CHAPTER 4

HIERARCHICAL SCENE REPRESENTATIONS

SOURCE CODE

LIBRARY
Engine

FILENAME
All Files

The graphics pipeline discussed in Chapter 3 requires that each drawable object be tested for culling against the view frustum and, if not culled, be passed to the renderer for clipping, lighting, and rasterizing. Given a 3D world with a large number of objects, the simplest method for processing the objects is to group them into a list and iterate over the items in the list for culling and rendering. Although this approach may be simple, it is not efficient since each drawable object in the world must be tested for culling.

A better method for processing the objects is to group them hierarchically according to spatial location. The grouping structure discussed in this chapter is a *tree*. The tree has leaf nodes that contain geometric data and internal nodes that provide a grouping mechanism. Each node has one parent (except for the root node, which has none) and any number of child nodes. It is possible to use a directed acyclic graph as an attempt to support high-level sharing of objects. Each node in the graph can have multiple parents, each parent sharing the object represented by the subgraph rooted at the node. However, the memory costs and code complexity to maintain such a graph do not justify using it. Sharing should occur at a lower level so that leaf nodes can

share vertices, texture images, and other data that tends to use a lot of memory. The implied links from sharing are not part of the parent-child relationships in the hierarchy. Regardless of whether trees or directed acyclic graphs are used, the resulting set of grouped objects is called a *scene graph*.

The organization of content in a scene graph is quite important for games in many ways, of which four are listed here. First, the amount of content to manage is typically large and is built in small pieces by the artists. The level editor can assemble the content for a single level as a hierarchy by concentrating on the local items of interest. The global ramifications are effectively the responsibility of the hierarchy itself. For example, a light in the world can be chosen to illuminate only a subtree of the graph. The level editor's responsibility is to assign that light to a node in the graph. The effect of the light on the subtree rooted at that node is automatically handled by the scene graph management system. Second, hierarchical organization provides a form of locality of reference, a common concept in memory management by a computer system. Objects that are of current interest in the game tend to occur in the same spatial region. The scene graph allows the game program to quickly eliminate other regions from consideration for further processing. Although minimizing the data sent to the renderer is an obvious goal to keep the game running fast, focusing on a small amount of data is particularly important in the context of collision detection. The collision system can become quite slow when the number of potentially colliding objects is large. A hierarchical scene graph supports grouping only a small number of potentially colliding objects, those objects occurring only in the local region of interest in the game. Third, many objects are naturally modeled with a hierarchy, most notably humanoid characters. The location and orientation of the hand of a character is naturally dependent on the locations and orientations of the wrist, elbow, and shoulder. Fourth, invariably the game must deal with persistence issues. A player wants to save the current game, and the game is to be continued at a later time. Hierarchical organization makes it quite simple to save the state of the world by asking the root node of the scene graph to save itself, the descendants saving themselves in a naturally recursive fashion.

Section 4.1 provides the basic concepts for management of a tree-based representation of a scene, including specification and composition of local and world transforms, construction of bounding volumes for use both in rapid view frustum culling and fast determination of nonintersection of objects managed by a collision system, selection and scope of renderer state at internal or leaf nodes, and control of animated quantities.

Changes in the world environment of the game are handled by changing various attributes at the nodes of the tree. A change at a single node affects the subtree for which that node is the root. Therefore, all nodes in the subtree must be notified of the change so that appropriate action can be taken. One typical action that requires an update of the scene graph is moving an object by changing its local transform. The world transforms of the object's descendants in the tree must be recalculated. Additionally, the object's bounding volume has changed, in turn affecting all the bounding volumes of its ancestors in the tree. The new bounding volume at a node

involves computing a single bounding volume that contains all the bounding volumes of its children, a process called *merging*. Another typical action that requires an update of the scene graph is changing renderer state at a node. The renderer state at all the leaf nodes in the affected tree must be updated. The update process is the topic of Section 4.2.

After a scene graph is updated, it is ready for processing by the renderer. The drawing pass uses the bounding volumes to cull entire subtrees at once, thereby reducing the amount of time the renderer has to spend on low-level processing of objects that ultimately will not appear on the computer screen. Section 4.3 presents culling algorithms for various bounding volumes compared to a plane at a time in the view frustum. The general drawing algorithm for a hierarchy is also discussed.

