10
Miscellaneous Discussion in Depth

In this chapter, we will cover:

- Playing with the Delaunay triangulator
- Understanding and using the pseudo loaders
- Managing customized data with the metadata system
- Designing customized serializers
- Reflecting classes with serializers
- Taking a photo of the scene
- Designing customized intersectors
- Implementing the depth peeling method
- Using OSG in C# applications
- Using osgSwig for language binding
- Contributing your code to OSG
- Playing with osgEarth: another way to visualize the world
- Use osgEarth to display a VPB-generated database

Introduction

This is the last chapter of this book, and we are going to solve some problems which still remain in suspense. In *Chapter 3*, I promised to provide a custom intersector example, and in *Chapter 6* we were still struggling with the transparent problems on complex meshes. Here we will figure all of them out.

We will also have some other discussions about the geometry triangulator, special uses of plugins and serializers, how to capture snapshots of the scene, and topics about OSG and other languages. These functionalities may not really apply to you, but it should be interesting to learn something new and use them some day in the future.

Playing with the Delaunay triangulator

Triangulation is the process of making up a mesh with triangles from a set of points. There are many different polygon triangulation algorithms used for tessellating geometries and many other purposes.

OSG provides a specific class for implementing the **Delaunay triangulation**. It is a famous triangulation solution which is actually a subdivision of the convex hull of the points. But we are not going to discuss the algorithm used in depth, instead we will demonstrate how to quickly create geometry from a discrete set of points.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/Geometry>
#include <osg/Geode>
#include <osgDB/ReadFile>
#include <osgUtil/DelaunayTriangulator>
#include <osgViewer/Viewer>
```

2. We create a new array object with a series of independent 3D points. They didn't have any relations with each other at the beginning.

```
osg::ref_ptr<osg::Vec3Array> va =
  new osg::Vec3Array(9);
(*va)[0].set(-5.0f,-5.0f, 0.4f);
(*va)[1].set( 1.0f,-5.6f, 0.0f);
(*va)[2].set( 5.0f,-4.0f,-0.5f);
(*va)[3].set(-6.2f, 0.0f, 4.2f);
(*va)[4].set(-1.0f,-0.5f, 4.8f);
(*va)[5].set( 4.3f, 1.0f, 3.0f);
(*va)[6].set(-4.8f, 5.4f, 0.3f);
(*va)[7].set( 0.6f, 5.1f,-0.8f);
(*va)[8].set( 5.2f, 4.5f, 0.1f);
```

3. Now we allocate the `osgUtil::DelaunayTriangulator` object and set the vertex array as input. We also specify an empty array as the output normal array and the triangulator itself will fill it in the `triangulate()` method.

```
osg::ref_ptr<osgUtil::DelaunayTriangulator> dt =
  new osgUtil::DelaunayTriangulator;
dt->setInputPointArray( va.get() );
dt->setOutputNormalArray( new osg::Vec3Array );
dt->triangulate();
```

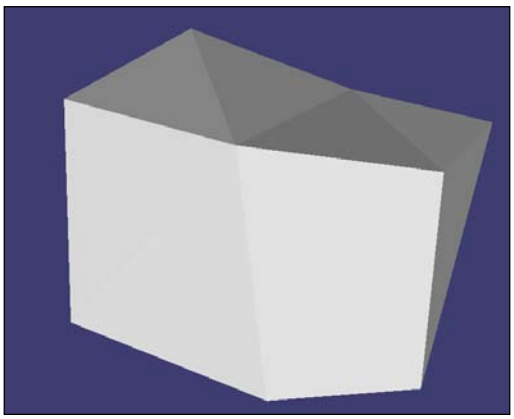
4. Add the point array, output normal, and generated triangles to a new geometry object. Note that the normal binding method should be set to the `BIND_PER_PRIMITIVE` parameter here as the output normal array is for each triangle instead of each vertex.

```
osg::ref_ptr<osg::Geometry> geometry =
  new osg::Geometry;
geometry->setVertexArray( dt->getInputPointArray() );
geometry->setNormalArray( dt->getOutputNormalArray() );
geometry->setNormalBinding(
  osg::Geometry::BIND_PER_PRIMITIVE );
geometry->addPrimitiveSet( dt->getTriangles() );
```

5. Start the viewer to see the triangulation result.

```
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( geometry.get() );

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

6. You can see a simple convex mesh in the scene. It is constructed by our input points (duplicated points will be ignored as they are not permitted) but we never set up its primitives. The triangulator does this for us automatically.



How it works...

The Delaunay triangulator accepts two kinds of input: the vertex array, and the constraints. A set of points is enough for the triangulator to construct final triangles because it can gradually find each point's location by computing the angles of previously fixed points by itself. But it will just output triangle lists and you may make triangle strips later using another utility class `osgUtil::TriStripVisitor`.

Sometimes we may have to force certain parts of the point set into the triangulation, preserve boundary edges to be split, or insertion of additional points. This is called a **constrained Delaunay triangulation** (**CDT**). We can use the `addInputConstraint()` method to add new `osgUtil::DelaunayConstraint` objects to define such constraints. You may read the `osgdelaunay` example in the OSG source code for more information.

Understanding and using the pseudo loaders

A **pseudo loader** in OSG is different from normal osgDB plugins. In most cases, its filename does not correspond to an actual file on disk, or at least it does not read from the actual disk file directly. The input filename here often represents some parameter information for the plugin to implement. Thus, the pseudo loaders may be used to quickly extend OSG functionalities using the dynamic plugin mechanism, which gives more convenience and flexibility to developers and application maintainers.

OSG has already implemented some useful pseudo loaders such as the `osgdb_trans`, `osgdb_scale`, and `osgdb_rot`, which can easily manipulate the transformation matrix of the file to be loaded. For example, we can move the cow along the X axis for 5 units before rendering it with the following line:

```
# osgviewer cow.osg.5,0,0.trans
```

Now we are going to make another plugin which will simplify the loaded model before it is rendered. The `osgUtil::Simplifier` class can do the actual work for us, and the pseudo-loader mechanism will also be included here as the simplifier's user-level interface.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osgDB/ReadFile>
#include <osgDB/FileNameUtils>
#include <osgDB/ReaderWriter>
#include <osgUtil/Simplifier>
#include <osgViewer/Viewer>
#include <sstream>
#include <iostream>
```

2. Declare the class `ReaderWriterSIMP` as if we are designing a new file reading/writing plugin. The `acceptExtension()` method should be rewritten to support a new file extension, `.simp`. We are going to add this extension to filenames later to help OSG redirect to this plugin's implementation.

```
class ReaderWriterSIMP : public osgDB::ReaderWriter
{
public:
  ReaderWriterSIMP() { supportsExtension("simp",
    "Simplification Pseudo-loader"); }

  virtual const char* className() const
  {
    return "simplification pseudo-loader";
  }

  virtual bool acceptsExtension(
    const std::string& ext ) const {
      return osgDB::equalCaseInsensitive(ext, "simp"); }

  virtual ReadResult readNode(
    const std::string& fileName,
      const osgDB::ReaderWriter::Options* options ) const;
};
```

3. The `readNode()` method is the main component which does the actual work. We first obtain the extension of the input filename and check if it can be handled here.

```
std::string ext = osgDB::getLowerCaseFileExtension(fileName);
if ( !acceptsExtension(ext) )
return ReadResult::FILE_NOT_HANDLED;
```

4. The `options` object here can also be used for determining simplification parameters (for example, the ratio). Of course, you may also record such parameters in the filename directly, just like the `osgdb_trans` plugin (as well as other inbuilt pseudo loaders).

```
double ratio = 1.0;
if ( options )
{
  std::stringstream ss( options->getOptionString() );
  ss >> ratio;
}
```

5. After the pseudo extension part is removed, the remainder of filename becomes clear. In our example, we treat it as a real file on the disk and use `readNodeFile()` again to read it from other plugins. If the file exists and can be read back as scene nodes, we simplify it using the simplifier class and the optional ratio parameter (0.5 by default).

```
osg::Node* scene = osgDB::readNodeFile(
  osgDB::getNameLessExtension(fileName) );
if ( scene )
{
  osgUtil::Simplifier simplifier(ratio);
  scene->accept( simplifier );
}
return scene;
```

6. Don't forget that we have to register a plugin to OSG with the `REGISTER_OSGPLUGIN` macro. Without the following line, we can never recognize and use the new `.simp` parser in our applications.

```
REGISTER_OSGPLUGIN( simp, ReaderWriterSIMP )
```

7. In the main entry, we will have a default ratio option (`0.2`) which will be set to the `osgDB::readNodeFile()` function as an argument. The file to be read is `cow.osg.simp`. Of course it doesn't exist on the disk, but OSG's file-loading system can still handle it, and will let the plugin decide whether the filename is valid or not.

```
osg::ArgumentParser arguments( &argc, argv );
std::string ratioStr("0.2");
if ( argc>1 ) ratioStr = arguments[1];

osg::ref_ptr<osg::Node> scene = osgDB::readNodeFile(
  "cow.osg.simp", new osgDB::ReaderWriter::Options(ratioStr) );
```

8. At last, we start the viewer to see if the cow is simplified as expected.

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

9. As you can see from the following screenshot, the model we loaded is much simpler than the original one now, but it still has a generic cow shape. The SIMP plugin helps us simplify the model at the time it is loaded, rather than searching and reading a new file itself.

How it works...

In this recipe, we place the plugin implementation and the main-scene viewer together to form one executable. The macro `REGISTER_OSGPLUGIN` is used to register the plugin instance into the `osgDB::Registry` singleton, and then the plugin can be considered and called when a new name is passed into the file reading function.

It is also possible to encapsulate the class into a separate library file, with the name `osgdb_simp`. The `REGISTER_OSGPLUGIN` macro is still needed for registering the plugin, and it will be immediately done when the library file is loaded into memory. Normally, OSG will try to load the library with the prefix `osgdb_` and the extension as name when reading new files. The plugin will be initialized and registered at that time so that it can work properly. In that case, we can even use utilities such as `osgviewer` to show the reading and parsing results of our pseudo loaders.

Managing customized data with the metadata system

The concept of metadata can be simply described as 'data about data'. It provides extra information to describe digital data and instances in an application. For example, we may have to record the author, date, tools used, external links, and some other notes of the creation and modification of a data object, and these will allow the application automatically maintain the object efficiently, or improve the user experience in some cases.

In 3D programming, metadata can also be used for different purposes. We can use it to save things such as the brief introduction of a loaded model, the additional attributes of GIS shape files, or original euler values of a quaternion (as we can hardly retrieve euler angles from quaternion directly). For this purpose, OSG provides a series of getter/setter methods to handle user values of an `osg::Object` instance. These data can be directly outputted to a `.osgt`, `.osgb`, or `.osgx` file, and read back again at any time.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/io_utils>
#include <osg/MatrixTransform>
#include <osg/ValueObject>
#include <osgDB/WriteFile>
#include <iostream>
```

