Single-pass rendering of objects is performed by `Renderer::DrawPrimitive`. This function is in the base class, so it necessarily hides any dependencies of the back-end graphics API by requiring a `Renderer`-derived class to implement a collection of pure virtual functions. At a high level, the order of operations is

- set global state
- enable lighting
- enable vertices
- enable vertex normals
- enable vertex colors
- enable texture units
- set the transformation matrix
- draw the object
- restore the transformation matrix
- disable texture units
- disable vertex colors
- disable vertex normals
- disable vertices
- disable lighting

Notice the symmetry. Each "enable" step has a corresponding "disable" step. The transformation is set and then restored. The only item without a counterpart is the setting of global state. Since each object sets *all* global state, there is no need to restore the previous global state as the last step. It is possible to create a pipeline in which the objects do not bother disabling features once the objects are drawn. The problem appears, however, if one object uses two texture units, but the next object uses only one texture unit. The first object enabled the second texture unit, so someone needs to disable that unit for the second object. Either the first object disables the unit (as shown previously) after it is drawn, or the second object disables the unit before it is drawn. In either case, the texture unit is disabled before the second object is drawn. I believe my proposed pipeline is the cleanest solution—let each object clean up after itself.

Wild Magic version 2 did not use this philosophy. A reported bug showed that vertex or material colors from one triangle mesh were causing a sibling triangle mesh to be tinted with those colors. To this day I still do not know where the problem is. Wild Magic version 3 appears not to have this bug. If a bug were to show up, I guarantee the current pipeline is easier to debug.

Portions of the actual DrawPrimitive code are shown next. The block for setting the global state is

```
if ( m_bAllowGlobalState )
    SetGlobalState(m_pkGeometry->States);
```

The default value for the Boolean data member m_bAllowGlobalState is true. It exists just to give advanced rendering features the ability not to set global state if they do not want it set. The SetGlobalState function is

```
void Renderer::SetGlobalState (
    GlobalStatePtr aspkState[GlobalState::MAX_STATE])
{
    GlobalState* pkState;

    if ( m_bAllowAlphaState )
    {
        pkState = aspkState[GlobalState::ALPHA];
        SetAlphaState((AlphaState*)pkState);
    }

    //... similar blocks for the other global states go here ...
}
```

Each global state has an associated Boolean data member that allows you to prevent that state from being set. This is useful in advanced rendering features that require multiple passes through a subtree of a scene hierarchy, because each pass tends to have different requirements about the global state. For example, the projected, planar shadow sample application needs control over the individual global states. The function SetAlphaState is pure virtual in Renderer, so the derived renderer classes need to implement it. Similar "set" functions exist for the other global state classes. The implementations of the "set" functions involve direct manipulation of the graphics API calls.

The blocks for enabling and disabling lighting are

```
if ( m_bAllowLighting )
    EnableLighting();

// ... other pipeline operations go here ...

if ( m_bAllowLighting )
    DisableLighting();
```

A Boolean data member also allows you to control whether or not lighting is enabled independent of whether there are lights in the scene that illuminate the object. The default value for the data member is `true`. The base class implements `EnableLighting` and `DisableLighting`:

```
void Renderer::EnableLighting (int eEnable)
{
    int iQuantity = m_pkGeometry->Lights.GetQuantity();
    if ( iQuantity >= m_iMaxLights )
        iQuantity = m_iMaxLights;

    for (int i = 0; i < iQuantity; i++)
    {
        const Light* pkLight = m_pkGeometry->Lights[i];
        if ( pkLight->On )
            EnableLight(eEnable,i,pkLight);
    }
}

void Renderer::DisableLighting ()
{
    int iQuantity = m_pkGeometry->Lights.GetQuantity();
    if ( iQuantity >= m_iMaxLights )
        iQuantity = m_iMaxLights;

    for (int i = 0; i < iQuantity; i++)
    {
        const Light* pkLight = m_pkGeometry->Lights[i];
        if ( pkLight->On )
            DisableLight(i,pkLight);
    }
}
```

The first block of code in each function makes sure that the quantity of lights that illuminate the geometry object does not exceed the total quantity supported by the graphics API. The data member `m_iMaxLights` must be set during the construction of a derived-class renderer. In OpenGL, this number is eight. The second block of code iterates over the lights. If a light is on, it is enabled/disabled. The functions `EnableLight` and `DisableLight` are pure virtual in `Renderer`, so the derived renderer classes need to implement them. The implementations involve direct manipulation of the graphics API calls.