4.1 Tree-Based Representation

A simple grouping structure for objects in the world is a tree. Each node in the tree has exactly one parent, except for the root node, which has none. The root is the first node to be processed when attempting to render objects in the tree. The simplest example of a tree is illustrated in Figure 4.1. The top-level node is a *grouping node* (bicycle) and acts as a *parent* for the two *child nodes* (wheels). The children are grouped because they are part of the same object both spatially and semantically.

To take advantage of this structure, the nodes must maintain spatial and semantic information about the objects they represent. The main categories of information are *transforms*, *bounding volumes*, *render state*, and *animation state*. Transforms are used to position, orient, and size the objects in the hierarchy. Bounding volumes are used for hierarchical culling purposes and intersection testing. Render state is used to set up the renderer to properly draw the objects. Animation state is used to represent any time-varying node data.

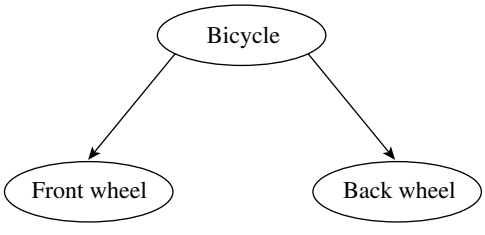

Figure 4.1 A simple tree with one grouping node.

4.1.1 TRANSFORMS

In Figure 4.1, it is not enough to know the semantic information that the two wheels are part of the bicycle. The spatial information, the location of the wheels, must also be specified. Moreover, it is necessary to know a coordinate system in which to specify that information. The parent node has its own coordinate system, and the location of a child is given relative to its parent's coordinates.

Local Transforms

The location of a node relative to its parent is represented abstractly as a homogeneous matrix with no perspective component. The matrix, called a *local transform,* represents any translation, rotation, scaling, and shearing of the node within the parent's coordinate system. While an implementation of scene graph nodes could directly store the homogeneous matrix as a 4×4 array, it is not recommended. The last row of the matrix is always $[0\ 0\ 0\ 1]$. Less memory is used if the homogeneous matrix is stored as a 3×3 matrix representing the upper-left block and a 3×1 vector representing the translation component of the matrix. This also avoids the inefficient general multiplication of homogeneous matrices and vectors since in that multiplication, there would be three multiplies by 0 and one multiply by 1. Given a homogeneous matrix with no perspective component, the matrix is denoted by

$$\left\langle M \;\middle|\; \vec{T} \right\rangle := \left[\begin{array}{c|c} M & \vec{T} \\ \hline \vec{0}^{\mathrm{T}} & 1 \end{array}\right]. \tag{4.1}$$

Using this compressed notation, the product of two homogeneous matrices is

$$\left\langle M_1 \;\middle|\; \vec{T}_1 \right\rangle \left\langle M_2 \;\middle|\; \vec{T}_2 \right\rangle = \left\langle M_1 M_2 \;\middle|\; M_1 \vec{T}_2 + \vec{T}_1 \right\rangle \tag{4.2}$$

and the product of a homogeneous matrix with a homogeneous vector $[\vec{V}|1]^{\mathrm{T}}$ is

$$\left\langle M \;\middle|\; \vec{T} \right\rangle \; \vec{V} = M\vec{V} + \vec{T}. \tag{4.3}$$

To keep the update time to a minimum and to avoid using numerical inversion of matrices in various settings, it is better to require that the local transform have only translation, rotation, and uniform scaling components. The general form of such a matrix is

$$\left\langle sR \;\middle|\; \vec{T} \right\rangle \tag{4.4}$$

and is called an *SRT-transform*. The uniform scaling factor is $s > 0$, the rotational component is the orthogonal matrix R whose determinant is one, and the translational component is $\vec{T}$. The product of two *SRT*-transforms is

$$\left\langle s_1 R_1 \;\middle|\; \vec{T}_1 \right\rangle \left\langle s_2 R_2 \;\middle|\; \vec{T}_2 \right\rangle = \left\langle s_1 s_2 R_1 R_2 \;\middle|\; s_1 R_1 \vec{T}_2 + \vec{T}_1 \right\rangle, \tag{4.5}$$

the product of an *SRT*-transform and a vector $\vec{V}$ is

$$\left\langle sR \;\middle|\; \vec{T} \right\rangle \vec{V} = sR\vec{V} + \vec{T}, \tag{4.6}$$

and the inverse of an *SRT*-transform is

$$\left\langle sR \;\middle|\; \vec{T} \right\rangle^{-1} = \left\langle \frac{1}{s} R^{\mathrm{T}} \;\middle|\; -\frac{1}{s} R^{\mathrm{T}} \vec{T} \right\rangle. \tag{4.7}$$