2. The class derived from `osg::ValueObject::GetValueVisitor` will be used to read user values from the data container. We will just print what we can get in the virtual methods.

```
class GetValueVisitor : public
  osg::ValueObject::GetValueVisitor
{
public:
  virtual void apply( bool value )
  { std::cout << "Bool: " << value << std::endl; }

  virtual void apply( int value )
  { std::cout << "Integer: " << value << std::endl; }

  virtual void apply( const std::string& value )
  { std::cout << "String: " << value << std::endl; }

  virtual void apply( const osg::Matrix& value )
  { std::cout << "Matrix: " << value << std::endl; }
};
```

3. In the main entry, we will create a new node and add some meta information with `setUserValue()` method. These user data can be created using any common data type with a name string as the key.

```
osg::Matrix matrix;
matrix.makeRotate( osg::PI_2, osg::Z_AXIS );

osg::ref_ptr<osg::MatrixTransform> node =
  new osg::MatrixTransform;
node->setMatrix( matrix );
```

```
node->setUserValue( "Creator", std::string("Rui Wang") );
node->setUserValue( "NodeID", 101 );
node->setUserValue( "IsMain", true );
node->setUserValue( "OriginMatrix", matrix );
```

4. Write out the node, as well as the metadata we just set, into a `.osgx` file (OSG XML).

```
osgDB::writeNodeFile( *node, "result.osgx" );
```

5. Now we will read the data back using the `GetValueVisitor` class we created before. We have to get the `osg::UserDataContainer` object of the node and then visit it to retrieve different types of values. The method similar to `getOrCreateStateSet()` is used to ensure that the container object we get is always valid.

```
osg::UserDataContainer* udc =
  node->getOrCreateUserDataContainer();
for ( unsigned int i=0; i<udc->getNumUserObjects(); ++i )
{
  osg::ValueObject* valueObject =
    dynamic_cast<osg::ValueObject*>( udc->getUserObject(i) );
  if ( valueObject )
  {
    GetValueVisitor gvv;
    valueObject->get( gvv );
  }
}
return 0;
```

6. As we are ready to print the value content in `apply()` method of the `GetValueVisitor` class, after the application runs, we can see several lines of text on the terminal, which indicates the values we have set to the node. That is all about how we input and obtain metadata from any scene node object.

```
String: Rui Wang
Integer: 101
Bool: 1
Matrix: (
        2.22045e-016 1 0 0
        -1 2.22045e-016 0 0
        0 0 1 0
        0 0 0 1
)
```

How it works...

You will find in the application folder a new `.osgx` file is generated. It records the content of the scene graph we just created, as well as all the metadata. Change the extension to `.xml` and open it with any XML editor or browser (such as Chrome). You will see something similar to the following screenshot:

The hierarchy of the scene graph is shown clearly here (although there is actually only one node). The user values are stored in the `UserDataContainer` element's children, each of which has a name and a value tag. OSG can parse these metadata when the `.osgx` file is read again and put them into the node for further use. Of course, other XML editors or applications can read this file for necessary information they want, no matter if they can render the scene or not.

`.osgt` and `.osgb` formats can also be used to manage metadata, but old native formats (`.osg` and `.ive`) are not available here (in previous versions, user data could be recorded by the `setUserData()` method but couldn't be saved to files automatically).

There's more...

You may have a look at the example `osguserdata` in the core OSG source code. It provides another solution to handle user values, that is, to define a new container type for complex managements and extending metadata types.

Designing customized serializers

From the 3.0 version, OSG provides a new native format that uses the serialization IO mechanism. As C/C++ doesn't have such a direct support, OSG introduced **class wrappers** and implemented read and write functions for recording class information. All class wrappers are located at `src/osgWrappers/serializers` and the compilation process will create dynamic libraries (with the same prefix `osgdb_serializers_*`) from these wrappers.

The serialization support can be extended to read/write user classes as well. The "*OpenSceneGraph 3.0*: *Beginner's Guide*", *Rui Wang* and *Xuelei Qian*, *Packt Publishing*, introduces how to use some predefined macros to quickly construct a class wrapper. The following link includes the short introductions of all usable macros:

`http://www.openscenegraph.org/projects/osg/wiki/Support/KnowledgeBase/SerializationSupport`

In this recipe, we will try to face some more complex situations. A user node with customized enum types and internal struct data will be required to output using serializers. **Enum serializer** macros and **user serializer** functions will be used here for such purpose.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/ShapeDrawable>
#include <osgDB/ObjectWrapper>
#include <osgDB/Registry>
#include <osgDB/ReadFile>
#include <osgDB/WriteFile>
#include <iostream>
```

2. The `ComplexNode` class definition is placed in the `testNS` namespace (a named namespace is required for the serializer to work properly). It may be meaningless in practical use, but is good for demonstrating the use of serializer macros.

```
namespace testNS
{

  class ComplexNode : public osg::Node
```

```
{
public:
  enum StyleType
  { NO_STYLE, NORMAL_STYLE, ADVANCED_STYLE, USER_STYLE };

  struct ChildData
  {
    int active;
    std::string name;
    osg::ref_ptr<osg::Image> image;
  };

  ComplexNode() : osg::Node(), _type(NO_STYLE) {}

  ComplexNode(const ComplexNode& copy,
    const osg::CopyOp& copyop=osg::CopyOp::SHALLOW_COPY)
    : osg::Node(copy, copyop), _type(copy._type),
    _shape(copy._shape), _children(copy._children)
  {}

  META_Node(testNS, ComplexNode)

  void setStyleType( StyleType type ) { _type = type; }
  StyleType getStyleType() const { return _type; }

  void setShape( osg::Shape* shape )
  {
    _shape = shape;
  }
  const osg::Shape* getShape() const
  {
    return _shape.get();
  }

  void addChildData( ChildData data )
  {
    _children.push_back(data);
  }
  const ChildData& getChildData( unsigned int i ) const
  {
    return _children[i];
  }
  unsigned int sizeOfChildren() const
  {
```

```
      return _children.size();
    }

  protected:
    StyleType _type;
    osg::ref_ptr<osg::Shape> _shape;
    std::vector<ChildData> _children;
  };

}
```

The `ComplexNode` class has an enum `StyleType`, and a struct `ChildData`, and member getter/setter methods for manipulating them. It can also be attached with an `osg::Shape` object. All these data should be written to native OSG formats (`.osgt`, `.osgb`, or `.osgx`), and read back when needed.

3. Before we start to write the serialization wrapper, we have to declare a few functions used by customized serializers (for handling non-standard data types).

```
extern bool checkChildData(
  const testNS::ComplexNode& );
extern bool readChildData( osgDB::InputStream&,
  testNS::ComplexNode& );
extern bool writeChildData( osgDB::OutputStream&,
  const testNS::ComplexNode& );
```

4. Now it's time to implement the class wrapper. A lot of macros are used here to make the implementation easier to achieve and understand. We use the `REGISTER_OBJECT_WRAPPER` macro with four arguments to design the wrapper: the unique wrapper name, the prototype (always a newly allocated instance), the class name, and the inheritance relations (string, for calling parent class wrappers).

```
REGISTER_OBJECT_WRAPPER( ComplexNode_Wrapper,
  new testNS::ComplexNode, testNS::ComplexNode,
    "osg::Object osg::Node testNS::ComplexNode" )
{
  BEGIN_ENUM_SERIALIZER( StyleType, NO_STYLE );
  ADD_ENUM_VALUE( NO_STYLE );
  ADD_ENUM_VALUE( NORMAL_STYLE );
  ADD_ENUM_VALUE( ADVANCED_STYLE );
  ADD_ENUM_VALUE( USER_STYLE );
  END_ENUM_SERIALIZER();

  ADD_OBJECT_SERIALIZER( Shape, osg::Shape, NULL );
  ADD_USER_SERIALIZER( ChildData );
}
```

You should have no problem reading the main wrapper body as it is self-explained. Enum items are added in the `BEGIN_ENUM_SERIALIZER()` and `END_ENUM_SERIALIZER()` pair. And `NO_STYLE` enumeration value is set as default at the beginning. Scene objects (derived from `osg::Object`) are directly recorded with `ADD_OBJECT_SERIALIZER()` macro. The `ChildData` vectors, however, must be handled with user serializer functions because they are unrecognizable to OSG itself.

5. The `checkChildData()` function is used for checking if the child data should be written to files or not.

```
bool checkChildData( const testNS::ComplexNode& node )
{
  return node.sizeOfChildren()>0;
}
```

6. The `writeChildData()` function is used for outputting child data to files. The `osg::OutputStream` class works like the `std::ostream`, but it supports OSG scene objects (the image object here) too. Note that we first check and write a Boolean value to the stream, and then use `BEGIN_BRACKET` and `END_BRACKET` to add brackets before and after writing the image. This style, which can be seen in section *How it works*, is always recommended for saving OSG objects in customized functions.

```
bool writeChildData( osgDB::OutputStream& os,
  const testNS::ComplexNode& node )
{
  // Get the children list size
  unsigned int size = node.sizeOfChildren();
  os << size << osgDB::BEGIN_BRACKET << std::endl;
  for ( unsigned int i=0; i<size; ++i )
  {
    const testNS::ComplexNode::ChildData& data =
      node.getChildData(i);
    os << std::string("Child") << data.active << ofdata.name;
    if ( data.image.valid() )
    {
      os << true << osgDB::BEGIN_BRACKET << std::endl;
      os << data.image << osgDB::END_BRACKET << std::endl;
    }
    else os << false << std::endl;
  }
  os << osgDB::END_BRACKET << std::endl;
  return true;
}
```

7. The `readChildData()` function is used for re-reading the data list from external files. The `osg::InputStream` class works nearly in the same sequence of the output stream, except that it ignores end flags and uses `>>` instead of `<<`. Every result read is pushed into the node's `ChildData` list with the `addChildData()` method.

```
bool readChildData( osgDB::InputStream& is,
  testNS::ComplexNode& node )
{
  unsigned int size = 0; is >> size >> osgDB::BEGIN_BRACKET;
  for ( unsigned int i=0; i<size; ++i )
  {
    testNS::ComplexNode::ChildData data;
    std::string childFlag("Child");
    is >> childFlag >> data.active >> data.name;

    bool hasImage = false;
    is >> hasImage >> osgDB::BEGIN_BRACKET;
    if ( hasImage ) is >> data.image >> osgDB::END_BRACKET;
    node.addChildData( data );
  }
  is >> osgDB::END_BRACKET;
  return true;
}
```

8. Now let's test our class and wrappers. In the main entry (or you can generate a dynamic library with past code), we randomly add some data and set properties to the `ComplexNode` instance.

```
testNS::ComplexNode::ChildData data1;
data1.active = 10;
data1.name = "data1";
data1.image = osgDB::readImageFile("Images/smoke.rgb");

testNS::ComplexNode::ChildData data2;
data2.active = 20;
data2.name = "data2";

osg::ref_ptr<testNS::ComplexNode> node =
  new testNS::ComplexNode;
node->setStyleType( testNS::ComplexNode::ADVANCED_STYLE );
node->setShape( new osg::Box() );
node->addChildData( data1 );
node->addChildData( data2 );
```

9. Write the node content to a new `.osgt` file. Of course, you may change the extension to `.osgb` or `.osgx`, but not others.

