A Short Introduction to the Basic Principles of the Open Scene Graph

Leandro Motta Barros

Start: August 17$^{\text{th}}$ 02005

Contents

Chapter 1

The Basics

. . . [1] TODO

Before talking about the Open Scene Graph (OSG), it is interesting to spend a little time giving some clues about a slightly more fundamental question.

1.1 The question is: "what is a scene graph?"

As the name suggests, a scene graph is data structure used to organize a scene in a Computer Graphics application. The idea is that a scene is usually decomposed in several different parts, and somehow these parts have to be tied together. So, a scene graph is a graph where every node represents one of the parts in which a scene can be divided. Being a little more strict, a scene graph is a directed acyclic graph, so it establishes a hierarchical relationship among the nodes.

Suppose you want to render a scene consisting of a road and a truck. A scene graph representing this scene is depicted in Figure 1.1.

It turns out that there is a great chance that if you render this scene just like it is, the truck will not appear on the place you want. Most likely, you'll have to translate it to its right position. Fortunately, scene graph nodes don't

[1] **TODO: Something to think about: I'm always using the notation `namespace::class`. Perhaps, omitting the namespace can improve readability. Perhaps the namespace should be included just on the first time a class name is mentioned?**

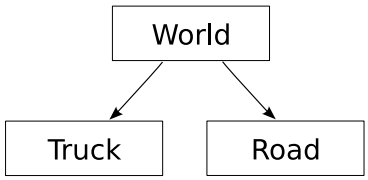

Figure 1.1: A scene graph, consisting of a road and a truck.

always represent geometry.[2] In this case, you can add a node representing a translation, yielding the scene graph shown on Figure 1.2.

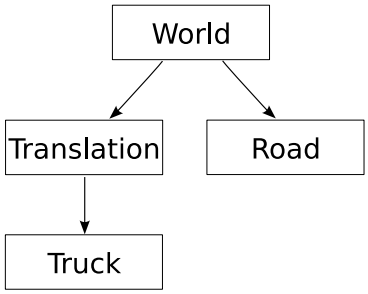

Figure 1.2: A scene graph, consisting of a road and a translated truck.

Perhaps you are now wondering why is a scene graph called a graph if they all look like trees? Well, the examples so far were trees, but that is not always the case. Let's add two boxes to the scene, one on the truck, the other one on the road. Both boxes will have translation nodes above them, so that they can be placed at their proper locations. Furthermore, the box on the truck will also be translated by the truck translation, so that if we move the truck, the box will move, too. The news is that, since both boxes look exactly the same, you don't have to create a node for each one of them. One node "referenced" twice does the trick, as Figure 1.3 illustrates. During rendering, the "Box" node will be visited (and rendered) twice, but some memory is spared because the model is loaded just once.

Of course, scene graphs can get more complicated than this. Hopefully, though, the simple notion just presented is enough for now. So, it's time to say a couple more words concerning a second fundamental question.

[2]Indeed, the node labeled "World" in Figure 1.1 doesn't represent geometry, it represents a group of some nodes (namely, "Truck" and "Road").

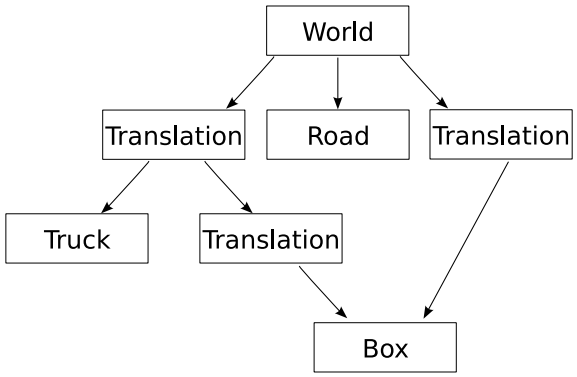

Figure 1.3: A scene graph, consisting of a road, a truck and a pair of boxes.

1.2 The question is: "who cares?"

Anyone needing a neat data structure to organize a Computer Graphics scene and wanting to render the scene efficiently cares.

. . .[3] TODO

1.3 Something OSG-related, at last

Up to this point, the discussion was around "generic" scene graphs. From now on, all examples will use exclusively OSG scene graphs, that is, instead of using a generic "Translation" node, we'll be using an instance of a real class defined in the OSG hierarchy.

