Minicurso apresentado no:
IX Symposium on Virtual and Augmented Reality - SVR 2007. Petrópolis, RJ, Maio 2007.

SVR 2007 - Minicurso MC-4

Open Scene Graph: conceitos básicos e aplicações em realidade virtual

Open Scene Graph: Basic Concepts and Applications in Virtual Reality

Leandro Motta Barros
UNISINOS
lmb <at> stackedboxes.org

Luiz Gonzaga da Silveira Jr
UNISINOS
luiz <at> omni3.org

Alberto Barbosa Raposo
Tecgraf / PUC-Rio
abraposo <at> tecgraf.puc-rio.br

April 27, 2007

Contents

1 Introduction

This document is a short introduction to Open Scene Graph (OSG). It is open source, cross-platform and high performance 3-D graphics toolkit. It intends to be useful in fields such as visual simulation, games, virtual reality, scientific visualization and modelling. So, Written in Standard C++

and OpenGL, it runs on all Windows platforms, OSX, GNU/Linux, IRIX, Solaris, HP-Ux, AIX and FreeBSD operating systems.

The OSGtoolkit is an extensible high-level abstraction layer above OpenGL, that provides more productive programming interface to support the development of computer graphics application. A hierarchical structure for efficient rendering, employment of memory management techniques, and capabilities for 2-D/3-D models, give all conditions to OSG to be a good choice for graphics programmers.

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.

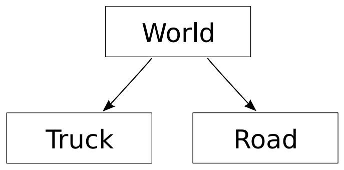

Figure 1: A scene graph, consisting of a road and a truck.

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 always represent geometry.[1] In this case, you can add a node representing a translation, yielding the scene graph shown on Figure 2.

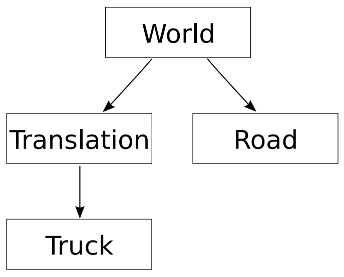

Figure 2: A scene graph, consisting of a road and a translated truck.

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 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.

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. Scene graphs expose model's geometry and rendering state, in case OSG do it related to OpenGL and also provide additional features and functionalities. The scene graph

[1]Indeed, the node labeled "World" in Figure 1 doesn't represent geometry, it represents a group of some nodes (namely, "Truck" and "Road").

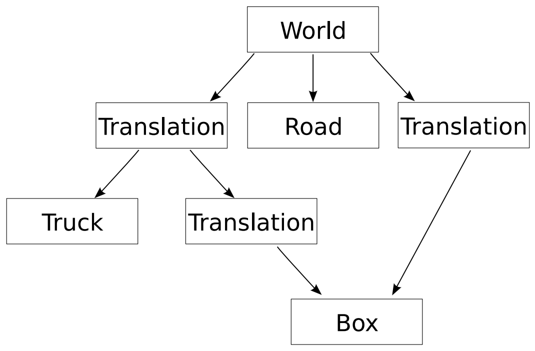

Figure 3: A scene graph, consisting of a road, a truck and a pair of boxes.

tree structure lends an intuitive spatial organization, view frustum and occlusion culling, view-dependent rendering at varying level of details, state change minimization, file I/O, and rendering effects support.

A simplest scene graph implementation provides mechanisms to store geometry and appearance, and draw the scene by performing structure traversal. So, the main traversal task is to send stored data to the graphic card, trough OpenGL calls. OSG performs traversal taking account scene updates, culling and finally the drawing. Observe that stereo applications require more than one traversal by frame.

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 section, 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 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 using real classes from OSG. The result is shown in Figure 4.

