Pro OGRE 3D Programming

Gregory Junker

Apress®

Pro OGRE 3D Programming

ISBN-13: 978-1-59059-710-1

ISBN-10: 1-59059-710-9

Library of Congress Cataloging-in-Publication data is available upon request.

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Matt Wade
Technical Reviewer: Steve Streeting
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Kylie Johnston
Copy Edit Manager: Nicole LeClerc
Copy Editor: Ami Knox
Assistant Production Director: Kari Brooks-Copony
Production Editor: Laura Esterman
Compositor/Artist: Kinetic Publishing Services, LLC
Proofreader: Lori Bring
Indexer: Broccoli Information Management
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit http://www.springeronline.com.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Source Code/Download section.

CHAPTER 4

Ogre First Steps

Now that you understand more about how Ogre 3D is designed and how the various parts work together, you are ready to begin writing actual programs that use the Ogre API. I will start you off easy, with a very basic bare-bones Ogre application, and work in more complicated topics as we go.

One of Ogre's best "selling points," if you will, is its flexibility. You have as much or as little control over the execution of your Ogre application as you care to exert. Ogre is rather adept at providing acceptable default behavior, and just as accommodating when you want to turn off the autopilot and grab the yoke yourself.

This chapter will cover the entire spectrum of execution patterns available to the Ogre programmer, from fully automatic to fully manual. Working source code is available for download from the book's web site, and will build and run out of the box. The code snippets in this chapter are provided to highlight points made in the text, but I will not be providing any complete source code listings in this book.

Ogre Initialization: Quick Start

The first thing you do in any Ogre application is create an instance of **Root**. You must do this before calling any other Ogre operations (except for altering **LogManager** settings as you will see). Ogre's **Root** constructor takes several optional string arguments, all of them file names.

```
Root *root = new Root();
Root *root = new Root("plugins.cfg");
Root *root = new Root("plugins.cfg", "ogre.cfg");
Root *root = new Root("plugins.cfg", "ogre.cfg", "ogre.log");
Root *root = new Root("", "");
```

All of the preceding will run just fine. The second-to-last line in the example contains the default file names for the **Root** constructor (these are the names that are assumed in the first line, with no parameters).

plugins.cfg

An Ogre *plug-in* is any code module (DLL or Linux `.so` file) that implements one of the Ogre plug-in interfaces, such as **SceneManager** or **RenderSystem**. `plugins.cfg` in the examples earlier contains the list of plug-ins that Ogre will load at startup, as well as their location (see Listing 4-1).

Listing 4-1. *Contents of the "Stock"* `plugins.cfg` *File That Is Included in the Ogre Source and SDK Distributions*

```
# Define plugin folder
PluginFolder=.

# Define plugins
Plugin=RenderSystem_Direct3D9
Plugin=RenderSystem_GL
Plugin=Plugin_ParticleFX
Plugin=Plugin_BSPSceneManager
Plugin=Plugin_OctreeSceneManager
Plugin=Plugin_CgProgramManager
```

Listing 4-1 is the stock `plugins.cfg` file that ships with the Ogre samples. If you choose to use a plug-in configuration file with your application, it will probably look a lot like this one. Of course, you do not have to call your plug-in configuration file "plugins.cfg"; you can call it whatever you like, and supply that file name to the **Root** constructor. If you do not supply an argument for this parameter, **Root** will look for a file called "plugins.cfg" in the same directory as the application and try to load whatever it finds in there. If you supply an empty argument (`""`, as I did in the fifth **Root** constructor example line earlier), then **Root** will not look for a plug-in configuration file at all, and you will have to load your plug-ins manually (as you will see later in the section "Ogre Initialization: Manual").

The `PluginFolder` directive tells Ogre where to look to find the plug-ins listed in the file. How this path is interpreted is entirely up to you: if you use an absolute path, it will look only in that directory for the plug-ins. If you use a relative specifier (i.e., a path that does not begin with `/` or `\`), then it will look in that path relative to the current working directory (usually the directory containing your application, or the directory containing the `.vcproj` file when running in the Microsoft Visual Studio debugger, for example). The `PluginFolder` specified in our example file tells Ogre to look for the plug-ins in the current working directory ("`.`"). Note that Ogre will append a `/` or `\` (depending on the operating system) to whatever is (or is not) in this setting, so leaving it blank will cause Ogre to look for your plug-ins in "`/`" or "`\`" (the root of the current drive on Windows, or the root directory on Linux). This setting is required if you use a plug-in configuration file; commenting out this line is identical to leaving its value empty (lines beginning with a `#` are treated as comments in Ogre config files). Finally, on Mac OS X, this setting is ignored since OS X looks in `Resources/` for the plug-ins.

The remainder of the file contains the list of plug-ins you want Ogre to load. Notice how the extensions are left off; this is on purpose, and allows you to use the same configuration file on multiple platforms without having to sweat the details of file naming conventions (even though Linux is rather accommodating and does not care one way or another what extensions you give your files).

■ **Caution** Also notice that no spaces are used on either side of the `=` in the file. Spaces in plug-in definition lines is a syntax error and will cause the plug-in **not** to be loaded.

The top two plug-ins listed previously are render system implementations; the rest are feature and scene manager plug-ins. You do not have to include all of these in your program; at the very least you need only a render system. If you plan to do anything beyond very simple scenes (and I mean **very** simple), you will want at least the **OctreeSceneManager** plug-in as well.

Ogre.cfg

Ogre provides a simple means of setting basic video rendering options via native GUI dialogs, such as the one in Figure 4-1.

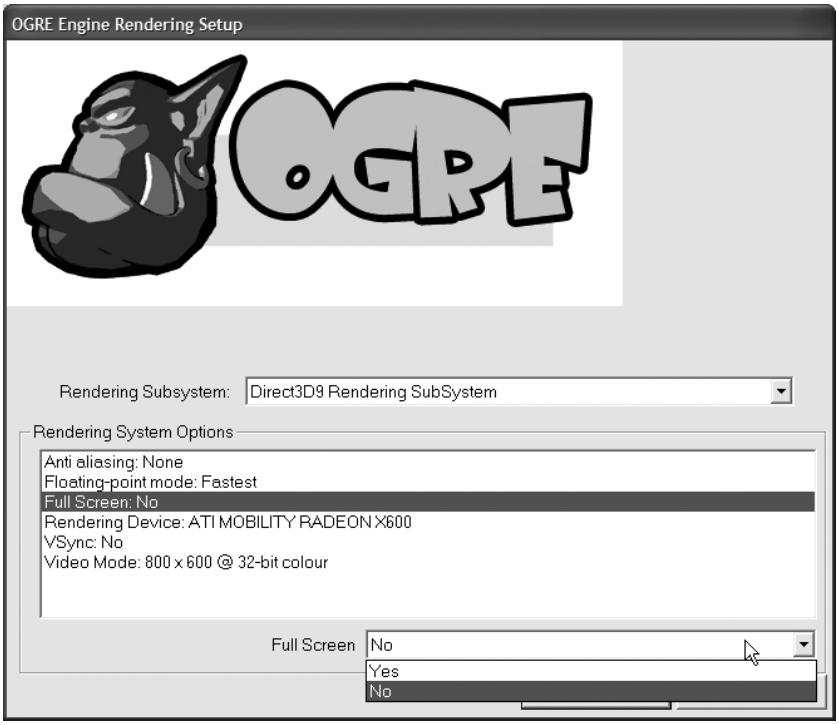

Figure 4-1. *Ogre Win32 configuration dialog*

I can hear you saying to yourself, "Neat, Ogre provides my application's setup dialog for me." Well, not quite. If you are doing only 3D graphics (for example, visualization) with no input, sound, or anything else that needs user configuration, then yes, it is probably sufficient to use Ogre's config dialog, and you can even customize it a bit (for example, if you want to use a different logo). However, you really cannot change much else about it; it is dedicated solely to managing Ogre settings, and nothing else. You should write your own configuration applet and use the manual initialization methods you will see later in this chapter.