A node in OSG is represented by the `osg::Node` class. Although technically possible, there is not much use in instantiating `osg::Node`s. Things start to get interesting when we look at some of `osg::Node`'s subclasses. In this chapter, three of these subclasses will be introduced: `osg::Geode`, `osg::Group` and `osg::PositionAttitudeTransform`.

Renderable things in OSG are represented by instances of the `osg::Drawable` class. But `osg::Drawable`s are not nodes, so we cannot attach them directly to a scene graph. It is necessary to use a "geometry node", `osg::Geode`, instead.

Not every node in an OSG scene graph can have other nodes attached to them as children. In fact, we can only add children to nodes that are

[3] **TODO: This is the place to say what is a scene graph good for.**

instances of `osg::Group` or one of its subclasses.

Using `osg::Geode`s and an `osg::Group`, it is possible to recreate the scene graph from Figure 1.1 using real classes from OSG. The result is shown in Figure 1.4.

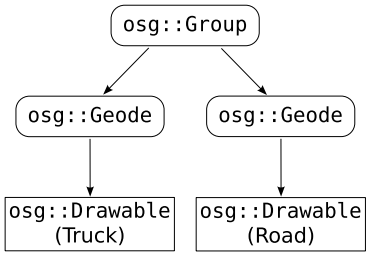

Figure 1.4: An OSG scene graph, consisting of a road and a truck. Instances of OSG classes derived from `osg::Node` are drawn in rounded boxes with the class name inside it. `osg::Drawable`s are represented as rectangles.

That's not the only way to translate the scene graph from Figure 1.1 to a real OSG scene graph. More than one `osg::Drawable` can be attached to a single `osg::Geode`, so that the scene graph depicted in Figure 1.5 is also an OSG version of Figure 1.1.

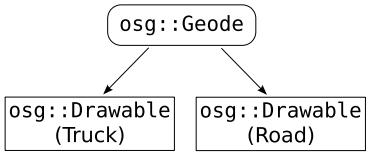

Figure 1.5: An alternative OSG scene graph representing the same scene as the one in Figure 1.4.

The scene graphs of Figures 1.4 and 1.5 has the same problem as the one in the Figure 1.1: the truck will probably be at the wrong position. And the solution is the same as before: translating the truck. In OSG, probably the simplest way to translate a node is by adding an `osg::PositionAttitudeTransform` node above it. An `osg::PositionAttitudeTransform` has associate to it not only a translation, but also an attitude and a scale. Although not exactly the same thing, this can be though as the OSG equivalent to the OpenGL calls `glTranslate()`, `glRotate()` and `glScale()`. Figure 1.6 is the OSGfied version of Figure 1.2.

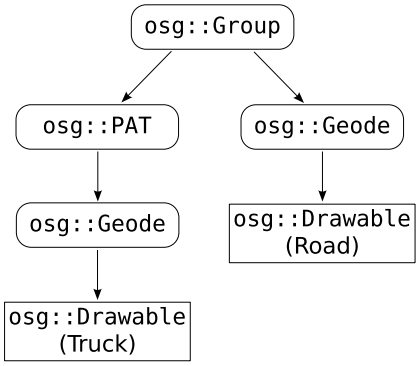

Figure 1.6: An OSG scene graph, consisting of a road and a translated truck. For compactness reasons, `osg::PositionAttitudeTransform` is written as `osg::PAT`.

For completeness, Figure 1.7 the OSG way to represent the "generic" scene graph from Figure 1.3.

1.4 Smart pointers and OSG

> *Save the whales. Feed the hungry. Free the mallocs.*
> — *fortune(6)*

Sadly, it looks like quite a few C++ users are unfortunate enough to not be proficient with smart pointers. Since OSG uses smart pointers extensively[4], it seems worthwhile to spend some time explaining them. Don't dare to skip this section if "smart pointers" sounds like Greek for you (and you are not Greek, that's it).

Let's start with a definition: a *resource* is something must be allocated before being used and deallocated when no longer needed. Perhaps the most common resource we use when programming is heap memory, but many other examples exist. Two common cases are files (which must be closed after being opened) and database transactions (which have to be committed or rolled back after being "beginned"). Also in OpenGL there are some examples of resources (one example are texture names generated by `glGenTextures()` which must be freed by `glDeleteTextures()`).

[4]Indeed, every program should.

