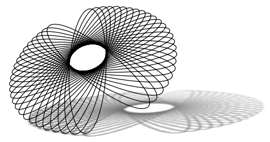
Appendix D
Scene Graphs



D.1 Introduction

Source Code
Demo
SceneGraph

In Chapter 4 of the book, we briefly discussed how to implement hierarchical data structures, and mentioned one such data structure: the scene graph. In this chapter we'll go into more detail about one possible implementation of a scene graph, used in the SceneGraph examples for Chapters 4 and 12 in the book.

The *scene graph* is meant to be used for managing hierarchical scenes, such as a collection of rooms and the objects contained within each room. While a generalized scene graph can be quite powerful, for now we will focus only on the basic structures needed for controlling hierarchical models efficiently. Although the scene graph is not necessarily a tree, for the purposes of this discussion we will be using it as such. Each object in the scene graph will have at most one parent, with one object (called the *root of the scene graph*) having no parent.

D.2 Base Implementation

The implementation that we will present is only one of many possibilities. The more we want our scene graph to do, the more complex the implementation needs to be, but for simple purposes the following will serve. It consists of three classes: `IvSpatial`, `IvNode`, and `IvGeometry`.

`IvSpatial` is the base class. It contains two copies of the transformations as member variables. The first is a transformation relative to the parent; the transformation of a propeller relative to the submarine body, for example. We'll call this the local transformation. The second is the full transformation from the object's local space into world space, which we'll use to render and interact with the subobject—we'll call this the world transformation. This is generated by an `UpdateWorldTransform()` virtual method, which multiplies the local transformation by the parent's world transformation. Finally, we define a method called `Render()`, which uses the world transformation to render each level of the hierarchy. An abbreviated class definition looks like the following.

```
class IvSpatial
{
public:
  IvSpatial();
  virtual ~IvSpatial();

  virtual void UpdateWorldTransform();
  virtual void Render() = 0;

protected:
  IvSpatial* mParent;

  float      mLocalScale;
  IvMatrix33 mLocalRotate;
  IvVector3  mLocalTranslate;

  float      mWorldScale;
  IvMatrix33 mWorldRotate;
  IvVector3  mWorldTranslate;
};
```

We define `UpdateWorldTransform()` as follows.

```
void IvSpatial::UpdateWorldTransform()
{
  if (mParent)
  {
    mWorldScale = mParent->mWorldScale*mLocalScale;
    mWorldRotate = mParent->mWorldRotate*mLocalRotate;
    mWorldTranslate = mParent->mWorldTranslate
       +mParent->mWorldScale*mParent->mWorldRotate*mLocalTranslate;
  }
  else
  {
    mWorldScale = mLocalScale;
```

```
      mWorldRotate = mLocalRotate;
      mWorldTranslate = mLocalTranslate;
    }
}
```

The method Render() has no data to work with in this case, so it will remain undefined.

While IvSpatial provides a framework for managing transformations, we will never actually allocate an instance of it as an object in our scene graph. Instead, we will use one of the following subclasses.

The subclass of IvSpatial that acts as the root and intermediary nodes of the hierarchy is called IvNode. It contains a list or array of pointers to IvSpatial objects that are the children of the node, as well as a method for adding children to the node. The UpdateWorldTransform() method overrides the default method and calls UpdateWorldTransform() for all the child IvSpatials in addition to the current node.

```
class IvNode : public IvSpatial
{
public:
  IvNode();
  virtual ~IvNode();

  virtual void UpdateWorldTransform();
  virtual void Render();

protected:
  unsigned int mNumChildren;
  IvSpatial** mChildren;
};
```

The methods UpdateWorldTransform() and Render() become as follows.

```
void IvNode::UpdateWorldTransform()
{
  IvSpatial::UpdateWorldTransform();
  unsigned int i;
  for (i = 0; i < mNumChildren; ++i )
  {
    mChildren[i]->UpdateWorldTransform();
  }
}
```

```
void IvNode::Render()
{
  unsigned int i;
  for (i = 0; i < mNumChildren; ++i )
  {
    mChildren[i]->Render();
  }
}
```