```
osgDB::writeNodeFile( *node, "ComplexNode.osgt" );
return 0;
```

10. Open the ASCII file with any text editor and you will see the node is written nicely in the file, and can be read back with the `osgDB::readNodeFile()` function at any time.

```
testNS::ComplexNode
{
  UniqueID 1
  StyleType ADVANCED_STYLE
  Shape TRUE
  {
    osg::Box
    {
      UniqueID 2
      HalfLengths 0.5 0.5 0.5
    }
  }
  ChildData 2
  {
    Child 10 data1 TRUE
    {
      UniqueID 3
      FileName "Images/smoke.rgb"
      WriteHint 0 2
      DataVariance STATIC
    }
  Child 20 data2 FALSE
  }
}
```

How it works...

We must be very careful about the code style of serializable classes. The `ADD_*_SERIALZIER` macro with the name `SomeValue` will automatically call corresponding methods of the original class to set or obtain values. So you have to always write your setter/getters as shown in the following code block:

```
// For basic data types (bool, integer, etc.)
void setSomeValue( T v ) {...}
T getSomeValue() const {...}

// For pointers, objects
void setSomeValue( T* v ) {...}
const T* getSomeValue() const {...}
```

The only exception is `ADD_USER_SERIALIZER()` macro. It requires the following three functions with specific names:

```
bool checkSomeValue(const Class& );
bool writeSomeValue( osgDB::OutputStream& os, const Class& );
bool readSomeValue( osgDB::OutputStream& os, Class& );
```

Be aware that improper implementations in user functions may cause the reading/writing process to fail. For example, when saving OSG objects, you should always write a Boolean value to indicate if the object exists or not, as shown in the following block of code:

```
if ( data.image.valid() )
{
  os << true << osgDB::BEGIN_BRACKET << std::endl;
  os << data.image << osgDB::END_BRACKET << std::endl;
}
else os << false << std::endl;
```

The read function will, thus, know whether there is image data saved and decide to continue reading it or skip. This is extremely important when you save scene in a binary format (`.osgb`).

Reflecting classes with serializers

Reflection is initially a feature in the Java language. It allows a running program to examine itself and obtain and manipulate internal classes, member methods, and properties without recompiling or rebuilding. For example, we can easily obtain a Java class instance by its name:

```
Class c = Class.forName("MyClass");
```

But C++ doesn't have these abilities, and neither do OSG and other C++ based engines in the past. But thanks to the serialization mechanism, now we may go beyond the language obstacle and implement part of the reflection feature by ourselves. In this recipe, we are going to create such a reflection system and make it work properly.

Getting ready

We will have to make the serialization IO system work outside the osgDB framework. It is not an easy task, and so we must borrow some code from the core OSG source code. Jump to the folder `src/osgPlugins/osg` in your OSG source code, and copy the file `BinaryStreamOperator.h` to your project directory. We will include and use it later in our own program.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osgDB/OutputStream>
#include <osgDB/InputStream>
#include <osgDB/Registry>
#include <osgDB/ReadFile>
#include <osgViewer/Viewer>
#include <iostream>
#include <sstream>
#include "BinaryStreamOperator.h"
```

2. Declare three important classes: `ClassInfo`, `ClassInstance`, and `Method`. They describe the class declaration (name, constructors, and methods from the associated wrapper), class instance (a new object of current class type), and the class method (the methods that could be used by the class instance) which are necessary for reflecting purposes.

```
class ClassInfo;
class ClassInstance;
class Method;
```

3. `ClassInfo` is the container of an OSG class, including its constructor (for creating new instance) and methods. It must be attached with the class wrapper `osgDB::ObjectWrapper` to obtain data from serializers.

```
class ClassInfo : public osg::Referenced
{
public:
  ClassInfo( osgDB::ObjectWrapper* w=0 ) : _wrapper(w) {}

  ClassInstance* createInstance();
  Method* getMethod( const std::string& );

protected:
  virtual ~ClassInfo() {}
  osgDB::ObjectWrapper* _wrapper;
};
```

4. The `ClassInstance` will always keep the newly allocated object internally without exposing it to high-level developers (and they don't need to face the real object either).

```
class ClassInstance : public osg::Referenced
{
public:
  ClassInstance( osg::Object* obj ) : _object(obj) {}

  void setObject( osg::Object* obj ) { _object = obj; }
  osg::Object* getObject()
  {
    return _object.get();
  }
  const osg::Object* getObject() const
  {
    return _object.get();
  }

protected:
  virtual ~ClassInstance() {}
  osg::ref_ptr<osg::Object> _object;
};
```

5. The `Method` class is used to store and execute an object's getter/setter methods, and is used as a bridge between the value and the real interfaces. It can also set/get a `ClassInstance` variable, that is, set/get the pointer of a scene object actually. Every `Method` object is attached with an `osg::BaseSerializer` object, which internally holds the setter/getter methods encapsulated by the `ADD_*_SERIALIZER()` macros.

```
class Method : public osg::Referenced
{
public:
  Method( osgDB::BaseSerializer* s=0 ) : _serializer(s) {}

  template<typename T> bool set( ClassInstance*, T );
  template<typename T> bool get( ClassInstance*, T& );
  bool set( ClassInstance*, const ClassInstance& );
  bool get( ClassInstance*, ClassInstance& );

protected:
  virtual ~Method() {}
  osgDB::BaseSerializer* _serializer;
};
```

6. The `ReflectionManager` class, which is designed as a singleton, will be used for managing all class information and methods. It uses the `osgDB::OutputStream` and `osgDB::InputStream` classes to save or load data through serializers, which in fact calls the original class' functionalities indirectly.

```
class ReflectionManager : public osg::Referenced
{
public:
  static ReflectionManager* instance()
  {
    static osg::ref_ptr<ReflectionManager> s_manager =
      new ReflectionManager;
    return s_manager.get();
  }

  osgDB::OutputStream& getOutputStream()
  {
    return *_outputStream;
  }
  osgDB::InputStream& getInputStream()
  {
    return *_inputStream;
  }
  std::stringstream& getSource()
  {
    return _source;
  }

  ClassInfo* getClassInfo( const std::string& name );
  ClassInfo* getClassInfo( osgDB::ObjectWrapper* wrapper );
  Method* getMethod( osgDB::BaseSerializer* serializer );

protected:
  ReflectionManager();
  virtual ~ReflectionManager();

  typedef std::map<osgDB::ObjectWrapper*,
    osg::ref_ptr<ClassInfo> > ClassInfoMap;
  ClassInfoMap _classInfoMap;

  typedef std::map<osgDB::BaseSerializer*,
    osg::ref_ptr<Method> > MethodMap;
  MethodMap _methodMap;

  osgDB::OutputStream* _outputStream;
  osgDB::InputStream* _inputStream;
  std::stringstream _source;
```

7. Now let's implement the methods of the these classes. First, in the constructor of `ReflectionManager`, we allocate new stream objects and set their iterators (using the class in the `BinaryStreamOperator.h` header file). The iterators here will read or write to the same `_source` variable when new I/O requests are parsed by the streams. At last, don't forget to delete allocated resources in the class destructor.

```
ReflectionManager::ReflectionManager()
{
  _outputStream = new osgDB::OutputStream(NULL);
  _inputStream = new osgDB::InputStream(NULL);
  _outputStream->setOutputIterator(
    new BinaryOutputIterator(&_source) );
  _inputStream->setInputIterator(
    new BinaryInputIterator(&_source) );
}

ReflectionManager::~ReflectionManager()
{
  delete _outputStream; delete _inputStream;
}
```

8. The method `getClassInfo()` can find an existing class by its name string. It simply calls the internal `osgDB::ObjectWrapperManager` and returns a `ClassInfo` object attached with the found wrapper.

```
ClassInfo* ReflectionManager::getClassInfo(
  const std::string& name )
{
  osgDB::ObjectWrapperManager* wrapperManager =
    osgDB::Registry::instance()->getObjectWrapperManager();
  if ( wrapperManager )
  {
    osgDB::ObjectWrapper* wrapper =
      wrapperManager->findWrapper(name);
    if ( wrapper ) return getClassInfo(wrapper);
  }
  return NULL;
}
```

9. The overloaded method creates a new `ClassInfo` object with the wrapper, or returns a previous one instead if the wrapper has already been called before.

```
ClassInfo* ReflectionManager::getClassInfo(
  osgDB::ObjectWrapper* wrapper )
{
  ClassInfoMap::iterator itr =
    _classInfoMap.find(wrapper);
```

```
  if ( itr!=_classInfoMap.end() ) return itr->
    second.get();

  ClassInfo* info = new ClassInfo( wrapper );
  _classInfoMap[wrapper] = info;
  return info;
}
```

10. The `getMethod()` method can find or create new `Method` object attached with a certain osgDB serializer. It is used internally by the `ClassInfo` class.

```
Method* ReflectionManager::getMethod(
  osgDB::BaseSerializer* serializer )
{
  MethodMap::iterator itr =
    _methodMap.find(serializer);
  if ( itr!=_methodMap.end() ) return itr->
    second.get();

  Method* method = new Method( serializer );
  _methodMap[serializer] = method;
  return method;
}
```

11. The `createInstance()` method of `ClassInfo` will allocate a new class instance. It replaces the `new` operator in standard C++ programs.

```
ClassInstance* ClassInfo::createInstance()
{
  if ( _wrapper )
  {
    const osg::Object* obj = _wrapper->getProto();
    if ( obj ) return new ClassInstance( obj->cloneType() );
  }
  return NULL;
}
```

12. The `getMethod()` method will try to find the setter/getter method name of the real class. It searches the `ClassInfo`'s serializer list and returns the `Method` object attached.

```
Method* ClassInfo::getMethod( const std::string& name )
{
  if ( _wrapper )
  {
    osgDB::BaseSerializer* serializer =
      _wrapper->getSerializer(name);
```

```
    if ( serializer ) return ReflectionManager::
      instance()->getMethod(serializer);
  }
  return NULL;
}
```

13. The `Method` class implements `set()` and `get()` methods for high-level developers to operate on hidden, native OSG classes. The basic concept of the `set()` method is to put the `value` into the output stream first, and then use serializer to read the stream and set to actual OSG object automatically.

```
template<typename T>
bool Method::set( ClassInstance* clsObject, T value )
{
  bool ok = false;
  ReflectionManager::instance()->getOutputStream() << value;
  if ( _serializer )
  {
    ok = _serializer->read( ReflectionManager::
      instance()->getInputStream(), *(clsObject->
        getObject()) );
  }
  ReflectionManager::instance()->getSource().clear();
  return ok;
}
```

14. The getter does the opposite. It first writes the serializer value to the output stream and then uses the input one to obtain the value.

```
template<typename T>
bool Method::get( ClassInstance* clsObject, T& value )
{
  bool ok = false;
  if ( _serializer )
  {
    ok = _serializer->write( ReflectionManager::
      instance()->getOutputStream(), *(clsObject->
        getObject()) );
  }
  if ( ok ) ReflectionManager::instance()->
    getInputStream() >> value;
  ReflectionManager::instance()->getSource().clear();
  return ok;
}
```

15. We also have `set()` and `get()` methods for `ClassInstance` variable. It actually means to handle the OSG object instead of common data types.

```
bool Method::set( ClassInstance* clsObject,
  const ClassInstance& instance )
{
  // Please find details in the source code
}

bool Method::get( ClassInstance* clsObject,
  ClassInstance& instance )
{
  // Please find details in the source code
}
```

16. OK, so we finally finished the reflection system. Let's do something in the main entry now. We are going to create an `osg::Box` object first with the `getClassInfo()` method and set its half length with the `Method` object. The `osg::Box`'s `setHalfLengths()` is actually called inside these superficial interfaces.

```
ClassInfo* boxInfo = ReflectionManager::
  instance()->getClassInfo("osg::Box");
Method* boxMethod =
  boxInfo->getMethod("HalfLengths");
ClassInstance* box =
  boxInfo->createInstance();
boxMethod->set( box, osg::Vec3(5.0f, 5.0f, 2.0f) );
```

17. Then we set the shape to a new `osg::ShapeDrawable` object using its `setShape()` method. Of course, it is another `Method` object in our reflection framework.

