CHAPTER 3

Ogre Design Overview

A quick glance at the list of classes and methods provided by Ogre can quickly make your eyes cross. Luckily, you do not have to deal with Ogre on that basis. Ogre is an object-oriented class library, and its sophisticated hierarchical design allows you to deal with it on as simple or involved a basis as you need. It is possible to create a running Ogre-based application in a dozen lines of code or less, but you don't get to see much along the way, and you are restricted by several assumptions made on your behalf.

Design Philosophy

Ogre provides an object-oriented method of access to what inherently is procedural data processing: rendering simple geometric primitives to a render target (usually a screen buffer displayed on a CRT or LCD device). Traditionally, when using OpenGL or Direct3D to render your scenes and objects, you would follow a series of steps—procedural processing flow, in other words: set up render state with various API calls, send geometry information with various API calls, and tell the API and GPU to render your geometry with another API call. Lather, rinse, repeat until a frame is fully rendered, then start it all over for the next frame.

With an object-oriented approach to rendering geometry, the need to deal with geometry is removed entirely, and you can instead deal with your scene in terms of the objects that make up the scene: movable objects in the scene, static objects that make up the world geometry, lights, cameras, and so on. No 3D API calls needed; just place the objects in the scene, and Ogre takes care of the messy details. Furthermore, you get to manipulate the objects in your scene using far more intuitive methods than managing transformation matrices: it is simpler to instruct an object to rotate and translate in terms of degrees (or radians) and world units (with local-, world-, or parent-space qualifiers) than it is to try to work up the proper transformation matrix that makes all the rotations and translations happen. In short, you can deal with objects, their properties, and intuitive methods of manipulation instead of trying to manage them in terms of vertex lists and triangle lists and rotation matrices and so on.

Ogre provides an object-oriented framework that involves all parts of the rendering process in the object model. Render systems abstract the complexities of the underlying 3D APIs (OpenGL and Direct3D, for example) into a common interface to their functionality; scene graph functionality is abstracted into another interface that allows simple plug-and-play usage of different scene graph management algorithms; all renderable objects in a scene, whether movable or static, are abstracted by a common interface that encapsulates the actual rendering operations such as techniques and their contained passes; movable objects are represented in the scene by a common interface that allows robust methods of manipulation.

Design Highlights

For the experienced developer, the architecture of Ogre might be self-evident. For those new to object-oriented design, or new to software engineering in general, design decisions in Ogre might make a bit less sense. Let's go over some of the design features at a high level.

Intelligent Use of Common Design Patterns

Ogre makes good use of many useful and common design patterns. *Design pattern* simply refers to a common and well-tested solution for a particular type of software problem, and the name "design pattern" more or less was immortalized in the popular "Gang of Four" book, *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma, Helm, Johnson, and Vlissides (Addison-Wesley, 1995).

Design patterns in Ogre are employed to enhance the usability and flexibility of the library. For example, Ogre is rather eager to inform the application of everything it does via the *Observer* pattern, in which client code registers to receive notifications of events or state changes within various parts of Ogre (such as the ubiquitous *FrameListener* in the Ogre demo applications, which is how the application is notified of frame-started and frame-ended events). The *Singleton* pattern is used to enforce the notion of a "single" instance of a class, and the *Iterator* pattern is used to walk the contents of a data structure. The *Visitor* pattern is used to enable operations to be performed on an object, without having to alter the object (for instance, all nodes in a scene graph). The *Façade* pattern is used to consolidate access to commonly used operations, implemented in many different subsystems, within a single class interface. And finally, the *Factory* (and cousin, *Abstract Factory*) are widely used for creation of concrete instances of abstract interfaces.

Scene Graph Decoupled from Scene Contents

The decision to decouple the scene graph from the scene contents was probably one of the most brilliant, yet underappreciated, design features in the entire Ogre project. This is such a simple design to understand, yet one of the hardest to comprehend for those used to more "traditional" scene graph designs.