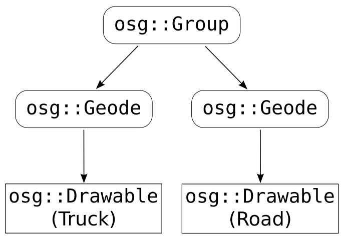

Figure 4: An OSG scene graph, consisting of a road and a truck. Instances of OSG classes derived from `osg::Node` are drawn in rounded boxes with the class name inside it. `osg::Drawable`s are represented as rectangles.

That's not the only way to translate the scene graph from Figure 1 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 5 is also an OSG version of Figure 1.

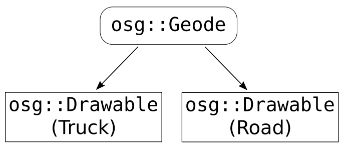

Figure 5: An alternative OSG scene graph representing the same scene as the one in Figure 4.

The scene graphs of Figures 4 and 5 has the same problem as the one in the Figure 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 6 is the OSGfied version of Figure 2.

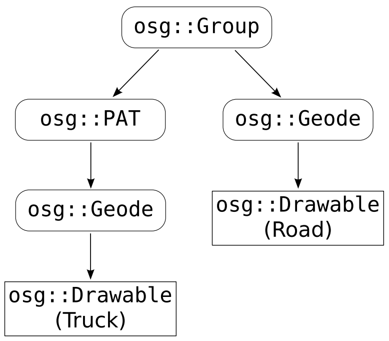

Figure 6: An OSG scene graph, consisting of a road and a translated truck. For compactness reasons, `osg::PositionAttitudeTransform` is written as `osg::PAT`.

For completeness, Figure 7 the OSG way to represent the "generic" scene graph from Figure 3.

1.4 Memory Management

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[2], 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 that must be allocated before being used and deallocated when no longer needed. Perhaps

[2]Indeed, every program should.

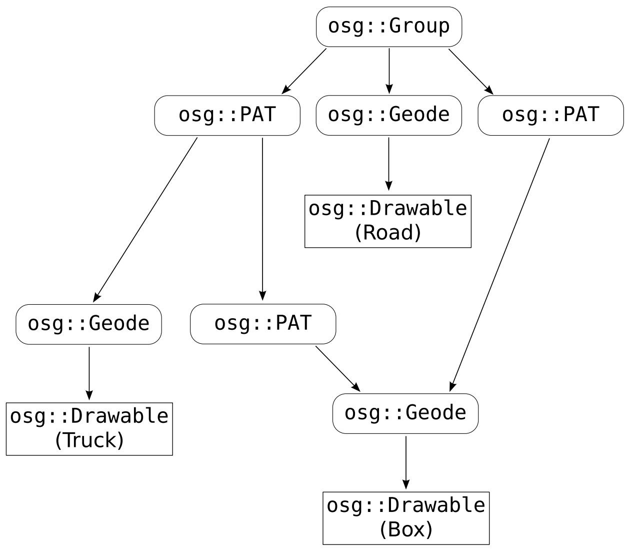

Figure 7: An OSG scene graph, consisting of a road, a truck and a pair of boxes.

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()`).

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. 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:

```
class ThingWrapper
{
   public:
      ThingWrapper() { handle_ = AllocateThing(); }
      ~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[3] 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.[4] For example, in the scene

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

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

graph shown in Figure 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.[5] The resource (that is, the object) will only be destroyed when its internal reference count goes down to zero.

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 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.

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

SmartPointers.cpp

```
#include <cstdlib>
#include <iostream>
#include <osg/Geode>
#include <osg/Group>

void MayThrow()
{
   if (rand() % 2)
      throw "Aaaargh!";
}

int main()
{
   try
   {
      srand(time(0));
      osg::ref_ptr<osg::Group> group (new osg::Group());

      // This is OK, albeit a little verbose.
```

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

```
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    {
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 section 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.