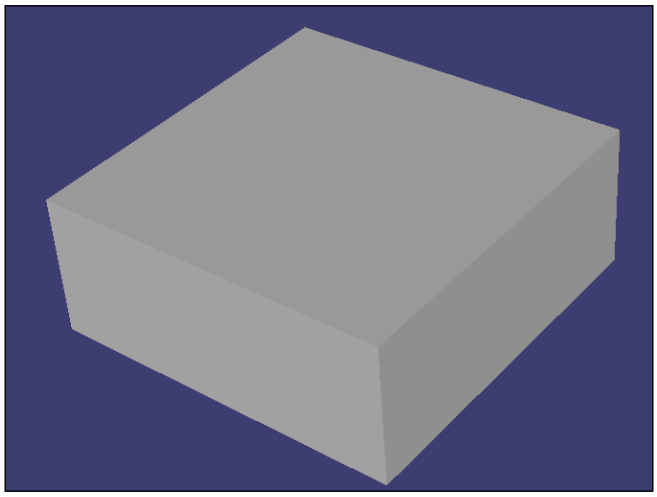
```
ClassInfo* drawableInfo = ReflectionManager::
  instance()->getClassInfo("osg::ShapeDrawable");
Method* drawableMethod =
  drawableInfo->getMethod("Shape");
ClassInstance* drawable =
  drawableInfo->createInstance();
drawableMethod->set( drawable, *box );
```

18. A limitation of our example framework is that we can't absolutely depend on the reflection framework at present, because we can't get and execute `addDrawable()` method from related serializers. It is unfortunate here but we will still go on and render the scene.

```
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
geode->addDrawable( dynamic_cast<osg::Drawable*>(
  drawable->getObject()) );
```

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

The result is still good. We never included headers for `osg::Box` and `osg::ShapeDrawable`, and never actually allocated these two classes' instances or call their methods. But we could still finish our goal by reflecting them for accessing and manipulating their constructors and methods.



How it works...

We built a common reflection framework here with four main components. The `ReflectionManager` singleton is used for obtaining a new class' information from the name string. The `ClassInfo` class, which stores the information, can then be used to create instances of the class (`ClassInstance`), or get class method by name and return a `Method` container.

All methods must be defined as OSG serializers in the class wrapper so that we can manipulate them. In this recipe, we used some tricks to implement the setter/getter interfaces, that is, the `osgDB::OutputStream` and `osgDB::InputStream`. These two objects are designed for file I/O only, but here we used them to pass method arguments to the real class methods and get them back when necessary.

The code doesn't work with all kinds of OSG classes because it can't handle user serializers, which always have unpredictable operations to the output/input streams. You may have to consider some other solutions if you really want your work to completely reflect the entire OSG project.

There's more...

Reflections are useful for integrating the library with other programming languages, but, unfortunately, the serialization functionalities are still too weak to be extended for such purposes. In this chapter, I also provided two examples about this topic that integrates OSG with C# and Python separately. The Python example makes use of osgSWIG, another good project reflecting OSG classes and methods to different environments. You can have a look at it later.

Taking a photo of the scene

How would you take a photo of your 3D scene? Some of you would want to use their *Print Screen* key to capture the whole screen or a window, and open the painter program, paste, and save to disk file. Others may consider finishing the capturing work in a post draw callback with the `glReadPixels()` function, which in OpenGL can return pixels from the frame buffer. But how can we really work out such a scene capturing utility with only a few lines of code? That is what we are going to see in this recipe.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/ValueObject>
#include <osg/Camera>
#include <osgDB/ReadFile>
#include <osgDB/WriteFile>
#include <osgViewer/Viewer>
#include <iostream>
#include <sstream>
```

2. We should execute the OpenGL pixel reading function after the graphics context is set as the current one, so an `osg::Camera::DrawCallback` is a good choice. It will maintain an image object, a text for displaying on the screen, and a `_fileIndex` for generating the filename.

```
class PhotoCallback : public osg::Camera::DrawCallback
{
public:
  PhotoCallback( osg::Image* img, osgText::Text* text )
  : _image(img), _text(text), _fileIndex(0) {}

  virtual void operator()( osg::RenderInfo&
    renderInfo ) const;
```

```
protected:
  osg::ref_ptr<osg::Image> _image;
  osg::observer_ptr<osgText::Text> _text;
  mutable int _fileIndex;
};
```

3. Implement the `operator()` method. Here we will also make use of the metadata (bool value `Capture`) we just learned in previous recipes.

```
bool capturing = false;
if ( _image.valid() && _image->getUserValue(
  "Capture", capturing) )
{
  osg::GraphicsContext* gc =
    renderInfo.getState()->getGraphicsContext();
  if ( capturing && gc->getTraits() )
  {
    int width = gc->getTraits()->width;
    int height = gc->getTraits()->height;
    GLenum pixelFormat = (gc->getTraits()->alpha ?
      GL_RGBA : GL_RGB);
    _image->readPixels( 0, 0, width, height,
      pixelFormat, GL_UNSIGNED_BYTE );

    std::stringstream ss; ss << "Image_" <<
      (++_fileIndex) << ".bmp";
    if ( osgDB::writeImageFile(*_image, ss.str())
      && _text.valid() )
    _text->setText( std::string("Saved to ") + ss.str() );
  }
  _image->setUserValue( "Capture", false );
}
```

4. We need a `PhotoHandler` class to receive user input and capture the picture once. When the user presses the *P* key, the image object will set its metadata `Capture` to `true`, and the drawing callback will do the actual work.

```
class PhotoHandler : public osgGA::GUIEventHandler
{
public:
  PhotoHandler( osg::Image* img ) : _image(img) {}

  virtual bool handle( const osgGA::GUIEventAdapter& ea,
    osgGA::GUIActionAdapter& aa )
  {
    if ( _image.valid() && ea.getEventType()==
```

```
      osgGA::GUIEventAdapter::KEYUP )
    {
      if ( ea.getKey()=='p' || ea.getKey()=='P' )
      _image->setUserValue( "Capture", true );
    }
    return false;
  }

protected:
  osg::ref_ptr<osg::Image> _image;
};
```

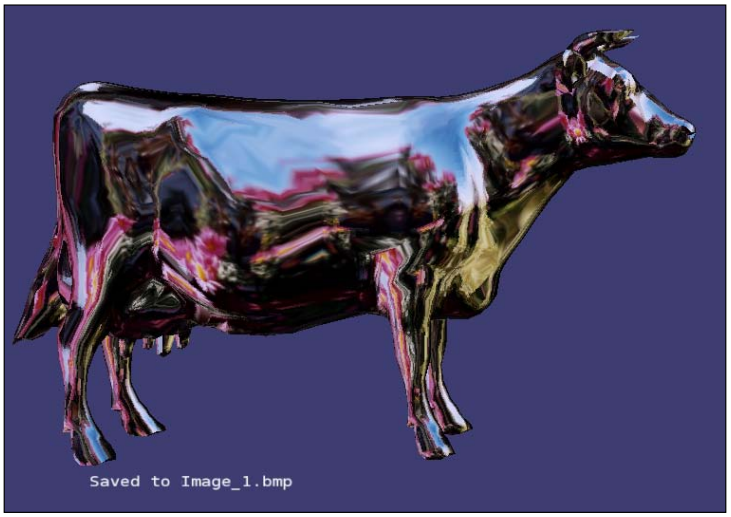
5. In the main entry, we construct a simple scene with a cow model and an HUD camera with a text (which will tell if the file is already saved).

```
osg::ArgumentParser arguments( &argc, argv );
osg::ref_ptr<osg::Node> scene =
  osgDB::readNodeFiles( arguments );
if ( !scene ) scene = osgDB::readNodeFile("cow.osg");

osgText::Text* text = osgCookBook::createText(
  osg::Vec3(50.0f, 50.0f, 0.0f), "", 10.0f);
osg::ref_ptr<osg::Geode> textGeode = new osg::Geode;
textGeode->addDrawable( text );

osg::ref_ptr<osg::Camera> hudCamera =
  osgCookBook::createHUDCamera(0, 800, 0, 600);
hudCamera->addChild( textGeode.get() );

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( hudCamera.get() );
root->addChild( scene.get() );
```

6. Allocate a new image object and set it to both the callback and the handler. Now start the viewer.

```
osg::ref_ptr<osg::Image> image = new osg::Image;
osg::ref_ptr<PhotoCallback> pcb =
  new PhotoCallback( image.get(), text );
osg::ref_ptr<PhotoHandler> ph =
  new PhotoHandler( image.get() );

osgViewer::Viewer viewer;
viewer.getCamera()->setPostDrawCallback( pcb.get() );
viewer.addEventHandler( ph.get() );
viewer.setSceneData( root.get() );
return viewer.run();
```

7. Press *P* and you will see a line on the screen showing the latest saved filename; meanwhile, you can find the BMP snapshot of the screen in your working directory. You may take more than one photo and the file index will automatically increase.



How it works...

During the capturing process, we will first obtain the Boolean user value `Capture` from the image, which indicates if there is a request for capturing picture of current scene. It will obtain the window size and use the `readPixels()` function (which encapsulates `glReadPixels()`) to take the photo. The filename is generated according to the index number, and the user value should be set to `false` at last. Note that `readPixels()` will automatically allocate memory space for the image object.

Another interesting appearance of this recipe is that the text on the screen is never included in the saved image, no matter how many times we press *P* to save the file. The reason is the HUD camera we used. An HUD camera is a 'post-drawing' camera that will work after the parent camera's children are rendered, and after the post-draw callback. The occasions of calling camera callbacks and child cameras can be described as follows:

- To execute the initial callback `getInitialDrawCallback()`
- To traverse and render child cameras marked as 'pre-render'
- To execute the pre-drawing callback `getPreDrawCallback()`
- To traverse and render child nodes (except cameras)
- To execute the post-drawing callback `getPostDrawCallback()`
- To traverse and render child cameras marked as 'post-render'
- To execute the final callback `getFinalDrawCallback()`

Now, let us change the line which sets the camera callback in main entry:

```
viewer.getCamera()->setFinalDrawCallback( pcb.get() );
```

Rebuild the project. You will find that the displayed text appears on the snapshot too.

There's more...

There are different methods to capture frame buffer data besides `readPixels()`. For instance, the `osgposter` example in the OSG source code uses multiple cameras with `FRAME_BUFFER_OBJECT` parameter to capture the screen into a huge poster-like image (very high resolution).

The `osgautocapture` example shows how to capture images in offline mode (no window displayed). There is also an `osgViewer::ScreenCaptureHandler` class which uses **Pixel Buffer Object** for capturing snapshots. You may compare them and choose one for practical use as you wish.

Designing customized intersectors

Do you remember that in *Chapter 3* we had created an example to pick and select a point on the geometry model? In that example, we encountered a small but annoying problem—the point selection solution is based on the line segment intersection result, so the mouse cursor must first intersect with the model, and then we are able to check if the distance of the intersection point and any vertex is under the threshold.

A considerable substitute is a customized point-picking intersector. As we already known, the line segment intersection is done by creating and using the `osgUtil::LineSegmentIntersector` class. So why not derive this class and make it support point intersection as well? We will work on such a topic in the following sections of this recipe.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/Geometry>
#include <osg/Geode>
#include <osg/MatrixTransform>
#include <osg/Point>
#include <osg/PolygonOffset>
#include <osgUtil/SmoothingVisitor>
#include <osgUtil/LineSegmentIntersector>
#include <osgViewer/Viewer>
```

2. We will derive the `osgUtil::LineSegmentIntersector` class to design our `PointIntersector` class. It needs a bias value which is used to check if the distance between the intersection line and model point is acceptable.

```
class PointIntersector :public osgUtil::LineSegmentIntersector
{
public:
  PointIntersector();
  PointIntersector( const osg::Vec3& start,
    const osg::Vec3& end );
  PointIntersector( CoordinateFrame cf, double x,
    double y );

  void setPickBias( float bias ) { _pickBias = bias; }
  float getPickBias() const { return _pickBias; }

  virtual Intersector* clone(
    osgUtil::IntersectionVisitor& iv );
  virtual void intersect( osgUtil::IntersectionVisitor& iv,
    osg::Drawable* drawable );

protected:
  virtual ~PointIntersector() {}
  float _pickBias;
};
```

3. The three constructors will simply pass arguments to the line segment intersector (the parent class).

```
PointIntersector::PointIntersector()
: osgUtil::LineSegmentIntersector(MODEL, 0.0, 0.0),
  _pickBias(2.0f)
{}

PointIntersector::PointIntersector( const osg::Vec3& start,
  const osg::Vec3& end )
: osgUtil::LineSegmentIntersector(start, end),
  _pickBias(2.0f)
{}

PointIntersector::PointIntersector( CoordinateFrame cf,
  double x, double y )
: osgUtil::LineSegmentIntersector(cf, x, y),
  _pickBias(2.0f)
{}
```

4. Now we will work on the `clone()` method, which is called by the intersection visitor to create local space intersectors.

```
osgUtil::Intersector* PointIntersector::clone(
  osgUtil::IntersectionVisitor& iv )
{
  ...
}
```

5. The first method to overwrite is `clone()`. As the constructor may specify a different coordinate frame, we must do a conversion here from specified space (window, projection, view, or model world) to the object's local space.

```
osg::Matrix matrix;
switch ( _coordinateFrame )
// same as previous variable 'cf'
{
  // Please find details in the source code
}
```

6. The second step is to clone a new intersector with the start and end points in the local space and return it. We will explain how it is used in the next section.

```
osg::Matrix inverse = osg::Matrix::inverse(matrix);
osg::ref_ptr<PointIntersector> cloned =
  new PointIntersector( _start*inverse, _end*inverse );
cloned->_parent = this;
cloned->_pickBias = _pickBias;
return cloned.release();
```

7. Now we will work on another important method `intersect()`. It will be called when a drawable is traversed and we want to know if it has intersections with the line segment.

8. We first check if the bounding box has intersections with the line segment with the internal `intersectAndClip()` method. The bounding box is altered by the bias value so that points at corners and edges of the box are pickable too.

```
osg::BoundingBox bb = drawable->getBound();
bb.xMin() -= _pickBias; bb.xMax() += _pickBias;
bb.yMin() -= _pickBias; bb.yMax() += _pickBias;
bb.zMin() -= _pickBias; bb.zMax() += _pickBias;

osg::Vec3d s(_start), e(_end);
if ( !intersectAndClip(s, e, bb) ) return;

// Just check if the intersector is disabled by someone
// using the setDoDummyTraversal(true) method.
if ( iv.getDoDummyTraversal() ) return;
```

9. The main line-point intersection computing process is not so complex. We first obtain the vertex array and find the distances between the line segment and each vertex. If the distance is acceptable, we will write data to the result list. Related structure and methods are already implemented by the parent class.

```
osg::Geometry* geometry = drawable->asGeometry();
if ( geometry )
{
  osg::Vec3Array* vertices = dynamic_cast<osg::
    Vec3Array*>( geometry->getVertexArray() );
  if ( !vertices ) return;

  osg::Vec3d dir = e - s;
  double invLength = 1.0 / dir.length();
  for ( unsigned int i=0; i<vertices->size(); ++i )
  {
    double distance =  fabs( (((*vertices)[i] -
      s)^dir).length() );
    distance *= invLength;
    if ( _pickBias<distance ) continue;

    Intersection hit;
    hit.ratio = distance;
    hit.nodePath = iv.getNodePath();
    hit.drawable = drawable;
    hit.matrix = iv.getModelMatrix();
    hit.localIntersectionPoint = (*vertices)[i];
    insertIntersection( hit );
  }
}
```

10. Now we will copy the main code from the *Selecting a point of the model* recipe in *Chapter 3*. Remove the virtual method, `doUserOperations()`, in `SelectModelHandler`, and overwrite `handle()` to support use of the new `PointIntersector` class.

11. The usage of `PointIntersector` is exactly the same as original `osgUtil::LineSegmentIntersector` class. The only difference is that it will check intersections between the line segment and points instead of triangles.

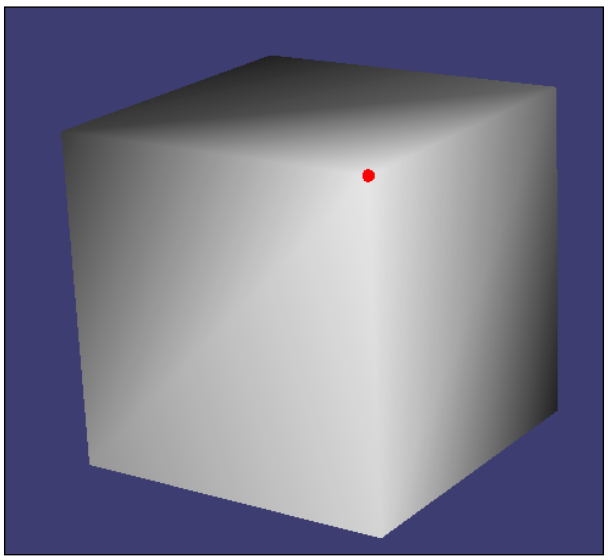
```
osgViewer::View* viewer =
  dynamic_cast<osgViewer::View*>(&aa);
if ( viewer )
{
  osg::ref_ptr<PointIntersector> intersector =
    new PointIntersector(osgUtil::Intersector::WINDOW,
      ea.getX(), ea.getY());
```

```
    osgUtil::IntersectionVisitor iv( intersector.get() );
    viewer->getCamera()->accept( iv );

    if ( intersector->containsIntersections() )
    {
      ...
    }
}
```

12. If there are any intersection results, set the first one to the `_selector` object which displays the picked point in red.

```
// The selVertices variable records the vertices of the
// selector, not the intersection result.
osg::Vec3Array* selVertices = dynamic_cast<osg::
  Vec3Array*>( _selector->getVertexArray() );
if ( !selVertices )
return false;

PointIntersector::Intersection result =
  *(intersector->getIntersections().begin());
selVertices->front() =
  result.getWorldIntersectPoint();
selVertices->dirty();
_selector->dirtyBound();
```

13. The main entry and other parts of the `SelectModelHandler` class are same as the one in *Chapter 6*. Now we can build the application and press *Ctrl* and mouse button to select a point on the model. Our customized intersector works, as shown in the following screenshot:



How it works...

Customized intersectors often need to be careful with the following virtual methods:

Method	Description
`Intersector* clone(osgUtil::IntersectionVisitor &iv);`	Clones the intersector in current local coordinate
`bool enter(const osg::Node &node);`	Checks whether there are intersections for current node's bounding sphere
`void leave();`	Removes temporary data allocated in `enter()`
`void intersect(osgUtil::IntersectionVisitor &iv, osg::Drawable *drawable);`	Computes the intersections of the intersector and the drawable
`bool containsIntersections();`	Returns `true` if there are any intersections

Intersectors must be attached with an `osgUtil::IntersectionVisitor`, which is a node visitor as the name suggests. It traverses the scene graph which is going to be computed and applies the intersector to every node in the scene graph.

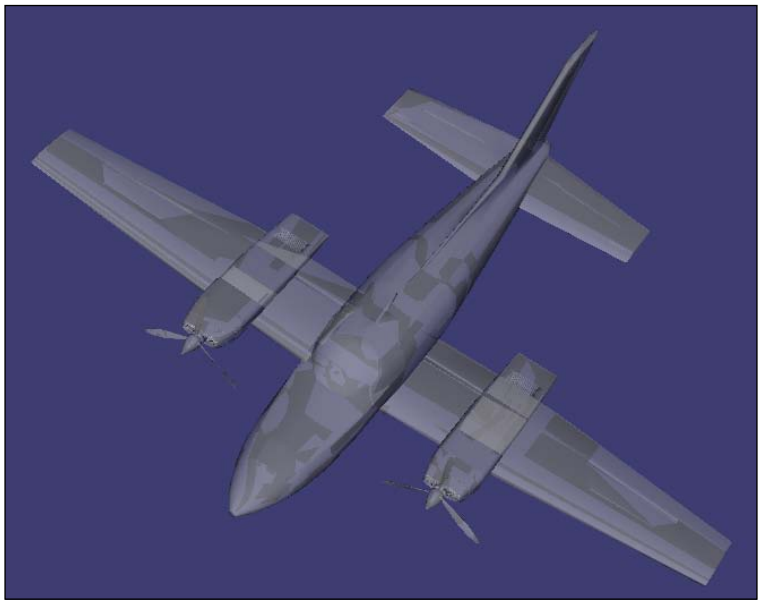
When we arrived at a node, firstly we used `enter()` to see if the bounding sphere has intersections with the intersector. If not, there is no need to continue traversing its children and we can directly end the current branch; otherwise, we have to convert the intersector's parameters to the local node's coordinate frame to make sure the following computation is correct; so `clone()` is used here to create a new 'local space' intersector.

If current node is an `osg::Geode`, we will have to work with its drawables using the `intersect()` method. This method contains actual computation code, such as line-triangle intersection algorithms, and will add new intersection data to the final result list. The `containsIntersection()` method can then be valid and we can obtain results in user applications.