World Transforms

The local transform at a node specifies how the node is positioned with respect to its parent. The entire scene graph represents the world itself. The world location of the node depends on all the local transforms of the node and its predecessors in the scene graph. Given a parent node P with child node C, the *world transform* of C is the product of P's world transform with C's local transform,

$$\begin{aligned} \left\langle M_{\text{world}}^{(C)} \;\middle|\; \vec{T}_{\text{world}}^{(C)} \right\rangle &= \left\langle M_{\text{world}}^{(P)} \;\middle|\; \vec{T}_{\text{world}}^{(P)} \right\rangle \left\langle M_{\text{local}}^{(C)} \;\middle|\; \vec{T}_{\text{local}}^{(C)} \right\rangle \\ &= \left\langle M_{\text{world}}^{(P)} M_{\text{local}}^{(C)} \;\middle|\; M_{\text{world}}^{(P)} \vec{T}_{\text{local}}^{(C)} + \vec{T}_{\text{world}}^{(P)} \right\rangle. \end{aligned}$$

The world transform of the root node in the scene graph is just its local transform. The world position of a node N_k in a path $N_0 \cdots N_k$, where N_0 is the root node, is generated recursively by the above definition as

$$\left\langle M_{\text{world}}^{(N_k)} \;\middle|\; \vec{T}_{\text{world}}^{(N_k)} \right\rangle = \left\langle M_{\text{local}}^{(N_0)} \;\middle|\; \vec{T}_{\text{local}}^{(N_0)} \right\rangle \cdots \left\langle M_{\text{local}}^{(N_k)} \;\middle|\; \vec{T}_{\text{local}}^{(N_k)} \right\rangle.$$

4.1.2 Bounding Volumes

Object-based culling within a scene graph is very efficient whenever the bounding volumes of the nodes are properly nested. If the bounding volume of the parent node encloses the bounding volumes of the child nodes, culling of entire subtrees is supported. If the bounding volume of the parent node is outside the view frustum, then

the child nodes must be outside the view frustum and no culling tests need be done on the children. Hierarchical culling provides a fast way for eliminating large portions of the world from being processed by the renderer. The same nested bounding volumes support collision detection. If the bounding volume of the parent node does not intersect an object of interest, then neither do the child nodes. Hierarchical collision detection provides a fast way for determining that two objects do not intersect. The bounding volumes that are discussed in this chapter include spheres, oriented boxes, capsules, lozenges, cylinders, and ellipsoids.

A leaf node containing geometric data will also contain a bounding volume based on the model space coordinates of the data. However, the leaf node has a world space representation based on the product of local transforms from scene graph root to that leaf. That means the leaf node must also contain a world bounding volume, obtained by applying the world transform to the model bounding volume.

To support the efficiencies of a hierarchical organization of the world, an internal node requires a world bounding volume that contains the world bounding volumes of all its children. It is not necessary to maintain a model bounding volume at an internal node since such a node does not contain its own geometric data. While transforms are propagated from the root of the scene graph toward the leaf nodes, the bounding sphere calculations must occur from leaf node to root. A parent bounding volume cannot be known until its child bounding volumes are known. A recursive traversal downward allows computation of the world transforms. The upward return from the traversal allows computation of the world bounding volumes.

4.1.3 Renderer State

Renderer state can also be maintained in a hierarchical fashion. For example, if a subtree rooted at a node has all leaf nodes that want their textures to be alpha blended, the node can be tagged with state information that indicates alpha blending should be enabled for the entire subtree. Alternatively, tagging all the leaf nodes with the same renderer state information is an efficient use of memory. A traversal along a single path in the tree from root to leaf node accumulates the renderer state necessary to draw the geometry of the leaf node. Just before a leaf node is about to be drawn, the renderer processes the state information at that node and decides whether or not it needs to change its own internal state. As changes in rendering state can be expensive, the number of changes should be minimal. A typical expensive change involves using different textures. If a texture is in system memory but not in video memory, the texture must be copied to video memory, and that takes time. For sorting purposes, it is convenient to allow each leaf node to store a copy of the renderer state. A sorter can select a renderer state for which it wants to minimize changes, then sort the leaf nodes accordingly.

