Chapter No. 5
"Managing Scene Graph"
In this package, you will find:
A Biography of the authors of the book
A preview chapter from the book, Chapter NO.5 "Managing Scene Graph"
A synopsis of the book’s content
Information on where to buy this book

About the Authors

Rui Wang is a software engineer at the Chinese Academy of Surveying and Mapping and the manager of osgChina, the largest OSG discussion website in China. He is one of the most active members of the official OSG community, who contributes to the serialization I/O, GPU-based particle functionalities, BVH and animated GIF plugins, and other fixes and improvements to the OSG project. He translated Paul Martz's OpenSceneGraph Quick Start Guide into Chinese in 2008, and wrote his own Chinese book OpenSceneGraph Design and Implementation in 2009, cooperating with Xuelei Qian. He is also a novel writer and a guitar lover.

For More Information:
www.packtpub.com/openscenegraph-3-0-beginners-guide/book
Xuelei Qian received his B.Sc. degree in Precision Instrument Engineering from Southeast University, Jiangsu, China, and his Ph.D. degree in applied graphic computing from the University of Derby, Derby, UK in 1998 and 2005, respectively. Upon completion of his Ph.D. degree, he worked as a postdoctoral research fellow in the Dept. of Precision Instrument and Mechanology at Tsinghua University and his current research interests include E-manufacturing, STEP-NC and intelligent CNC, and virtual reality engineering.
Real-time rendering is in quite demand in computer science today, and OpenSceneGraph, being one of the best 3D graphics toolkits, is being used widely in the fields of virtual reality, scientific visualization, visual simulation, modeling, games, mobile applications, and so on. Although you can use the powerful OpenSceneGraph, which is based on the low-level OpenGL API, to implement applications that simulate different environments in the 3D world, developing picture-perfect applications is easier said than done.

This book has been written with the goal of helping readers become familiar with the structure and main functionalities of OpenSceneGraph, and guiding them to develop virtual-reality applications using this powerful 3D graphics engine. This book covers the essence of OpenSceneGraph, providing programmers with detailed explanations and examples of scene graph APIs.

This book helps you take full advantages of the key features and functionalities of OpenSceneGraph. You will learn almost all of the core elements required in a virtual reality application, including memory management, geometry creation, the structure of the scene graph, realistic rendering effects, scene navigation, animation, interaction with input devices and external user interfaces, file reading and writing, and so on.

With the essential knowledge contained in this book, you will be able to start using OpenSceneGraph in your own projects and research fields, and extend its functionalities by referring to OpenSceneGraph's source code, official examples, and API documentation.

This handy book divides the core functionalities of the proved and comprehensive OpenSceneGraph 3D graphics engine into different aspects, which are introduced in separate chapters. Each chapter can be treated as an individual lesson that covers one important field of OpenSceneGraph programming, along with several examples illustrating concrete usages and solutions. The sequence of the chapters is organized from the easy topics to the more difficult concepts, to help you to gradually build your knowledge and skills in with OpenSceneGraph.

By the end of the whole book, you will have gained a ready-to-use OpenSceneGraph development environment for yourself, and will have the ability to develop OpenSceneGraph-based applications and extend practical functionalities for your own purposes.

With plenty of examples to get you started quickly, you'll master developing with OpenSceneGraph in no time.

For More Information:
www.packtpub.com/openscenegraph-3-0-beginners-guide/book
What This Book Covers

Chapter 1, *The Journey into OpenSceneGraph* introduces the history, structure and features of OpenSceneGraph (OSG), and introduces the general concept of scene graph.

Chapter 2, *Compilation and Installation of OpenSceneGraph* guides readers through compiling, installing and configuring an OSG development environment, either by using the prebuilt binaries or building an environment wholly from the source code.

Chapter 3, *Creating Your First OSG Program* shows how to code an OSG-based application, highlighting the utilization of smart pointers, notifying system, object instances and data variances.

Chapter 4, *Building Geometry Models* explains how to create a geometry entity simply with vertices and the drawing primitives defined within OSG.

Chapter 5, *Managing Scene Graph* is all about the implementation of a typical scene graph using OSG, and shows the usages of the various types of scene graph nodes with special focus on some commonly-used node types.

Chapter 6, *Creating Realistic Rendering Effects* introduces some basic knowledge about OSG implementation of rendering states, texture mapping, shaders, and the render-to-texture technique.

Chapter 7, *Viewing the World* shows the means by which developers can encapsulate the cameras, manipulators, and stereo supports, and have them work together.

Chapter 8, * Animating Scene Objects* shows OSG's capability of creating animated graphic presentations by using the built-in animation library, and showcases the implementations of path animations, vertex-level animations, state and texture animations, and character animations that a 3D application can use.

Chapter 9, *Interacting with Outside Elements* focuses on the implementation of human computer interaction using OSG, including input device handling and GUI toolkit integration.

Chapter 10, *Saving and Loading Files* explains in detail the working mechanism of reading and writing scene data, and gives tips for creating user-customized I/O plugins.

Chapter 11, *Developing Visual Components* covers a wide range of advanced scene graph components, including billboards, texts, height mapped terrains, shadows, and volume rendering.

Chapter 12, *Improving Rendering Efficiency* introduces the techniques necessary for building a fast real time rendering system. It helps users to load, organize, and render massive datasets in a very efficient manner.

For More Information:  
www.packtpub.com/openscenegraph-3-0-beginners-guide/book
Managing Scene Graph

**Scene graph** is a hierarchy graph of nodes representing the spatial layout of graphic and state objects. It encapsulates the lowest-level graphics primitives and state combined to visualize anything that can be created through a low-level graphical API. OpenSceneGraph has leveraged the strength of **scene graph** and developed optimized mechanisms to manage and render 3D scenes, thus allowing the developers to use simple but powerful code in a standard way, in order to realize things such as object assembling, traversal, transform stack, culling of the scene, level-of-detail management, and other basic or advanced graphics characteristics.

In this chapter, we will cover the following topics:

- Understanding the concept of group nodes and leaf nodes
- How to handle parent and child node interfaces
- Making use of various nodes, including the transformation node, switch node, level-of-detail node, and proxy node
- How to derive your own nodes from the basic node class
- How to traverse the scene graph structure of a loaded model

For More Information:
www.packtpub.com/openscenegraph-3-0-beginners-guide/book
Managing Scene Graph

The Group interface

The `osg::Group` type represents the group nodes of an OSG scene graph. It can have any number of child nodes, including the `osg::Geode` leaf nodes and other `osg::Group` nodes. It is the most commonly-used base class of the various NodeKits—that is, nodes with various functionalities.