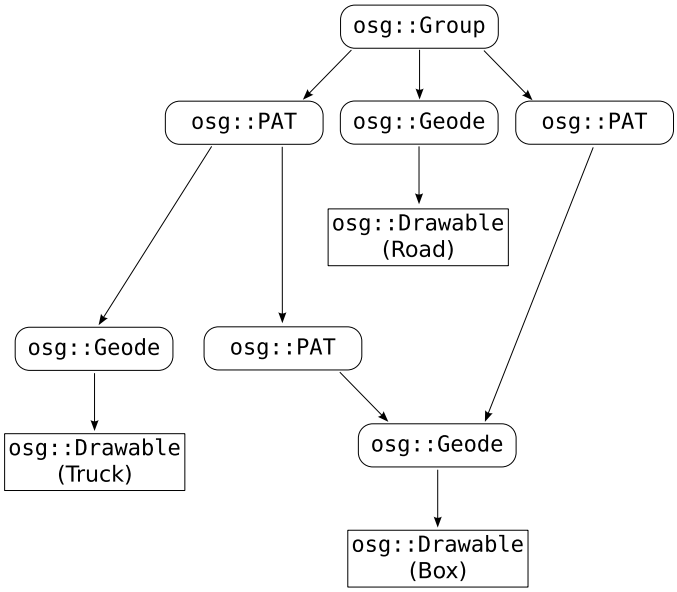

Figure 1.7: An OSG scene graph, consisting of a road, a truck and a pair of boxes.

The most fascinating thing related to resources is the fact that there exist so many programmers who believe that they can handwrite code capable of freeing them in every case *and* will never forget to write such code. This thinking only leads to resource leaks. The good news is that, with some discipline, this freeing task can be passed to the C++ compiler, which is much more reliable than us for tasks like this.

The main ideas behind resource management in C++ are worth of mentioning here, but complete discussion about this is beyond the scope of this text.[5] Speaking of "scope", the scope of "automatic" variables (that is, variables allocated on the stack) plays a central role in resource management in C++: the language rules guarantee that the destructor of an object allocated on the stack will be called when it gets out of scope. How does this help to avoid resource leaks? Take a look at the following class:

TODO

```
class ThingWrapper
{
   public:
      ThingWrapper() { handle_ = AllocateThing(); }
```

[5] **TODO: Indicate a good reference about this.**

```
        ~ThingWrapper() { DeallocateThing (handle_); }
        ThingHandle& get() { return handle_; }
    private:
        ThingHandle handle_;
};
```

It allocates a `Thing` in the constructor and frees it in the destructor. So, whenever we need a `Thing` we can do something like this:

```
ThingWrapper thing;
UseThing (thing.get());
```

Instantiating a `ThingWrapper` allocates a Thing (in `ThingWrapper`'s constructor). But the nice part is that the Thing will be automatically freed when `thing` gets out of scope, since its destructor is guaranteed to execute when this happens. Voilà. Automatic resource management.

The class `ThingWrapper` is an example of a C++ programming technique usually called "resource acquisition is initialization" (RAII). A smart pointer is simply a class[6] that uses the RAII technique to automatically manage heap memory. Quite like `ThingWrapper`, but instead of calling hypothetical `AllocateThing()` and `DeallocateThing()` functions, a smart pointer typically receives a pointer to newly allocated memory in its constructor and uses the C++ operator `delete` to free that memory in the destructor.

In the `ThingWrapper` example, `thing` is said to be owner of the Thing allocated with `AllocateThing()`, and therefore is responsible for deallocating it. In OSG, there is an extra detail to complicate the things a little bit: sometimes an object has more than one owner.[7] For example, in the scene graph shown in Figure 1.7, the `osg::Geode` with the box attached to it has two parents. Which one should be responsible for deallocating it?

In these cases, the resource shall not be deallocated while there is at least one reference pointing to it. So, most objects in OSG have an internal counter on the number of references pointing to it.[8] The resource (that is, the object) will only be destroyed when its internal reference count goes down to zero.

[6] Or, more commonly, a class template.

[7] This additional complication is not an OSG exclusivity. "Shared ownership", as it is also called, is a common situation in practice.

[8] To be more exact, the objects with an embedded reference count are all those that are instances of classes derived from `osg::Referenced`.

Fortunately, we programmers are not expected to manage these reference counts manually: that's why smart pointers exist for. So, in OSG, smart pointers are implemented as a class template named `osg::ref_ptr<>`. Whenever[9] an OSG object receives a pointer to another OSG object, it is immediately stored in an `osg::ref_ptr<>`. This way, the reference count of the underlying object is automatically managed, and the object will be automatically deallocated when it is no longer being referenced by anyone.