Don't forget that we have `setReadCallback()` method in `osgUtil::IntersectionVisitor` class. It traverses into paged nodes to obtain precise results. The callback executes simply after the `enter()` method. It will add new scene graph under current node and the visitor will automatically traverse it for computation purposes.

Implementing the depth peeling method

In *Chapter 6*, we implemented a multi-pass program that can avoid some bad rendering effects of complex transparent objects. Without this special handling, a translucent cessna will result in a bad image, as shown in the following screenshot:



But multi-pass rendering doesn't solve the problem. The result is in fact still incorrect because we can never arrange the meshes to be rendered in a depth-sorted order. The cessna is self-occluded but it can't be divided again, unless we derive the `osg::Geometry` class and sort each triangle. Of course, the solution is impossible in most complex cases.

So there comes the **depth peeling** algorithm. You may read a short introduction at the following site:

`http://developer.nvidia.com/content/interactive-order-independent-transparency`

It solves the issue with a number of passes over the same scene. The first pass can give us a 'nearest surface' for the second one to 'peel away' depths that are less or equal to it. And the third one will do the same work using the second pass' surface, and so on. We won't see its principle in depth here, but will quickly see how to achieve it with as simple as possible OSG code.

How to do it...

Let us start.

1. Include necessary headers:

```
#include <osg/AlphaFunc>
#include <osg/BlendFunc>
#include <osg/Camera>
#include <osg/Geometry>
#include <osg/Material>
#include <osg/TexGenNode>
#include <osg/TexMat>
#include <osg/TextureRectangle>
#include <osgDB/ReadFile>
#include <osgUtil/CullVisitor>
#include <osgViewer/Viewer>
```

2. We will define a few global variables for the depth-peeling implementation.

```
unsigned int g_width = 800, g_height = 600;
unsigned int g_offset = 8;
int g_unit = 1;
```

3. The `CullCallback` class will be attached to peeling cameras and affects the mapping of the scene.

```
class CullCallback : public osg::NodeCallback
{
public:
  CullCallback( unsigned int unit, unsigned int off )
  : _unit(unit), _offset(off) {}

  virtual void operator()( osg::Node* node,
    osg::NodeVisitor* nv );

protected:
  osg::ref_ptr<osg::StateSet> _stateset;
  unsigned int _unit, _offset;
};
```

4. In the `operator()` method, we mainly compute the texture coordinate matrix for representing the scene's projective mapping. The matrix scales the projected scene to window space, which will then cooperate with an `osg::TextureRectangle` texture to render the scene to a quad with a `_offset` variable.

```
osgUtil::CullVisitor* cullVisitor =
  static_cast<osgUtil::CullVisitor*>(nv);
osgUtil::RenderStage* renderStage =
  cullVisitor->getCurrentRenderStage();
const osg::Viewport* vp = renderStage->getViewport();
if ( !vp ) return;

osg::Matrixd m( *cullVisitor->getProjectionMatrix() );
m.postMultTranslate( osg::Vec3d(1.0, 1.0, 1.0) );
m.postMultScale( osg::Vec3d(0.5, 0.5, 0.5) );
m.postMultScale( osg::Vec3d(
  vp->width(), vp->height(), 1.0) );
m.postMultTranslate( osg::Vec3d(0.0, 0.0,
  -ldexp(double(_offset), -24.0)) );

_stateset = new osg::StateSet;
_stateset->setTextureAttribute(
  _unit, new osg::TexMat(m) );
cullVisitor->pushStateSet( _stateset.get() );
traverse( node, nv );
cullVisitor->popStateSet();
```

   The `pushStateSet()` and `popStateSet()` methods can apply new rendering states to child nodes. It is in fact the essence of OSG's multi-pass rendering mechanism.

5. The `createTexture()` function will create a texture without mipmapping. If it is the middle layer of the depth peeling passes, we should also enable shadow comparison (`GL_TEXTURE_COMPARE_MODE_ARB`) to disable writing to fragments if depth test fails.

```
osg::Texture* createTexture( GLenum format,
  bool asMidLayer )
{
  osg::ref_ptr<osg::TextureRectangle> texture =
    new osg::TextureRectangle;
  texture->setTextureSize( g_width, g_height );
  texture->setFilter( osg::Texture::MIN_FILTER,
    osg::Texture::NEAREST );
  texture->setFilter( osg::Texture::MAG_FILTER,
    osg::Texture::NEAREST );
  texture->setWrap( osg::Texture::WRAP_S,
```

```
    osg::Texture::CLAMP_TO_BORDER );
  texture->setWrap( osg::Texture::WRAP_T,
    osg::Texture::CLAMP_TO_BORDER );
  texture->setInternalFormat( format );

  if ( asMidLayer )
  {
    texture->setSourceFormat( GL_DEPTH_STENCIL_EXT );
    texture->setSourceType( GL_UNSIGNED_INT_24_8_EXT );

    texture->setShadowComparison( true );
    texture->setShadowAmbient( 0 );
    texture->setShadowCompareFunc( osg::Texture::GREATER );
    texture->setShadowTextureMode( osg::Texture::INTENSITY );
  }
  return texture.release();
}
```

6. Middle layer cameras are created with the `createProcessCamera()` function. It is a render-to-texture camera which outputs the color buffer and depth buffer of scene image to textures. Depth peeling uses dual-depth textures for passing clipped depth data. The `prevDepth` is applied to texture unit 1 of current scene, and `depth` is linked with the output buffer.

```
typedef std::pair<osg::Camera*,
  osg::Texture*> CameraAndTexture;
CameraAndTexture createProcessCamera(
  int order, osg::Node* scene, osg::Texture* depth,
    osg::Texture* prevDepth )
{
  osg::ref_ptr<osg::Camera> camera = new osg::Camera;
  ...  // Please find details in the source code
}
```

7. The `createCompositionCamera()` function will composite all middle layers together to generate the final output. It must ensure that all depth layers are rendered in correct order for the blending function to work. The `TraversalOrderBin` parameter can work for such situations (it renders children using the order that node visitors traverse them).

```
osg::Camera* createCompositionCamera()
{
  osg::ref_ptr<osg::Camera> camera = new osg::Camera;
  camera->setDataVariance( osg::Object::DYNAMIC );
  camera->setReferenceFrame( osg::Transform::ABSOLUTE_RF );
  camera->setInheritanceMask( osg::Camera::READ_BUFFER|
```

```
    osg::Camera::DRAW_BUFFER );
  camera->setRenderOrder( osg::Camera::POST_RENDER );
  camera->setComputeNearFarMode(
    osg::Camera::COMPUTE_NEAR_FAR_USING_PRIMITIVES );
  camera->setClearMask( 0 );

  camera->setViewMatrix( osg::Matrix() );
  camera->setProjectionMatrix(
    osg::Matrix::ortho2D(0, 1, 0, 1) );
  camera->addCullCallback( new CullCallback(0, 0) );

  osg::StateSet* ss = camera->getOrCreateStateSet();
  ss->setRenderBinDetails( 0, "TraversalOrderBin" );
  return camera.release();
}
```

8. In the main entry, we will load a cessna model and set its material and rendering states for translucent effects.

```
osg::ref_ptr<osg::Material> material =
  new osg::Material;
material->setAmbient( osg::Material::FRONT_AND_BACK,
  osg::Vec4(0.0f, 0.0f, 0.0f, 1.0f) );
material->setDiffuse( osg::Material::FRONT_AND_BACK,
  osg::Vec4(1.0f, 1.0f, 1.0f, 0.5f) );

osg::Node* loadedModel =
  osgDB::readNodeFile( "cessna.osg" );
loadedModel->getOrCreateStateSet()->setAttributeAndModes(
  material.get(), osg::StateAttribute::ON|
    osg::StateAttribute::OVERRIDE );
loadedModel->getOrCreateStateSet()->setAttributeAndModes(
  new osg::BlendFunc );
loadedModel->getOrCreateStateSet()->setRenderingHint(
  osg::StateSet::TRANSPARENT_BIN );
```

9. Create the composite camera and two depth buffers for ping-pong use.

```
osg::Camera* compositeCamera =
  createCompositionCamera();
osg::ref_ptr<osg::Texture> depth[2];
depth[0] = createTexture(GL_DEPTH24_STENCIL8_EXT, true);
depth[1] = createTexture(GL_DEPTH24_STENCIL8_EXT, true);

osg::ref_ptr<osg::Group> root = new osg::Group;
root->addChild( compositeCamera );
```

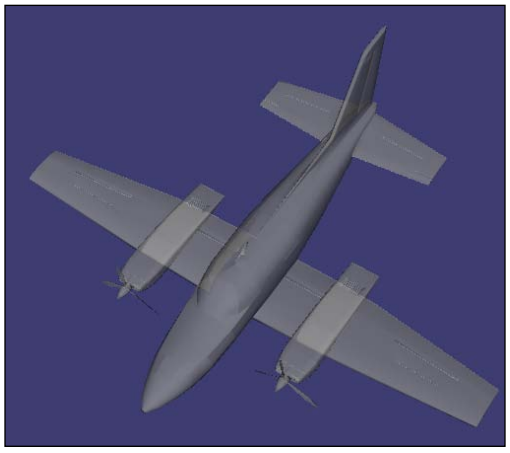
10. We will have eight passes to generate render-to-texture images of the original scene. The resultant quads with textures are inserted into the composite camera in a reversed order.

```
unsigned int numPasses = 8;
for ( unsigned int i=0; i<numPasses; ++i )
{
  CameraAndTexture cat = createProcessCamera(
    i, loadedModel, depth[i%2].get(), depth[(i+1)%2].get() );
  root->addChild( cat.first );

  osg::ref_ptr<osg::Geode> geode = new osg::Geode;
  geode->addDrawable( osg::createTexturedQuadGeometry(
    osg::Vec3(), osg::X_AXIS, osg::Y_AXIS) );

  osg::StateSet* ss = geode->getOrCreateStateSet();
  ss->setTextureAttributeAndModes( 0, cat.second );
  ss->setAttributeAndModes( new osg::BlendFunc );
  ss->setMode( GL_LIGHTING, osg::StateAttribute::OFF );
  ss->setMode( GL_DEPTH_TEST, osg::StateAttribute::OFF );
  compositeCamera->insertChild( 0, geode.get() );
}
```

11. Now start the viewer. It must have the same size as the textures allocated.

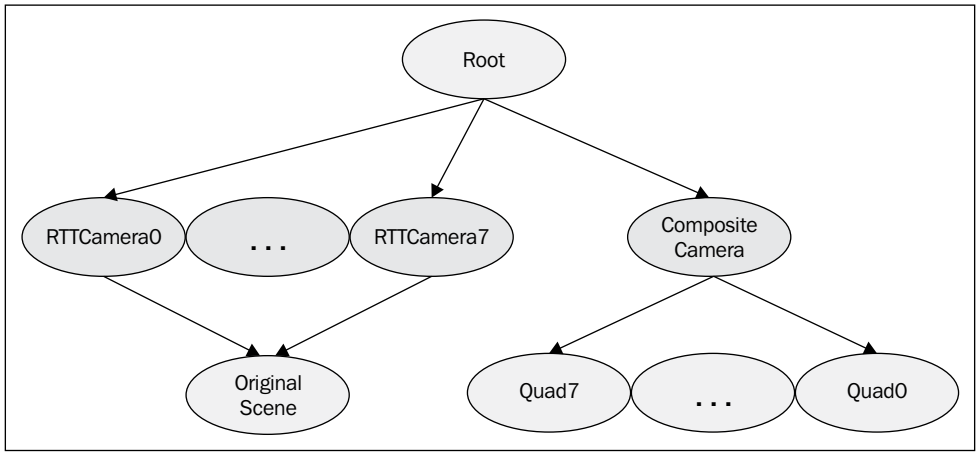
```
osgViewer::Viewer viewer;
viewer.setSceneData( root.get() );
viewer.setUpViewInWindow( 50, 50, g_width, g_height );
return viewer.run();
```

It's done! Now the cessna is rendered correctly. Both the front and back faces are translucent, and no weird spot appears. Although it is not so efficient, we already have a solution for blending complex objects and make them render more realistically.