The other subclass of `IvSpatial` is called `IvGeometry`. These are the leaf nodes of the scene graph, and contain the geometric data for each subobject. One way to use `IvGeometry` is to subclass it and hard-code our geometry information, but most of the time it will contain a pointer to model data. In both cases, the world transformations are updated using the base `IvSpatial` method, called by the parent `IvNode`, so we don't implement it. However, we will need to implement a `Render()` call, which builds the 4×4 matrix that is set as our world transform.

```
class IvGeometry : public IvSpatial
{
public:
  IvGeometry();
  virtual ~IvGeometry();

virtual void Render();
};

void IvGeometry::Render()
{
    // build 4x4 matrix
    IvMatrix44 transform( mWorldRotate );
    transform(0,0) *= mWorldScale;
    transform(1,0) *= mWorldScale;
    transform(2,0) *= mWorldScale;
    transform(0,1) *= mWorldScale;
    transform(1,1) *= mWorldScale;
    transform(2,1) *= mWorldScale;
    transform(0,2) *= mWorldScale;
    transform(1,2) *= mWorldScale;
    transform(2,2) *= mWorldScale;
    transform(0,3) = mWorldTranslate.x;
    transform(1,3) = mWorldTranslate.y;
    transform(2,3) = mWorldTranslate.z;
```

```
        // set transform
        ::SetWorldMatrix( transform );

        // render geometry
    }
```

Using the scene graph is a two-step process. In step 1, we call UpdateWorldTransform() at the root level, which updates transforms via a recursive traversal from the top of the tree down to the leaf nodes. At each level, we store the updated world transforms. These transforms may now be used by the game engine for other purposes. Step 2 occurs once we're ready to render the object, when we do another recursive tree traversal by calling Render() on it. UpdateWorldTransform() does *not* have to be called on the root of the scene graph in every rendered frame. Generally, it is called once on the root object, directly following the creation of the scene graph. Thereafter, it only needs to be called at or above any and all IvNodes whose local transforms have changed since the last call to UpdateWorldTransform(). This is often a small subset of the scene graph. In other words, it is often sufficient and much faster to call UpdateWorldTransform() several times on disjoint subsections (subtrees) of the scene graph that have changed than it is to make the single call to UpdateWorldTransform() at the root of the scene.

One clarification is in order here. Consider an example of a body with arms and legs stored as a scene graph. The model data for the body itself is not itself a root node — it is a geometry leaf node that hangs directly off of the root node. This leads to some duplication of transformation information, but that is the price we pay for maintaining transformations in the base class.

One might wonder why we have two recursive calls — one for generating the new transformations and one for rendering — or, for that matter, why we bother storing the transforms at all. We could just have one recursive call that generates the world transformation at each level and then passes the result down as a function argument. At the leaf level, we would create the transformation data and then render the data directly. However, there is usually a culling step where we try to avoid rendering models that are not currently visible on the screen. As we will see, it is convenient to keep the transformation data around for this and other purposes. Scene graphs are a very flexible and modular technique, and can be mixed with other data structures and rendering systems. For example, scene graphs are sometimes used to compute hierarchical transforms without using a hierarchical Render() function to draw the scene graph. In such cases, the scene graph is used only to manipulate the local transforms of objects and update the world transforms of visible geometry. Another method (such as a flat list of all of the leaf IvGeometry objects) is used to render the scene.

D.3 Bounding Hierarchies

In Chapter 12 of the book, we mention bounding hierarchies as a way to get a better approximation to the shapes of objects. Bounding hierarchies work very well with scene graphs, and it's fairly simple to add this functionality to our existing classes. We begin by adding an `IvBoundingSphere` member to each `IvSpatial` object. In addition, our `IvGeometry` leaf nodes will have two `IvCapsule` members: one for local space and one that we'll transform into world space. This gives us our culling sphere hierarchy, with capsules as the lowest-level test (Figure D.1). Now we need to pregenerate the bounding parameters before initiating the collision test process. This is done as a part of the recursive propagation of transform information. We propagate the transform information down from the root. When we reach a leaf node, we generate a new world-space sphere and capsule from the updated transform data. Then as we undo the recursion, we propagate the changes in the bounding spheres back up. At each `IvNode` level, we merge its children's bounding spheres to obtain the sphere for the node. Note that if an update is called on a node other than the root, undoing the recursion is not sufficient. We must contain propagating the new bound upward, all the way to the root.

The procedure to merge two spheres is as follows. If one sphere completely surrounds the other, then the larger sphere is clearly the minimum

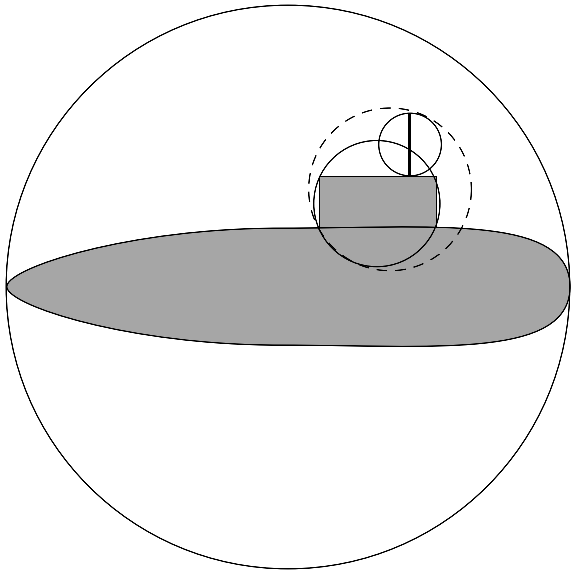


FIGURE D.1 Using bounding hierarchy.