The blocks of code for handling the vertex positions for the geometry object are

```
EnableVertices();

// ... other pipeline operations go here ...

DisableVertices();
```

I assume that any geometry object has vertices. Otherwise, what would you draw? No Boolean member is provided to prevent the enabling of vertices. The functions `EnableVertices` and `DisableVertices` are pure virtual in `Renderer`, so the derived renderer classes need to implement them. The implementations involve direct manipulation of the graphics API calls. The OpenGL versions tell the graphics driver the vertex array to use. The engine supports caching of vertex data on the graphics card itself to avoid constantly sending vertices over an AGP bus. The enable/disable functions do all the graphics-API-specific work to make this happen.

The blocks of code for handling the vertex normals for the geometry object are

```
if ( m_bAllowNormals && m_pkGeometry->Normals )
    EnableNormals();

// ... other pipeline operations go here ...

if ( m_bAllowNormals && m_pkGeometry->Normals )
    DisableNormals();
```

The vertex normals are passed through the graphics API calls only if the geometry object has normals and the application has not prevented the enabling by setting `m_bAllowNormals` to `false`. The default value for the data member is `true`. The functions `EnableNormals` and `DisableNormals` are pure virtual in `Renderer`, so the derived renderer classes need to implement them. The implementations involve direct manipulation of the graphics API calls. The OpenGL versions tell the graphics driver the vertex normal array to use. The engine supports caching of vertex data on the graphics card itself to avoid constantly sending vertices over an AGP bus. The enable/disable functions do all the graphics-API-specific work to make this happen.

The blocks of code for handling the vertex colors for the geometry object are

```
if ( m_bAllowColors && m_pkLocalEffect )
{
    if ( m_pkLocalEffect->ColorRGBAs )
        EnableColorRGBAs();
    else if ( m_pkLocalEffect->ColorRGBs )
        EnableColorRGBs();
}

// ... other pipeline operations go here ...

if ( m_bAllowColors && m_pkLocalEffect )
{
    if ( m_pkLocalEffect->ColorRGBAs )
        DisableColorRGBAs();
    else if ( m_pkLocalEffect->ColorRGBs )
        DisableColorRGBs();
}
```

Once again, a Boolean data member controls whether or not the vertex color handling is allowed. The default value for `m_bAllowColors` is `true`. Vertex colors are not stored in the geometry object, but are considered to be one of the local effects that you can attach to an object. As such, the vertex colors are stored in the `m_pkLocalEffect` object that belongs to the `m_pkGeometry` object. Since `Effect` objects allow you to store RGB or RGBA colors, the renderer must decide which one to use. Only one set of colors is used, so setting both in the `Effect` object will lead to use of only the RGBA colors. The functions `EnableColorRGBAs`, `EnableColorRGBs`, `DisableColorRGBAs`, and `DisableColorRGBs` are pure virtual in `Renderer`, so the derived renderer classes need to implement them. The implementations involve direct manipulation of the graphics API calls. The OpenGL versions tell the graphics driver the vertex color array to use. The engine supports caching of vertex data on the graphics card itself to avoid constantly sending vertices over an AGP bus. The enable/disable functions do all the graphics-API-specific work to make this happen.

The texture units are enabled and disabled by the following code blocks:

```
if ( m_bAllowTextures )
    EnableTextures();

// ... other pipeline operations go here ...

if ( m_bAllowTextures )
    DisableTextures();
```