2 Two 3D Viewers

In this section 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

```
#include <iostream>
#include <osgDB/ReadFile>
#include <osgProducer/Viewer>

int main (int argc, char* argv[])
{
   // Check command-line parameters
   if (argc != 2)
   {
      std::cerr << "Usage: " << argv[0] << " <model file>\n";
      exit (1);
   }

   // Create a Producer-based viewer
   osgProducer::Viewer viewer;
   viewer.setUpViewer (osgProducer::Viewer::STANDARD_SETTINGS);

   // Load the model
   osg::ref_ptr<osg::Node> loadedModel = osgDB::readNodeFile(argv[1]);

   if (!loadedModel)
   {
      std::cerr << "Problem opening '" << argv[1] << "'\n";
      exit (1);
   }

   viewer.setSceneData (loadedModel.get());

   // Enter rendering loop
   viewer.realize();

   while (!viewer.done())
   {
      // Wait for all cull and draw threads to complete.
      viewer.sync();

      // Update the scene by traversing it with the the update visitor which
      // will call all node update callbacks and animations.
      viewer.update();

      // Fire off the cull and draw traversals of the scene.
      viewer.frame();
   }

   // Wait for all cull and draw threads to complete before exit.
   viewer.sync();
}
```

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) 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.

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. 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.

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 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.[6] The loop from line 32 to line 43 is the typical main loop of an application based on OSG and

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

Producer. And the final call at line 46 simply waits for any remaining threads to complete before going on.

2.2 A simple (and somewhat buggy) 3D viewer

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 section, in which the problem will be fixed by using a `StateSet`.

Figure 8 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.

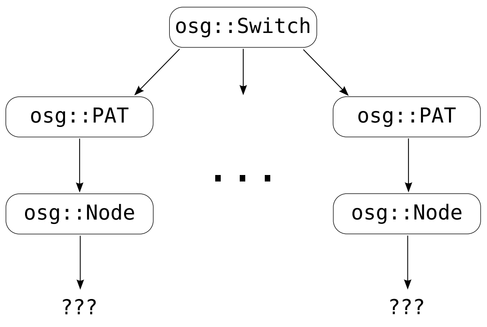

Figure 8: The OSG scene graph used in the "simple and buggy viewer". For compactness reasons, `osg::PositionAttitudeTransform` is written as `osg::PAT`.

SimpleAndBuggyViewer.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:
               {
                  osg::ref_ptr<osg::PositionAttitudeTransform> pat =
                     dynamic_cast<osg::PositionAttitudeTransform*>(
                        TheSwitch->getChild (CurrentModel));
                  pat->setScale (pat->getScale() * 1.1);

                  return true;
               }

               // Down key: decrease the current model scale
               case osgGA::GUIEventAdapter::KEY_Down:
               {
                  osg::ref_ptr<osg::PositionAttitudeTransform> pat =
                     dynamic_cast<osg::PositionAttitudeTransform*>(
                        TheSwitch->getChild (CurrentModel));
                  pat->setScale (pat->getScale() / 1.1);
                  return true;
               }

               // Don't handle other keys
               default:
                  return false;
            }
         }
         else
            return false;
      }
};


int main (int argc, char* argv[])
{
   // Check command-line parameters
   if (argc < 2)
   {
      std::cerr << "Usage: " << argv[0]
                << " <model file> [ <model file> ... ]\n";
      exit (1);
   }

   // Create a Producer-based viewer
   osgProducer::Viewer viewer;
   viewer.setUpViewer (osgProducer::Viewer::STANDARD_SETTINGS);

   // Create the event handler and attach it to the viewer
   osg::ref_ptr<osgGA::GUIEventHandler> eh (new ViewerEventHandler());
        viewer.getEventHandlerList().push_front (eh);

   // Construct the scene graph
   TheSwitch = osg::ref_ptr<osg::Switch> (new osg::Switch());

   for (int i = 1; i < argc; ++i)
   {
      osg::ref_ptr<osg::Node> loadedNode = osgDB::readNodeFile(argv[i]);
      if (!loadedNode)
         std::cerr << "Problem opening '" << argv[i] << "'\n";
      else
      {
         osg::ref_ptr<osg::PositionAttitudeTransform> pat(
            new osg::PositionAttitudeTransform());
         pat->addChild (loadedNode.get());
         TheSwitch->addChild (pat.get());
      }
   }

