OpenSceneGraph – Sample Code Snippets

Fundamentals

General Notes: You should have a basic understanding of basic graphics concepts, including scene graphs, before attempting to apply the examples given in this document.

Build Notes: To make your Visual Studio projects build properly, you must add **osg.lib** to your *Additional Dependencies* under the Linker->Input settings (under Unix/Linux it is **libosg.so**).

Create a simple shape

```
#include <osg/Geode>
#include <osg/ShapeDrawable>
```

```
osg::ref_ptr<osg::Geode> myNode = new osg::Geode

myGeode->addDrawable(new osg::ShapeDrawable(new osg::Sphere());
```

Notes: Predefined shape primitives include Box, Capsule, Cone, Cylinder, InfinitePlane & Sphere. Other classes are provided also to support more complex shapes. Initial properties of the shape primitives can be set at creation time or afterward (see API documentation for details).

Load object(s) from file

```
#include <osg/Node>
```

```
osg::ref_ptr<osg::Node> myNode
    = osgDB::readNodeFile("MyModel.obj");
```

Notes: Node files must be in a format supported by OSG. The node returned may be a Group node with children corresponding to the various objects defined in the file. Linking requires the library **osgDB.lib** (or **libosgDB.so** under Unix/Linux).

Attach a node under another

```
someNode->addChild(otherNode.get()); // if otherNode is a ref_ptr

someNode->addChild(otherNode); // if otherNode is a raw pointer
```

Notes: The parent node must be a Group or a node type derived from Group. This code creates a subgraph, which must ultimately be attached to your scenegraph at the root node or elsewhere.

Transform a node/subgraph

```
#include <osg/PositionAttitudeTransform>
```

```
// Create transformation node
osg::ref_ptr<osg::PositionAttitudeTransform> myPat
    = new osg::PositionAttitudeTransform;
myPat->setPosition(osg::Vec3(1.0, ydist, zdist));
myPat->setScale(osg::Vec3(xscale, yscale, zscale));
myPat->setAttitude(osg::Quat(angle_in_rads, axis_of_rot_vec));

// Attach node to be transformed
myPat->addChild(myNode.get());
```

Notes: Don't forget to attach the transform node to the graph, whether at the root or elsewhere. Other types of transform nodes are also available; the PAT is easy to use for simple transforms.

Setup the Viewer object

```
#include <osgViewer/Viewer>
#include <osgViewer/ViewerEventHandlers>
```

```
osgViewer::Viewer viewer;
viewer.setSceneData(root.get());

viewer.addEventHandler(new osgViewer::StatsHandler);
viewer.addEventHandler(new osgViewer::WindowSizeHandler);
viewer.addEventHandler(
    new osgGA::StateSetManipulator(
        viewer.getCamera()->getOrCreateStateSet()));
viewer.setCameraManipulator(new osgGA::TrackballManipulator());
viewer.realize();
```

Notes: The handlers above provide user controls to display performance statistics, switch rendering mode, toggle fullscreen. Linking requires **osgViewer.lib** (or **libosgViewer.so** under Unix/Linux).

Begin the main event loop

```
return (viewer.run());
```

Notes: This should be the last line in your `main` function.

Add a light source to the scene

```
#include <osg/LightSource>
```

```
osg::ref_ptr<osg::LightSource> ls = new osg::LightSource;

ls->getLight()->setPosition(osg::Vec4(1,-1, 1, 0)); // make 4th coord 1 for point
ls->getLight()->setAmbient(osg::Vec4(0.2, 0.2, 0.2, 1.0));
ls->getLight()->setDiffuse(osg::Vec4(0.7, 0.4, 0.6, 1.0));
ls->getLight()->setSpecular(osg::Vec4(1.0, 1.0, 1.0, 1.0));

root->addChild(ls.get());
```

Notes: This code creates a directional (“distant”) light source; change the fourth coordinate to 1 in order to make it a point source. Feel free to set the light properties to whatever values suit your specific scene; some properties such as attenuation an spot-exponent are not shown here. If you want multiple lights, you can add them all to a Group node, and then add the group to the root of the scene graph. You will also need to set the light number for each light so they don't interfere with one another.