How it works...

The scene structure can be described as shown in the following diagram:



There are eight RTT cameras rendering the same scene and outputting to eight color and eight depth buffer values as textures. The composite camera collects all color textures and renders them on eight quads separately. The quads are rendered and blended in traversing order, that is, the 7th quad, which is inserted to front at last, will be rendered first, and the first one is rendered at the end.

Because the depth textures enable shadow comparison, they only represent part of the scene according to previous camera's depth values, and, thus, change the behavior of corresponding colored ones. To sum it up, this is a fixed function pipeline method. It works without any shader, so it can perform well under some old and low-level graphics systems.

The ping-pong technique is also used in our recipe. It means to use the output of a given rendering pass as the input of the next one. You may find an OpenGL-based implementation at `http://www.mathematik.uni-dortmund.de/~goeddeke/gpgpu/tutorial.html#feedback2`.

Using OSG in C# applications

What is C# (pronounced C sharp)? It is a programming language developed by Microsoft that supports strong typing, object-oriented or component-oriented framework, and many other features. It is declarative, functional, and generic, and used and suggested by a huge number of high-level and GUI developers. C# can directly reflect the underlying **Common Language Infrastructure** (**CLI**), which represents a complete reversal of the C++ object model. In short, CLI helps you implement .NET programming using C++, and CLI works with C# perfectly as a component of related projects.

Now it is clear that OSG can cooperate with C#. We will use the C++/CLI framework to construct an OSG window first, and then it is easy to be embedded into C# applications at any time.

Getting ready

You need to have Visual Studio installed before creating C++/CLI applications. You may download the Express version at the following website if you don't have one yet:

`http://www.microsoft.com/express`

How to do it...

Let us start.

1. We will create a new **CLR console application** named `cliTest` and integrate OSG functionalities in it. You may choose other **CLR** items if you wish.
2. Add a new `Windows Form` named `osgWindow`, or any other name of your choice. A new `osgWindow.h` and an `osgWindow.cpp` file will be automatically generated, and you will be directed into the window designer.
3. Double click on the form and you will now get to the `osgWindow_Load()` implementation. We want to initialize OSG components here. So let us just add a line:

```
private: System::Void osgWindow_Load(System::Object^ sender,
  System::EventArgs^ e) {
    initializeOSG();
}
```

4. Declare two protected methods in the `osgWindow` class:

```
void runThread();
void initializeOSG();
```

5. Include necessary headers and define a global viewer variable in `osgWindow.cpp`:

```
#include <osgDB/ReadFile>
#include <osgGA/TrackballManipulator>
#include <osgViewer/api/Win32/GraphicsWindowWin32>
#include <osgViewer/Viewer>

using namespace System::Threading;

osg::ref_ptr<osgViewer::Viewer> g_viewer;
```

6. In the `initializeOSG()` method, we first get the Win32 window handle and window rectangle.

```
RECT rect;
HWND hwnd = (HWND)Handle.ToInt32();
GetWindowRect( hwnd, &rect );
```

7. After we obtain the window handle, we can now create window traits and attach graphics context with the scene camera one by one, as we usually did in earlier examples.

```
osg::ref_ptr<osg::GraphicsContext::Traits> traits=
  new osg::GraphicsContext::Traits;
traits->inheritedWindowData = new osgViewer::
  GraphicsWindowWin32::WindowData(hwnd);
...  // Please find details in the source code

osg::ref_ptr<osg::Camera> camera = new osg::Camera;
camera->setGraphicsContext(
  osg::GraphicsContext::createGraphicsContext(
    traits.get()) );
...  // Please find details in the source code
```

8. Allocate the viewer and set a common scene to it.

```
g_viewer = new osgViewer::Viewer;
g_viewer->setCameraManipulator(
  new osgGA::TrackballManipulator );
g_viewer->setCamera( camera.get() );
g_viewer->setSceneData( osgDB::readNodeFile(
  "cessna.osg") );
```

9. Start the thread with the `runThread()` method as the threading callback.

```
System::Threading::Thread^ threadObject =
  gcnew System::Threading::Thread(
    gcnew System::Threading::ThreadStart(
      this, &cliTest::osgWindow::runThread) );
threadObject->Priority =
  Threading::ThreadPriority::BelowNormal;
threadObject->Start();
```

10. In the `runThread()` method, we will run `frame()` method to advance the viewer; meanwhile, we use critical section mechanism to ensure threading safety.

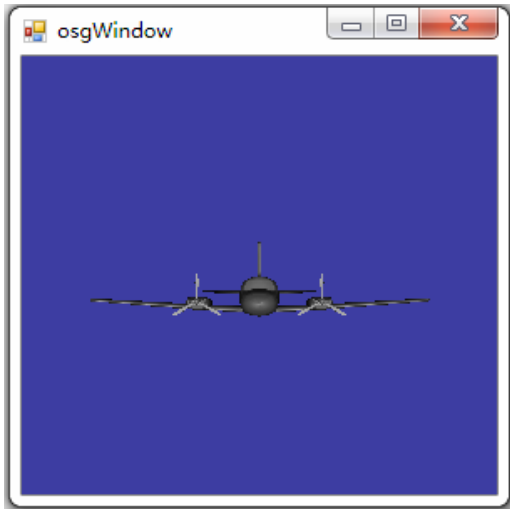
```
void cliTest::osgWindow::runThread()
{
  Object^ dummyViewerObject = gcnew Object;
  while ( !g_viewer->done() )
```

```
    {
      Monitor::Enter(dummyViewerObject);
      g_viewer->frame();
      Monitor::Exit(dummyViewerObject);
    }
}
```

11. At last, we have to start the window form in the main entry.

```
int main(array<System::String ^> ^args)
{
  cliTest::osgWindow^ form = gcnew cliTest::osgWindow();
   Application::Run(form);
  return 0;
}
```

12. Build and start the program. The OSG scene will be successfully embedded into the window form.



How it works...

The next step is to build the functionality to a DLL file (dynamic library). Then we can add the C++/CLI DLL to C#'s "References" section. After that, we will be able to use any public "ref" classes (`cliTest::osgWindow`) and methods in the DLL.

The main idea of using OSG functionalities in C++/CLI, and then in C#, is simple (and also works for Java ports)—first obtain the window handle (for example, a form, a dialog, or a label), and then use the handle to create the graphics context. Attach the graphics context to the camera and we will have the viewer render on current window. To execute `frame()` method to refresh the viewer, we can either choose a timer event or a separate thread. The latter is always of high efficiency in frame rates, but should be paid more attention due to potential multithreading problems.

Using osgSwig for language binding

You may already be wondering if OSG can be integrated into more kinds of programming languages, for example, Python, Java, Lua, Ruby, and so on. The answer is yes. And we don't even have to consider the integration solution separately. The **Simplified Wrapper and Interface Generator** (**SWIG**) library, which is an interface compiler that connects different programming languages, does the amazing work for us. You may have a look at it at `http://www.swig.org/`.

SWIG can automatically or manually find the declarations from C/C++ header files, and use them to generate wrapper code for scripting languages. The script languages which get access to the underlying code (class declarations in header files) can then work with C/C++ to create applications. Of course, it can work with OSG libraries too.

But it is still difficult to understand and extract interface information from C++ headers. Fortunately, there is another OSG-based project named **osgSWIG**. It helps build OSG libraries in Python, Perl, and Java languages. And we can make use of these libraries later for concrete programming work.

Getting ready

Download the SWIG library first. We will need the `swig` executable in the tarball for generating language wrappers later. You can download it from the following link:

`http://www.swig.org/download.html`

Download the osgSWIG library. As osgSWIG doesn't provide a prebuilt package for wrapping OSG 3.0 functionalities, we better check it out using source control utilities. The following is the osgSWIG website:

`http://code.google.com/p/osgswig/`

To check out the latest version, we must use the **Mercurial Hg tool**:

`http://mercurial.selenic.com/downloads/`

We won't introduce the usage of Hg or the compilation of SWIG in this book because of the page limitation. Please read their own instructions carefully, and discuss in related forums if necessary.

In the recipe, we will choose Python for wrapping OSG classes and methods. That is because osgSWIG has already provided full support for this commonly used script language. Python should be already installed under most Linux environments. Developers under other platforms should download the package for their operation systems at `http://www.python.org/download/`.

How to do it...

Let us start.

1. I will assume you have already cloned the osgSWIG source code into the local folder. It also uses CMake as the build system so there should be no problems starting `cmake-gui`; specify the source location.
2. The most important option groups are `BUILD`, `PYTHON`, and `SWIG`. There are also a series of groups with `OSG` as the prefix. If you have set the `OSG_ROOT` environment variable before, they will be automatically filled with correct paths and libraries.
3. The `SWIG` group should be implemented first. We have to specify the `SWIG_EXECUTABLE` to the `swig` program so that the wrapper generator could actually work.
4. In the `BUILD` group, click only the option `BUILD_OSGPYTHON` and `BUILD_WITH_OSGANIMATION`. The first one indicates that we are working with Python bindings of OSG, not Java or Perl ones (but if you have interests, you can try them too). The second one forces osgSWIG to bind **osgAnimation** library as well.

   Note that there are a few more options that can bind some third-party OSG-based projects. We won't choose them this time in order to make this instruction simple enough.
5. The `PYTHON` group should be automatically filled under Linux. You may also specify your own Python `include` directory and libraries if required. Python 2.x and 3.x are both supported.
6. Generate the solution and run `make` (or open the Visual Studio file) to generate the wrapper libraries. After some time of compilation, you will find the libraries (`.pyd`) and scripts (`.py`) generated in `lib/python/osgswig-3.1.0` (or other version number) in your `build` directory.
7. Well done! Now you may either copy these `.pyd` and `.py` files to Python executable directory, or set the `PYTHONPATH` environment variable to direct Python to import libraries from the library folder to enable them. A third way is to add a line `sys.path` in your own Python script files.

8. osgSWIG provides a number of examples that can be used to quickly test your generated libraries. Choose one from `examples/python` in the source code directory. For example, you may execute `simpleosg.py` with Python using the following command:

   ```
   # env python simpleosg.py
   ```

9. The result is nothing special, but don't forget that we just ran a program using Python instead of C/C++! Take a look at the `simpleosg.py` file and you may find something familiar inside.