Traditional designs (as used in many commercial and open source 3D engines) typically couple the scene contents and the scene graph in an inheritance hierarchy that forces the subclassing of content classes as types of scene nodes. This turns out to be an incredibly poor design decision in the long run, as it makes it virtually impossible to change graph algorithms later, without forcing a lot of code changes at the leaf-node level if the base node interfaces change at all (and they usually do). Furthermore, this "all nodes derive from a common node type" design is, in the long run, inherently inflexible and nonextensible (at least from a maintenance standpoint): functionality invariably is forced up the inheritance hierarchy to the root nodes, and myriad subclasses are required, and typically end up as minor adjustments to base functionality. This is, at the very least, a poor object-oriented design practice, and those who adopt this design philosophy almost always end up wishing they had done it a different way in the beginning.

Ogre did. First of all, Ogre operates on its scene graph(s) at an interface level; Ogre makes no assumption as to what sort of graph algorithm is implemented. Instead, Ogre operates on the scene graph only through its signature (its methods, in other words) and is completely ignorant of the underlying graph algorithm implementation. Second, Ogre's scene graph interface is concerned only with the graph structure. Nodes do not contain any inherent content access

or management functionality. Instead, Ogre pushes that down into what it calls *renderable*, from which all bits of geometry in your scene (movable or otherwise) are derived. The rendering properties (also known as *materials*) for these renderables are contained in **Entity** objects, which in turn contain one or more **SubEntity** objects. These subentities are the actual renderable objects. See Figure 3-1 for a visual description of the relationship between the scene graph structure and contents. Note that even the scene nodes are attached to the scene graph; the scene graph does not manipulate the nodes' state directly.

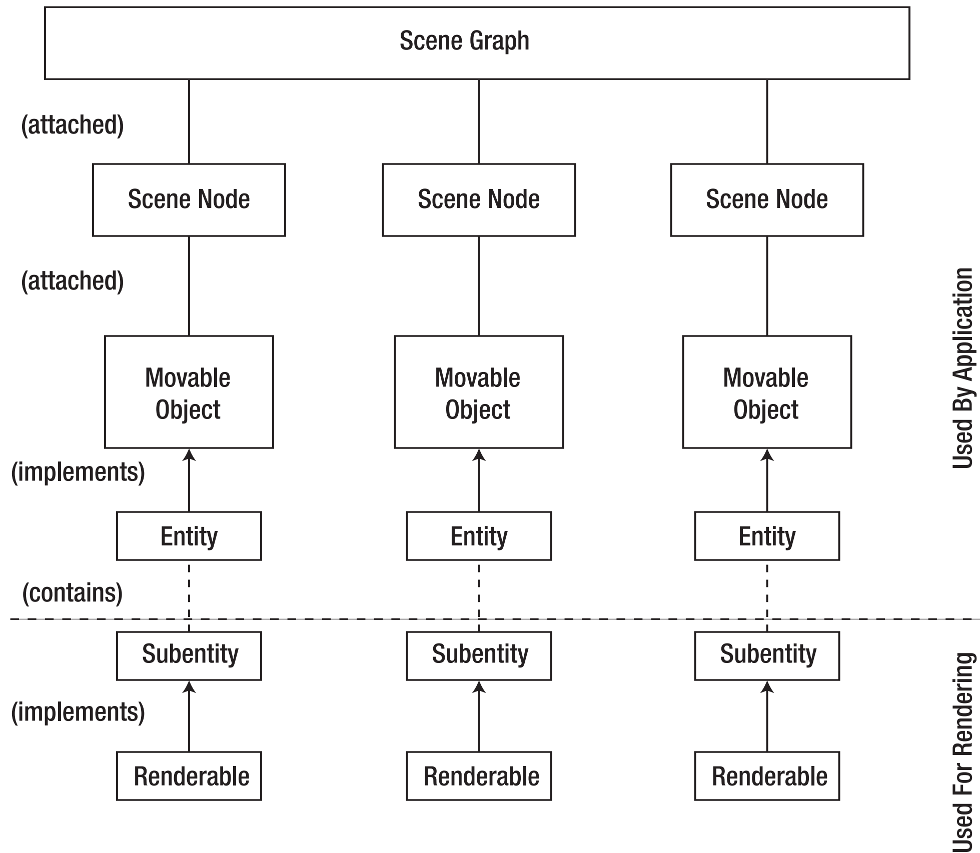

Figure 3-1. *Relationship between the scene graph structure and content management objects in Ogre*

All of this geometry and these rendering properties are made available to the scene graph via `MovableObject`. Instead of subclassing the movable object from a scene node, it is attached to the scene node. This means that scene nodes can exist without anything renderable actually attached to them, if your application has a need for that. It also means that extending, changing, refactoring, or otherwise altering the scene graph implementation has no impact on the design and implementation of the implementation and interface of the content objects; they are entirely independent of the scene graph. The scene graph interface can even change completely and the content classes would not be affected in the least.