Set custom material properties

```
#include <osg/Material>
```

```
osg::ref_ptr<osg::Material> mat = new osg::Material;

mat->setColorMode(osg::Material::DIFFUSE);
mat->setAmbient (osg::Material::FRONT_AND_BACK, osg::Vec4(0.2, 0.2, 0.2, 1.0));
mat->setDiffuse (osg::Material::FRONT_AND_BACK, osg::Vec4(0.9, 0.3, 0.4, 1.0));
mat->setSpecular(osg::Material::FRONT_AND_BACK, osg::Vec4(1.0, 1.0, 1.0, 1.0));
mat->setShininess(osg::Material::FRONT_AND_BACK, 64);

someObjNode->getOrCreateStateSet()->
    setAttributeAndModes(mat.get(), osg::StateAttribute::ON);
```

Notes: You can set the RGBA color values as you wish to achieve the desired effect.

Optimize your scene graph for better performance

```
#include <osgUtil/Optimizer>
```

```
osgUtil::Optimizer optimizer;
optimizer.optimize( root.get() );
```

Notes: Check the frame rate before and after you optimize to see the difference. For simple scenes, the difference may be negligible, but more complex scenes should noticeably benefit. Linking requires **osgUtil.lib** (or **libosgUtil.so** under Unix/Linux).

Attach a texture to an object

```
#include <osg/Texture2D>
```

```
osg::ref_ptr<osg::Texture2D> myTex
    = new osg::Texture2D(osgDB::readImageFile("myTexture.jpg"));

myNode->getOrCreateStateSet()->
    setTextureAttributeAndModes(0, myTex.get());
```

Notes: Image file must be a format supported by OSG. Don't forget to attach the object node to your scene graph, whether at the root or elsewhere. Linker requirements here are the same as those described above for loading a model from a node file.

Set a custom camera

```
#include <osg/Camera>
```

```
osg::ref_ptr<osg::Camera> myCam = new osg::Camera;
myCam->setClearColor(osg::Vec4(0, 0, 0, 1)); // black background

// set dimensions of the view volume
myCam->setProjectionMatrixAsPerspective(30, 4.0 / 3.0, 0.1, 100);

// set position and orientation of the viewer
myCam->setViewMatrixAsLookAt(
    osg::Vec3(0, -10, 10), // eye above xy-plane
    osg::Vec3(0, 0, 0),    // gaze at origin
    osg::Vec3(0, 0, 1));   // usual up vector

viewer.setCamera(myCam); // attach camera to the viewer
```

Notes: The *clear color* may be set to any desired RGBA color. Values shown for the projection matrix give a view very similar to the default, but feel free to experiment. The Viewer object can have multiple cameras attached, but it is simplest to start out with just one.

Make the camera track a specific node

```
#include <osgGA/NodeTrackerManipulator>
```

```
osg::ref_ptr<osgGA::NodeTrackerManipulator> manip
   = new osgGA::NodeTrackerManipulator;

manip->setTrackNode(frisbeeNode.get());
manip->setTrackerMode(osgGA::NodeTrackerManipulator::NODE_CENTER);

viewer.setCameraManipulator(manip.get());
```

Notes: Consult the OpenSceneGraph API reference for information about other tracker modes.

Add a custom shader program to an object

```
#include <osg/Shader>
```

```
osg::ref_ptr<osg::Program> shadeProg(new osg::Program);

osg::ref_ptr<osg::Shader> vertShader(
    osg::Shader::readShaderFile(osg::Shader::VERTEX, filename1));

osg::ref_ptr<osg::Shader> fragShader(
    osg::Shader::readShaderFile(osg::Shader::FRAGMENT, filename2));

//Bind each shader to the program
shadeProg->addShader(vertShader.get());
shadeProg->addShader(vertShader.get());

//Attaching the shader program to the node
osg::ref_ptr<osg::StateSet> objSS = objNode->getOrCreateStateSet();
objSS->setAttribute(shadeProg.get());
```

Notes: An alternative to providing shaders as files is to directly include them in your source code.