How it works...

Let us take a look at the `simpleosg.py` file in any text editor:

```
import osgDB, osgViewer

loadedModel = osgDB.readNodeFile("cow.osg")
viewer = osgViewer.Viewer()
viewer.setSceneData(loadedModel)
viewer.addEventHandler(osgViewer.StatsHandler());
viewer.run()
```

It works just like the traditional C++ version. The `loadedModel`, which is a generic type, will be used to keep the loaded cow model. Then we will set it to the `viewer` object and execute `run()` to start the simulation loop. osgSWIG binds namespaces and classes/functions together with a dot (`.`) separator. And as there is no need to explicitly define the node and viewer types, it is much easier for Python developers to learn and use OSG in their own ways.

Contributing your code to OSG

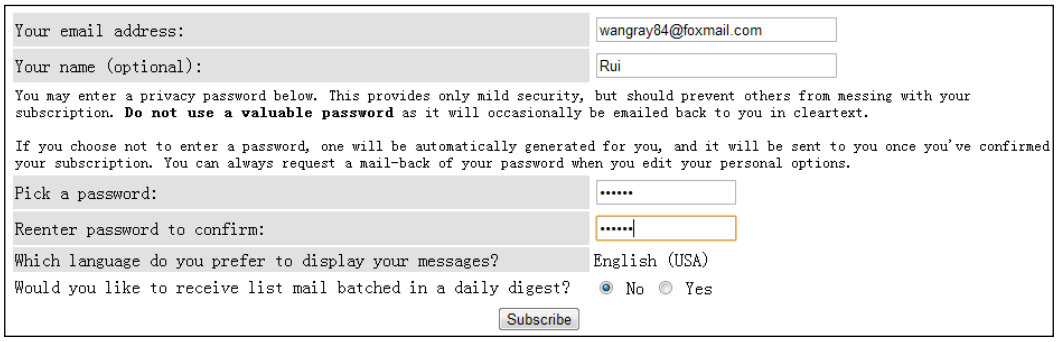
We have finally come to the last recipe of this book. What are your feelings about this book now? Interesting? Boring? Or is it hard to understand the recipes? I would be very glad if you could learn anything from all these recipes. And don't hesitate to mail me to give suggestions, questions, or point out problems in this book.

I am sure at present that you have the ability to work on independent functionalities based on OSG. So why not contribute it to the OSG trunk and share your experience with people all over the world? In this recipe, we will tell you how to register in the `osg-submissions` mail list, and what are the common steps to submit your changes.

How to do it...

Let us start.

1. You should have a valid e-mail account and should not object to receiving mails from the `osg-submissions` mail list. You can also commit on other's submissions, but don't ask programming questions here. The `osg-users` mail list would be a better choice.
2. Open the following link: `http://lists.openscenegraph.org/listinfo.cgi/osg-submissions-openscenegraph.org`.
3. Input your e-mail address and name in the edit boxes on the page, as well as your private password, as shown in the following screenshot:



4. It is not recommended to receive mails in a daily digest, as it is hard to find useful information and reply to others in time.
5. Click on **Subscribe** and soon you will find a confirmation mail in your mail box. Visit the link provided to be added to this mailing list. You may also directly reply to the mail to confirm the instructions.

Everything goes well. Now you can receive submissions daily and contribute to OSG yourselves. There are three suggestions for you before submitting a new functionality or fixing existing bugs:

- Always use real names, not nicknames or meaningless characters. Your name will also be added to the contributor list so that everyone who uses OSG can benefit from your work.
- Write and test your code under the latest trunk version of OSG. Somebody else may have already discover the same problem and his/her submissions may already be accepted. Working on the latest code is also of much help for merging your changes in time.
- Always attach complete source files instead of patches or code segments. And if you have multiple files to modify or add, try to organize them using the same directory structure as current OSG trunk.

How it works...

Note that although OSG uses source control utility (**Subversion**) to manage the source code trunk and branches, only a few people have the right to directly upload the source code to the server. You have to wait until some of them have time to review your changes and give advice before accepting them. Remember that your work will affect everyone who is using OSG, so tests and reviews are necessary all the time.

Thank you for your great efforts on improving this world famous rendering engine. I will be eager to see you in the mail list and community!

Playing with osgEarth: another way to visualize the world

The **osgEarth** project provides another way to view the earth. It uses an XML file to point to necessary imagery, elevation, and vector data and create the whole earth database on the fly. Different from VPB's generation tool, osgEarth doesn't need any pre-processing operations. It creates a quad-tree structure and splits small tiles dynamically from various sources, and sets up all visible layers automatically to maximize performance.

The official website of osgEarth is at `http://osgearth.org/`.

osgEarth supports visualizing earth data at runtime, so it has advantages in handling patches with high resolution and parsing original data from the Internet. But VPB generated files, which are built statically, always perform more efficient in practical applications.

Getting ready

You should download the latest osgEarth source code from the website before compiling and using it. As we are already familiar with the SVN tool, we can directly check out the source code by calling it:

```
# svn co http://svn.github.com/gwaldron/osgearth.git osgEarth
```

But osgEarth actually uses another version control utility called **Git** to manage the project. If you already have a Git client installed, it is recommended that you use the `git clone` command to obtain the source code from the remote server:

```
# git clone git://github.com/gwaldron/osgearth.git osgEarth
```

And then you can use the `git pull` command to update the local source directory at any time.

You can also find available prebuilt binaries at `http://osgearth.org/wiki/Downloads`.

How to do it...

Let us start.

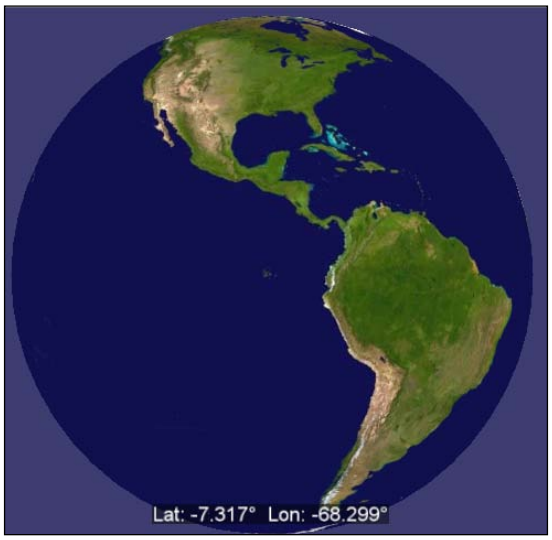
1. Open the `cmake-gui` program and select the `CMakeLists.txt` in the `osgEarth` root directory as the source code location. Configure and generate the makefile or solution. Remember that you must have OSG, CURL, and GDAL installed before configuring osgEarth options.
2. If you have GEOS (`http://trac.osgeo.org/geos/`), LibZip (`http://www.nih.at/libzip/`) or Sqlite (`http://www.sqlite.org/`) installed, you can make osgEarth support more features. But it is OK if you don't, as these are only additional components and plugins.
3. In the build directory, use the following commands to build and install the osgEarth library and applications:

   ```
   # sudo make
   # sudo make install
   ```

4. You can find a new `osgearth_viewer` application in the `bin` directory, and a new plugin named `osgdb_earth`. There are plenty of utilities besides these, but in this book we will only introduce the usages of these two.
5. Find a `.earth` file in the `tests` subdirectory of the source code folder. For example, `gdal_tiff.earth` is a good example that uses a local TIFF file for terrain rendering.
6. Directly execute `osgearth_viewer` in the tests folder:

   ```
   # osgearth_viewer gdal_tiff.earth
   ```

7. The result will be quickly shown in full screen:



How it works...

Open the `gdal_tiff.earth` file with a text editor. The main content of this file is the following:

```
<map version="2">
  <image driver="gdal" name="world-tiff">
    <url>../data/world.tif</url>
  </image>
  <options lighting="false"/>
</map>
```

Although we don't have any understanding of osgEarth's internal implementation, we can still guess that the file indicates an image with the URL `../data/world.tif` that should be read into the scene. It will then be used to construct the earth model with an image driver named `gdal`. The driver plugin's real name is `osgdb_osgearth_gdal`. It is used for reading GDAL-supported formats and converts them to osgEarth's public-data structure.

There's more...

osgEarth provides many other drivers for loading data in other formats or from specific servers. Most drivers have a corresponding test file in the `tests` directory. For example, to load world-wide elevation data from the SRTM server, you can just run:

```
# osgearth_viewer srtm.earth
```

And it will use the TMS driver to load and handle image and height field data which are read from NASA's Blue Marble service.

Use osgEarth to display a VPB-generated database

osgEarth has a VPB driver to provide access to imagery and elevation data in the VPB-generated database, in order to provide high performance and a scalable terrain structure for rendering. To make this mechanism work, we should first use VPB to create an earth (for instance, the earth model using True Marble and SRTM data created earlier in this chapter), and then create a new `.earth` file to point to the outputted `.osgb` file. The driver plugin `osgearth_osgdb_vpb` must exist before we perform the following steps:

How to do it...

Let us start.

1. Let us first create a new text file with any name (`simple_earth.earth` here) and `.earth` as extension:

```
<map name="Virtual Planet Builder model" type="geocentric"
  version="2">
  <image name="imagery layer 0" driver="vpb">
    <url>output/out.osgb</url>
  </image>

  <heightfield name="dem" driver="vpb">
    <url>output/out.osgb</url>
  </heightfield>
</map>
```

2. The URL content `output/out.osgb` must be from the second recipe. We should copy the entire `output` directory to a reachable place for the `.earth` file to be located at. The parameter `type="geocentric"` means that osgEarth is going to handle models in the geographic coordinate system, for example, the earth.

   You can easily understand the meaning of XML elements `<image>` and `<heightfield>`. But for a VPB-generated terrain, image and height field information are already embedded in the scene graph, so we will directly set the root `.osgb` file as the URL of both layers and use the `osgdb_osgearth_vpb` plugin as the terrain rendering driver.

3. Use `osgearth_viewer` to view the new `.earth` file, and you will see the earlier TrueMarble earth built again.

4. You can also have a look at the `tests/vpb_earth_bayarea.earth` file stored in the osgEarth source directory. It will load the prebuilt earth model `bayarea` from the Internet and render with the `osgearth_viewer` executable. It has more levels and is very detailed in some areas of the USA.

How it works...

As VPB generated files always have similar node structures, and they can save both build options, height, and image layers with one `osgTerrain::TerrainTile` class, it is possible to parse and convert them into osgEarth's data format. And because osgEarth is an on-the-fly terrain generator and renderer, the VPB database added can directly combine with other dynamic imagery layers, or have new hi-resolution insets on existing data, without any terrain-building having to be done in advance. This makes the manipulation of earth surface and textures much more efficient and much easy and quick.

To note, only databases created with the `--terrain` option can work with osgEarth at present, as they contain osgTerrain classes for different data operation. VPB terrain created with `--polygonal` are regular geometries without geographical information and cannot be recognized by osgEarth anymore.