The `osg::Group` class derives from `osg::Node`, and thus indirectly derives from `osg::Referenced`. The `osg::Group` class contains a children list with each child node managed by the smart pointer `osg::ref_ptr<`. This ensures that there will be no memory leaks whenever deleting a set of cascading nodes in the scene graph.

The `osg::Group` class provides a set of public methods for defining interfaces for handling children. These are very similar to the drawable managing methods of `osg::Geode`, but most of the input parameters are `osg::Node` pointers.

1. The public method `addChild()` attaches a node to the end of the children list. Meanwhile, there is an `insertChild()` method for inserting nodes to `osg::Group` at a specific location, which accepts an integer index and a node pointer as parameters.
2. The public methods `removeChild()` and `removeChildren()` will remove one or more child nodes from the current `osg::Group` object. The latter uses two parameters: the zero-based index of the start element, and the number of elements to be removed.
3. The `getChlid()` returns the `osg::Node` pointer stored at a specified zero-based index.
4. The `getNumChildren()` returns the total number of children.

You will be able to handle the child interface of `osg::Group` with ease because of your previous experience of handling `osg::Geode` and drawables.

Managing parent nodes

We have already learnt that `osg::Group` is used as the group node, and `osg::Geode` as the leaf node of a scene graph. Their methods were introduced in the last chapter, and are also used in this chapter. Additionally, both classes should have an interface for managing parent nodes.

OSG allows a node to have multiple parents, as will be explained later. In this section, we will first have a glimpse of parent management methods, which are declared in the `osg::Node` class directly:

1. The method `getParent()` returns an `osg::Group` pointer as the parent node. It requires an integer parameter that indicates the index in the parent's list.

For More Information:

www.packtpub.com/openscenegraph-3-0-beginners-guide/book
2. The method `getNumParents()` returns the total number of parents. If the node has a single parent, this method will return 1, and only `getParent(0)` is available at this time.

3. The method `getParentalNodePaths()` returns all possible paths from the root node of the scene to the current node (but excluding the current node). It returns a list of `osg::NodePath` variables.

The `osg::NodePath` is actually a `std::vector` object of node pointers, for example, assuming we have a graphical scene:

![Diagram of a scene graph with nodes Root, Child1, Child2, and Child3.](image)

The following code snippet will find the only path from the scene root to the node `child3`:

```cpp
osg::NodePath& nodePath = child3->getParentalNodePaths()[0];
for ( unsigned int i=0; i<nodePath.size(); ++i )
{
    osg::Node* node = nodePath[i];
    // Do something...
}
```

You will successively receive the nodes `Root`, `Child1`, and `Child2` in the loop.

We don't need to use the memory management system to reference a node's parents. When a parent node is deleted, it will be automatically removed from its child nodes' records as well.

A node without any parents can only be considered as the root node of the scene graph. In that case, the `getNumParents()` method will return 0 and no parent node can be retrieved.
Managing Scene Graph

Time for action – adding models to the scene graph

In the past examples, we always loaded a single model, like the Cessna, by using the osgDB::readNodeFile() function. This time we will try to import and manage multiple models. Each model will be assigned to a node pointer and then added to a group node. The group node, which is defined as the scene root, is going to be used by the program to render the whole scene graph at last:

1. Include the necessary headers:
   ```cpp
   #include <osg/Group>
   #include <osgDB/ReadFile>
   #include <osgViewer/Viewer>
   ```

2. In the main function, we will first load two different models and assign them to osg::Node pointers. A loaded model is also a sub-scene graph constructed with group and leaf nodes. The osg::Node class is able to represent any kind of sub graphs, and if necessary, it can be converted to osg::Group or osg::Geode with either the C++ dynamic_cast<> operator, or convenient conversion methods like asGroup() and asGeode(), which are less time-costly than dynamic_cast<>.
   ```cpp
   osg::ref_ptr<osg::Node> model1 = osgDB::readNodeFile( "cessna.osg" );
   osg::ref_ptr<osg::Node> model2 = osgDB::readNodeFile( "cow.osg" );
   ```

3. Add the two models to an osg::Group node by using the addChild() method:
   ```cpp
   osg::ref_ptr<osg::Group> root = new osg::Group;
   root->addChild( model1.get() );
   root->addChild( model2.get() );
   ```

4. Initialize and start the viewer:
   ```cpp
   osgViewer::Viewer viewer;
   viewer.setSceneData( root.get() );
   return viewer.run();
   ```

5. Now you will see a cow getting stuck in the Cessna model! It is a little incredible to see that in reality, but in a virtual world, these two models just belong to uncorrelated child nodes managed by a group node, and then rendered separately by the scene viewer.

For More Information:
www.packtpub.com/openscenegraph-3-0-beginners-guide/book
What just happened?

Both `osg::Group` and `osg::Geode` are derived from the `osg::Node` base class. The `osg::Group` class allows the addition of any types of child nodes, including the `osg::Group` itself. However, the `osg::Geode` class contains no group or leaf nodes. It only accepts drawables for rendering purposes.

It is convenient if we can find out whether the type of a certain node is `osg::Group`, `osg::Geode`, or other derived type especially those read from files and managed by ambiguous `osg::Node` classes, such as:

```cpp
osg::ref_ptr<osg::Node> model = osgDB::readNodeFile( "cessna.osg" );
```

Both the `dynamic_cast<>` operator and the conversion methods like `asGroup()`, `asGeode()`, among others, will help to convert from one pointer or reference type to another. Firstly, we take the `dynamic_cast<>` operator as an example. This can be used to perform downcast conversions of the class inheritance hierarchy, such as:

```cpp
osg::ref_ptr<osg::Group> model =
    dynamic_cast<osg::Group*>( osgDB::readNodeFile("cessna.osg") );
```

The return value of the `osgDB::readNodeFile()` function is always `osg::Node*`, but we can also try to manage it with an `osg::Group` pointer. If, the root node of the Cessna sub graph is a group node, then the conversion will succeed, otherwise it will fail and the variable `model` will be NULL.

You may also perform an upcast conversion, which is actually an implicit conversion:

```cpp
osg::ref_ptr<osg::Group> group = ...;
osg::Node* node1 = dynamic_cast<osg::Node*>( group.get() );
osg::Node* node2 = group.get();
```

On most compilers, both `node1` and `node2` will compile and work fine.