The reverse is also true: the scene graph has no need to know about any changes to your content classes, so long as they implement a simple interface that the scene graph **does** know about. Ogre even allows you to attach arbitrary "user-defined" content to scene nodes, so if you want to carry around, say, audio cue information in your scene graph, you can do that as well. You do not need to subclass anything, you simply need to implement a very simple interface on your custom data object in order to attach it to the scene graph nodes.

This decoupling has turned out to be one of the best, yet sometimes most often misunderstood, design decisions in the history of the library.

Plug-In Architecture

Ogre is designed to be extensible. Contrary to many other rendering API designs, Ogre does not force any particular implementations on the user. Ogre accomplishes this through a *contract-based design*, which is a fancy way of saying that Ogre is designed as a set of cooperating components that communicate with each other through a known interface.

This allows incredible freedom in creating new or different implementations of various bits of functionality. For example, Ogre itself deals with its scene graph at an interface level, which means that the user is not limited to one or two choices of scene graph algorithm, choices made by the Ogre developers. Instead, scene graph implementations can be "plugged into" the Ogre library as needed, as discussed in the previous section. If a kd-tree implementation is required for a particular application, then it is simply a matter of creating a kd-tree scene graph that conforms to the interface defined by Ogre and making that scene graph plug-in available to Ogre (and therefore your application).

The same is true for all pluggable functionality: file archives and render systems are the most common forms of plug-in, but alternate functionalities such as the Ogre Particle system are also implemented as plug-ins.

One of the most attractive aspects of plug-ins is the fact that they do not require rebuilding of the Ogre library in order to be incorporated. Ogre provides a simple means of loading plug-in libraries at runtime and initializing them in order to expose their contained classes and functionality. Pluggable functionality supports a registration mechanism that allows an entirely code-free plug-in incorporation process. Each pluggable mechanism defines its own particular syntax or protocol for loading plug-ins at runtime, but typically it is simply a matter of telling Ogre "this is what I am called" or "this is what sort of resource I am here to handle" and providing a reference or pointer to the main class within the plug-in.

Hardware-Accelerated Renderer Support

Ogre is designed, on purpose, to support only hardware-accelerated graphics rendering. This means that Ogre requires a graphics coprocessor (such as those made by NVIDIA and ATI); direct software rendering is not an option. This design decision allows Ogre the freedom to work, in an optimized fashion, with *hardware buffers*, which are areas of memory shared between the graphics hardware and the application (Ogre).

This decision has a great impact on Ogre's capabilities. Since it is a hardware-based rendering API, it can take full advantage of all hardware acceleration capabilities, including programmable shaders. The integration between the programmable graphics pipeline and Ogre puts Ogre on the same level of capability as most commercial 3D rendering engines: since much of the "fancy" graphics processing in modern 3D applications and games is done via the programmable GPU pipeline, anything that, say, the Unreal Engine or CryENGINE can

do, Ogre can do. The bits of additional functionality not present in Ogre (for example, direct engine support for advanced global illumination solutions such as Ambient Occlusion or Precomputed Radiance Transfer) still have to be done by the application. However, since computation of many advanced algorithms is still done “offline” at this time (not in real time, in other words), this is hardly a limiting factor.

Currently, Ogre offers two choices for render system support: Direct3D 9 and OpenGL. Given that there are no other hardware acceleration APIs of any consequence (on the platforms currently supported by Ogre), it is likely that for the foreseeable future, Direct3D and OpenGL will remain the only two render system options supported within Ogre.