   // Ensure that we have at least on model before going on
   if (TheSwitch->getNumChildren() == 0)
   {
      std::cerr << "No 3D model was loaded. Aborting...\n";
      exit (1);
   }

   viewer.setSceneData (TheSwitch.get());

   TheSwitch->setSingleChildOn (0);

   // Enter rendering loop
   viewer.realize();

   while (!viewer.done())
   {
      viewer.sync();
      viewer.update();
      viewer.frame();
   }

   // Wait for all cull and draw threads to complete before exit
   viewer.sync();
}
```

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 section 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 next 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 section 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 section.

3.2 OSG and the OpenGL state

OSG provides a mechanism to manipulate OpenGL state directly in scene graph. In cull traversal geometry with same state are grouped to minimize state changes, while in draw traversal the current state is tracked to avoid redundant state changes.

The **StateSet** class store a set of state values, identified as *mode* and *attribute*. So, any node in a graph can be associate to **StateSet**. The *modes* are analogous to *glEnable* and *glDisable* calls. While, *attributes* allow to specify state parameters, such as fog color, material properties blending functions and so on.

OSGprovides a mechanism for controlling how state is inherited in scene graph. By default, all children inherit stateset parameters of your parents, but can override them, too. However, you can force parent state overrride child state node, and can protect child node from parent overriding, too.

`ON`, `OFF`, `OVERRIDE`, `PROTECTED` and `INHERIT` are parameters to `osg::StateSet::setAttribute()`.

3.3 A simple (and bugless) 3D viewer

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`.

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:
               {
                  osg::ref_ptr<osg::PositionAttitudeTransform> pat =
                     dynamic_cast<osg::PositionAttitudeTransform*>(
                        TheSwitch->getChild (CurrentModel));
                  pat->setScale (pat->getScale() * 1.1);

                  return true;
               }

               // Down key: decrease the current model scale
               case osgGA::GUIEventAdapter::KEY_Down:
               {
                  osg::ref_ptr<osg::PositionAttitudeTransform> pat =
                     dynamic_cast<osg::PositionAttitudeTransform*>(
                        TheSwitch->getChild (CurrentModel));
                  pat->setScale (pat->getScale() / 1.1);
                  return true;
               }

               // Don't handle other keys
               default:
                  return false;
            }
         }
         else
            return false;
      }
```

```
};


int main (int argc, char* argv[])
{
   // Check command-line parameters
   if (argc < 2)
   {
      std::cerr << "Usage: " << argv[0]
                << " <model file> [ <model file> ... ]\n";
      exit (1);
   }

   // Create a Producer-based viewer
   osgProducer::Viewer viewer;
   viewer.setUpViewer (osgProducer::Viewer::STANDARD_SETTINGS);

   // Create the event handler and attach it to the viewer
   osg::ref_ptr<osgGA::GUIEventHandler> eh (new ViewerEventHandler());
        viewer.getEventHandlerList().push_front (eh);

   // Construct the scene graph
   TheSwitch = osg::ref_ptr<osg::Switch> (new osg::Switch());

   for (int i = 1; i < argc; ++i)
   {
      osg::ref_ptr<osg::Node> loadedNode = osgDB::readNodeFile(argv[i]);
      if (!loadedNode)
         std::cerr << "Problem opening '" << argv[i] << "'\n";
      else
      {
         osg::ref_ptr<osg::StateSet> ss (loadedNode->getOrCreateStateSet());
         ss->setMode (GL_NORMALIZE, osg::StateAttribute::ON);
         osg::ref_ptr<osg::PositionAttitudeTransform> pat(
            new osg::PositionAttitudeTransform());
         pat->addChild (loadedNode.get());
         TheSwitch->addChild (pat.get());
      }
   }

   // Ensure that we have at least on model before going on
   if (TheSwitch->getNumChildren() == 0)
   {
      std::cerr << "No 3D model was loaded. Aborting...\n";
      exit (1);
   }
```

```
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 }
```

The only difference from the previous example are lines 103 and 104. But, this little difference provides better results, mainly when 3-D models, modeling in distinc scale, are imported to application.