Again we have a Boolean data member, `m_bAllowTextures`, that gives advanced rendering features the chance to control whether or not the texture units are enabled. The default value is `true`. The base class implements `EnableTextures` and `DisableTextures`:

```
void Renderer::EnableTextures ()
{
    int iTMax, i;
    int iUnit = 0;

    // set the local-effect texture units
    if ( m_pkLocalEffect )
    {
        iTMax = m_pkLocalEffect->Textures.GetQuantity();
        if ( iTMax > m_iMaxTextures )
            iTMax = m_iMaxTextures;

        for (i = 0; i < iTMax; i++)
            EnableTexture(iUnit++,i,m_pkLocalEffect);
    }
```

```
    // set the global-effect texture units
    if ( m_pkGlobalEffect )
    {
        iTMax = m_pkGlobalEffect->Textures.GetQuantity();
        if ( iTMax > m_iMaxTextures )
            iTMax = m_iMaxTextures;

        for (i = 0; i < iTMax; i++)
            EnableTexture(iUnit++,i,m_pkGlobalEffect);
    }
}

void Renderer::DisableTextures ()
{
    int iTMax, i;
    int iUnit = 0;

    // disable the local-effect texture units
    if ( m_pkLocalEffect )
    {
        iTMax = m_pkLocalEffect->Textures.GetQuantity();
        if ( iTMax > m_iMaxTextures )
            iTMax = m_iMaxTextures;

        for (i = 0; i < iTMax; i++)
            DisableTexture(iUnit++,i,m_pkLocalEffect);
    }

    // disable the global-effect texture units
    if ( m_pkGlobalEffect )
    {
        iTMax = m_pkGlobalEffect->Textures.GetQuantity();
        if ( iTMax > m_iMaxTextures )
            iTMax = m_iMaxTextures;

        for (i = 0; i < iTMax; i++)
            DisableTexture(iUnit++,i,m_pkGlobalEffect);
    }
}
```

The first block of code in each function makes sure that the quantity of texture units that the geometry object requires does not exceed the total quantity supported by the graphics API and, in fact, by the graphics card. As you are aware, the more texture units the graphics card has, the more expensive it is. Since your clients will have cards with different numbers of texture units, you have to make sure you only try to use what is there. The data member m_iMaxTextures must be set during the construction of a derived-class renderer. In OpenGL, the graphics card driver is queried for this information. The functions EnableTexture and DisableTexture are pure virtual in Renderer, so the derived renderer classes need to implement them. The implementations involve direct manipulation of the graphics API calls. The data member m_pkPostEffect is part of the multipass rendering system that is described later in this section.

As I noted earlier, Wild Magic version 2 had a design flaw regarding multitexturing. The flaw surfaced when trying to add a Node-derived class for projected texture. The assignment of textures to texture units was the programmer's responsibility, unintentionally so. To make sure the projected texture appears as the last texture and not have any texture units just pass the previous unit's data through it, the programmer needed to know for each geometry object in the subtree how many textures it used and what units they were assigned to, which is needlessly burdensome. In Wild Magic version 3, a projected texture shows up as a "post-effect." As you can see in the EnableTextures, the texture units are enabled as needed and in order.

The transformation handling is

```
if ( m_bAllowWorldTransform )
    SetWorldTransformation();
else
    SetScreenTransformation();

// ... the drawing call goes here ...

if ( m_bAllowWorldTransform )
    RestoreWorldTransformation();
else
    RestoreScreenTransformation();
```

The graphics system needs to know the model-to-world transformation for the geometry object. The transformation is set by the function SetWorldTransformation. The world translation, world rotation, and world scales are combined into a single homogeneous matrix and passed to the graphics API. The world transformation is restored by the function RestoreWorldTransformation.

The engine supports *screen space polygons*. The polygons are intended to be drawn either as part of the background of the window or as an overlay on top of all the other rendered data. As such, the vertices are two-dimensional and are already in screen space coordinates. The perspective viewing model does not apply. Instead we need an orthonormal projection. The function SetScreenTransformation must handle both the selection of projection type and setting of the transformation. The function RestoreScreenTransformation restores the projection type and transformation. All the transformation handlers are pure virtual in the base class. This allows hiding the matrix representation that each graphics API chooses.

The *z*-values (depth values) need to be provided for the polygon vertices. The *z*-values may depend on the graphics API, so the `ScreenPolygon` class requires you only to specify whether it is a foreground or a background polygon. `ScreenPolygon` derives from `TriMesh`, which allows you to attach render state and effects just like any other geometry object. A classical use for screen space polygons is to overlay the rendered scene with a fancy border, perhaps with simulated controls such as menu selection, buttons, sliders, and so on. An overlay can have an RGBA texture assigned to it. By setting selected image pixels to have an alpha of zero, you can make the overlay as fancy as you like, with curved components, for example.