But, let's say you want to use the Ogre configuration dialog. How to get it to show up?

```
Root *root = new Root();
bool rtn = root->showConfigDialog();
```

That's all there is to it. `showConfigDialog()` returns `true` or `false` depending on whether the user clicked the OK or Cancel button: `true` for OK, `false` for Cancel. You should really consider shutting down the application if the user clicks the Cancel button. Using the Ogre config dialog takes care of setting the selected render system along with all of the parameters that you can change in the dialog.

What does all of this have to do with `Ogre.cfg`? Well, this dialog is what generates that file. Of course, you can create one yourself by hand, and if you are using the more manual methods that you'll see later, you'll likely not even have one of these, but for now, let's look at what it contains (see Listing 4-2).

Listing 4-2. *Contents of a Sample* `Ogre.cfg` *File*

```
Render System=Direct3D9 Rendering Subsystem

[Direct3D9 Rendering Subsystem]
Allow NVPerfHUD=No
Anti aliasing=None
Floating-point mode=Fastest
Full Screen=No
Rendering Device=ATI MOBILITY RADEON X600
VSync=No
Video Mode=800 x 600 @ 32-bit colour

[OpenGL Rendering Subsystem]
Colour Depth=32
Display Frequency=60
FSAA=0
Full Screen=Yes
RTT Preferred Mode=FBO
VSync=No
Video Mode=1024 x 768
```

Note that these options correspond to the options available in the configuration dialog. Even though these are perfectly human-readable and intuitive, this file really is meant for machine consumption, and it's probably best, if you are using an `Ogre.cfg` file, to let the configuration dialog handle loading this file in case the format changes. Case is important, so it's easy to screw something up editing this file by hand, and not easily know why.

Ogre also provides a method to load an existing `Ogre.cfg` configuration file:

```
if (!root->restoreConfig())
     root->showConfigDialog();
```

This is a very common sequence when using the `Ogre.cfg` method. If the `restoreConfig()` call fails, then no `Ogre.cfg` file exists, and the application shows the Ogre configuration dialog to obtain user preferences (and save them to an `Ogre.cfg` file when the user clicks OK). In this way, you can avoid forcing users to see this dialog every time they run your app.

You can also save the current state of Ogre to `Ogre.cfg` (or whatever name you supplied when you called the **Root** constructor) at any time you wish, with the `saveConfig()` call:

```
root->saveConfig();
```

Ogre.log

Ogre provides diagnostic and exception logging facilities using its log management classes. This is useful for obtaining details about a crash on a client machine without having to ask users technical details about their setup. The log output that Ogre generates contains all events and system initialization, state, and capabilities information from each run of the Ogre-based program. This output is sent to a disk file; while you can change the name of this file, you cannot provide an empty value to the third **Root** constructor parameter (unless you have already created the log directly by calling `LogManager::createLog()`, as in Listing 4-3, prior to creating your **Root** instance). You simply are required to have an Ogre log file, regardless of its name.

Listing 4-3. *Creating a Log Manager Instance, the Custom Way*

```
// create an instance of LogManager prior to using LogManager::getSingleton()
LogManager* logMgr = new LogManager;
Log *log = LogManager::getSingleton().createLog("mylog.log", true, true, false);

// third param is not used since we already created a log in the previous step
Root *root = new Root("", "");
```

This code will direct Ogre to create a custom log file named `mylog.log`. This is the preferred method (only method, actually) of customizing your log output, if you need to alter more logging options than just the output file name. The second parameter tells Ogre to make this the default log file. The third tells Ogre to send messages to `std::cerr` as well as the log file, and the fourth tells Ogre **not** to suppress file output (write to the custom log file as normal, in other words). If you do not do any of this, and either do not supply this parameter to the **Root** constructor (which causes Ogre to default to `Ogre.log`) or supply an alternate file name for the log file, Ogre will send all logged data to this file as the default data logging location.

The Ogre **Log** and **LogManager** classes do not support alternate stream types. However, you can register a *log listener* with the Ogre **LogManager** class and redirect the log data any way you like. If you want no file output at all, you can direct the custom log to suppress output, but Ogre must have a log data destination.

■ **Note** There is another way to suppress log files entirely, and that is by simply not calling `createLog()` after instancing **LogManager** prior to instancing **Root**. Since the log messages will not have anywhere to go by default, they are lost. This practice as default behavior, however, is **highly discouraged**, because if something goes wrong in your application, it becomes very difficult to figure out what happened without logging. Granted, you can walk the user through running your app with custom logging command-line options if you want, but unless you have a very good reason to suppress the log entirely, then don't.

Listing 4-4 will intercept log messages and allow you to deal with them as you see fit in the body of the `write()` callback method.

Listing 4-4. *Intercepting Ogre Logging*

```
class MyLogListener : public LogListener
{
public:
     void write (const String& name, const String& message,
     LogMessageLevel level, bool maskDebug)
{
     // redirect log output here as needed
};

MyLogListener *myListener = new MyLogListener;

// this is the same as calling LogManager::getSingletonPtr() after the
// LogManager has first been instanced; the same pointer value is returned
LogManager *logMgr = new LogManager;

LogMgr->addListener(myListener);
logMgr->createLog("mylog.log", true, false, true);
logMgr->setLogDetail(LL_NORMAL);

Root *root = new Root("", "", "mylog.log");
```

Notice that we changed the last parameter to the `createLog()` method to `true`; this will disable file output to the named log file, freeing you up to handle the log data as you see fit (send it to a debugging window, network stream, etc.). Since we turned off writing to `std::cerr` by setting the third parameter to `false`, the user should see no log messages anywhere, other than the messages you log in your `write()` method. We set the logging detail level to "normal"; other options are `LL_LOW` (very little detail) and `LL_BOREME` (probably way too much detail, but useful for particularly tricky debugging).

Render Window

At any point after a render system has been selected (in the current example, that would be when the user selects and configures a render system in the config dialog and clicks the OK button), you can call the **Root** `initialise()` method:

```
root->initialise(true, "My Render Window");
RenderWindow *window = root->getAutoCreatedWindow();
```

The first line will instruct **Root** to complete its initialization and create a render window with the settings the user selected in the config dialog and "My Render Window" for its title. If you do not supply a window title, it defaults to "OGRE Render Window." The first parameter, which is required, tells Ogre whether it should automatically create a render window; for the sake of this section, we take the easy road and tell Ogre to create our render window for us. The second line obtains a pointer to that **RenderWindow** instance.