TODO

The example below shows OSG's smart pointers in action. The example is followed by some notes about it.

SmartPointers.cpp

```
1  #include <cstdlib>
2  #include <iostream>
3  #include <osg/Geode>
4  #include <osg/Group>
5
6  void MayThrow()
7  {
8     if (rand() % 2)
9        throw "Aaaargh!";
10 }
11
12 int main()
13 {
14    try
15    {
16       srand(time(0));
17       osg::ref_ptr<osg::Group> group (new osg::Group());
18
19       // This is OK, albeit a little verbose.
20       osg::ref_ptr<osg::Geode> aGeode (new osg::Geode());
21       MayThrow();
22       group->addChild (aGeode.get());
23
24       // This is quite safe, too.
25       group->addChild (new osg::Geode());
26
27       // This is dangerous! Don't do this!
28       osg::Geode* anotherGeode = new osg::Geode();
29       MayThrow();
30       group->addChild (anotherGeode);
31
32       // Say goodbye
33       std::cout << "Oh, fortunate one. No exceptions, no leaks.\n";
34    }
35    catch (...)
36    {
```

[9] **TODO: "Whenever"? This *really* "whenever"? No exceptions? I think I read somewhere that this is at least how things should be (it's a bug otherwise). Anyway, I'd like to check before telling this to everybody.**

```
37        std::cerr << "'anotherGeode' possibly leaked!\n";
38    }
39 }
```

Concerning the example above, the first thing to notice is that it gives a first and rough idea on how to create "compose" scene graphs like the ones shown in the figures on Section 1.3. (For now, this is just for curiosity sake. The next chapter will address this properly). The real intent of this example is showing two safe ways of using OSG's smart pointers and one dangerous way to not use them:

- Lines 20 to 22, show one safe way to use the smart pointers: an `osg::ref_ptr<>` (called `aGeode`) is explicitly created and initialized with a newly allocated `osg::Geode` (the resource) in line 20. At this point, the reference count of the geode allocated on the heap equals to one (since there is just one `osg::ref_ptr<>`, namely `aGeode`, pointing to it.)

  A little bit latter, on line 22, the geode is added as a child of a group. As soon as this happens, the group increments the geode's reference to two.

  Now, what happens if something bad happens? What happens if the call to `MayThrow()` at line 21 actually throws? Well, `aGeode` will get out of scope and will be destroyed. Its destructor will decrement the geode's reference count. And, since it was decremented to zero, it will also properly dispose the geode. There is no memory leak.

- Line 25 does more or less the same thing as the previous case. The difference is that the geode is allocated with `new` and added as group's child in a single line of code. This is quite safe, too, because there are not many bad things that can happen in between (after all, there is no in between.)

- The bad, wrong, dangerous and condemned way to manage memory is shown from line 28 to line 30. It looks like the first case, but geode is allocated with `new` but stored in a "dumb" pointer. If the `MayThrow()` at line 29 throws, nobody will call `delete` on the geode and it will leak.

  There is another thing that can be said here: `osg::Referenced`'s destructor isn't even public, so you are not able to say `delete anotherGeode`.

Instances of classes derived from `osg::Referenced` (like `osg::Geode`) are simply meant to be managed automatically by using `osg::ref_ptr<>`s.

So, do the right thing and never write code like in this third case.

TODO . . . [10]

[10] **TODO: I wonder if I should say *en passant* that one can `ref()` and `unref()` `osg::ref_ptr<>`s manually if strictly necessary. This is most likely a bad idea, but. . .**

Chapter 2

Two 3D Viewers

In this chapter we'll finally have OSG programs that actually show something on the screen. Both are viewers of 3D models, and illustrate many concepts.

2.1 A very simple viewer

The first viewer is a very simple one. Basically, all it does is loading the file passed as a command-line parameter and displaying it on the screen. So, without further delays, here is its source code.