Flexible Render Queue Architecture

Ogre’s design takes a somewhat novel approach to the problem of ordering the rendering of various parts of a scene. The standard process (at a coarse, high level) typically works as follows: render terrain and/or world geometry, render movable objects, render effects, render overlays, then render backgrounds and/or skyboxes. However, as typically implemented (meaning, as a monolithic procedural block), this process is difficult to change if needed. For example, your application might need to render static world geometry in multiple “layers,” interleaved with 3D scene objects, perhaps in a “fighting” game like Mortal Kombat or Street Fighter. Or perhaps you need to render various bits of geometry “out of order,” so to speak, to create certain effects (such as with real-time shadowing algorithms). It is difficult in many cases to alter the order of rendering, or to effect “conditional rendering” directly in the main loop; the result is often a ton of hard-to-maintain special-case code and an inflexible design.

Ogre overcomes this inflexibility with the use of *render queues*. The concept is not hard to grasp: Ogre will render the contents of several ordered queues, one at a time, and will render the queues in order as well.

Figure 3-2 visually describes the render queue organization in Ogre. Queues themselves have an order, or *priority*, and objects within a queue have their own priority as well. For example, Ogre will render the set of queues in Figure 3-2 from back to front (from lower to higher priority, or order). Within each queue, Ogre will render in order as well. For example, in the Overlays queue in Figure 3-2, Ogre will render the HUD objects, and then the reticle objects, and then the UI menu objects, in that order.

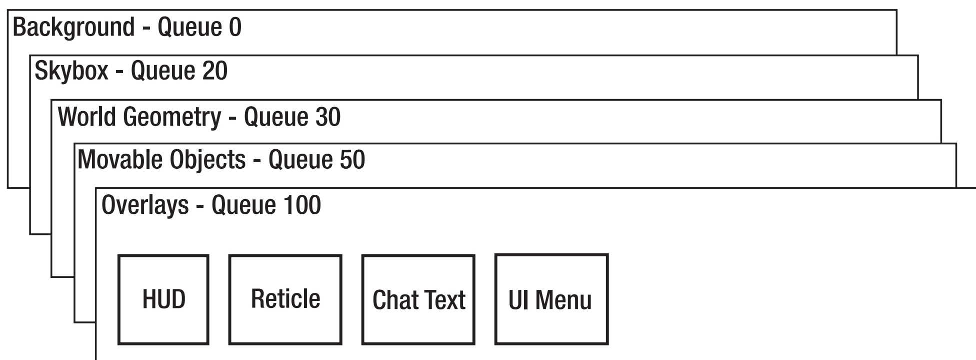

Figure 3-2. *Conceptual render queue organization in Ogre*

The flexibility of this design is in the fact that reorganizing rendering order is as simple as reassigning queue priorities. Render queues can be created at custom priorities, and the objects within any queue can be ordered at will as well. Entire queues can be turned on and off, and objects within a queue likewise can be turned on and off.

Finally, each queue provides notifications of events within the queue (such as prerender and postrender), so that the application has the opportunity to alter the rendering of the objects within the queue if needed. In terms of code development and maintenance, this is invaluable for the encapsulation of render queue management into small, easily understood chunks of code, as opposed to trying to figure out what bits of a huge monolithic procedural block are responsible for rendering which bits of the scene.

In other words, Ogre's render queue design provides an elegant object-oriented solution to what, in many complex applications, typically fast becomes an intractable and unmanageable problem.

Robust Material System

It is possible to create and render objects in a scene without ever touching a single line of code (beyond the obvious work involved in actually loading the objects in the first place). Ogre's material scripting system is one of the most flexible and powerful available in its class of software.

Ogre materials are composed of one or more *techniques*, which are simply collections of *passes*. *Pass* refers to a rendering pass, and is the unit of rendering at the material level within Ogre. In other words, a single pass on an object will result in exactly one *draw call* to the graphics hardware for the geometry being rendered. You can have as many passes in a technique as you like, but understand that in most cases, each pass will cause a completely new rendering operation (complete with full render state changes on the hardware for each pass). This has what should be obvious implications on performance, but in some cases there simply is no other way to create a particular rendering effect.

The most impressive feature of Ogre's material system is its *automatic fallback* design; Ogre can automatically apply the "best" technique available in a material, and will search "downwards" through the list of techniques until it finds one that is compatible with the graphics hardware being used. Ogre will also do its best to reorganize passes within a technique if the hardware cannot support even the least technically demanding technique in a material. For example, if a particular set of graphics hardware supports only a single texture unit in its fixed-function pipeline, and your least-complex technique requires a minimum of two texture units, then Ogre will break up the pass into two separate passes and blend the two renderings to achieve the same effect.