For More Information:
www.packtpub.com/openscenegraph-3-0-beginners-guide/book
Managing Scene Graph

The conversion methods will do a similar job. Actually, it is preferable to use those methods instead of `dynamic_cast<>` if one exists for the type you need, especially in a performance-critical section of code:

```cpp
// Assumes the Cessna's root node is a group node.
osg::ref_ptr<osg::Node> model = osgDB::readNodeFile("cessna.osg");
osg::Group* convModel1 = model->asGroup();  // OK!
osg::Geode* convModel2 = model->asGeode();  // Returns NULL.
```

Pop quiz – fast dynamic casting

In C++ programs, the `dynamic_cast<>` can perform typecasts with a safety at runtime check, which requires the run-time type information (RTTI) to be enabled. It is sometimes not recommended to compare with the `osg::Node` class's converting methods, which have been overridden by subclasses like `osg::Group` and `osg::Geode`. Can you tell the reason? When would you prefer to use `asGroup()` and `asGeode()`, and when would you use the `dynamic_cast<>`?

Traversing the scene graph

A typical traversal consists of the following steps:

1. First, start at an arbitrary node (for example, the root node).
2. Move down (or sometimes up) the scene graph recursively to the child nodes, until a leaf node is reached, or a node with no children is reached.
3. Backtrack to the most recent node that doesn't finish exploring, and repeat the above steps. This can be called a depth-first search of a scene graph.

Different updating and rendering operations will be applied to all scene nodes during traversals, which makes traversing a key feature of scene graphs. There are several types of traversals, with different purposes:

1. An event traversal firstly processes mouse and keyboard inputs, and other user events, while traversing the nodes.
2. An update traversal (or application traversal) allows the user application to modify the scene graph, such as setting node and geometry properties, applying node functionalities, executing callbacks, and so on.
3. A cull traversal tests whether a node is within the viewport and worthy of being rendered. It culls invisible and unavailable nodes, and outputs the optimized scene graph to an internal rendering list.

For More Information:

www.packtpub.com/openscenegraph-3-0-beginners-guide/book
4. A **draw traversal** (or rendering traversal) issues low-level OpenGL API calls to actually render the scene. Note that it has no correlation with the **scene graph**, but only works on the rendering list generated by the **cull traversal**.

In the common sense, these traversals should be executed per frame, one after another. But for systems with multiple processors and graphics cards, OSG can process them in parallel and therefore improve the rendering efficiency.

The **visitor pattern** can be used to implement traversals. It will be discussed later in this chapter.

**Transformation nodes**

The **osg::Group** nodes do nothing except for traversing down to their children. However, OSG also supports the **osg::Transform** family of classes, which is created during the traversal-concatenated transformations to be applied to geometry. The **osg::Transform** class is derived from **osg::Group**. It can’t be instantiated directly. Instead, it provides a set of subclasses for implementing different transformation interfaces.

When traversing down the scene graph hierarchy, the **osg::Transform node** always adds its own transformation to the current transformation matrix, that is, the OpenGL model-view matrix. It is equivalent to concatenating OpenGL matrix commands such as **glMultMatrix()**, for instance:

This example **scene graph** can be translated into following OpenGL code:

```c
    glPushMatrix();
    glMultMatrix( matrixOfTransform1 );
    renderGeode1();  // Assume this will render Geode1
    glPushMatrix();
    glMultMatrix( matrixOfTransform2 );
    renderGeode2();    // Assume this will render Geode2
```

For More Information:

glPopMatrix();
glPopMatrix();

To describe the procedure using the concept of coordinate frame, we could say that Geode1 and Transform2 are under the relative reference frame of Transform1, and Geode2 is under the relative frame of Transform2. However, OSG also allows the setting of an absolute reference frame instead, which will result in the behavior equivalent to the OpenGL command glLoadMatrix():

```
transformNode->setReferenceFrame( osg::Transform::ABSOLUTE_RF );
```

And to switch back to the default coordinate frame:

```
transformNode->setReferenceFrame( osg::Transform::RELATIVE_RF );
```

### Understanding the matrix

The `osg::Matrix` is a basic OSG data type which needs not be managed by smart pointers. It supports an interface for 4x4 matrix transformations, such as translate, rotate, scale, and projection operations. It can be set explicitly:

```
osg::Matrix mat( 1.0f, 0.0f, 0.0f, 0.0f,
0.0f, 1.0f, 0.0f, 0.0f,
0.0f, 0.0f, 1.0f, 0.0f,
0.0f, 0.0f, 0.0f, 1.0f ); // Just an identity matrix
```

Other methods and operators include:

1. The public methods `postMult()` and `operator*()` post multiply the current matrix object with an input matrix or vector parameter. And the method `preMult()` performs pre-multiplications.
2. The `makeTranslate()`, `makeRotate()`, and `makeScale()` methods reset the current matrix and create a 4x4 translation, rotation, or scale matrix. Their static versions, `translate()`, `rotate()`, and `scale()`, can be used to allocate a new matrix object with specified parameters.
3. The public method `invert()` inverts the matrix. Its static version `inverse()` requires a matrix parameter and returns a new inversed `osg::Matrix` object.

You will notice that OSG uses row-major matrix to indicate transformations. It means that OSG will treat vectors as rows and pre-multiply matrices with row vectors. Thus, the way to apply a transformation matrix `mat` to a coordinate `vec` is:

```
osg::Matrix mat = ...;
osg::Vec3 vec = ...;
osg::Vec3 resultVec = vec * mat;
```

For More Information:

The order of OSG row-major matrix operations is also easy to understand when concatenating matrices, for example:

```cpp
osg::Matrix mat1 = osg::Matrix::scale(sx, sy, sz);
osg::Matrix mat2 = osg::Matrix::translate(x, y, z);
osg::Matrix resultMat = mat1 * mat2;
```

Developers can always read the transformation process from left to right, that is, the resultMat means to first scale a vector with mat1, and then translate it with mat2. This explanation always sounds clear and comfortable.

The `osg::Matrixf` class represents a 4x4 float type matrix. It can be converted by using `osg::Matrix` using overloaded `set()` methods directly.