The final piece of `DrawPrimitive` is the drawing call itself:

```
DrawElements();
```

This function tells the graphics system what type of geometric object is to be drawn (points, polyline, triangle mesh, etc.). In the case of an object that has an array of indices into the vertex array, the indices are passed to the graphics system.

3.5.5 Cached Textures and Vertex Attributes

Consumer graphics hardware has become very powerful and allows a lot of computations to be off-loaded from the CPU to the GPU. For practical applications, the amount of data the GPU has to process will not cause the computational aspects to be the bottleneck in the graphics system. What has become the bottleneck now is the transfer of the data from the host machine to the graphics hardware. On a PC, this is the process of sending the vertex data and texture images across the AGP bus to the graphics card.

In Wild Magic version 2, vertex data is transferred to the graphics card each time a scene is rendered. However, the graphics APIs support caching on the graphics card for textures and their corresponding images, thus avoiding the transfer. When a texture is *bound* to the graphics card (i.e., cached on the card) the first time it is handed to the graphics API, you are given an identifier so that the next time the texture needs to be used in a drawing operation the graphics API knows it is already in VRAM and can access it directly. Support for this mechanism requires some communication between the `Renderer` and `Texture` classes.

I still use this mechanism in Wild Magic version 3. The relevant interface for the `Texture` class is

```
class Texture : public Object
{
protected:
    class BindInfo
    {
    public:
```

```
        BindInfo ();
        Renderer* User;
        char ID[8];
    };

    TArray<BindInfo> m_kBind;

// internal use
public:
    void Bind (Renderer* pkUser, int iSize, const void* pvID);
    void Unbind (Renderer* pkUser);
    void GetID (Renderer* pkUser, int iSize, void* pvID);
};
```

The nested class `BindInfo` is used by the graphics system to store the identifier to a texture. A pointer to the renderer to which the texture is bound is part of the binding information. The renderer is responsible for storing a unique identifier in the `ID` field of `BindInfo`. The array has 8 bytes to allow storage of the identifier on a 64-bit graphics system. Of course, if a graphics system requires more than 8 bytes for the identifier, this number must change. The number of bytes used is known only to the derived-class renderer and is irrelevant to the scene graph system. The size information is not saved in the `BindInfo` class for this reason. The `ID` array elements are all initialized to zero; a value of zero indicates the texture is not bound to any renderer. Public access to the binding system is labeled for internal use, so an application should not directly manipulate the functions.

When the renderer is told to use a texture, it calls `GetID` and checks the identifier. If it is zero, this is the first time the renderer has seen the texture. It then calls `Bind` and passes a pointer to itself, the size of the identifier in bytes, and a pointer to the identifier. Notice that a texture may be bound to multiple renderers; the class stores an array of `BindInfo` objects, one per renderer. The derived-class renderer does whatever is necessary to cache the data on the graphics card. The second time the renderer is told to use the texture, it calls the `GetID` function and discovers that the identifier is not zero. The derived-class renderer simply tells the graphics API that the texture is already in VRAM and should use it directly. All of the logic for this occurs through the function `Renderer::EnableTexture`. Recall that this is a pure virtual function that a derived class must implement.

At some point your application might no longer need a texture and deletes it from the scene. If that texture was bound to a renderer, you need to tell the renderer to unbind it in order to free up VRAM for other data. The smart pointer system makes the notification somewhat challenging. It is possible that the texture object was deleted automatically because its reference count went to zero. For example, this happens if a scene graph is deleted by assigning `NULL` to its smart pointer:

```
NodePtr m_spkScene = <a scene graph>;
// ... do some application stuff ...
m_spkScene = NULL;
// scene is deleted, including any Texture objects
```

If I had required you to search the scene graph for any `Texture` objects and somehow unbind them manually, that would have been a large burden to place on your shoulders. Instead, the destructor of the `Texture` class notifies the renderer that the texture is being deleted. To notify the renderer, you need to have access to it. Conveniently, the `BindInfo` nested class has a member `User` that is a pointer to the renderer to which the texture is bound. No coincidence. The destructor is

```
Texture::~Texture ()
{
    // Inform all renderers using this texture that it is being
    // destroyed.  This allows the renderer to free up any
    // associated resources.
    for (int i = 0; i < m_kBind.GetQuantity(); i++)
        m_kBind[i].User->ReleaseTexture(this);
}
```

The `Texture` object iterates over all its binding information and informs each renderer that it is being deleted. This gives the renderers a chance to unbind the texture, whereby it frees up the VRAM that the texture occupied. Once freed, the renderer in turn notifies the texture object that it is no longer bound to the renderer. The notification is via the member function `Texture::Unbind`.