4 File IO: loading and saving

To develop serious applications, like games and virtual reality complex scenes, we need fix limitations of low level APIs related to just create models using few primitives. We need to read complex level, maps and characters from files. So, that's the point! The osgDB library will help us. It provides these functionalities, through letting your program read and write 3-D models. A huge variety of formats are supported, including natives OSG(.osg - ASCII) and (.ive - binary), OpenFlight (.flt), TerraPage (.txp) with multi-threading support, LightWave (.lwo), Alias Wavefront (.obj), Carbon Graphics GEO (.geo), 3D Studio MAX (.3ds), Peformer (.pfb), Quake Character Models (.md2). Direct X (.x), and Inventor Ascii 2.0 (.iv)/ VRML 1.0 (.wrl), Designer Workshop (.dw) and AC3D (.ac). Besides OSGincludes image loaders to read and write 2D images. Supported formats include .rgb, .gif, .jpg, .png, .tiff, .pic, .bmp, .dds (include compressed mip mapped imagery), .tga and quicktime (under MacOSX). “A whole set of high quality, anti-aliased fonts can also be loaded via the freetype plugin” . But, it’s another history.

The OSG provides an extensible dynamic plugin mechansim for reading and writing 3D and 2D models from/to files. You just need add following headers to the applications:

```
#include <osgDB/ReadFile>
#include <osgDB/WriteFile>
```

Let’s go. Just reading a lovely cow:

```
  osg::ref_ptr<osg::Node> lmodel = osgDB::readNodeFile(‘‘cow.osg’’);
```

Right now, just saving a new cow:

```
  bool osgDB::writeNodeFile(lmodel,‘‘cow.osg’’);
```

If this operation fail, the function returns *false*. Else, it returns *true*. Bingo! The location where file will be save can be absolute or relative, and existing files are overwritten. For avoiding this behaviour, please check for existing files.

5 Working together: OSG and Cal3D, OSG and ODE

Many computer graphics and virtual reality applications require a simple way way for simulating vehicles, objects in virtual reality environments and virtual creatures, character animation and efficiently display entire simulation and animation in the 3-D space. Besides, models, images, collision detection, scene database management, intergration with GUI and human-interation must be supported.

We have exploited the skeleton animation and rigid body dynamics using Cal3D (`http://cal3d.sourceforge.net`) and ODE (`www.ode.org`), respectively. Here, these libraries have been employ with OSGṠo, OSG provides functionalities concerned with efficient scene display.

5.1 Skeleton animation: Cal3D

Cal3D is a free skeletal-based 3-D character animation library, written in C++, in a platform/graphic-API independent way. It is compounded by two main parts: the library and the exporter. The first one is used in the application development, while exporter take characters (built in a 3D modeling package) and create the Cal3D-format files read by the library. So, exporters are actually plug-ins for these 3D modeling packages. For example, there exists exporters for 3D Studio MAX[7], Milkshape 3D and Blender (under construction).

The basic concept in the Cal3D library is to separate data that can be shared between several objects from data that is tied to one specific object instance. In the realm of skeletal character animation there is quite a lot of shared data: Take the animations and the meshes as examples. The exporters usually exports skeleton, meshes, materials and animation from a character model. It allows cal3D library load and animate character in fashion way.

The Cal3D library does not do graphics, ok?! Programmers are responsible to put 3D characters in application, by making the relationship between Cal3D and graphics API. In case, *osgCal*[8] is an adapter for using cal3d inside OpenSceneGraph. Let's go...how can we do this?! First install Cal3D, and then install *osgCal*, too. I hope you just have installed OSG. Not yet?! hummm...do it, right now. So, let's go! give one looked at following. It shows a simple demonstration about synergy between Cal3D and OSG using *osgCal*.

osgcal.cpp

```


// headers
#include <osgGA/TrackballManipulator>
#include <osg/Drawable>
#include <osg/Timer>
```

[7]For Creating Cal3D Characters with 3D Studio MAX, see a basic tutorial at `http://cal3d.sourceforge.net/modeling/tutorial.html`.