### The MatrixTransform class

The `osg::MatrixTransform` class is derived from `osg::Transform`. It uses an `osg::Matrix` variable internally to apply 4x4 double type matrix transformations. The public methods `setMatrix()` and `getMatrix()` will assign an `osg::Matrix` parameter onto the member variable of `osg::MatrixTransform`.

### Time for action – performing translations of child nodes

Now we are going to make use of the transformation node. The `osg::MatrixTransform` node, which multiplies the current model-view matrix with a specified one directly, will help to transfer our model to different places in the viewing space.

1. Include the necessary headers:
   ```cpp
   #include <osg/MatrixTransform>
   #include <osgDB/ReadFile>
   #include <osgViewer/Viewer>
   
   2. Load the Cessna model first:
   ```cpp
   osg::ref_ptr<osg::Node> model = osgDB::readNodeFile("cessna.osg");
   ```

   3. The `osg::MatrixTransform` class is derived from `osg::Group`, so it can use the `addChild()` method to add more children. All child nodes will be affected by the `osg::MatrixTransform` node and be transformed according to the presetting matrix. Here, we will transform the loaded model twice, in order to obtain two instances displayed separately at the same time:
   ```cpp
   osg::ref_ptr<osg::MatrixTransform> transformation1 = new osg::MatrixTransform;
   ```

For More Information:

www.packtpub.com/openscenegraph-3-0-beginners-guide/book
Managing Scene Graph

transform1->setMatrix(osg::Matrix::translate(-25.0f, 0.0f, 0.0f));
transform1->addChild(model.get());

osg::ref_ptr<osg::MatrixTransform> transform2 = new osg::MatrixTransform;
transform2->setMatrix(osg::Matrix::translate(25.0f, 0.0f, 0.0f));
transform2->addChild(model.get());

4. Add the two transformation nodes to the root node and start the viewer:

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild(transformation1.get());
root->addChild(transformation2.get());

osgViewer::Viewer viewer;
viewer.setSceneData(root.get());
return viewer.run();

5. The Cessna model, which is initially placed at the axis origin, is duplicated and shown at different positions. One is transformed to the coordinate (-25.0, 0.0, 0.0), and the other to (25.0, 0.0, 0.0):

What just happened?
You may be puzzled by the scene graph structure because the model pointer is attached to two parent nodes. In a typical tree structure, a node should have at most one parent, so sharing child nodes is impossible. However, OSG supports the object sharing mechanism, that is, a child node (the model pointer), can be instantiated by different ancestors (transform1 and transform2). Then there will be multiple paths leading from the root node to the instantiated node while traversing and rendering scene graph, which causes the instanced node to be displayed more than one time.

For More Information:
www.packtpub.com/openscenegraph-3-0-beginners-guide/book
This is extremely useful for reducing the scene memory, because the application will keep only one copy of the shared data and simply call the implementation method (for instance, `drawImplementation()` of `osg::Drawable` derived classes) many times in different contexts managed by its multiple parents.

Each parent of a shared child node keeps its own `osg::ref_ptr<>` pointer to the child. In that case, the **referenced counting number** will not decrease to 0 and the child will not be released until all of its parents unreference it. You will find that the `getParent()` and `getNumParents()` methods are helpful in managing multiple parents of a node.

It is suggested that we share leaf nodes, geometries, textures, and OpenGL rendering states in one application as much as possible.

**Pop quiz – matrix multiplications**

As we have discussed, OSG uses row vectors and **row-major** matrices to perform pre-multiplications (vector * matrix) under the right-hand rule. However, OpenGL is said to use **column-major** matrices and column vectors to perform post-multiplications (matrix * vector). So, what do you think is important when converting OpenGL transformations to OSG ones?
Managing Scene Graph

Have a go hero – making use of the PositionAttitudeTransform class

The \texttt{osg::MatrixTransform} class performs like the OpenGL \texttt{glMultMatrix()} or \texttt{glLoadMatrix()} functions, which can realize almost all kinds of space transformations, but is not easy to use. The \texttt{osg::PositionAttitudeTransform} class, however, works like an integration of the OpenGL \texttt{glTranslate()}, \texttt{glScale()}, and \texttt{glRotate()} functions. It provides public methods to transform child nodes in the 3D world, including \texttt{setPosition()}, \texttt{setScale()}, and \texttt{setAttitude()}. The first two both require the \texttt{osg::Vec3} input value, and \texttt{setAttitude()} uses an \texttt{osg::Quat} variable as the parameter. The \texttt{osg::Quat} is a quaternion class, which is used to represent an orientation. Its constructor can apply a float angle and an \texttt{osg::Vec3} vector as the parameters. Euler rotations (rotating about three fixed axes) is also acceptable, using the \texttt{osg::Quat} overloaded constructor:

\begin{verbatim}
    osg::Quat quat(xAngle, osg::X_AXIS,
                   yAngle, osg::Y_AXIS,
                   zangle, osg::Z_AXIS); // Angles should be radians!
\end{verbatim}

Now, let’s rewrite the last example to replace the \texttt{osg::MatrixTransform} nodes with \texttt{osg::PositionAttitudeTransform} ones. Use \texttt{setPosition()} to specify the translation, and \texttt{setRotate()} to specify the rotation of child models, and see if it is more convenient to you in some cases.

Switch nodes

The \texttt{osg::Switch} node is able to render or skip specific children conditionally. It inherits the methods of \texttt{osg::Group} super class and attaches a Boolean value to each child node. It has a few useful public methods:

1. The overloaded \texttt{addChild()} method is able to have a Boolean parameter in addition to the \texttt{osg::Node} pointer. When the Boolean parameter is set to false, the added node will be invisible to the viewer.
2. The \texttt{setValue()} method will set the visibility value of the child node at the specified index. It has two parameters: the zero-based index and the Boolean value. And \texttt{getValue()} can get the value of child node at the input index.
3. The \texttt{setNewChildDefaultValue()} method sets the default visibility for new children. If a child is simply added without specifying a value, its value will be decided by \texttt{setNewChildDefaultValue()}, for instance:

\begin{verbatim}
    switchNode->setNewChildDefaultValue( false );
    switchNode->addChild( childNode ); // Turned off by default now!
\end{verbatim}
**Time for action – switching between the normal and damaged Cessna**

We are going to construct a scene with the `osg::Switch` node. It can even be used to implement state-switching animations and more complicated work, but at present we will only demonstrate how to predefine the visibilities of child nodes before the scene viewer starts.

1. Include the necessary headers:
   ```cpp
   #include <osg/Switch>
   #include <osgDB/ReadFile>
   #include <osgViewer/Viewer>
   ```

2. We will read two models from files and use a switch to control them. We can find a normal Cessna and a damaged one in the OSG sample data directory. They are good for simulating different states (normal/damaged) of an aircraft:
   ```cpp
   osg::ref_ptr<osg::Node> model1 = osgDB::readNodeFile("cessna.osg");
   osg::ref_ptr<osg::Node> model2 = osgDB::readNodeFile("cessnafire.osg");
   ```

3. The `osg::Switch` node is able to display one or more children and hide others. It does not work like the `osg::Group` parent class, which always displays all of its children while rendering the scene. This functionality will be quite useful if we are going to develop a flight game, and would like to manage some aircraft objects which may be destroyed at any time. The following code will set `model2` (the damaged Cessna) to visible when adding it to root, and hide `model1` (the normal one) at the same time:
   ```cpp
   osg::ref_ptr<osg::Switch> root = new osg::Switch;
   root->addChild( model1.get(), false );
   root->addChild( model2.get(), true );
   ```

4. Start the viewer:
   ```cpp
   osgViewer::Viewer viewer;
   viewer.setSceneData( root.get() );
   return viewer.run();
   ```

For More Information:
5. Now you will see an afire Cessna instead of the normal one:

![Afire Cessna](image)

**What just happened?**

The `osg::Switch` class adds a switch value list, in addition to the children list managed by its super class `osg::Group`. The two lists have the same size, and each element of one list is put into a one-to-one relationship with the element of another list. Thus, any changes in the switch value list will take effects on the related children nodes, turning their visibilities on or off.

The switch value changes that are triggered by `addChild()` or `setValue()` will be saved as properties and performed in the next rendering frame, while the OSG backend traverses the scene graph and applies different `NodeKit`'s functionalities. In the following code fragment, only the last switch values of child nodes at index 0 and 1 will be put into actual operation:

For More Information:
www.packtpub.com/openscenegraph-3-0-beginners-guide/book
switchNode->setValue( 0, false );
switchNode->setValue( 0, true );
switchNode->setValue( 1, true );
switchNode->setValue( 1, false );

Redundant calls of setValue() methods will simply be overwritten and will not affect the scene graph.

**Level-of-detail nodes**

The level-of-detail technique creates levels of detail or complexity for a given object, and provides certain hints to automatically choose the appropriate level of the object, for instance, according to the distance from the viewer. It decreases the complexity of the object's representation in the 3D world, and often has an unnoticeable quality loss on a distant object's appearance.

The osg::LOD node is derived from osg::Group and will use child nodes to represent the same object at varying levels of detail, ordered from the highest level to the lowest. Each level requires the minimum and maximum visible ranges to specify the ideal opportunity to switch with adjacent levels. The result of an osg::LOD node is a discrete amount of children as levels, which can also be named discrete LOD.

The osg::LOD class can either specify ranges along with the addition of children, or make use of the setRange() method on existing child nodes:

```
osg::ref_ptr<osg::LOD> lodNode = new osg::LOD;
lodNode->addChild( node2, 500.0f, FLT_MAX );
lodNode->addChild( node1 );
...
lodNode->setRange( 1, 0.0f, 500.0f );
```

In the previous code snippet, we first add a node, node2, which will be displayed when the distance to the eye is greater than 500 units. After that, we add a high-resolution model, node1, and reset its visible range for close observation by using setRange().
Time for action – constructing a LOD Cessna

We will create a discrete LOD node with a set of predefined objects to represent the same model. These objects are used as child nodes of the `osg::LOD` node and displayed at different distances. We will use the internal polygon reduction technique class `osgUtil::Simplifier` to generate various detailed objects from an original model. You may also read low-polygon and high-polygon models from disk files.

1. Include the necessary headers:
   ```cpp
   #include <osg/LOD>
   #include <osgDB/ReadFile>
   #include <osgUtil/Simplifier>
   #include <osgViewer/Viewer>
   ```

2. We would like to build three levels of model details. First, we need to create three copies of the original model. It is OK to read the Cessna from the file three times, but here a `clone()` method is called to duplicate the loaded model for immediate uses:
   ```cpp
   osg::ref_ptr<osg::Node> modelL3 = osgDB::readNodeFile("cessna osg");
   osg::ref_ptr<osg::Node> modelL2 = dynamic_cast<osg::Node*>(
       modelL3->clone(osg::CopyOp::DEEP_COPY_ALL));
   osg::ref_ptr<osg::Node> modelL1 = dynamic_cast<osg::Node*>(
       modelL3->clone(osg::CopyOp::DEEP_COPY_ALL));
   ```

3. We hope that level three will be the original Cessna, which has the maximum number of polygons for close-up viewing. Level two has fewer polygons to show, and level one will be the least detailed, which is displayed only at a very far distance. The `osgUtil::Simplifier` class is used here to reduce the vertices and faces. We apply the `setSampleRatio()` method to the level 1 and level 2 models with different values, which results in different simplifying rates:
   ```cpp
   osgUtil::Simplifier simplifier;
   simplifier.setSampleRatio( 0.5 );
   modelL2->accept( simplifier );
   