The render window is only part of rendering your scene. It is the canvas, the surface onto which Ogre renders your content. Ogre needs at least one camera to "take pictures" of your scene, and one or more viewports, which are regions on a rendering surface (such as a render window) into which the camera places its contents.

You will learn more about the Ogre scene manager classes later in this chapter in the section "Scene Manager"; for now, it is sufficient to know that the scene manager API acts as a "factory" for many different types of objects that exist in your scene, including cameras. In order to obtain a new camera for rendering your scene, you call the `createCamera()` method on the **SceneManager** interface:

```
Camera *cam = sceneMgr->createCamera("MainCamera");
cam->setNearClipDistance(5);
cam->setFarClipDistance(1000);
cam->setAspectRatio(Real(1.333333));
```

In this example, `sceneMgr` is a pointer to an existing instance of **SceneManager**. **Camera** has many properties that can be adjusted and methods to adjust them.

The preceding code demonstrates the basics you will need for a minimal Ogre application. It sets the aspect ratio equivalent to a 4:3 display setup (that of most CRTs and nonwidescreen LCD displays). This value can be set or reset at any time, and usually is set to the ratio of the viewport's width to its height (as we did here, since 4/3 is nearly equal to 1.33333).

This code also sets the near and far clip plane distances. I used my standard outdoor settings of 5 and 1000 units, but you can use whatever you like, so long as the ratio between the far and near is in the neighborhood of 1000, or less.

■ **Tip** A popular misconception is that clip planes are a cheap method to reduce the amount of "stuff" that a card has to render. While this certainly is a side effect of clip distance selection (and most modern cards support infinite far clip planes anyway), the primary reason for a particular set of clip plane distances is to maintain optimal depth buffer resolution. The depth buffer resolution is a direct function of the ratio between the far and near clip distances, and selecting distances that result in too coarse a resolution will invariably result in *depth fighting*. This phenomenon occurs when the depth-sorting algorithm on the GPU cannot tell which objects are "in front of" other objects. You end up with bits of objects that are at relative equivalent depths in the scene, rendering "through" each other. This occurs because the depth resolution is low enough that objects at slightly different depths were assigned the same depth value. The only solution in this case is to increase the depth precision, typically by altering the near plane distances (which give much more bang for your precision buck compared to altering the far clip distances). Google "depth fighting" for more.

Later in this chapter, you will see how to manipulate the Ogre **Camera** in more advanced ways, but for now, this is sufficient for the purposes of creating a viewport in our rendering window.

```
Viewport *vp = window->addViewport(camera);
vp->setBackgroundColour(ColourValue(0, 0, 0));
```

This code creates a new viewport in the render window we created when we called the `initialise()` method earlier on **Root**. It also sets the background color of the viewport to black.

Render Loop

The simplest way to set Ogre about the task of rendering your scene is to invoke the `startRendering()` method on **Root**:

```
root->startRendering();
```

This will cause Ogre to render endlessly whatever renderable content you have in your scene. It exits either when the render window is closed using the normal windowing system means (for example, clicking the small x button in the upper-right corner of a Windows app or closing it from the taskbar), or when a registered *frame listener* returns `false`. An alternate method of ending the rendering loop is to call `Root::getSingleton().queueEndRendering()` anywhere in your code, but typically you will just return `false` from your frame listener when you are using the `startRendering()` method.

Frame Listener

Frame listeners are the only way you can invoke your own code during the Ogre render loop when using the `startRendering()` method. A frame listener is simply a class that implements the **FrameListener** interface, and is just a callback that allows Ogre to invoke your code at the beginning and/or end of each frame (see Listing 4-5).

Listing 4-5. *Creating and Adding a Frame Listener to the Ogre* ***Root***

```
class myFrameListener : public FrameListener {
public:
     bool frameStarted (const FrameEvent &evt);
     bool frameEnded (const FrameEvent &evt);
};

bool myFrameListener::frameStarted(const FrameEvent &evt) {

     // really cool stuff to do before a frame is rendered
     return true;
}

bool myFrameListener::frameEnded(const FrameEvent &evt) {

     // really cool stuff to do after a frame is rendered
     return true;
}

Root *root = new Root();
MyFrameListener myListener;

// YOU HAVE TO ADD A FRAMELISTENER BEFORE YOU CALL startRendering()!!!
root->addFrameListener(myListener);
root->startRendering();
```

The implementation of the `frameStarted()` method in Listing 4-5 will be called before Ogre invokes the rendering pipeline. The `frameEnded()` method is less often used, and is useful if you need to clean up after your app each frame. It likewise is called after Ogre completes the rendering process each frame.

Typically during each frame, you process HID (Human Interface Device, such as keyboard, mouse, or joystick) input events. Depending on the particular event, you might cause a model in your scene to move and/or rotate, cause a camera to move or rotate, start a process for a player-character to put a sleeping hex on an NPC troll for 10 exp, or whatever. Regardless of what it is, it occurs during one or both of the **FrameListener** callback methods—usually during `frameStarted()`, because more than likely you will process changes to the scene **before** the scene is rendered, rather than after.

I have put together a very simple working application that implements the concepts you have learned so far; you can find it with this book's source, downloadable from the Apress web site's Source Code section; it appears in the QuickStart project in the CH04 solution. It has just about the bare minimum functionality. The frame listener I have implemented simply counts the elapsed time until 15 seconds have passed, then exits the application gracefully. Pretty boring, but a fantastic way to ensure you have your build environment set up correctly.

Ogre Initialization: Manual

In this section, I will walk you through all of the steps needed to set up an Ogre application without using any of the automatic, "behind the scenes" features at all. You will also learn how to write your own main loop that invokes Ogre's frame rendering.

I will go over, in more detail, each of the main Ogre subsystems involved in initializing and starting up Ogre, and outline by example your options for each step of the process.

Root

In the previous section, you learned that a great deal of functionality is invoked on your behalf when you supply the **Root** constructor with the names of config files that contain the plug-ins and render settings you wish to use for your application. You also saw the easy way to create a render window in Ogre, and the most basic way of running a loop to render to that window. All of this functionality is available through **Root**, and as you will find out, so is the manual way.

"Manual" simply means that you take complete control over particular aspects of the Ogre initialization process. You don't have to do everything manually; for example, you can still run an automatic render loop with an automatically created window, but with manual render system selection and setup (common in games that provide a GUI for video settings or options). In this section, you will see how to do all parts manually, and which various options exist for each part.

plugins.cfg

Loading plug-ins is probably the most straightforward part of initializing Ogre, so we will start here. **Root** provides two methods to deal manually with plug-ins:

```
void loadPlugin(const String& pluginName);
void unloadPlugin(const String& pluginName);
```

The first method, not surprisingly, loads the named plug-in. The second likewise unloads the named plug-in. The name used is the actual file name of the plug-in; for example, "Plugin_ParticleFX". Note that it is not required to provide an extension when supplying a plug-in name. Ogre will detect "missing" extensions and append the proper extension for the platform (`.dll` on Windows and `.so` on Mac OS X and Linux). You may supply an extension, of course, but then you are making your application platform-specific (Ogre will not strip extensions and append the proper one if the extension is incongruent with the platform). Moral of the story: be easy on yourself and let Ogre handle the housekeeping.