[8]http://osgcal.sourceforge.net/

```
#include <osgProducer/Viewer>
#include <osgCal/CoreModel>
#include <osgCal/Model>

//...
// a lot of omitted code!
//...

//
// osgCal stuffs
//
// 1. create a core model, the template
osgCal::CoreModel *core = new osgCal::CoreModel("dummy");
load(core); //< loading some data into it

// 2. create a concrete model using the core template
osgCal::Model *model = new osgCal::Model();
model->create(core);

osgCal::Model *model_2 = new osgCal::Model();
model_2->create(core);

// 3. set the first animation in loop mode,
//    weight 1, and starting just now
model->startLoop(0, 1.0f, 0.0f);
model_2->startLoop(1, 1.0f, 0.0f);

model_2->setTimeScale(1.5f);

// 4. add to scene graph
osg::ref_ptr<osg::Group> escena = new osg::Group();
escena->addChild(model);
escena->addChild(model_2);

//
// Porducer/OSG roducer stuffs
//
// 5. construct the viewer.
osgProducer::Viewer viewer;

// 6. set up the value with sensible default event handlers.
//    see, viewer/model glue!
viewer.setUpViewer(osgProducer::Viewer::STANDARD_SETTINGS);
viewer.setSceneData(escena.get());
unsigned int pos = viewer.addCameraManipulator(new MyManipulator(model));
viewer.selectCameraManipulator(pos);
viewer.realize();

// 7. main loop
while ( !viewer.done() ) {
  viewer.sync();
  viewer.update();
  viewer.frame();
}
viewer.sync();
return 0;

```

The function *void load(osgCal::CoreModel *core)* loads animation, msh, material and texture from a model. *osgCal::CoreModel* is an adapter that enables reference-counting of core models. It is a reference (a template) for creating real models, which allows sharing lot of data, like the base geometry (the geometry before deforming any vertex). It inherits from *osg::Object*, then it can be easly added to scene graph (see code: item 4.).

5.2 Rigid Body Dynamics: ODE

ODE (Open Dynamics Engine) is an open source, high performance library for simulating rigid body dynamics. It is platform independent with an easy to use C/C++ API. It has advanced joint types and integrated collision detection with friction. ODE is useful for simulating vehicles, objects in virtual reality environments and virtual creatures. It is currently used in many computer games, 3D authoring tools and simulation tools. ODE has a number of base concepts. Let's start with the two fundamental ones:

Body this is a solid rigid body whose dynamics will be computed by ODE during simulation. As so, it does not have a "shape" or a visual appearance. It is just a position and orientation , a linear and angular velocity, and a mass together with an intertia tensor.

World this is what holds a bunch of bodies (dynamic and static) along with forces, and a current time. Running the physical simulation is just advancing the world's time by small steps and updating each bodies dynamics according to the existing forces and the fundamental law of dynamics.

Besides, a physical simulation is interesting if bodies interact, namely if they collide, bounce and push each other. To compute such interactions, you know need to know the shape of objects, because this is what commands when/where/how bodies interact. Thus, ODE has two other concepts: **Geom** (describe the shape the geometry of a body) and **Space** (holds a bunch of geoms and manages the collision detection). ODE comes with several flavor of space. The following code shows a simple example, about an ODE simulation, displayed using OSG. First, required headers:

```
#include <osg/Geometry>
#include <osg/Shape>
#include <osg/ShapeDrawable>
#include <osg/PositionAttitudeTransform>
#include <osg/Texture2D>
#include <osg/Timer>
#include <osgDB/ReadFile>
#include <osgProducer/Viewer>
#include <osgGA/GUIEventHandler>
// ODE!
#include <ode/ode.h>
```