   simplifier.setSampleRatio( 0.1 );
   modelL1->accept( simplifier );
   ```

4. Add level models to the LOD node and set their visible range in descending order. Don’t make overlapping ranges when you are configuring minimum and maximum range values with the `addChild()` or `setRange()` method, otherwise there will be more than one level of model shown at the same position, which results in incorrect behaviors:
osg::ref_ptr<osg::LOD> root = new osg::LOD;
root->addChild( modelL1.get(), 200.0f, FLT_MAX );
root->addChild( modelL2.get(), 50.0f, 200.0f );
root->addChild( modelL3.get(), 0.0f, 50.0f );

5. Start the viewer. The application will need a little more time to compute and reduce model faces this time:

osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
return viewer.run();

6. The Cessna model comes out again. Try pressing and holding the right mouse button to zoom in and out. You will find that the model is still well-represented when looking close, as shown in the left part of the following image. However, the model is slightly simpler when viewing from far distances, as in the right two parts of the image. This difference will not affect the rendering result a lot, but will enhance the system's efficiency if properly used.

What just happened?

Have you noticed that the Cessna model should be copied twice to prepare for different level polygons? The modelL3 pointer can't be shared here, because the simplifier will directly work on the geometric data in application memory, which will affect all pointers sharing the same memory. In fact, this is called a shallow copy.

In this example, we introduce a clone() method, which can be used by all scene nodes, drawables, and objects. It is able to perform a deep copy, that is, to make copies of all dynamically-allocated memory used by the source object. The modelL2 and modelL1 pointers thus manage newly-allocated memories, which are filled with the same data as modelL3.
Managing Scene Graph

The osgUtil::Simplifier class then starts to simplify the model for decreasing the workload on the graphics pipeline. To apply the simplifier, we have to call the accept() method of a node. In the Visiting scene graph structures section, you will learn more of it and the visitor pattern.

Proxy and paging nodes

The proxy node osg::ProxyNode, and the paging node osg::PagedLOD are provided for scene load balancing. Both of them are derived from the osg::Group class directly or indirectly.

The osg::ProxyNode node will reduce the start time of the viewer if there are huge numbers of models to be loaded and displayed in the scene graph. It is able to function as the interface of external files, help applications to start up as soon as possible, and then read those waiting models by using an independent data thread. It uses setFileName() rather than addChild() to set a model file and dynamically load it as a child.

The osg::PagedLOD node also inherits methods of osg::LOD, but dynamically loads and unloads levels of detail in order to avoid overloading the graphics pipeline and keep the rendering process as smooth as possible.

Time for action – loading a model at runtime

We are going to demonstrate the loading of a model file by using the osg::ProxyNode. The proxy will record the filename of the original model, and defer loading it until the viewer is running and sending corresponding requests.

1. Include the necessary headers:
   
   ```
   #include <osg/ProxyNode>
   #include <osgViewer/Viewer>
   ```

2. Instead of just loading model files as child nodes, we will set a filename to the specified index of children. This is similar to the insertChild() method, which puts a node into the specified position of the children list, but the list will not be filled until the dynamic loading process has finished.

   ```
   osg::ref_ptr<osg::ProxyNode> root = new osg::ProxyNode;
   root->setFileName( 0, "cow.osg" );
   ```

3. Start the viewer:

   ```
   osgViewer::Viewer viewer;
   viewer.setSceneData( root.get() );
   return viewer.run();
   ```
4. The model seems to be loaded as usual, but you may have noticed that it came out a little suddenly, and the viewpoint is not adjusted to a better position. That is because the proxy node, which is invisible, is used as if it contains no child at the start of rendering. Then the cow model will be loaded from the presetting file at runtime, and automatically added and rendered as the child node of the proxy then:

What just happened?
The osg::ProxyNode and osg::PagedLOD are pretty tiny themselves; they mainly just work as containers. OSG's internal data loading manager osgDB::DatabasePager will actually do the work of sending requests and loading the scene graph when new filenames or levels of detail are available, or falling back to the next available children.

The database pager works in several background threads and drives the loading of both static database (data generated files organized by proxy and paged nodes) and dynamic database data (paged nodes generated and added at runtime).

The database pager automatically recycles paged nodes that don't appear in the current view port, and removes them from the scene graph when the rendering backend is nearly overloaded, which is when it needs to support multi-threaded paging of massive rendering data. However, this doesn't affect osg::ProxyNode nodes.

For More Information:
www.packtpub.com/openscenegraph-3-0-beginners-guide/book
Managing Scene Graph

**Have a go hero – working with the PagedLOD class**

Like the proxy node, the `osg::PagedLOD` class also has a `setFileName()` method to set the filename to load to the specified child position. However, as a LOD node, it should also set the minimum and maximum visible ranges of each dynamic loading child. Assuming that we have the `cessna.osg` file and a low-polygon version `modelL1`, we can organize a paged node like this:

```cpp
osg::ref_ptr<osg::PagedLOD> pagedLOD = new osg::PagedLOD;
pagedLOD->addChild( modelL1, 200.0f, FLT_MAX );
pagedLOD->setFileName( 1, "cessna.osg" );
pagedLOD->setRange( 1, 0.0f, 200.0f );
```

Note that the `modelL1` pointer will never be unloaded from memory, because it is a direct child and not a proxy to a file.

You will see no difference between using `osg::LOD` and `osg::PagedLOD` if displaying only one level-of-detail model. A better idea is to try using `osg::MatrixTransform` to construct a huge cluster of Cessnbas. For example, you may use an independent function to build a transformable LOD Cessna:

```cpp
osg::Node* createLODNode( const osg::Vec3& pos )
{
    osg::ref_ptr<osg::PagedLOD> pagedLOD = new osg::PagedLOD;
    ... 
    osg::ref_ptr<osg::MatrixTransform> mt = new osg::MatrixTransform;
    mt->setMatrix( osg::Matrix::translate(pos) );
    mt->addChild( pagedLOD.get() );
    return mt.release();
}
```

Set different position parameters and add multiple `createLODNode()` nodes to the scene root. See how paged nodes are rendered. Switch to use `osg::LOD` instead and have a look at the difference in performance and memory usage.

**Customizing your own NodeKits**

The most important step in customizing a node and extending new features is to override the virtual method `traverse()`. This method is called every frame by the OSG rendering backend. The `traverse()` method has an input parameter, `osg::NodeVisitor*`, which actually indicates the type of traversals (update, event, or cull). Most OSG NodeKits override `traverse()` to implement their own functionalities, along with some other exclusive attributes and methods.

For More Information:

Note that overriding the `traverse()` method is a bit dangerous sometimes, because it affects the traversing process and may lead to the incorrect rendering of results if developers are not careful enough. It is also a little awkward if you want to add the same new feature to multiple node types by extending each node type to a new customized class. In these cases, consider using node callbacks instead, which will be discussed in Chapter 8, Animate Scene Objects.

### Time for action – animating the switch node

The `osg::Switch` class can display specified child nodes while hiding others. It could be used to represent the animation states of various objects, for instance, traffic lights. However, a typical `osg::Switch` node is not able to automatically switch between children at different times. Based on this idea, we will develop a new `AnimatingSwitch` node, which will display its children at one time, and reverse the switch states according to a user-defined internal counter.

1. Include the necessary headers:
   ```cpp
   #include <osg/Switch>
   #include <osgDB/ReadFile>
   #include <osgViewer/Viewer>
   ```

2. Declare the `AnimatingSwitch` class. This will be derived from the `osg::Switch` class to take advantage of the `setValue()` method. We also make use of an OSG macro definition, `META_Node`, which is a little similar to the `META_Object` introduced in the last chapter, to define basic properties (library and class name) of a node:
   ```cpp
class AnimatingSwitch : public osg::Switch
{
public:
    AnimatingSwitch() : osg::Switch(), _count(0) {}
    AnimatingSwitch( const AnimatingSwitch& copy,
        const osg::CopyOp& copyop=osg::CopyOp::SHALLOW_COPY ) :
        osg::Switch(copy, copyop), _count(copy._count) {}
    META_Node( osg, AnimatingSwitch );

    virtual void traverse( osg::NodeVisitor& nv );

protected:
    unsigned int _count;
};
```
3. In the `traverse()` implementation, we will increase the internal counter and see if it reaches a multiple of 60, and reverse the states of the first and second child nodes:

```cpp
void AnimatingSwitch::traverse( osg::NodeVisitor& nv )
{
    if ( !((++_count)%60) )
    {
        setValue( 0, !getValue(0) );
        setValue( 1, !getValue(1) );
    }
    osg::Switch::traverse( nv );
}
```

4. Read the Cessna model and the fire model again and add them to the customized `AnimatingSwitch` instance:

```cpp
osg::ref_ptr<osg::Node> model1= osgDB::readNodeFile("cessna.osg");
osg::ref_ptr<osg::Node> model2= osgDB::readNodeFile("cessnafire.osg");
osg::ref_ptr<AnimatingSwitch> root = new AnimatingSwitch;
root->addChild( model1.get(), true );
root->addChild( model2.get(), false );
```

5. Start the viewer:

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

6. Because the hardware refresh rate is often at 60 Hz, the `if` condition in `traverse()` will become true once per second, which achieves the animation. Then you will see the Cessna is intact in the first second, and afire and smoking in the next, acting in cycles:
What just happened?

Because the \texttt{traverse()} method is widely re-implemented to extend different node types, it should involve a mechanism for reading transformation matrices and rendering states for actual use. For example, the \texttt{osg::LOD} node must calculate the distance from a child node’s center to the viewer’s eye point, which will be used as the visibility range for switching between levels.

The input parameter \texttt{osg::NodeVisitor\&} is the key to various kinds of node operations. It indicates the type of traversals visiting this node, such as the update, the event, and the cull traversal. The first two are associated with \texttt{callbacks} and will be introduced in detail in \textit{Chapter 8, Animating Scene Objects}.

The cull traversal, named \texttt{osgUtil::CullVisitor}, can be retrieved from the \texttt{osg::NodeVisitor\&} parameter with following code snippet:

\begin{verbatim}
    osgUtil::CullVisitor* cv = dynamic_cast<osgUtil::CullVisitor*>(nv);
    if ( cv )
    {
        // Do something
    }
\end{verbatim}

For More Information:

www.packtpub.com/openscenegraph-3-0-beginners-guide/book
Managing Scene Graph

You should include the `<osgUtil/CullVisitor>` header at the beginning of your program. The cull visitor class is able to obtain lots of scene states with different methods, and even change the structure and order of the internal rendering list. The concept and usage of `osgUtil::CullVisitor` is beyond the scope of this beginner's book, but is still worth understanding and learning from the source code of OSG NodeKits.

Have a go hero – creating a tracker node

Have you ever thought of implementing a tracker node, which will follow up the position of another node at all times? The trailer had better be an `osg::MatrixTransform` derived subclass. It can use a smart pointer member to record the node to be tracked and obtain the position in the 3D world in the `traverse()` override method. Then the tracker will use the `setMatrix()` method to set itself to a relative position, in order to realize the tracking operation.

You can compute a vertex in the absolute coordinate frame by using the `osg::computeLocalToWorld()` function:

```cpp
osg::Vec3 posInWorld = node->getBound().center() *
  osg::computeLocalToWorld(node->getParentalNodePaths()[0]);
```

The `getBound()` method here will return an `osg::BoundingSphere` object. The `osg::BoundingSphere` class represents the bounding sphere of a node, which is used to decide if the node is invisible and cullable in the view frustum culling process. It has two main methods: the `center()` method simply reads the center point of the bounding sphere in the local coordinate; and the `radius()` method returns the radius.

Using the `getParentalNodePaths()` method mentioned in the Managing parent nodes section, we can get the parent node path and compute the transformation matrix from the node's relative reference frame to the world reference frame.

The visitor design pattern

The visitor pattern is used to represent a user operation to be performed on elements of a graph structure, without modifying classes of these elements. The visitor class implements all of the appropriate virtual functions to be applied to various element types, and archive the goal through the mechanism of double dispatch, that is, the dispatch of certain virtual function calls, depending on the runtime types of both the receiver element and the visitor itself.

Based on the theory of double dispatch, developers can customize their visitors with special operation requests, and bind the visitor to different types of elements at runtime without changing the element interfaces. This is a great way to extend element functionalities without defining many new element subclasses.

For More Information:
www.packtpub.com/openscenegraph-3-0-beginners-guide/book
OSG supports `osg::NodeVisitor` class to implement the **visitor pattern**. In essence, an `osg::NodeVisitor` derived class traverses a **scene graph**, visits each node, and applies user-defined operations to them. It is the basic class of implementations of the update, event, and cull traversals (for example, `osgUtil::CullVisitor`), as well as some other scene graph utilities, including `osgUtil::SmoothingVisitor`, `osgUtil::Simplifier`, and `osgUtil::TriStripVisitor`, all of which will traverse the given **sub-scene graph** and apply polygon modifications to geometries found in `osg::Geode` nodes.

### Visiting scene graph structures

To create a visitor subclass, we have to re-implement one or several `apply()` virtual overloaded methods declared in the `osg::NodeVisitor` base class. These methods are designed for most major OSG node types. The visitor will automatically call the appropriate `apply()` method for each node it visits during the traversal. User customized visitor classes should override only the `apply()` methods for required node types.

In the implementation of an `apply()` method, developers have to call the `traverse()` method of `osg::NodeVisitor` at the appropriate time. It will instruct the visitor to traverse to the next node, maybe a child, or a sibling node if the current node has no children to visit. Not calling the `traverse()` method means to stop the traversal at once, and the rest of the scene graph is ignored without performing any operations.

The `apply()` methods have the unified formats of:

```
virtual void apply( osg::Node& );
virtual void apply( osg::Geode& );
virtual void apply( osg::Group& );
virtual void apply( osg::Transform& );
```

To traverse a specified node’s **sub-scene graph** and call these methods, we first need to select a traversal mode for the visitor object. Take an assumed `ExampleVisitor` class as an example; there are two steps to initialize and start this visitor on a certain node:

```cpp
ExampleVisitor visitor;
visitor->setTraversalMode( osg::NodeVisitor::TRAVERSE_ALL_CHILDREN );
node->accept( visitor );
```

The enumerate or `TRAVERSE_ALL_CHILDREN` means to traverse all of the node’s children. There are two other options: `TRAVERSE_PARENTS`, which backtracks from current node until arriving at the root node, and `TRAVERSE_ACTIVE_CHILDREN`, which only visits active child nodes, for instance, the visible children of an `osg::Switch` node.
**Managing Scene Graph**

**Time for action – analyzing the Cessna structure**

User applications may always search the loaded scene graph for nodes of interest after loading a model file. For example, we might like to take charge of the transformation or visibility of the loaded model if the root node is `osg::Transform` or `osg::Switch`. We might also be interested in collecting all transformation nodes at the joints of a skeleton, which can be used to perform character animations later.

The analysis of the loaded model structure is important in that case. We will implement an information printing visitor here, which prints the basic information of visited nodes and arranges them in a tree structure.

1. Include the necessary headers:
   ```cpp
   #include <osgDB/ReadFile>
   #include <osgViewer/Viewer>
   #include <iostream>
   ```

2. Declare the `InfoVisitor` class, and define the necessary virtual methods. We only handle leaf nodes and common `osg::Node` objects. The inline function `spaces()` is used for printing spaces before node information, to indicate its level in the tree structure:
   ```cpp
class InfoVisitor : public osg::NodeVisitor
{
public:
    InfoVisitor() : _level(0) 
        { setTraversalMode(osg::NodeVisitor::TRAVERSE_ALL_CHILDREN); }

    std::string spaces() 
        { return std::string(_level*2, ' '); }

    Virtual void apply( osg::Node& node );
    virtual void apply( osg::Geode& geode );

protected:
    unsigned int _level;
};
```

3. We will introduce two methods `className()` and `libraryName()`, both of which return `const char*` values, for instance, "Node" as the class name and "osg" as the library name. There is no trick in re-implementing these two methods for different classes. The `META_Object` and `META_Node` macro definitions will do the work internally:

---

**For More Information:**

void InfoVisitor::apply( osg::Node& node )
{
    std::cout << spaces() << node.libraryName() << "::"
    << node.className() << std::endl;

    _level++;
    traverse( node );
    _level--;
}

4. The implementation of the apply() overloaded method with the osg::Geode& parameter is slightly different from the previous one. It iterates all attached drawables of the osg::Geode node and prints their information, too. Be aware of the calling time of traverse() here, which ensures that the level of each node in the tree is correct.
void apply( osg::Geode& geode )
{
    std::cout << spaces() << geode.libraryName() << "::"
    << geode.className() << std::endl;

    _level++;    
    for ( unsigned int i=0; i<geode.getNumDrawables(); ++i )
    {
        osg::Drawable* drawable = geode.getDrawable(i);
        std::cout << spaces() << drawable->libraryName() << "::"
                    << drawable->className() << std::endl;
    }

    traverse( geode );
    _level--;
}

5. In the main function, use osgDB::readNodeFiles() to read a file from command line arguments:
osg::ArgumentParser arguments( &argc, argv );
osg::ref_ptr<osg::Node> root = osgDB::readNodeFiles( arguments );
if ( !root )
{
    OSG_FATAL << arguments.getApplicationName() <<": No data
    loaded." << std::endl;
    return -1;
}
6. Use the customized InfoVisitor to visit the loaded model now. You will have noticed that the setTraversalMode() method is called in the constructor of the visitor in order to enable the traversal of all of its children:

```cpp
InfoVisitor infoVisitor;
root->accept( infoVisitor );
```

7. Start the viewer or not. This depends on your opinion, because our visitor has already finished its mission:

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

8. Assuming that your executable file is `MyProject.exe`, in the prompt, type:

```
# MyProject.exe cessnafire.osg
```

9. You may get following information on the console:

```
osg::Group
  osg::MatrixTransform
    osg::Geode
      osg::Geometry
    osg::MatrixTransform
      osgParticle::ModularEmitter
      osgParticle::ModularEmitter
      osgParticle::ParticleSystemUpdater
    osg::Geode
      osgParticle::ParticleSystem
      osgParticle::ParticleSystem
```

**What just happened?**

You can easily draw the structure of the input fire Cessna model now. It explicitly includes an `osg::Geode` node with a geometry object, which contains the geometric data of the Cessna. The geometry node can be transformed by its parent `osg::MatrixTransform` node. The whole model is managed under an `osg::Group` node, which is returned by the `osgDB::readNodeFile()` or `osgDB::readNodeFiles()` functions.

Other classes with the prefix `osgParticle` may still seem confusing at present. They actually represent the smoke and fire particle effects of the Cessna, which will be introduced in Chapter 8, *Animating Scene Objects*.

For More Information:

Now we are able to modify the primitive sets of the model, or control the particle system, based on the results of visiting the scene graph. To archive this purpose, now let’s just save the specified node pointer to a member variable of your own visitor class, and reuse it in future code.

Summary

This chapters taught how to implement a typical scene graph by using OSG, which shows the usage of various types of scene graph nodes, with a special focus on the assembly of the graph tree and how to add state objects like the commonly used osg::Transform, osg::Switch, osg::LOD, and osg::ProxyNode classes. We specifically covered:

- How to utilize various osg::Group and osg::Geode nodes to assemble a basic hierarchy graph and handle parent and children nodes
- How to realize the spatial transform by using osg::Transform, based on the understanding of the concept of matrix and its implementation—the osg::Matrix variables
- How to use the osg::Switch node to shift the rendering status of scene nodes
- How to decide upon the detail of rendering complexity for scene nodes, by using the osg::LOD class
- Using the osg::ProxyNode and osg::PagedLOD classes to balance the runtime scene load
- How to customize a node and enhance its features
- The basic concept of the visitor design pattern and its implementation in OSG
- Traversing a node and its sub-scene graph with the osg::NodeVisitor derived classes

For More Information:
www.packtpub.com/openscenegraph-3-0-beginners-guide/book
Where to buy this book

You can buy OpenSceneGraph 3.0 from the Packt Publishing website:

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.