Paths to the plug-ins follow the path searching rules of the particular platform, since Ogre uses the platform-specific dynamic library loader APIs on each platform (`LoadLibrary()` on Windows, `dlopen()` on Mac OS X and Linux). Typically, the directory containing the executable is searched first, then the path defined in the system `PATH` environment variable, and then (on Linux) the `LD_LIBRARY_PATH` environment variable.

By default, Ogre does not name differently the Debug and Release builds of its plug-ins; this is a very common source of confusion, and the main culprit for the infamous "ms_singleton" assert encountered when mixing Debug and Release versions of the same DLL/`.so`.

■ **Tip** The "ms_singleton" assert occurs when your application loads a Debug and Release version of the same DLL at execution time. The reason is the nature of the singleton implementation in Ogre. Ogre creates the one-and-only single instance of each of its singleton classes in the **Root** constructor and access to them is via static member methods. This is all fine and good until your app loads the "opposite" DLL type at run-time, usually because of a plug-in mismatch (Debug plug-ins into a Release app or vice versa). This plug-in will in turn have been linked against the "opposite" OgreMain library, and when they are loaded, the operating system will load that opposite OgreMain library. The next time one of the singleton classes residing in OgreMain is called, the new singleton will try to create an instance of itself, detect that an instance already exists, and BANG! . . . instant assert.

The simplest way to avoid Debug/Release clashes is to name your plug-ins accordingly; the naming standard within Ogre is to append a "_d" to the file name, as in `OgreMain_d.dll`. This would also require `#if defined` statements in your code to allow conditional builds, as in

```
Root *root = new Root;

#if defined(_DEBUG)
     root->loadPlugin("Plugin_ParticleFX_d.dll");
#else
     root->loadPlugin("Plugin_ParticleFX.dll");
#endif
```

This is a small price to pay to avoid the time (and hair loss) involved in trying to figure out which of your plug-ins is responsible for the "ms_singleton" assert. Of course, if you never use Debug builds, then you won't have this problem . . . but if you don't use the debugger, then you probably are not reading this anyway, opting instead to spend copious amounts of time staring at your code, trying to figure out (presumably using the Vulcan mind-meld) why it doesn't work. Use the debugger; that's why it's there.

You do not need to unload plug-ins. **Root** will clean up after itself rather nicely, and releasing and unloading the plug-ins is one of those cleanup activities. Of course, you may wish to unload a plug-in early in order to shut down gracefully other parts of your application, but calling `delete root` will work just fine.

Ogre comes with several plug-ins:

- **Plugin_OctreeSceneManager**: Octree-based scene graph and manager. Also contains the **TerrainSceneManager**, which is a derivative of the **OctreeSceneManager** (OSM) optimized for dealing with heightmapped terrain scenes.
- **Plugin_BSPSceneManager**: Reference implementation of a BSP scene manager for loading and dealing with Quake 3–level files. Growing more obsolete by the day, and is not regularly maintained or supported (provided as a demo reference only).
- **Plugin_CgProgramManager**: Plug-in provided for loading, parsing, compiling, and managing GPU shader programs written in Cg. As Cg falls farther and farther behind the technology curve (it supports only shader profiles less than 3.0), it becomes less useful overall; Ogre can deal with HLSL and GLSL programs internally.
- **Plugin_ParticleFX**: Particle system plug-in; provides several affectors and emitters for most common particle effects.
- **RenderSystem_Direct3D9**: Abstraction layer providing access to the native Direct3D 9 API on Windows.
- **RenderSystem_GL**: Abstraction layer providing access to the OpenGL API on all platforms.

We will go further into the **OctreeSceneManager** plug-in when we discuss Ogre scene management, and we will dive into the **ParticleFX** plug-in when we deal with the more advanced topics later in the book. As mentioned, the other two exist increasingly more for reference than actual use (though both of them work quite well), and we won't discuss them much more in this book.

You may find it odd to see the render systems included with the plug-ins. They are, however, just that: plug-ins. True to the Ogre design, they implement an abstract API and are "plugged into" Ogre using the same mechanism as all of the other plug-ins.

Render Systems

Ogre needs to have a render system loaded in order for it to be available for use. You do this with the same `loadPlugin()` API you already saw:

```
// create a new Root without config files
Root *root = new Root("", "");

root->loadPlugin("RenderSystem_Direct3D9");
root->loadPlugin("RenderSystem_GL");
```

This will load and make available both of the supported render systems (assuming the proper hardware drivers are installed and operational). **Root** provides a means to find out which render systems are loaded and available, via the `getAvailableRenderers()` method:

```
RenderSystemList* getAvailableRenderers(void);
```

Root also provides several other methods for obtaining information about and manipulating the render system list:

```
void addRenderSystem(RenderSystem* newRend);
RenderSystem* getRenderSystemByName(const String& name);
void setRenderSystem(RenderSystem* system);
RenderSystem* getRenderSystem(void);
```

Typically you will only use the `setRenderSystem()` method in conjunction with the `getAvailableRenderers()` method, to instruct Ogre to use a render system, usually as a result of a user's choice (see Listing 4-6).

Listing 4-6. *Setting the Render System to Use in an Ogre Application*

```
// RenderSystemList is a std::vector
RenderSystemList *rlist = root->getAvailableRenderers();
RenderSystemList::iterator it = rList->begin();

while (it != rList->end()) {

     // Ogre strings are typedefs of std::string
     RenderSystem *rSys = *(it++);
     if (rSys->getName().find("OpenGL")) {

          // set the OpenGL render system for use
          root->setRenderSystem(rSys);
          break;
     }
}
// note that if OpenGL wasn't found, we haven't set a render system yet! This
// will cause an exception in Ogre's startup.
```

The code in Listing 4-6 will look for the OpenGL render system in the list of available render systems, and set it if it exists. Like the warning says at the end of the listing, this code will cause an exception within the `initialise()` method of **Root** if no render system has been set, so treat this example as just that: an example. Most likely what you will do is populate a drop-down menu in your application's GUI with the names of the available render systems as provided by `getAvailableRenderers()`, and then call `setRenderSystem()` with the user's selection.

`addRenderSystem()` typically should only be called from a plug-in initialization function, unless the render system implementation comes from another source (such as a custom application logic loading system). `getRenderSystemByName()` is useful for quick detection of installed render systems, and returns `NULL` if the named render system has not been loaded. Keep in mind that spelling and case sensitivity are paramount when accessing render systems by name. As of this writing, the proper spellings and case for the two provided render systems are "Direct3D9 Rendering Subsystem" and "OpenGL Rendering Subsystem." Finally, `getRenderSystem()` simply returns the active render system, or `NULL` if no render system is active.