Ogre materials also support the notion of *schemes*. A material scheme can best be understood as support for the common "Ultra High, High, Medium, Low" graphics settings. In this case, you would define four schemes and assign material techniques to each as you see fit (each technique obviously would be developed to fit the particular scheme). Then you can limit Ogre's technique fallback search to stay within the techniques that belong to a particular scheme, making material management for your application that much easier.

You are not limited only to scripting for material management. All classes and methods that Ogre uses to create a material from the script are fully available to the application; you can create a material completely in code, procedurally, and in fact this is commonly done. The same material scheme feature and technique fallback processing are just as available to procedurally created materials as they are to scripted materials. Of course, with material scripting, no code changes are required (and in fact, material creation can be placed entirely in the

hands of your artists, since material scripts typically are exported from 3D modeling packages along with mesh and animation data).

Native Optimized Geometry and Skeleton Format

Ogre utilizes a single format for its mesh and skeleton data. As a result, it does not have the ability to load third-party mesh formats, such as those used for character data in commercial games. Community-developed converters may exist for such items, but they are not part of the Ogre library.

Ogre uses this format to allow for fast, efficient loading of its mesh and skeleton data. This efficiency is enabled by the ability for Ogre to preoptimize the layout of the binary mesh and skeleton files in an exporter or offline tool (the command-line *OgreXMLConverter* tool, discussed in Appendix A). Of course, the classes used in the *OgreXMLConverter* tool are available for use in your application if you wish to employ them (for example, if you wish to export binary mesh data files directly from a 3D modeling package). One method of creating binary mesh and skeleton files is first to export your scene or character data from your 3D tool into an intermediate, human-readable XML format (Ogre XML), and then convert this data to binary format with the command-line tool. Exporters exist for most current modeling tools (both commercial and open source), such as Softimage|XSI, Autodesk 3D Studio Max and Maya, and Blender (as well as many others).

Along similar lines, the notion of loading raw XML at runtime is a performance nightmare. XML is an incredible format for exchanging data between disparate systems (which is exactly how it is used in Ogre: exchanging data between an arbitrary 3D modeling tool and the Ogre binary mesh and skeleton serializer), but it is a horrible format for any sort of performance-oriented application . . . which describes precisely the requirements of runtime asset loading. An additional bonus of the intermediate XML format is the ability to inspect or change the exported data. The inspection ability makes it much simpler to debug an exporter, as well as verify the structure and composition of an exported scene or object. Plus, you can easily insert additional tools into your asset pipeline if you wish; it often is easier to deal with the textual XML format for minor systematic tweaking than it is to work with a serialized binary file.

The optimization of the binary format is primarily in the ordering of vertex, geometry, and skeleton data, but the offline process also has other features, such as available automatic LoD (level of detail) and object tangent generation for the meshes. Performing these processes offline removes the need to perform them at runtime, enabling reduced load times.

Multiple Types of Animation

Ogre supports three types of animation: skeletal, morph, and pose.

Skeletal animation refers to the binding of vertices to bones in a skeleton (also known as *palette matrix skinning*, or just *skinning* for short). Each vertex in an object can have up to four independent bone influences. Each influence is assigned a weight along with its bone, so that when that bone moves, its influence on the position of the vertex is weighted by that amount. This is useful for realistic deformation of vertices, approximating, say, the effect that moving your arm might have on the shape of your shoulders (that is, how the muscles bunch up over your shoulder socket when you raise your arm). Skeletal animation is performed in keyframed *forward kinematic* mode only; Ogre does not support inverse kinematics (IK) at this time; if you modeled your animation using IK in a 3D tool, you must sample the positions of the bones

at arbitrary intervals (a process known as *keyframing*). Typically, the Ogre exporter for your modeling tool does this for you.

Morph animation is a vertex animation technique that stores absolute vertex locations each keyframe, and interpolates between those positions at runtime. It differs from *pose animation* in that pose animation stores vertex offsets instead of absolute positions, and therefore multiple pose tracks in an animation can be blended to create complex vertex-based animations. Morph animation is far more limited than pose animation, as it cannot be blended with other morph animations due to the use of absolute vertex positions. Both types can be blended with skeletal animation.