For ODE simulation, only one include is required. The remain includes are from OSG. Variables declaration and definition have been omitted to make code cleaner. Now, you can to enjoy the ODE code, exhaustively commented code, in the main function. It gives us an overview about a simple ODE simulation.

odedemo.cpp

```
// 1. Create a dynamics world.
World = dWorldCreate();
dWorldSetGravity (World, 0.0, 0.0, -9.8);
dWorldSetERP (World, ERP);

// 2. Create bodies in the dynamics world.
Box = dBodyCreate (World);
```

```
for (int i = 0; i < NUM_SEGMENTS; ++i)
  Snake[i] = dBodyCreate (World);

// 3. Set the state (position etc) of all bodies.
dMass mass;

dBodySetPosition (Box, 0.0, 0.0, BOX_SIDE/2.0);
dMassSetBoxTotal (&mass, BOX_MASS, BOX_SIDE, BOX_SIDE, BOX_SIDE);
dBodySetMass (Box, &mass);

dMassSetSphereTotal (&mass, SPHERE_MASS, SNAKE_RADIUS);
for (int i = 0; i < NUM_SEGMENTS; ++i)
{
  dBodySetPosition (Snake[i], SmallRand(), SmallRand(),
                    5.0 + 2*SNAKE_RADIUS * i);
  dBodySetMass (Snake[i], &mass);
}

// 4. Create joints in the dynamics world.
for (int i = 0; i < NUM_SEGMENTS - 1; ++i)
  Joints[i] = dJointCreateUniversal (World, 0); // 0 is "joint group"

// 5. Attach the joints to the bodies.
for (int i = 0; i < NUM_SEGMENTS - 1; ++i)
  dJointAttach (Joints[i], Snake[i], Snake[i+1]);

// 6. Set the parameters of all joints.
for (int i = 0; i < NUM_SEGMENTS - 1; ++i)
{
  dJointSetUniversalAnchor (Joints[i], 0.0, 0.0, 5.0 + SNAKE_RADIUS + 2*i*SNAKE_RADIUS);
  dJointSetUniversalAxis1 (Joints[i], 0.0, 1.0, 0.0);
  dJointSetUniversalAxis2 (Joints[i], 1.0, 0.0, 0.0);
  dJointSetUniversalParam (Joints[i], dParamCFM, CFM);
}

// 7. Create a collision world and collision geometry objects, as necessary.
Space = dSimpleSpaceCreate(0);

BoxGeom = dCreateBox (Space, 2.0, 2.0, 2.0);
dGeomSetPosition (BoxGeom, 0.0, 0.0, 1.0);
dGeomSetBody (BoxGeom, Box);

for (int i = 0; i < NUM_SEGMENTS; ++i)
{
  SnakeGeom[i] = dCreateSphere (Space, SNAKE_RADIUS);
  dGeomSetPosition (SnakeGeom[i], 0.0, 0.0, 5.0 + 1.2*i);
  dGeomSetBody (SnakeGeom[i], Snake[i]);
}

GroundGeom = dCreatePlane (Space, 0.0, 0.0, 1.0, 0.0);

// 8. Create a joint group to hold the contact joints.
ContactGroup = dJointGroupCreate(0); // param unused; kept for backward compatibility

// Create a Producer-based viewer
osg::ref_ptr<ODEController> eh (osg::ref_ptr<ODEController>(new ODEController));
osgProducer::Viewer viewer;
viewer.setUpViewer (osgProducer::Viewer::STANDARD_SETTINGS);

viewer.getEventHandlerList().push_front (eh.get());

CreateOSGWorld();
viewer.setSceneData (SGRoot.get());
UpdateOSGWorld();

osg::Timer_t prevTime = osg::Timer::instance()->tick();
viewer.realize();

// 9. Loop:
while (!viewer.done())
{
  viewer.sync();
  viewer.update();
  viewer.frame();

  const double MAX_STEP = 0.01;
  const osg::Timer_t now = osg::Timer::instance()->tick();
  double deltaSecs = osg::Timer::instance()->delta_s (prevTime, now);
  prevTime = now;

  while (deltaSecs > 0.0)
    {
      // 9.1. Apply forces to the bodies as necessary.
      // (forces are valid for one time step only, so this must be done inside
      // this inner loop that keeps the time steps short)
      if (eh->isForceOn())
        dBodyAddForce (Box, 0.0, 0.0, 800.0);

      // 9.2. Adjust the joint parameters as necessary.
      // (nothing necessary here)

      // 9.3. Call collision detection.
      dSpaceCollide (Space, 0, NearCallback);

      // 9.4. Create a contact joint for every collision point, and put it in
      //      the contact joint group.
      // See 'NearCallback()', please.

      // 9.5. Take a simulation step
      const double step = std::min (MAX_STEP, deltaSecs);
      deltaSecs -= MAX_STEP;

      dWorldStep (World, step);
      // Alternatively, you may wish to try this:
      // dWorldQuickStep (World, step);

      // 9.6. Remove all joints in the contact joint group.
      dJointGroupEmpty (ContactGroup);
    }

  UpdateOSGWorld();
}

// 10. Destroy the dynamics and collision worlds.
dSpaceDestroy (Space);
dWorldDestroy (World);

viewer.sync();