Render Window

The main reason you would want to create your application's render window(s) manually is closer management of both the creation parameters and the point at which the window is created. One common reason for creating your render window manually is to embed it in an alternate windowing toolkit or system (such as the Qt or wxWidgets cross-platform windowing libraries), e.g., for use in a level or map editor application. You may also wish to supply different or additional parameters to the window, even if you are using the Ogre configuration dialog to allow the user to select some of the video options.

The first thing to know is that when manually creating a render window, the method that is used is on the **RenderSystem** class selected earlier, not on **Root**. Also, you may create your render window(s) at any time once you have a pointer to a valid **RenderSystem** instance. As you can see from the code in the "Ogre Initialization: Quick Start" section, you will need to have a **RenderWindow** pointer in order to create a viewport, but when you are manually creating the render window, the call to `initialise()` on **Root** can be done at any time. The call to `initialise()` will initialize the plug-ins you have specified, so if you have special startup sequence needs pertaining to plug-ins and your application's render window(s), you can deal with the two separately (see Listing 4-7).

Listing 4-7. *Basic Manual Ogre Application Setup Steps*

```
#include "Ogre.h"

// create a new Root without config files
Root *root = new Root("", "");

// load the render system plug-ins
root->loadPlugin("RenderSystem_Direct3D9");
root->loadPlugin("RenderSystem_GL");

// pretend the user used some other mechanism to select the
// OpenGL renderer
String rName("OpenGL Rendering Subsystem");
RenderSystemList *rList = root->getAvailableRenderers();
RenderSystemList::iterator it = rList->begin();
RenderSystem *rSys = 0;

while (it != rList->end()) {

     rSys = *(it++);
     if (rSys->getName() == rName) {

          // set this renderer and break out
          root->setRenderSystem(rSys);
          break;
     }
}
```

```
// end gracelessly if the preferred renderer is not available
if (root->getRenderSystem() == NULL) {

     delete root;
     return -1;
}

// We can initialize Root here if we want. "false" tells Root NOT to create
// a render window for us.
root->initialise(false);

// set up the render window with all default params
RenderWindow *window = rSys->createRenderWindow(
     "Manual Ogre Window",  // window name
     800,                   // window width, in pixels
     600,                   // window height, in pixels
     false,                 // fullscreen or not
     0);                    // use defaults for all other values

// from here you can set up your camera and viewports as normal
```

We have not done anything very earth-shattering in Listing 4-7; we supplied the render window creation call with values that likely would be derived from some other programmatic source (such as your game's video options configuration UI). Understand that the window *name* is the same as the window *title* (as it appears in the window's title bar and windowing system widgets) if it is not overridden in the list of parameter values that is optionally provided during this call.

■ **Note** The **RenderWindow** class is derived from the more general **RenderTarget** class, which implements the generalized notion of a *rendering surface*, useful when dealing with non-frame-buffer render targets such as textures. All **RenderTarget** instances are accessible via the name given them at their creation (a pattern used throughout Ogre, as you'll see), and the first parameter of the `createRenderWindow()` call is that name.

In this example, we accepted the defaults for all other parameters that the render window understands. You can see a full description of those parameters in the online API reference, or in the `OgreRenderSystem.h` header file in the SDK or the full source (in all of which I suggest looking first, as the list of options is always subject to change, and what I mention here will surely be outdated shortly after publication).

Let's say that we wanted to put the render window in the upper-left corner of the screen, and use a different string for the render window title than we used for the render window name. For this I introduce you to Ogre's **NameValuePairList** class, which is simply a typedef of the STL **map** class. The last parameter of the `createRenderWindow()` call takes a pointer to an instance of this class. This instance should be populated only with the parameters you want to change; all "missing" parameters will assume default values.

```
NameValuePairList params;
params["left"] = "0";
params["top"] = 0;
params["title"] = "Alternate Window Title";

RenderWindow *window = rSys->createRenderWindow(
     "MainWindow",  // RenderTarget name
     800,           // window width, in pixels
     600,           // window height, in pixels
     false,         // windowed mode
     &params);      // supply the param values set above
```

This code will create a render window named "MainWindow" that sets up in the upper-left corner of the screen, with the title "Alternate Window Title."

At this time, Ogre has no elegant method of reinitializing a render window or render system. For example, if your user wishes to switch from Direct3D rendering to OpenGL rendering while your application is running, obviously you would have to shut down the D3D renderer and initialize the OpenGL renderer. Less obvious is that you have to do this for the render window as well. You can adjust the size of the window (height and width) and move it around the screen, but the only way to change other settings (such as FSAA or full screen) is to destroy the existing window and create a new one.

As mentioned earlier, you may have multiple render windows open and running. This is often useful for tools such as level editors that provide multiple views of your scene. This is different from multiple viewports in that viewports are contained fully within a render window, while render windows are top-level windows.

You may also embed Ogre render windows inside windows of most window and widget systems (such as Qt and wxWidgets). The `getCustomAttribute()` method on **RenderWindow** enables you to obtain the system-dependent window handle for the render window. You can also supply Ogre with the parent window you would like to use for embedding the Ogre render window. For example:

```
// hWnd is a handle to an existing Win32 window
// renderSystem points to an existing, initialized instance of D3D9RenderSystem

NameValuePairList opts;
opts["parentWindowHandle"] =  StringConverter::toString(hWnd);

// everything but "opts" is somewhat irrelevant in the context of an
// explicitly parented window
RenderWindow *window = renderSystem->createRenderWindow(
     "WindowName",
     800, 600,
     false, &opts);
```

This code will allow you to embed the Ogre render window in an existing windowing system window of your choice. Keep in mind that when you do this, the Ogre window message processing functionality is bypassed, and you have to take care of cleaning up the Ogre render window when the user clicks the Close button, etc., things that otherwise are done for you by Ogre.

Camera and SceneManager

I have refrained thus far from talking too much about these two classes, using just enough of their functionality to demonstrate other classes. However, these two classes actually are responsible for rendering your scene.

Scene Manager

I want to avoid getting too deep into the topic of the scene manager in Ogre, since it is covered in far greater detail in the next chapter. However, for purposes of completeness in this chapter (since I have mentioned it so many times), I figured that you should know at least the basics of creating a scene manager for use in your application.

Before you can use a scene manager instance, you must create one.

```
SceneManager* sceneMgr = root->createSceneManager(ST_GENERIC, "MySceneManager");
```

When Ogre loads its plug-ins, among those plug-ins can be various scene manager implementations, as discussed previously. Each of these implementations will register itself as a particular type of scene manager:

- `ST_GENERIC`: Minimal scene manager implementation, not optimized for any particular scene content or structure. Most useful for minimally complex scenes (such as the GUI phases of an application).
- `ST_INTERIOR`: Scene manager implementation optimized for rendering interior, close-quarter, potentially high-density scenes.
- `ST_EXTERIOR_CLOSE`: Scene manager implementation optimized for rendering outdoor scenes with near-to-medium visibility, such as those based on tiled single-page terrain mesh or heightfield.
- `ST_EXTERIOR_FAR`: Anachronism in Ogre, typically no longer used. Use `ST_EXTERIOR_CLOSE` or `ST_EXTERIOR_REAL_FAR` instead.
- `ST_EXTERIOR_REAL_FAR`: Scene manager implementation typically suited for paged landscape or paged scene construction. Paged landscapes often are huge, possibly entire planets.