All animation types can be performed in software or on the GPU hardware using a vertex program. For straight skeletal animation, the positions of the bones are passed to the vertex program in a separate block of program constants, along with the positions of the vertices and the blend weights and indices. Morph animations do not have overbearing data requirements when performed on the hardware; only a second vertex buffer is required to be passed to the vertex shader. For pose animations, the amount of data passed can be considerable, especially since each additional pose requires an additional vertex buffer be passed to the shader.

For the same reason that morph animations cannot be blended together, morph animation cannot be blended with pose animation, and vice versa. Both types of vertex animation can be blended with skeletal animation, however.

Ogre's animation system operates on the principle of *controllers*; that is, objects that manage a changing value as a function of another value (in the case of animations, that "other" value is time). As mentioned, Ogre's animation system is keyframed; it will interpolate between keys in an animation track on two selectable bases: linear or cubic spline. You should match the type of interpolation used in your application to the type used in the modeling/animation package, or compensate by using a higher sampling frequency in your exporter.

Compositor Postprocessing

A relatively new addition to the Ogre feature package is the *Compositor framework*, which allows the user the ability to create sophisticated two-dimensional, full-screen postprocessing effects on a viewport. For example, a viewport can be enhanced with a full-screen glow or bloom effect, or the viewport can be postprocessed into a black-and-white or sepia-toned rendering, or the viewport can be transformed into a line-art drawing with hard edges, and so on. Anything you can think of to do to a viewport can be done with the Compositor framework.

The framework operates on much the same principles as the material scripting system. Compositor *techniques* are, like with materials, different ways of achieving a particular effect. Compositor *passes* are similar to material passes in that multiple calculations and/or refinements can be done to a viewport before the final output is created. And like material fallbacks, the Compositor framework provides fallback handling for cases where a desired output pixel format is not available.

The easiest way to think of the Compositor framework in Ogre is as an extension of the fragment program (pixel shader) pipeline. In fact, the Compositor framework utilizes the fragment processing features of the graphics hardware to perform its processing; Compositor passes are defined in terms of fragment programs defined in material scripts. The difference is that while the conventional graphics pipeline only allows one fragment program per material pass, the Compositor framework will "ping-pong" pixel buffers back and forth as many times as needed to perform all of the passes required of the particular Compositor script. Granted, you could do this processing yourself and handle the management of the multiple pixel buffers needed

to accomplish complex postprocessing effects, but with the introduction of the Compositor framework, there is no need.

Compositor scripts operate on viewports. As a result, they can target any render target, whether render textures or the main or secondary render windows. The final result is always drawn into a full-viewport quad overlay, whether or not geometry is rendered underneath the quad. As a result, you find yourself often rendering your geometry to offscreen buffers and displaying it in what essentially is a rendered texture applied to a quad.

As with the material system, you are not limited to scripting for Compositor effects: you can certainly create the effects entirely in code, using the same classes and methods that the Compositor parser uses. Also like materials, the Compositor supports material schemes the same way that materials do directly; in fact, schemes in the Compositor framework refer to material schemes.

Extensible Resource Management

Resources in Ogre are defined as “anything that is needed to render geometry to a render target.” This obviously includes meshes, skeletons, and materials, but also includes overlay scripts and fonts, as well as ancillary material items such as Compositor scripts, GPU programs, and textures.

All of these types of resources have their own manager in Ogre. This manager is responsible primarily for controlling the amount of memory that a particular type of resource occupies in memory; that is, the resource manager controls a resource instance’s lifetime. Actually, the resource manager controls this lifetime only to a point: first, it can only store as many instances of a particular resource type as there is memory allocated to that resource type (defined when the resource manager for that type starts up); second, Ogre will never remove from memory resources that are actively referenced by part of your application.

Resources themselves are actually responsible for loading themselves. This is to support a design feature of the resource system: manual resource loading. “Manual” refers to the fact that a resource is loaded, procedurally or otherwise, as a result of a method call on a class interface rather than an implicit load from the file system. Fonts and meshes are examples of manually loaded resources, as they typically require a bit of extra processing during load and initialization (compared to, say, a texture file that is already in its needed form when loaded from the file system).