Add rain/snow to your scene

```
#include <osgParticle/PrecipitationEffect>
```

```
osg::ref_ptr<osgParticle::PrecipitationEffect> precipNode
    = new osgParticle::PrecipitationEffect;

precipNode->setWind(osg::Vec3(xdir, ydir, zdir));
precipNode->setParticleSpeed(0.4);
precipNode->rain(0.3); // alternatively, use snow

root->addChild(precipNode.get());
```

Notes: You need not manually set each property of the precipitation, such as speed or wind; most of these settings have some sensible default value, but feel free to experiment. Linking requires **osgParticle.lib** (or **libosgParticle.so** under Unix/Linux).

Add other fancy particle effects

```
#include <osgParticle/FireEffect>
#include <osgParticle/SmokeEffect>
#include <osgParticle/SmokeTrailEffect>
#include <osgParticle/FireEffect>
```

```
osg::ref_ptr<osgParticle::ParticleEffect> effectNode =
   new osgParticle::FireEffect;

effectNode->setTextureFileName("fire.rgb");
effectNode->setIntensity(2.5);
effectNode->setScale(4);

someNode->addChild(effectNode.get());
```

Notes: If using only one kind of effect, then you only need the corresponding include directive. When using multiple effects, each effect must be added to the scene graph separately. Adding a particle effect as a child of a transformation will cause the effect to be modified by the transform. The texture file may be any texture image of your choosing, though some effects have color properties apart from the textures so the original texture colors may not show faithfully. Linker requirements are the same as those given above for precipitation.

Add simple shadowing to your scene

```
#include <osgShadow/ShadowedScene>
#include <osgShadow/ShadowMap>
```

```
osg::ref_ptr<osgShadow::ShadowedScene> shadowScene
    = new osgShadow::ShadowedScene;
osg::ref_ptr<osgShadow::ShadowMap> sm = new osgShadow::ShadowMap;
shadowScene->setShadowTechnique(sm.get());
shadowScene->addChild(lightSource.get());
shadowScene->addChild(scene.get());
```

Notes: The light source and scene nodes must already have been created. The scene node represents a Group node under which all the objects in your scene must appear. There are other shadow techniques besides ShadowMap; however, they require more effort to get them working. If you are using PrecipitationEffect, then you must be careful *not* to add the effect as a child under the ShadowedScene, otherwise your program will be overloaded as it tries to render shadows for each raindrop. Finally, don't forget to add your ShadowedScene to either your root node or your viewer (if the ShadowedScene is the root node). Linking requires **osgShadow.lib** (or **libosgShadow.so** under Unix/Linux).

Turn part of your graph on/off

```
#include <osg/Switch>
```

```
osg::ref_ptr<osg::Switch> switchNode = new osg::Switch;

switchNode->addChild(someNode.get());
switchNode->addChild(otherNode.get());

switchNode->setAllChildrenOff();
switchNode->setSingleChildOn(1);

root->addChild(switchNode.get());
```

Notes: You may add many children to a single switch, and the code used to turn on/off a given node(s) may be placed inside an event handler (for example, to toggle node in response to a keystroke or mouse click). Of course, we must remember to attach the switch itself to our scene graph at some desired point (not necessarily at the root, though the example here shows that design).

Animate your scene

```
#include <osg/NodeCallback>
```

```
class MyCallback : public osg::NodeCallback
{
public:
    virtual void operator() (osg::Node* n, osg::NodeVisitor* nv)
    {
        // your code to change the scene (one frame) goes here
    }
private:
    // perhaps some useful data variable can go here
};
```

```
someNode->setUpdateCallback(new MyCallback);
```

Notes: Any attached update callback will execute every time we process the attached node during the update traversal of the main processing loop. Be aware that the implementation of the update callback class may be a bit involved. It is possible to add multiple update callbacks so that each successive one wraps the previous, but it is again best to start with just one at a time.