4.1.4 Animation

Animation in the classic sense is the motion of articulated characters and objects in the scene. If a character is represented hierarchically, each node might represent a joint (neck, shoulder, elbow, wrist, knee, etc.) whose local transformations change over time. Moreover, the values of the transformations are usually controlled by procedural means (see Chapter 9) as compared to the application manually adjusting the transforms. This can be accomplished by allowing each node to store *controllers,* with each controller managing some quantity that changes over time. In the case of classic animation, a controller might represent the local transform as a matrix function of time. For each specified time in the application, the matrix is computed by the controller and the world transform is computed using this matrix.

It is possible to allow any quantity at a node to change over time. For example, a node might be tagged to indicate that fogging is to be used in its subtree. The fog depth can be made to vary with time. A controller can be used to procedurally compute the depth based on current time. In this way animation is controlling any time-varying quantity in a scene graph.

4.2 Updating a Scene Graph

The scene graph represents the state of the world at a given time. If the state changes for whatever reason, the scene graph must be updated to represent the new state. Typical state changes include model data changing at a node, local transforms changing at a node, the topological structure of the tree changing, renderer state changing, or some animated quantity changing. Updating the scene graph is only necessary in those subtrees affected by the changes. For example, if a local transform is changed at a single node, then only the subtree rooted at that node is affected. The world transforms of descendants must be recalculated to reflect the new position and orientation of the subtree's root node. It is possible that more than one change has been made at different locations in the scene graph. An implementation of a scene graph manager can attempt to maintain the minimum number of subtree root nodes that need to be updated. For example, if the local transforms are changed at nodes A and B, and if B is a descendant of A, the update of the subtree rooted at node A will automatically update the subtree rooted at B. It would be inefficient to first update the subtree at B, then update the subtree at A.

The updating is done in a recursive pass. Transforms are updated on the downward pass; bounding volumes are updated on the upward pass that is initiated as a return from the recursive calls. Note that the upward pass should not terminate at the node at which the initial update call was made. If the bounding volume of this node has changed as a result of changes in bounding volumes of the descendants, then the parent's bounding volume might also change. Thus, the upward pass must proceed

all the way to the root of the scene graph. If transforms are animated, the update pass is responsible for asking the controllers to make the necessary adjustments to the quantities they manage before the world transform is computed. Finally, if renderer state has changed, that information must be propagated to the leaf nodes (to support sorting as mentioned earlier). A single update call can be implemented to handle all changes in the scene graph, but since renderer state tends to change independently of geometry and transform changes, it might be desirable to have separate update passes.

The computation of model bounding volumes for geometric data was already discussed in Chapter 2. The main focus in the remainder of this section is on computing the parent's bounding volume from the child bounding volumes. The expense and algorithmic complexity depends on the type of volume used. It is possible to consider all child bounds simultaneously, but practice has shown that it is easier and faster to incrementally bound the children. For a node with three or more children, a bound is found for the first two children. That bound is increased in size to include the third child bound, and so on.

4.2.1 Merging Two Spheres

Source Code

Library
Containment

Filename
ContSphere

The algorithm described here computes the smallest sphere containing two spheres. Let the spheres S_i be $|\vec{X} - \vec{C}_i|^2 = r_i^2$ for $i = 0, 1$. Define $L = |\vec{C}_1 - \vec{C}_0|$ and unit-length vector $\vec{U} = (\vec{C}_1 - \vec{C}_0)/L$. The problem can be reduced to one dimension by projecting the spheres onto the line $\vec{C}_0 + t\vec{U}$. The projected intervals in terms of parameter t are $[-r_0, r_0]$ for S_0 and $[L - r_1, L + r_1]$ for S_1.

If $[-r_0, r_0] \subseteq [L - r_1, L + r_1]$, then $S_0 \subseteq S_1$ and the two spheres merge into S_1. The test for this case is $r_0 \le L + r_1$ and $L - r_1 \le -r_0$. A single test covers both conditions, $r_1 - r_0 \ge L$. To avoid the square root in computing L, compare instead $r_1 \ge r_0$ and $(r_1 - r_0)^2 \ge L^2$.

