Tutorial 5: Rendering a simple geometry in OSG

Franclin Foping
franclin@netcourrier.com

April 25, 2008

Abstract

In this tutorial, we will write our first OSG program. The code itself aims at rendering a simple geometry (a capsule) in OSG. From the previous tutorials, we have learnt in depth the architecture of OSG as well as some vital C++ requirements. We will see how OSG simplifies the process by providing wrappers. Therefore, coders should spend their time on the actual design not on the low-level programming.

1 Structure of the tutorial

The tutorial will begin by describing the scene graph in section 2, I will talk about OSG core classes in section 3. Output of the OSG code will be presented in section 4 paving the way for the conclusion in section 5. The tutorial will be closed by giving some exercises in section 6, remember that practice always makes perfect!

2 The scene graph

The first issue we need to address when building an OSG code is the scene graph. After all, what we want to simulate on screen has to be modelled first. That is where scene graphs are important. In this tutorial, we only want to render a simple geometry so our scene graph can be modelled as shown by the following sketch. As we can see, there are four actors on our

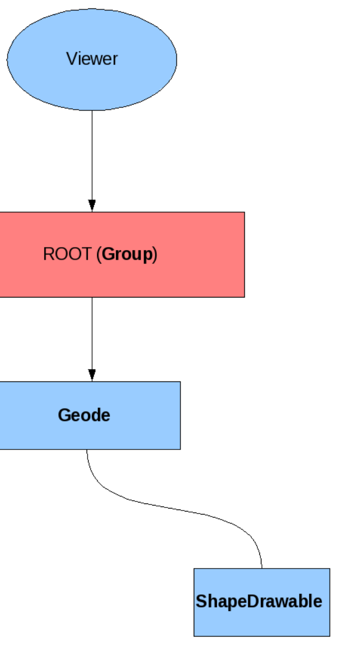

Figure 1: A scene graph

sketch: the **Viewer class**, the **Group class**, the **geode class** and finally the **ShapeDrawable class**. Of course, all of them are **node class**. The next section will focus on dwelling on these classes. Two of them are ubiquitous: the **viewer** and the **group class** so you have to understand how they work.

3 Description of core classes

1. *Node class*: this is the interface of every internal node in the scene graph. It also provides methods to enable operations on the scene graph such as traversals, culling, callbacks and state management.

2. *The Group class* is the base class for any node that can get children.

It provides a parent interface. This class is actually one of the most important classes in OSG because this is where users should organize the scene graph.

3. *The ShapeDrawable class* is a class that derives from *osg::Drawable*. It's purpose is to allow the creation of common predefined shapes. Once created, these shapes can be added to the scene graph using an *osg::Geode*.

4. *The Geode class* is derived from the *osg::Node* class and designed to hold osg::Drawable and their derivatives. In fact, Geode stands for Geometry-Node. Note that scene geometry (scene data) cannot be added directly to the SceneGraph. That is, I cannot add a osg::Geometry to the *osg::SceneView*. Instead, we must add the geometry to one of these intermediate containers, such as *osg::Geode*.

5. *The viewer class* manages multiple synchronized cameras to render a single view spanning multiple monitors. Viewer creates its own window (s) and graphics context (s) based on the underlying graphics system capabilities, so a single Viewer-based application executable run on single or multiple display systems.

4 Source code presentation

Less than 40 lines of code were necessary to do the job as we can see.

```

int main()
{
    //Creating the viewer
        osgViewer::Viewer viewer;

    //Creating the root node
    osg::ref_ptr<osg::Group> root (new osg::Group);

    //The geode containing our shape
    osg::ref_ptr<osg::Geode> myshapegeode (new osg::Geode);

    //Our shape: a capsule, it could have been any other geometry (a box, plane, cylinder etc.)
    osg::ref_ptr<osg::Capsule> myCapsule (new osg::Capsule(osg::Vec3f(),1,2));

    //Our shape drawable
    osg::ref_ptr<osg::ShapeDrawable> capsuledrawable (new osg::ShapeDrawable(myCapsule.get()));

    myshapegeode->addDrawable(capsuledrawable.get());

    root->addChild(myshapegeode.get());

    viewer.setSceneData( root.get() );

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

The first two lines were reserved to the preprocessor (like any other C++ code). We just needed to include the header files for the viewer and the

shapeDrawable. There is no need to include unneccessary header files. For instance including *osg/Geode* would have been a waste of time especially for the compiling process.
The rest of the code starts by the main function in which it is self-explanatory. We started by creating a *viewer* object and other objects (root, geode, shape-drawable etc.), we also added the so-called *geode* object to the root as its child. The pseudo-code can be summarized as follows:

1. Create scene objects (root, viewer, geode, shape, shapedrawable).
2. Add the geode node to the root as its child.
3. Add the shapedrawable object to the geode node
4. Specify to the viewer which node will be used as the reference node for rendering (the root node).
5. Fire-off threads and entering the simulation loop. This is where the traversals studied at the first tutorial will be started.

The output of all these is shown at the following picture:

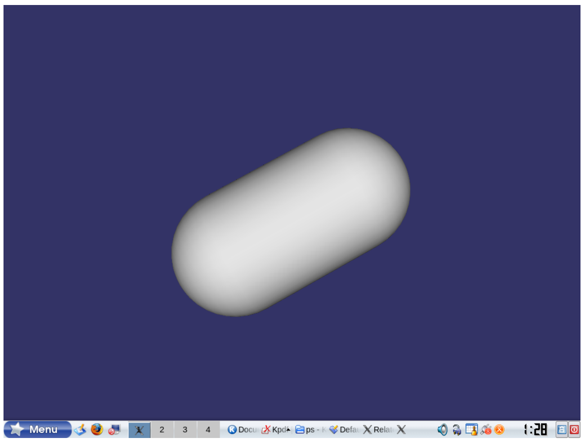

Figure 2: A capsule

5 Conclusion

Congratulations! you have just completed and written your first OSG code. Hopefully, you have managed to survive to all these lenghty steps. The reason for this is mainly because I wanted to give you a deep description of the whole process. So now, you are ready to do some exercises.
I have provided with this tutorial, the full source code, if you want to compile it you just need to type **make** in the prompt. I have also provided the **makefile** so everything is ready for you. Enjoy!!!

6 Exercises

1. There are many other shapes that could have been used in lieu of the capsule such as: Box, Sphere, Cylinders and so on. You task is to study all of them and try to change the capsule with each one. Hint: There are all defined in the osg header file.

2. You may have noticed that the majority of objects were created with smart pointers (*ref_ptr*). However, the viewer object has been created on the stack without a smart pointer attached to it, why do you think the viewer should not be created with smart pointers?

3. Look closely at the line 30 in which I specified to the viewer where to get the data for rendering. The node used there was the root node. What happen if you replace it with the geode node?

4. Still talking about the viewer, we are going to do a tweaking: uncomment the lines 4, 5, 34, 37 and 40 in the source code and recompile it. Describe what happen when you press the following keys at runtime? ’s’ , ’f’, ’l’ and ’w’ keys?