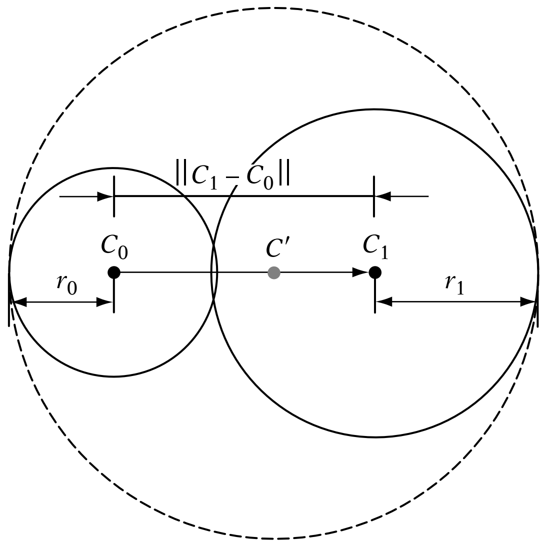


Figure D.2 Merging two spheres.

enclosing sphere. However, in most cases the two spheres are interpenetrating or separate. The situation can be seen in Figure D.2. We have two spheres: one with center C_0 and radius r_0, and the other with center C_1 and radius r_1. The diameter of the new sphere will be $r_0 + \|C_1 - C_0\| + r_1$, so the radius r will be $1/2(r_0 + r_1) + 1/2\|C_1 - C_0\|$. The new center will lie along the line $C_0 + t(C_1 - C_0)$. We determine t by moving r_0 in distance back along the line to the edge of the sphere, and then r units forward to the new center. The resulting code is as follows.

Source Code
Library
IvCollision
Filename
IvBoundingSphere

```
IvBoundingSphere Merge( const IvBoundingSphere& s0,
                        const IvBoundingSphere& s1 )
{
    IvVector3 diff = s1.mCenter - s0.mCenter;
    float distsq = diff.Dot(diff);
    float radiusdiff = s1.mRadius - s0.mRadius;
    // one sphere inside other
    if ( distsq <= radiusdiff*radiusdiff )
            if ( s0.mRadius > s1.mRadius )
                return s0;
            else
                return s1;

    // build new sphere
```

```
        float dist = ::IvSqrt( distsq );
        float newRadius = 0.5f*( s0.mRadius + s1.mRadius + dist );
        IvVector3 newCenter = s0.mCenter;
        if (!::IsZero( dist ))
            newCenter += ((newRadius-s0.mRadius)/dist)*diff;
        return IvBoundingSphere( newCenter, newRadius );
    }
```

Source Code
Library
IvScene
Filename
IvGeometry
IvNode

Finding collisions between the two hierarchies is another recursive process. We'll define a virtual method in IvSpatial called Colliding, which checks for collision between the current object and another IvSpatial object. Represented in pseudocode, this is as follows.

```
Boolean IvGeometry::Colliding(IvSpatial* other)
{
   if other is not IvGeometry node
       return other->Colliding( this )
   else
       if both spheres and capsules intersecting return TRUE
}
```

For IvNodes, we use the following.

```
Boolean IvNode::Colliding(IvSpatial* other)
{
   if bounding spheres are not colliding, return FALSE
   else
       if this node has children
           for each child do
               if child->Colliding(other)
                   return TRUE
           return FALSE
       else if other node has children
           for each child in other node do
               if other_child->Colliding(this)
                   return TRUE
           return FALSE
       else
           return FALSE  // shouldn't happen
}
```

This will find the first collision between the hierarchies. You may wish to find them all (there may be more than one if our models are not convex). If so, instead of returning TRUE immediately when a collision is found, store the collision information and proceed to the next child.

D.4 Chapter Summary

In this chapter we have covered the basics of the scene graph, a powerful data structure for managing hierarchies of transformations. We discussed the creation and management of scene graphs, first in terms of rendering functionality and secondly for the purposes of simulation. For those interested, further details on hierarchical transformation management and scene graph construction and usage can be found in Eberly [25].