```

You have observed that bodies do not interact only because of collision. They can also interact because they are "attached" together, through a *joint*. ODE defines different types of *joints*. Here, we show an example with independent objects, so only contact joints are described[9]. Why are *joints* important? Because the space manages the collision detection based on the given geoms. However, once collisions are detected, the dynamics must account for them. It turns out that collision/contact interaction can be described as temporary joints that exists only during the time of the contact. compiling and ODE_Demo.cpp, we can appreciate, a simple, but interesting demostration about ODE capabilities, display through OSG. So, let's go! give one looked at the source code.

[9]For detailed information, see ODE Manual `http://www.ode.org/ode-latest-userguide.html`.

5.3 Beyond the Straight Line

Cal3D library has been chosen to produce the virtual characters animation. However, it does not deal with the movement of the characters in the virtual world, but only with body animation. This task can be realized, for exemple, by OpenSteer[10]. It is the natural choice for steering behaviors.

An intermediate layer between Cal3D and steering has been required to syncronizeboth. This layer controls automatically the animation of one character (movement of legs and arms) due to the trajectories processed in OpenSteer library at each frame. To each animation keyframe, the displacement of the character from previous position is computed and then the body animation is updated. Then, all simulation is shown by cooperative work among OSG, Cal3D and OpenSteer. For more sophisticated simulation, this synergy can take account simulation on relief terrain. So, GDAL[11] makes easier the visualization of animated human and geometric data of city relief.

6 Past, Present and Future? Questions For God

This text is and adaptation of "Short Introduction to the Basic Principles of the Open Scene Graph" written by Leandro Motta Barros at Aug, 17, 2005. The original can be downloaded in OSG - site.

We intend to produce a Portuguese version of this document, and update current version according to the OSG evolution and toolkit updates, too. Visit us,

`http://www.stackedboxes.org`

`http://www.omni3.org/osg`

All source codes (and updates) are stored there. Go! go! You can get and use them!

About hot topics like *NodeKits*, *Visitors* and so on, you are working to add in a new version of this document. Visit our site and follow pieces of news.

[10]`http://opensteer.sourceforge.net`

[11]`http://www.gdal.org`

Please, explore this document and find errors. We hope you can help us to produce a high quality document. Contributions are welcome. Please, send all to luiz<at>omni3.org and lmb<at>stackedboxes.org.

All these libraries, mainly OSG, have a large set of examples. You can explore them to develop professional computer graphics applications and products. Besides, Cal3D and ODE have complete manuals, while Paul Martz has delivered a OSG Quick Start Guide.

A Rough equivalences between OpenGL and OSG

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.

OpenGL	**OSG**
`glTranslate()`	`osg::PositionAttitudeTransform`
`glRotate()`	`osg::PositionAttitudeTransform`
`glColor()`	`osg::Material`