VerySimpleViewer.cpp

```
1  #include <iostream>
2  #include <osgDB/ReadFile>
3  #include <osgProducer/Viewer>
4
5  int main (int argc, char* argv[])
6  {
7     // Check command-line parameters
8     if (argc != 2)
9     {
10       std::cerr << "Usage: " << argv[0] << " <model file>\n";
11       exit (1);
12    }
13
14    // Create a Producer-based viewer
15    osgProducer::Viewer viewer;
16    viewer.setUpViewer (osgProducer::Viewer::STANDARD_SETTINGS);
17
18    // Load the model
19    osg::ref_ptr<osg::Node> loadedModel = osgDB::readNodeFile(argv[1]);
20
21    if (!loadedModel)
22    {
```

```
23      std::cerr << "Problem opening '" << argv[1] << "'\n";
24      exit (1);
25   }
26
27   viewer.setSceneData (loadedModel.get());
28
29   // Enter rendering loop
30   viewer.realize();
31
32   while (!viewer.done())
33   {
34      // Wait for all cull and draw threads to complete.
35      viewer.sync();
36
37      // Update the scene by traversing it with the the update visitor which
38      // will call all node update callbacks and animations.
39      viewer.update();
40
41      // Fire off the cull and draw traversals of the scene.
42      viewer.frame();
43   }
44
45   // Wait for all cull and draw threads to complete before exit.
46   viewer.sync();
47 }
```

This example is pretty simple, but there are quite a few things that can be said about it. First, notice that OSG, just like OpenGL, is independent of windowing system. Thus, the task of creating a window with a proper OpenGL context to draw in is not handled by OSG. In this first example (and in most other examples to come[1]) this job will be handled by a library called Open Producer (or simply Producer). Producer is designed to be efficient, portable, scalable and it can be easily used with OSG.

TODO

So, our first example uses a Producer-based viewer instantiated on line 15. Line 16 sets the viewer up with its standard settings, which include quite a lot of useful features.[2]

TODO

OSG knows how to read (and write) several formats of 3D models and images, and all the functions and classes related to this are declared in the namespace `osgDB`. Line 19 uses of these functions, `osgDB::readNodeFile()`, which takes as parameter the name of a file containing a 3D model and returns a pointer to an `osg::Node`. The returned node contains all information

[1] **TODO: Perhaps all of them?**

[2] **TODO: Yes, it has quite a lot of useful features, but I'm not feeling like describing them right now. Try pressing keys and moving the mouse around while running the example to discover some of these features.**

necessary to render the 3D properly, including, for example, vertices, polygons, normals and texture maps.

The node returned by `osgDB::readNodeFile()` is ready to be added to a scene graph. In fact, in this simple example, it is the whole scene graph: notice that at line 27 we tell the viewer what it is expected to view, and it is exactly the node we got by calling this function.

The call at line 30 "realizes" the viewer's window, that is, it creates the window with a proper OpenGL context. And that's pretty much all the program does. From this point on, we just make some additional calls to ensure that our program keeps running forever.[3] The loop from line 32 to line 43 is the typical main loop of an application based on OSG and Producer.[4] And the final call at line 46 simply waits for any remaining threads to complete before going on. TODO

2.2 A simple (and somewhat buggy) 3D viewer

...[5] TODO

...[6] TODO

[3] Or until the user presses ESC, whatever comes first.

[4] **TODO: And should be better explained here. It is important to make a few notes about the multithreaded nature of Producer and a very brief explanation about the cull/draw/update phases (and this last point should be explained in more depth somewhere else).**

[5] **TODO: The next example loads n models passed as command-line parameters. They are attached to an `osg::Switch`, and things are made such that just of them is active at a time. The example also has an event handler that allows the user to select which one of them is the active. Also, every model has an `osg::PositionAttitudeTransform` above it, and the user can change their scale. The model will get darker or lighter as the scale changes (because normals are not normalized, this is the "buggy" part). This is the hook for the next chapter, in which the problem will be fixed by using a `StateSet`.**

[6] **TODO: Figure 2.1 shows the scene graph for the "simple and buggy viewer". Notice the "???": there may be things below the node returned by `osgDB::readNodeFile()` (at least, there is an `osg::Drawable`. A geode is also mandatory, but perhaps the node returned is this geode. Anyway, the point is that it doesn't matter.) Also, notice the ellipsis, indicating that all models passed as command-line parameters are added to the scene graph.**

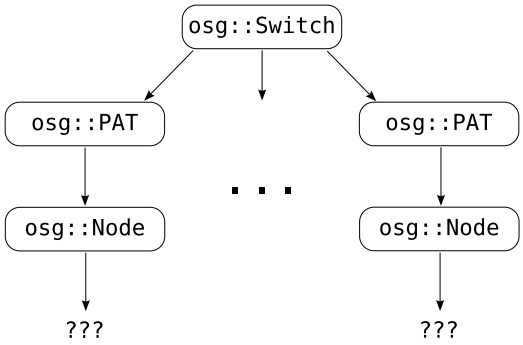

Figure 2.1: The OSG scene graph used in the "simple and buggy viewer". For compactness reasons, `osg::PositionAttitudeTransform` is written as `osg::PAT`.