Clearly, the `Renderer` class must have a function that the destructor calls to unbind the texture. This function is named `ReleaseTexture` and is a pure virtual function, so the derived class must implement it. The base class also has a function `ReleaseTextures`. This one is implemented to perform a depth-first traversal of a scene. Each time a `Texture` object is discovered, the renderer is told to release it. The texture objects are *not deleted*. If you were to redraw the scene, all the texture objects would be rebound to the renderer. Does this make the function useless? Not really. If you had a few scenes loaded into system memory, and you switch between them based on the current state of the game without deleting any of them, you certainly want to release the textures for one scene to make room for the next scene.

The graphics hardware does allow for you to cache vertex data on the card, as well as texture data. For meshes with a large number of vertices, this will also lead to a speedup in the frame rate because you do not spend all your time transferring data across a memory bus. Wild Magic version 2 did not support caching vertex data, but Wild Magic version 3 does. The vertex arrays (positions, normals, colors, indices, texture coordinates) are normally stored as shared arrays using the template class `TSharedArray`. The sharing is for the benefit of the scene graph management system. Each time a geometry object is to be drawn, its vertex arrays are given to the graphics

API for the purposes of drawing. The arrays are transferred across the memory bus on each drawing call.

To support caching, the graphics APIs need to provide a mechanism that is similar to what is used for textures, and they do. I chose to use *vertex buffer objects* for the caching. Just as the class `Texture` has the `BindInfo` nested class for storing binding information, the scene graph system needs to provide some place to store binding information for vertex data. I have done this by deriving a template class `TCachedArray` from `TSharedArray`. This class provides a system that is identical to the one in `Texture`. The texture binding occurs through the derived-class implementation of `Renderer::EnableTexture`. The vertex data caching occurs similarly through the derived-class implementations of `EnableVertices`, `EnableNormals`, `EnableColorRGBAs`, `EnableColorRGBs`, and `EnableUVs`. The derived-class implementations need only check the RTTI for the vertex arrays. If they are of type `TCachedArray`, the renderer binds the arrays and stores the identifiers in the `BindInfo` structures. If they are not of type `TCachedArray`, the renderer treats them normally and transfers the data across the memory bus on each draw operation.

The same issue arises as for textures. If the vertex data is to be deleted, and that data was bound to a renderer, the renderer needs to be notified that it should free up the VRAM used by that data. The texture objects notify the renderer through `Renderer::ReleaseTexture`. The vertex data objects notify the renderer through `Renderer::ReleaseArray` (there are five such functions—for positions, normals, color RGBs, color RGBAs, and texture coordinates). The notification occurs in the `TCachedArray` destructor. Finally, you may release all cached data by calling the function `Renderer::ReleaseArrays`. A depth-first traversal of the scene graph is made. Each cached vertex array is told to notify the renderer to free the corresponding resources and unbind the array.

3.5.6 Global Effects and Multipass Support

The single-pass rendering of a scene graph was discussed in Section 3.5.3. This system essentially draws a `Geometry` object at the leaf node of a scene hierarchy using all the global state, lights, and effects that are stored by the object. Some effects, though, may be desired for all the geometry leaf nodes in a subtree. For example, a projected texture can apply to multiple triangle meshes, as can an environment map. A projected planar shadow may be rendered for an object made up of many triangle meshes. Planar reflections also apply to objects that have multiple components. It would be convenient to allow an `Effect` object to influence an entire subtree. Wild Magic version 2 supported this by creating `Node`-derived classes to represent the effects, but that design was clunky and complicated when it came to handling *reentrancy*. An effect such as a planar projected shadow requires multiple passes to be made over a subtree of the scene. Each pass has different requirements regarding render state. If the drawing is initiated on a first pass through the subtree, and the renderer must use a second pass to complete the drawing, you have to be certain not to end up in an infinite recursion of the drawing function; that is, the drawing system must be reentrant.