In the preceding example we created a scene manager instance of the `ST_GENERIC` type. If we were loading a Quake 3 level using the **BSPSceneManager** plug-in (which registers as `ST_INTERIOR`), we would use the `ST_INTERIOR` scene type, and if we wanted to create a terrain heightfield-based scene using the **TerrainSceneManager** (TSM) plug-in, we would have created a scene manager of type `ST_EXTERIOR_CLOSE` (since that is what how the **TerrainSceneManager** registers itself). `ST_GENERIC` has no particular plug-in for its scene management, but if the **OctreeSceneManager** is loaded, it will take over `ST_GENERIC` duties.

Camera

A **Camera** is the same as its real-world analog: it "takes a picture" of your scene each frame, from a particular vantage point (meaning it has a position and an orientation). It is not typically a renderable object, so even if you have one camera in the field of view of another, the camera object will not render (if you're used to seeing a frustum representation of a camera in

a 3D modeling tool, you know what I mean). Cameras (like lights, as you also will find out later) can either be attached to scene nodes (and therefore be "animate-able" via an animation controller) or exist in free space (which means you get to move them around manually if you want their position and/or orientation changed each frame). As mentioned, cameras have a "field of view" with near and far clip planes. This complete geometry defines what is known as a *frustum*, which is a sort of pyramid with the point chopped off, as depicted in Figure 4-2.

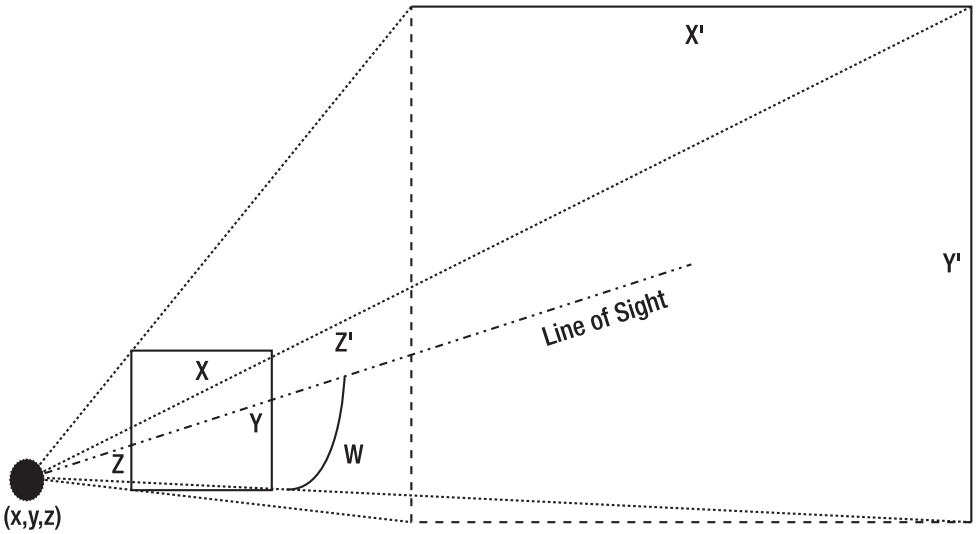

Figure 4-2. *Camera frustum*

In this figure, (x,y,z) indicates the location of the camera. X and Y are the size of the near clip plane, and are a function of the distance Z from the camera to the near clip plane. X' and Y' are the size of the far clip plane, and are a function of the distance (Z+Z') from the camera to the far clip plane. You supply the near and far clip plane distances, the aspect ratio of the camera (defined as X/Y), and the vertical angle W between the line of sight and the lower (or upper) frustum bounding plane (this is the field-of-view Y angle), and the **Camera** class calculates the horizontal angle and the size of the near and far planes.

Let's say you want to have a camera with the standard 4:3 aspect ratio, with near clip plane distance 5 units from the camera and far clip plane distance 1000 units, with a 30-degree angle between the line of sight and the lower (and upper) frustum bounding planes (W in Figure 4-2 is 30°, in other words). The following code creates a camera with these characteristics:

```
// sceneMgr is an existing instance of a SceneManager implementation. We are
// creating a camera named "MainCam" here.
Camera *camera = sceneMgr->createCamera("MainCam");

// normally you would calculate this from the size of the viewport
camera->setAspectRatio(1.33333f);

// 30 degrees will give more of a long, telescopic view
camera->setFOVy(30.0f);
```

```
camera->setNearClipDistance(5.0f);
camera->setFarClipDistance(1000.0f);
```

The view frustum defined by the camera settings defines the six clipping planes outside of which polygons are culled (discarded from the list of polygons to render for a given frame).

Rendering Modes

The **Camera** can render in one of three different modes: wireframe, solid, or "points" (only the vertices are rendered).

```
camera->setPolygonMode(PM_WIREFRAME);
camera->setPolygonMode(PM_POINTS);
camera->setPolygonMode(PM_SOLID);
PolygonMode mode = camera->getPolygonMode();
```

The mode set will continue in force until changed by a later call (in other words, this is not a one-frame-only setting). The default is `PM_SOLID`.

Position and Translation

A **Camera** (**Frustum**) is a **MovableObject** as well, and shares all of the methods and functionality of that class. The most common feature of a **MovableObject** is the ability to attach to a scene node and "piggyback" the camera along with renderable objects in the scene. There are reasons you might want to do this, such as various "chase" camera techniques. You will see different third-person camera techniques in later chapters; for now, we will discuss the inherent position and orientation features of the **Camera**.

```
// assume camera is a pointer to an existing, valid instance of "Camera"
camera->setPosition(200, 10, 200);

// you can also use a Vector3 to position the camera, useful for using the
// position obtained from a scene node getter
// camera->setPosition(Vector3(200, 10, 200));
```

This code will position the camera at absolute world coordinates (200,10,200). This is different from the `move()` and `moveRelative()` methods, which translate the camera from its current position in world and local space, respectively.

```
// camera is at world space 200, 10, 200 from before
camera->move(10, 0, 0); // camera moves to 210, 10, 200
camera->moveRelative(0, 0, 10); // camera moves to 210, 10, 210
```

We have to be careful with `moveRelative()`. Since it is carried out in local space, the translation applied is relative to the camera's current orientation. In the preceding examples, we assumed that the camera was originally aligned with the major axes, pointing in the normal +Z direction. If the camera had been rotated 90 degrees to the right, for example, the `moveRelative(0,0,10)` call would have moved the camera to (220,10,200).