SimpleAndBuggyViewer.cpp

```
1  #include <iostream>
2  #include <osg/PositionAttitudeTransform>
3  #include <osg/Switch>
4  #include <osgDB/ReadFile>
5  #include <osgGA/GUIEventHandler>
6  #include <osgProducer/Viewer>
7
8  osg::ref_ptr<osg::Switch> TheSwitch;
9  unsigned CurrentModel = 0;
10
11 class ViewerEventHandler: public osgGA::GUIEventHandler
12 {
13    public:
14       virtual bool handle (const osgGA::GUIEventAdapter& ea,
15                            osgGA::GUIActionAdapter&)
16       {
17          if (ea.getEventType() == osgGA::GUIEventAdapter::KEYUP)
18          {
19             switch (ea.getKey())
20             {
21                // Left key: select previous model
22                case osgGA::GUIEventAdapter::KEY_Left:
23                   if (CurrentModel == 0)
24                      CurrentModel = TheSwitch->getNumChildren() - 1;
25                   else
26                      --CurrentModel;
27
28                   TheSwitch->setSingleChildOn (CurrentModel);
29
30                   return true;
31
32                // Right key: select next model
33                case osgGA::GUIEventAdapter::KEY_Right:
34                   if (CurrentModel == TheSwitch->getNumChildren() - 1)
```

```
35                      CurrentModel = 0;
36                   else
37                      ++CurrentModel;
38
39                   TheSwitch->setSingleChildOn (CurrentModel);
40
41                   return true;
42
43                // Up key: increase the current model scale
44                case osgGA::GUIEventAdapter::KEY_Up:
45                {
46                   osg::ref_ptr<osg::PositionAttitudeTransform> pat =
47                      dynamic_cast<osg::PositionAttitudeTransform*>(
48                         TheSwitch->getChild (CurrentModel));
49                   pat->setScale (pat->getScale() * 1.1);
50
51                   return true;
52                }
53
54                // Down key: decrease the current model scale
55                case osgGA::GUIEventAdapter::KEY_Down:
56                {
57                   osg::ref_ptr<osg::PositionAttitudeTransform> pat =
58                      dynamic_cast<osg::PositionAttitudeTransform*>(
59                         TheSwitch->getChild (CurrentModel));
60                   pat->setScale (pat->getScale() / 1.1);
61                   return true;
62                }
63
64                // Don't handle other keys
65                default:
66                   return false;
67             }
68          }
69          else
70             return false;
71       }
72 };
73
74
75 int main (int argc, char* argv[])
76 {
77    // Check command-line parameters
78    if (argc < 2)
79    {
80       std::cerr << "Usage: " << argv[0]
81                 << " <model file> [ <model file> ... ]\n";
82       exit (1);
83    }
84
85    // Create a Producer-based viewer
86    osgProducer::Viewer viewer;
87    viewer.setUpViewer (osgProducer::Viewer::STANDARD_SETTINGS);
88
89    // Create the event handler and attach it to the viewer
90    osg::ref_ptr<osgGA::GUIEventHandler> eh (new ViewerEventHandler());
```

```
91          viewer.getEventHandlerList().push_front (eh);
92
93      // Construct the scene graph
94      TheSwitch = osg::ref_ptr<osg::Switch> (new osg::Switch());
95
96      for (int i = 1; i < argc; ++i)
97      {
98         osg::ref_ptr<osg::Node> loadedNode = osgDB::readNodeFile(argv[i]);
99         if (!loadedNode)
100           std::cerr << "Problem opening '" << argv[i] << "'\n";
101        else
102        {
103           osg::ref_ptr<osg::PositionAttitudeTransform> pat(
104              new osg::PositionAttitudeTransform());
105           pat->addChild (loadedNode.get());
106           TheSwitch->addChild (pat.get());
107        }
108     }
109
110     // Ensure that we have at least on model before going on
111     if (TheSwitch->getNumChildren() == 0)
112     {
113        std::cerr << "No 3D model was loaded. Aborting...\n";
114        exit (1);
115     }
116
117     viewer.setSceneData (TheSwitch.get());
118
119     TheSwitch->setSingleChildOn (0);
120
121     // Enter rendering loop
122     viewer.realize();
123
124     while (!viewer.done())
125     {
126        viewer.sync();
127        viewer.update();
128        viewer.frame();
129     }
130
131     // Wait for all cull and draw threads to complete before exit
132     viewer.sync();
133 }
```

TODO ...[7]

[7] **TODO: Say relevant things using this example as base.**

Chapter 3

Enter the `StateSets`

There is a very important class in OSG that was not mentioned so far: `osg::StateSet`. It is so important that this whole chapter is dedicated to it. But, in order to understand the importance of `osg::StateSet`s, one must have some basic understanding on how does OpenGL work. This OpenGL background is briefly discussed in the net section. If you are already tired of reading things entitled "OpenGL as a state machine" feel free to skip to Section 3.2. Otherwise, keep reading.