The Effect class introduced in Wild Magic version 3 was initially designed to represent a *local effect;* that is, the Effect object stores the vertex colors, textures, and texture coordinates and implements any semantics necessary to correctly render the geometry object to which the effect is attached. A natural class to store the effect is the Geometry class. I still wanted to support *global effects* such as projected textures and projected planar shadows, but with a system that was better designed than the one requiring you to derive a class from Node and have it encapsulate the relevant render state. My choice was to store the Effect object in the Spatial class. In this way, a Node has an Effect object that can represent a global effect. A pleasant consequence is that a multipass drawing operation is cleanly implemented without much fuss in the scene graph management system.

A recapitulation of my previous discussion: The top-level call to drawing a scene is

```
void Renderer::DrawScene (Node* pkScene)
{
    if ( pkScene )
    {
        pkScene->OnDraw(*this);

        if ( DrawDeferred )
        {
            (this->*DrawDeferred)();
            m_iDeferredQuantity = 0;
        }
    }
}
```

The OnDraw function is implemented in Spatial and handles any culling of objects. If a Node object is not culled, the function Draw is called on all the children in order to propagate the drawing down the hierarchy. The Node class's version of Draw is

```
void Node::Draw (Renderer& rkRenderer, bool bNoCull)
{
    if ( m_spkEffect == NULL )
    {
        for (int i = 0; i < m_kChild.GetQuantity(); i++)
        {
            Spatial* pkChild = m_kChild[i];
            if ( pkChild )
                pkChild->OnDraw(rkRenderer,bNoCull);
        }
    }
```

```
    else
    {
        // A "global" effect might require multipass rendering,
        // so the Node must be passed to the renderer for special
        // handling.
        rkRenderer.Draw(this);
    }
}
```

In the typical case, the node does not have an effect attached to it, in which case `m_spkEffect` is `NULL`, and the drawing operation is propagated to the node's children. If the node has an effect attached to it, then the renderer is immediately told to draw the subtree rooted at the node. From the scene graph management perspective, all you care about is that the renderer does the right thing and correctly draws the subtree. From the renderer's perspective, if multiple drawing passes must be made over the subtree, the `Node::Draw` function must be reentrant. The only natural solution is to require the renderer to keep a temporary handle to `m_spkEffect`, set `m_spkEffect` to `NULL`, draw the subtree with multiple passes (if necessary), and then restore `m_spkEffect` to its original value.

The function referenced by `rkRenderer.Draw(this)` in the previous displayed code block is listed next. The code for deferred drawing is discussed in Section 4.2.4.

```
void Renderer::Draw (Node* pkNode)
{
    if ( !DrawDeferred )
    {
        m_pkNode = pkNode;
        m_pkGlobalEffect = pkNode->GetEffect();

        assert( m_pkGlobalEffect );
        (this->*m_pkGlobalEffect->Draw)();

        m_pkNode = NULL;
        m_pkGlobalEffect = NULL;
    }
    else
    {
        m_kDeferredObject.SetElement(m_iDeferredQuantity,pkNode);
        m_kDeferredIsGeometry.SetElement(m_iDeferredQuantity,false);
        m_iDeferredQuantity++;
    }
}
```

The global effect has a Renderer function assigned to its Draw data member. The function encapsulates the semantics necessary to correctly draw the object. Pseudocode for the drawing function is

```
void DerivedRenderer::DrawGlobalFeature ()
{
    // Hang onto the effect with a smart pointer (prevent
    // destruction).
    EffectPtr spkSaveEffect = m_pkGlobalEffect;

    // Allow reentrancy to drawing at the node m_pkNode.  By
    // having a NULL effect, the node will just propagate the
    // drawing call to its children.
    m_pkNode->SetEffect(NULL);

    // do whatever, including calls to m_pkNode->Draw(*this,...)

    // Restore the effect.
    m_pkNode->SetEffect(spkSaveEffect);
}
```

Should you add a function such as the above to support a new effect, you must add a pure virtual function to the base class, Renderer::DrawGlobalFeature. Currently, the base class has

```
virtual void DrawBumpMap () = 0;
virtual void DrawEnvironmentMap () = 0;
virtual void DrawGlossMap () = 0;
virtual void DrawPlanarReflection () = 0;
virtual void DrawPlanarShadow () = 0;
virtual void DrawProjectedTexture () = 0;
```

Corresponding Effect-derived classes are in the scene graph management system. Each one sets its Draw data member to one of these function pointers.