Direction, Orientation, and "Look-At"

Ogre provides a plethora of methods to point your camera around the scene:

```
void setDirection(Real x, Real y, Real z);
void setDirection(const Vector3& vec);
Vector3 getDirection(void) const;
Vector3 getUp(void) const;
Vector3 getRight(void) const;
void lookAt( const Vector3& targetPoint );
void lookAt(Real x, Real y, Real z);
void roll(const Radian& angle);
void roll(Real degrees) { roll ( Angle(degrees) ); }
void yaw(const Radian& angle);
void yaw(Real degrees) { yaw ( Angle(degrees) ); }
void pitch(const Radian& angle);
void pitch(Real degrees) { pitch ( Angle(degrees) ); }
void rotate(const Vector3& axis, const Radian& angle);
void rotate(const Vector3& axis, Real degrees) {
rotate ( axis, Angle(degrees) ); }
void rotate(const Quaternion& q);
void setFixedYawAxis( bool useFixed, const Vector3& fixedAxis =
const Quaternion& getOrientation(void) const;
void setOrientation(const Quaternion& q);
void setAutoTracking(bool enabled, SceneNode* target = 0,
const Vector3& offset = Vector3::ZERO);
```

Most of these are self-explanatory. `roll()`, `yaw()`, and `pitch()` do precisely what they say: they rotate the camera around its local Z axis (roll), Y axis (yaw), or X axis (pitch) by the radial displacement specified, relative to its current orientation. `setDirection()` will orient the camera along the vector specified, again in local space. `rotate()` will cause the camera to rotate by the angular displacement around the given axis, as specified in the angle-axis versions, or by the quaternion in the quaternion version. `lookAt()` is a very commonly used method to orient a camera on the basis of a target point or object in world space, without having to try to do the Euclidean math to figure out the quaternion needed to get there. And finally, `setFixedYawAxis()` is provided to allow you to break the camera free from its own yaw (Y) axis. In first-person shooters, the camera often can look up as well as scan the X-Z plane; in this case, you want the default behavior, which is to yaw around the camera's local Y axis. However, in the case of a flight simulator, you want to be able to break the camera free of that restriction in order to implement a fully free camera.

`setAutoTracking()` is an interesting feature, and a useful one if you wish to have the camera always follow a certain node in the scene. Note that this is not the same as a true third-person chase camera, since that type of camera typically is not looking at a particular node, but instead is looking at whatever your character is looking at. The first parameter indicates whether to turn tracking on or off; this can be done at any time before a frame is rendered, and **must** be done (tracking turned off) prior to deleting the node that is being tracked (otherwise Ogre will throw an exception). The node to be tracked is the second parameter, and that node must exist before this method is called. The parameter can be `NULL` only if the first parameter is `false`. If the object being tracked is large and sighting on the center of it is not desirable, you can fine-tune

the actual look-at point with the third (offset) parameter, which operates in local space relative to the scene node being tracked.

The following methods are available to obtain information about the camera's actual orientation, taking into account rotations and translations of attached scene nodes (and also reflection matrices in the "derived" set of methods):

```
const Quaternion& getDerivedOrientation(void) const;
const Vector3& getDerivedPosition(void) const;
Vector3 getDerivedDirection(void) const;
Vector3 getDerivedUp(void) const;
Vector3 getDerivedRight(void) const;
const Quaternion& getRealOrientation(void) const;
const Vector3& getRealPosition(void) const;
Vector3 getRealDirection(void) const;
Vector3 getRealUp(void) const;
Vector3 getRealRight(void) const;
```

The "Real" set of methods returns values in world space, while the "Derived" set returns values in "axis-aligned" local space (which means the camera is at the origin in its local space, and its local axes are aligned with the world-space axes).

Advanced Camera Features

Ogre supports stereoscopic rendering via the `setFrustumOffset()` and `setFocalLength()` methods of **Camera (Frustum)**. For example, you can adjust a second camera (frustum) horizontally to simulate the distance between the viewer's eyes. This will render from the adjusted camera at a slightly different angle, and will produce the eye-crossing output (at least, without the special red/blue glasses that blocked the opposite image) common in the "3D" movies that were the craze in the 1950s. Of course, this technique is highly specialized and "niche" but is provided in case someone needs to do such a thing (usually in research laboratories).

Ogre also allows you to manipulate the view and projection matrices directly. This is definitely a more advanced topic, and falls under the heading of "do this only if you know exactly what you are doing, and why," since these matrices are already calculated for you by your setup of the camera and the position and orientation of the camera. The following methods are available for view/projection matrix manipulation:

```
const Matrix4& getProjectionMatrixRS(void) const;
const Matrix4& getProjectionMatrixWithRSDepth(void) const;
const Matrix4& getProjectionMatrix(void) const;
const Matrix4& getViewMatrix(void) const;
void setCustomViewMatrix(bool enable,
     const Matrix4& viewMatrix = Matrix4::IDENTITY);
bool isCustomViewMatrixEnabled(void) const;
void setCustomProjectionMatrix(bool enable,
     const Matrix4& projectionMatrix = Matrix4::IDENTITY);
bool isCustomProjectionMatrixEnabled(void) const;
```

The first two listed return the render system–specific projection matrix. `getProjectionMatrixRS()` will return the matrix in the render system's native coordinate system (right-handed or left-handed), while `getProjectionMatrixWithRSDepth()` will return the

matrix in Ogre's native format (right-handed). The difference is that the depth values will range from [0,1] or [–1,1] depending on the render system in use. You can avoid that and always get a depth range from [–1,1] by calling `getProjectionMatrix()` instead, which is essentially `getProjectionMatrixWithRSDepth()` with the consistent depth range.

When you set a custom view and/or projection matrix, you must be aware that Ogre will no longer update the matrix based on translation or orientation of the frustum (camera). You have to update the matrix manually every time you move its origin. You can turn the custom matrix on and off with the `enable` parameter in `setCustomViewMatrix()` and `setCustomProjectionMatrix()`. When the custom matrix is turned off, Ogre will resume updating the internal matrix for you as normal.

For the most part, Ogre handles LoD (level-of-detail) bias pretty well. However, there are times you may want to override its default factors, and Ogre provides **Camera** methods to do so:

```
void setLodBias(Real factor = 1.0);
Real getLodBias(void) const;
```

`setLodBias()` is not actually an immutable command; elements in the scene are free to ignore this directive, making it more of a hint if LoD overrides are widespread in the scene or application. The `factor` parameter instructs the camera to increase (or decrease) the amount of detail it renders. Higher values (>1.0) increase the detail, lower values (<1.0) decrease it. This is useful for implementing items such as rear-view cameras that do not need the full detail of the main viewport.

World Space to Screen Space

One common need, especially for mouse-picking applications, is translation of world space to screen space. The *centerline* of the camera (that is, the line in the world down which the camera is looking) intersects the screen at a particular point, namely [0.5,0.5] (in normalized coordinates). If you are moving a cursor around the screen, you probably want to determine a line from the camera origin through the screen at the location where the cursor. This line is called a *ray*, and it is returned from the `getCameraToViewportRay()` method:

```
// x and y are in "normalized" (0.0 to 1.0) screen coordinates
Ray getCameraToViewportRay(Real x, Real y) const;
```

With this ray, you can proceed to perform a query on the scene manager to determine what objects might intersect that ray; scene manager details are discussed later in this book.

Viewport

Referring again to Figure 4-2, consider the rectangle defined by X and Y. This rectangle is the viewport for that camera . . . or, more correctly, "a" viewport for that camera. If you recall from the example code earlier in this chapter, a **Viewport** instance is obtained from an instance of **RenderWindow**, and the method that retrieves the viewport takes an instance of **Camera** as the lone parameter. This means that a single **Camera** instance can drive zero or more **Viewport** objects.