3.1 OpenGL as a state machine

OpenGL can be roughly seen as something that transforms vertices into pixels. Essentially, the programmer says: "Hey, OpenGL, please process this list of points in 3D space for me." And, shortly after, OpenGL answers: "Done! The results are on your 2D screen." This is not a 100% accurate or complete description of OpenGL, but for the purposes of this chapter it is good enough.

So, OpenGL takes vertices and makes pixels. Suppose we pass four vertices to OpenGL. Let's call them v_1, v_2, v_3 and v_4. Which pixels should they originate? Or, rephrasing the question: how should they be rendered? To begin with, what do these vertices represent? Four "isolated" points? A quadrilateral? Two line segments (v_1–v_2 and v_3–v_4)? Perhaps three line segments (v_1–v_2, v_2–v_3 and v_3–v_4)? And why not something else?

Going in other direction, what color should the pixels be? Are the rendered things affected by any light source? If they are, how many light sources

are there, where they are and what are their characteristics? And are they texture mapped?

We could keep asking questions like these for ages (or pages, at least), but let's stop here. The important thing to notice is that, although OpenGL is essentially transforming vertices into pixels, there are lots of different ways to perform this transformation. And, somehow, we must be able to "configure" OpenGL so that it does what we want. But how to configure this plethora of settings?

Divide and conquer. There are tons of settings, but they are orthogonal. This means that we can change, for example, lighting settings without touching the texture mapping settings. Of course there is interaction among the settings, in the sense that the final color of a pixel depends on both the lighting and texture mapping settings (and others). The important idea is that they can be set independently.

From now on, let's call these OpenGL settings by their more proper names: attributes and modes (the difference between an attribute and a mode is not important right now). So, OpenGL has a set of attributes and modes, and this set of attributes and modes define precisely how OpenGL behaves. But people soon noticed that writing a long expression like "set of attributes and modes" is very tiresome, and hence they gave it a shorter name: "state".

And this explains the title of this section. OpenGL can be seen as a state machine. All the important details that define exactly how vertices are transformed into pixels are part of the OpenGL state. If we were drawing green things and now want to draw blue things, we have to change the OpenGL state. If we were drawing things with lighting enabled and now want to draw things with lighting disabled, we have to change the OpenGL state. The same goes for texture mapping and everything else.

The obvious question is "how do we change the OpenGL state when using OSG?" This is answered in the rest of this chapter.

3.2 OSG and the OpenGL state

TODO

. . . [1]

[1] **TODO: Write an introduction about `StateSets`. Tell the difference between "mode" and "attribute". Give some high level examples. Link to the next section.**

3.3 A simple (and bugless) 3D viewer

. . .[2] TODO

SimpleAndBuglessViewer.cpp

```
#include <iostream>
#include <osg/PositionAttitudeTransform>
#include <osg/Switch>
#include <osgDB/ReadFile>
#include <osgGA/GUIEventHandler>
#include <osgProducer/Viewer>

osg::ref_ptr<osg::Switch> TheSwitch;
unsigned CurrentModel = 0;

class ViewerEventHandler: public osgGA::GUIEventHandler
{
   public:
      virtual bool handle (const osgGA::GUIEventAdapter& ea,
                           osgGA::GUIActionAdapter&)
      {
         if (ea.getEventType() == osgGA::GUIEventAdapter::KEYUP)
         {
            switch (ea.getKey())
            {
               // Left key: select previous model
               case osgGA::GUIEventAdapter::KEY_Left:
                  if (CurrentModel == 0)
                     CurrentModel = TheSwitch->getNumChildren() - 1;
                  else
                     --CurrentModel;

                  TheSwitch->setSingleChildOn (CurrentModel);

                  return true;

               // Right key: select next model
               case osgGA::GUIEventAdapter::KEY_Right:
                  if (CurrentModel == TheSwitch->getNumChildren() - 1)
                     CurrentModel = 0;
                  else
                     ++CurrentModel;

                  TheSwitch->setSingleChildOn (CurrentModel);

                  return true;

               // Up key: increase the current model scale
               case osgGA::GUIEventAdapter::KEY_Up:
```