If $[L - r_1, L + r_1] \subseteq [-r_0, r_0]$, then $S_1 \subseteq S_0$ and the two spheres merge into S_0. The test for this case is $L + r_1 \le r_0$ and $-r_0 \le L - r_1$. A single test covers both conditions, $r_1 - r_0 \le -L$. Again to avoid the square root, compare instead $r_1 \le r_0$ and $(r_1 - r_0)^2 \ge L^2$.

Otherwise, the intervals either have partial overlap or are disjoint. The interval containing the two projected intervals is $[-r_0, L + r_1]$. The corresponding merged sphere whose projection is the containing interval has radius

$$r = \frac{L + r_1 + r_0}{2}.$$

The center t-value is $(L + r_1 - r_0)/2$ and corresponds to the point

$$\vec{C} = \vec{C}_0 + \frac{L + r_1 - r_0}{2}\vec{U} = \vec{C}_0 + \frac{L + r_1 - r_0}{2L}\left(\vec{C}_1 - \vec{C}_0\right).$$

The pseudocode is

```
Input:  Sphere(C0,r0) and Sphere(C1,r1)
centerDiff = C1 - C0;
radiusDiff = r1 - r0;
radiusDiffSqr = radiusDiff*radiusDiff;
Lsqr = centerDiff.SquaredLength();
if ( radiusDiffSqr >= LSqr )
{
    if ( radiusDiff >= 0.0f )
        return Sphere(C1,r1);
    else
        return Sphere(C0,r0);
}
else
{
    L = sqrt(Lsqr);
    t = (L+r1-r0)/(2*L);
    return Sphere(C0+t*centerDiff,(L+r1+r0)/2);
}
```

4.2.2 Merging Two Oriented Boxes

If two oriented boxes were built to contain two separate sets of data points, it is possible to build a single oriented bounding box that contains the union of the sets. That box might not contain the two original oriented boxes—something that is not desired in a hierarchical decomposition of an object. Moreover, the time it takes to build the single oriented box could be expensive.

Source Code

Library
Containment

Filename
ContBox

An alternative approach is to construct an oriented box from only the original boxes and that contains the original boxes. This can be done by interpolation of the box centers and axes, then growing the box to contain the originals. Let the original two boxes have centers $\vec{C}_i$ for $i = 0, 1$. Let the box axes be stored as columns of a rotation matrix R_i. Now represent the rotation matrices by unit quaternions q_i such that the dot product of the quaternions is nonnegative, $q_0 \cdot q_1 \geq 0$. The final box is assigned center $\vec{C} = (\vec{C}_0 + \vec{C}_1)/2$. The axes are obtained by interpolating the quaternions. The unit quaternion representing the final box is $q = (q_0 + q_1)/|q_0 + q_1|$, where the absolute value signs indicate length of the quaternion as a four-dimensional vector. The final box axes can be extracted from the quaternion using the methods described in Section 2.3. The extents of the final box are computed by projecting the vertices of the two original boxes onto the final box axes and computing the extreme values.

The pseudocode is

```
// Box has center, axis[3], extent[3]
Input:  Box box0, Box box1
Output:  Box box

// compute center
box.center = (box0.center + box1.center)/2;

// compute axes
Quaternion q0 = ConvertAxesToQuaternion(box0.axis);
Quaternion q1 = ConvertAxesToQuaternion(box1.axis);
Quaternion q = q0+q1;
Real length = Length(q);
q /= Length(q);
box.axis = ConvertQuaternionToAxes(q);

// compute extents
box.extent[0] = box.extent[1] = box.extent[2] = 0;
for each vertex V of box0 do
{
    Point3 delta = V - box.center;
    for (j = 0; j < 3; j++)
    {
        Real adot = |Dot(box0.axis[j],delta)|
        if ( adot > box.extent[j] )
            box.extent[j] = adot;
    }
}
for each vertex V of box1 do
{
    Point3 delta = V - box.center;
    for (j = 0; j < 3; j++)
    {
        Real adot = |Dot(box1.axis[j],delta)|
        if ( adot > box.extent[j] )
            box.extent[j] = adot;
    }
}
```

The function `ConvertAxesToQuaternion` stores the axes as columns of a rotation matrix, then uses the algorithm to convert a rotation matrix to a quaternion. The function `ConvertQuaternionToAxes` converts the quaternion to a rotation matrix, then extracts the axes as columns of the matrix.