



Working with Shapes, Changing State

Goals:

Build a scene from `osg::Shape` instances. Use `osg::StateSet` to control how the shapes are rendered.

Using the 'Shape' class:

The shape class is the base class for all shape types. Shapes are used either for culling and collision detection or to define the geometric shape of procedurally generated Geometry. The following classes inherit from the 'Shape' class:

- TriangleMesh
- Sphere
- InfinitePlane
- HeightField
- Cylinder
- Cone
- CompositeShape
- Box

To use one of these shapes for rendering we need to associate it with an instance of the 'Drawable' class. The class 'ShapeDrawable' allows us to do this. This class is derived from 'Drawable' and allows us to assign a 'Shape' instance to something we can render. Since the 'ShapeDrawable' class inherits from 'Drawable', ShapeDrawable instances can be added to a Geode class instance. The following steps demonstrate how to do this to add a unit cube to an empty scene:

```
// Declare a group to act as root node of a scene:
osg::Group* root = new osg::Group();

// Declare a box class (derived from shape class) instance
// This constructor takes an osg::Vec3 to define the center
// and a float to define the height, width and depth.
// (an overloaded constructor allows you to specify unique
// height, width and height values.)
osg::Box* unitCube = new osg::Box( osg::Vec3(0,0,0), 1.0f);

// Declare an instance of the shape drawable class and initialize
// it with the unitCube shape we created above.
// This class is derived from 'drawable' so instances of this
// class can be added to Geode instances.
osg::ShapeDrawable* unitCubeDrawable = new osg::ShapeDrawable(unitCube);

// Declare a instance of the geode class:
osg::Geode* basicShapesGeode = new osg::Geode();

// Add the unit cube drawable to the geode:
basicShapesGeode->addDrawable(unitCubeDrawable);

// Add the geode to the scene:
root->addChild(basicShapesGeode);
```

Creating a sphere would be similarly straight forward. Without many comments, the code would look something like this:

```
// Create a sphere centered at the origin, unit radius:
osg::Sphere* unitSphere = new osg::Sphere( osg::Vec3(0,0,0), 1.0);
osg::ShapeDrawable* unitSphereDrawable = new osg::ShapeDrawable(unitSphere);
```

Now we can add the sphere to the scene using a transform node to move it away from the cube we added at the origin. The unitSphereDrawable cannot be added to the scene directly (since it's not derived from 'node' class) so we need to use a new geode to add it:

```
osg::PositionAttitudeTransform* sphereXForm =
    new osg::PositionAttitudeTransform();
sphereXForm->setPosition(osg::Vec3(2.5,0,0));
osg::Geode* unitSphereGeode = new osg::Geode();
root->addChild(sphereXForm);
sphereXForm->addChild(unitSphereGeode);
unitSphereGeode->addDrawable(unitSphereDrawable);
```

Setting states.

The previous [tutorial](#) demonstrates how to create a texture, assign it to an image loaded from a file, and create a StateSet in which this texture is enabled. The following code sets up two state sets - one for BLEND texture mode and one for DECAL texture mode. First BLEND:

```
// Declare a state set for 'BLEND' texture mode
osg::StateSet* blendStateSet = new osg::StateSet();

// Declare a TexEnv instance, set the mode to 'BLEND'
osg::TexEnv* blendTexEnv = new osg::TexEnv();
blendTexEnv->setMode(osg::TexEnv::BLEND);

// Turn the attribute of texture 0 - the texture we loaded above - 'ON'
blendStateSet->setTextureAttributeAndModes
    (0,KLN89FaceTexture,osg::StateAttribute::ON);
// Set the texture texture environment for texture 0 to the
// texture environment we declared above:
blendStateSet->setTextureAttribute(0,blendTexEnv);
```

Repeat these steps to create a second state set for DECAL texture mode.

```
osg::StateSet* decalStateSet = new osg::StateSet();
osg::TexEnv* decalTexEnv = new osg::TexEnv();
decalTexEnv->setMode(osg::TexEnv::DECAL);
decalStateSet->setTextureAttributeAndModes
    (0,KLN89FaceTexture,osg::StateAttribute::ON); decalStateSet-
->setTextureAttribute(0,decalTexEnv);
```

Now that we've create state sets we can apply them to nodes in the scene graph. States are accumulated as the scene graph during a draw (root->leaf) traversal. Unless a node has a state assigned to it, it will inherit its state from its parent node. (This means that if a node has more than one parent, it can be rendered using more than one state.)

```
root->setStateSet(blendStateSet);
unitSphereGeode->setStateSet(decalStateSet);
```

The last step is to enter the simulation loop:

```
viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
viewer.setSceneData( root );

viewer.realize();

while( !viewer.done() )
{
```

```
viewer.sync();  
viewer.update();  
viewer.frame();  
}  
return 0;
```

Download source [here](#).

This tutorial set's [index](#), [contact page](#) [author](#). (jasullivan <at> nps <dot> edu)

Jason McVeigh's OpenSceneGraph [tutorial set](#).