[2] **TODO: The idea here is to fix the bug in the previous example by calling `ss->setMode(GL_NORMALIZE,osg::StateAttribute::ON)`. Perhaps this is too little change to justify a new example. If it is, we could also use a more complex `osg::StateAttribute`.**

```
45                {
46                   osg::ref_ptr<osg::PositionAttitudeTransform> pat =
47                      dynamic_cast<osg::PositionAttitudeTransform*>(
48                         TheSwitch->getChild (CurrentModel));
49                   pat->setScale (pat->getScale() * 1.1);
50
51                   return true;
52                }
53
54                // Down key: decrease the current model scale
55                case osgGA::GUIEventAdapter::KEY_Down:
56                {
57                   osg::ref_ptr<osg::PositionAttitudeTransform> pat =
58                      dynamic_cast<osg::PositionAttitudeTransform*>(
59                         TheSwitch->getChild (CurrentModel));
60                   pat->setScale (pat->getScale() / 1.1);
61                   return true;
62                }
63
64                // Don't handle other keys
65                default:
66                   return false;
67             }
68          }
69          else
70             return false;
71       }
72 };
73
74
75 int main (int argc, char* argv[])
76 {
77    // Check command-line parameters
78    if (argc < 2)
79    {
80       std::cerr << "Usage: " << argv[0]
81                 << " <model file> [ <model file> ... ]\n";
82       exit (1);
83    }
84
85    // Create a Producer-based viewer
86    osgProducer::Viewer viewer;
87    viewer.setUpViewer (osgProducer::Viewer::STANDARD_SETTINGS);
88
89    // Create the event handler and attach it to the viewer
90    osg::ref_ptr<osgGA::GUIEventHandler> eh (new ViewerEventHandler());
91         viewer.getEventHandlerList().push_front (eh);
92
93    // Construct the scene graph
94    TheSwitch = osg::ref_ptr<osg::Switch> (new osg::Switch());
95
96    for (int i = 1; i < argc; ++i)
97    {
98       osg::ref_ptr<osg::Node> loadedNode = osgDB::readNodeFile(argv[i]);
99       if (!loadedNode)
100         std::cerr << "Problem opening '" << argv[i] << "'\n";
```

```
101       else
102       {
103          osg::ref_ptr<osg::StateSet> ss (loadedNode->getOrCreateStateSet());
104          ss->setMode (GL_NORMALIZE, osg::StateAttribute::ON);
105          osg::ref_ptr<osg::PositionAttitudeTransform> pat(
106             new osg::PositionAttitudeTransform());
107          pat->addChild (loadedNode.get());
108          TheSwitch->addChild (pat.get());
109       }
110    }
111
112    // Ensure that we have at least on model before going on
113    if (TheSwitch->getNumChildren() == 0)
114    {
115       std::cerr << "No 3D model was loaded. Aborting...\n";
116       exit (1);
117    }
118
119    viewer.setSceneData (TheSwitch.get());
120
121    TheSwitch->setSingleChildOn (0);
122
123    // Enter rendering loop
124    viewer.realize();
125
126    while (!viewer.done())
127    {
128       viewer.sync();
129       viewer.update();
130       viewer.frame();
131    }
132
133    // Wait for all cull and draw threads to complete before exit
134    viewer.sync();
135 }
```

. . .[3] TODO

3.4 Beyond

. . .[4] TODO

[3] **TODO: The only difference from the previous example are lines 103 and 104.**

[4] **TODO: Perhaps this is the time to talk about those `ON`, `OFF`, `OVERRIDE`, `PROTECTED` and `INHERIT` parameters to `osg::StateSet::setAttribute()`. But... if I want to talk about them I must first understand all of them!**

Appendix A

Rough equivalences between OpenGL and OSG

. . . [1] TODO

OpenGL	OSG	Pages
`glTranslate()`	`osg::PositionAttitudeTransform`	Not yet
`glRotate()`	`osg::PositionAttitudeTransform`	Not yet
`glColor()`	`osg::Material`	Not yet

[1] **TODO: I think it is a good idea to have something like this. As can be easily seen, there is no real content in this table yet, just some examples (and they lack the page numbers).**