Viewports can overlap without ill effect. By default, Ogre clears the depth and color buffers before rendering viewports that are "on top" of other viewports, to avoid depth-blending issues. You can have as many cameras and viewports as your application needs (and memory allows). One common use for multiple cameras and viewports is "picture-in-picture" zoom windows in a game.

■ **Tip** Something to keep in mind when considering viewports and camera zoom is that it is not enough simply to narrow the field of view on the camera to create a zoomed image. This is because Ogre will still be using the camera position to calculate level of detail for the zoom, and what you will find is that the objects in your zoom window will look terrible, since they will still be at the correct level of detail for their distance from the camera. Solution: when you zoom, move the camera closer to the zoom target, or use a secondary camera (which is required for picture-in-picture zooms), or use `Camera::setLodBias()` to increase the level of detail rendered.

Viewports have a *z-order*, which determines what viewports will render "on top" of other viewports in use. Higher ordinal indicates "higher" placement in the stack (that is, a z-order of zero would indicate the viewport underneath all others). Only one viewport can occupy a given z-order for a render window. For example, you cannot have two viewports at z-order zero; this will cause an exception.

Each viewport can have an independent background color. For example, you can have a main viewport at z-order 1 that covers the entire window, with a black background, and a smaller viewport on top of that one (at z-order zero, in other words) with, say, a blue background, as in the following code:

```
// assume window is a valid pointer to an existing render window, and camera is
// a valid pointer to an existing camera instance
Viewport *vpTop, *vpBottom;

// second parameter is z-order, remaining params are position and size, respectively
vpBottom = window->addViewport(camera, 0);

// create a smaller viewport on top, in the center, 25% of main vp size
vpTop = window->addViewport(camera, 1,
     0.375f, 0.375f,
     0.25, 0.25);
// set the background of the top window to blue (the default is black so we don't
// need to set the bottom window explicitly)
vpTop->setBackgroundColour(ColourValue(0.0f, 0.0f, 1.0f));

// an alternate way to set the color is to use the manifest constant for blue
// vpTop->setBackgroundColour(ColourValue::Blue);
```

I mentioned earlier that Ogre defaults to clearing the depth and color buffers each frame; you can manage both of these settings independently, using the `setClearEveryFrame()` method on **Viewport**.

```
// As mentioned, these both default to "true". The flags are maskable; in other
// words, setClearEveryFrame(true, FBT_COLOUR|FBT_DEPTH) is valid.
vpTop->setClearEveryFrame(true, FBT_COLOUR);
vpTop->setClearEveryFrame(false);
```

Another important consideration when using picture-in-picture style viewports is the fact that overlays are rendered by default in all viewports. This is not something you would want in a zoom window (you do not want your HUD repeated in miniature in the zoom window), so you can turn off overlay rendering on a per-viewport basis. Likewise for skies (skyboxes) and shadows; these are renderable on a per-viewport basis.

```
vpTop->setOverlaysEnabled(false);
vpTop->setSkiesEnabled(false);
vpTop->setShadowsEnabled(true);
```

You can do more advanced things with viewports, such as altering the render queue sequence, and using a per-viewport material scheme, but we will cover those topics in more detail in later chapters devoted to these more advanced topics.

Main Rendering Loop

The typical Ogre application will render one frame after another, ceaselessly (at least until you tell it to stop). We saw earlier in this chapter one method of invoking this render loop: the `Root::startRendering()` method. However, this method simply starts a small loop that calls another method: `renderOneFrame()`.

The existence of `renderOneFrame()` is important for several reasons. For one, you may wish to incorporate Ogre into an existing application or framework, and if you had to rework your application's main rendering loop to accommodate Ogre's `startRendering()`/**FrameListener** *Observer* duolith, you might simply give up and choose another, less capable 3D rendering API . . . and we can't have that.

Another reason you might use `renderOneFrame()` over `startRendering()` is that it is literally otherwise impossible to integrate Ogre into a windowing system's message loop. Let's take the Windows `WM_PAINT` message, for example. When a Windows application processes this message, it goes about the task of redrawing the contents of the window's client area (at least the part that has been invalidated by whatever action another window performed, such as covering part of the current window). If you have embedded an Ogre rendering window inside the window processing the `WM_PAINT` message, you want to update the Ogre render window (by calling `renderOneFrame()`) only when the message is processed, not all the time as would be the case for `startRendering()`.

However, the primary reason for choosing a manual render loop and `renderOneFrame()` over `startRendering()` and a frame listener is that the design and architecture of many 3D game engines or applications are such that driving the application's main loop from the rendering engine turns out to be a poor choice. For example, a networked game engine design might run mostly the same loop on a dedicated server as it does on the full client, but without any rendering support. If the engine design has Ogre driving that server's main loop, and Ogre is not present on the server, then the problem is self-evident.

Luckily, Ogre does not force one method or the other on your application: you are free to choose the method that best suits your situation. For those that prefer the manual render loop, `renderOneFrame()` is for you.

Listing 4-8 is deliberately minimal; it is intended only to show you how you might implement the manual render loop in your application, and where it typically exists in the flow of your code.

Listing 4-8. *Skeleton Example of a Manual Main Rendering Loop in Action*

```
bool keepRendering = true;

// Do all of the Ogre setup we've covered so far in this chapter: loading plug-ins,
// creating render window and scene manager and camera and viewport, and putting
// some stuff in our scene.

while (keepRendering)
{
   // process some network events into engine messages

   // process some input events into engine messages

   // update scene graph (manager) based on new messages

   // render the next frame based on the new scene manager state
   root->renderOneFrame();

   // check to see if we should stop rendering
   // Note: NextMessageInQueue() is completely fictional and used here only
   // for purposes of illustration -- it does not exist in Ogre.

   if (NextMessageInQueue() == QUIT)
   {
      keepRendering = false;
   }
}

// Do whatever cleanup your application needs
// Then, shut down Ogre
delete root;
```

It is worth noting that you can still use **FrameListener** classes with `renderOneFrame()`. `renderOneFrame()` is actually the method that notifies any registered frame listeners in **Root**, as Listing 4-9 demonstrates. (Not much to it, is there?)

Listing 4-9. *The Full Implementation of the* `renderOneFrame()` *Method of* ***Root*** *(Found in* `Root.cpp` *in the Ogre Source)*

```
bool Root::renderOneFrame(void)
{
    if(!_fireFrameStarted())
        return false;

    _updateAllRenderTargets();

    return _fireFrameEnded();
}
```

Listing 4-9 contains the entire body of the `renderOneFrame()` method. As you can see, it fires the frame-started and frame-ended events in your **FrameListener** implementation; clearly **FrameListener** and `renderOneFrame()` are perfectly compatible if you prefer to use a **FrameListener** with a manual render loop.

Conclusion

In this chapter, you learned how to initialize and start up an Ogre application, and how the various visualization classes (**RenderWindow**, **Camera**, **Viewport**) work together to present your scene to the viewer. In the next chapter, we will begin putting actual content into the scene, which should make the techniques you learned in this chapter a lot more interesting.