A resource in Ogre can exist in one of four states at any given time. It can be *undefined* (which means Ogre knows nothing about it); it can be *declared*, which means the resource has been indexed in its archive, but that’s about it; it can be *unloaded*, which means that the resource has been initialized (if it is a script, then the script was parsed) and a reference to it was created; or it can be *loaded*, which means that it actually occupies space in its resource manager’s memory pool.

Ogre organizes its resources, at the highest level of management, into named *groups*. This is to facilitate the loading, unloading, creation, and initialization of resources in terms of a logically related collection. The relationship between the resources in a group is entirely up to you: they can be resources used in a particular game level; they can be all resources that are used to create your application’s GUI; they can be all resources that begin with the letter *A*. A group’s name and purposes really is completely arbitrary, entirely up to you and not in any way meaningful to Ogre (other than the name of the default “catch-all” resource group: General). When your application goes searching for a resource, Ogre can find it (if a reference to it has been created) regardless of the group in which it exists (if you tell Ogre to search all resource

groups). This demarcation between groups is another useful feature of resource groups: you can use resource group names as a sort of "namespace" for same-named resources (if your application design needs this sort of thing).

Non–manually loaded resources in Ogre exist solely in *archives*. The archive in Ogre is simply an abstraction of a generic file container. The archive can be searched (using file name wildcards), both recursively and nonrecursively; it can return a reference to a file within itself; it can be opened and closed. Sounds a lot like a file system, doesn't it? As you might expect, the file system is just another type of archive to Ogre. The two types of archive that Ogre understands are the FileSystem and the Zip archive (the latter is a simple file in PKZIP format, compressed or otherwise). You can implement any type of archive you like. For instance, if your application uses a custom archive format for its assets, you can create an implementation of an Ogre archive that reads and manipulates this file format, to provide Ogre with access to the assets within it.

Ogre will index archives based on known file extensions, such as `.material`, `.mesh`, `.overlay`, and so on. Unknown file types are ignored, so you can mix Ogre- and non-Ogre-related resources in the same file if you like.

Subsystem Overview

The design highlights and philosophy outlined previously are implemented in numerous classes within the Ogre API. Luckily, you do not need to be familiar with all of them in order to be productive with Ogre. With basic knowledge of just a few bits of Ogre (and of course some available art assets), you can have a 3D application running in no time at all.

Let's briefly tour the most basic and common systems with which you will interact in a typical Ogre application. These systems and classes will be covered in more detail later in this book, but in the interests of fostering familiarity early in your experience with Ogre, I will introduce them here. You may see some things in this section that do not make immediate sense; that's OK, more specific coverage occurs in later chapters.

Root Object

The main point of access to an Ogre application is through the **Root** object. As pointed out earlier, this is a façade class, and it provides a convenient point of access to every subsystem in an Ogre application.

The **Root** object is the simplest way to fire up and shut down Ogre; constructing an instance of **Root** starts Ogre, and destructing it (either by letting it go out of scope or executing the `delete` operator on it) shuts down Ogre cleanly. For all objects whose lifetime Ogre is responsible, it will clean them up in an orderly fashion.

Resource Management

Anything that Ogre needs in order to render a scene is known as a *resource*, and all resources ultimately are managed by a single object: **ResourceGroupManager**. This object is responsible for locating resources (within search paths defined by the application to the manager) and initializing (but not necessarily loading) the known types of resources that it finds.

By default, Ogre recognizes the following types of resources:

- **Mesh**: Ogre supports a single binary mesh format, one that is optimized for fast loading and is generated typically by the **OgreXMLConverter** command-line tool provided with Ogre. While you can create your own geometry on the fly (or provide a manual mesh loader if you have reason to do so), typically these resources will exist on the file system, and must be named with a `.mesh` extension for Ogre to recognize them as mesh data. Mesh files also contain animation data for morph and pose animations.
- **Skeleton**: Skeleton resources typically are referenced within a `.mesh` file (but can be used by themselves if you need) and define the bone hierarchy and keyframe data used with skeletal animation. These files use a `.skeleton` extension and also are created typically by the **OgreXMLConverter** command-line tool.
- **Material**: Material script files define the render state used when rendering a batch of geometry. Material scripts are referenced by mesh data either in a `.mesh` file or manually using the Ogre renderable object methods. These scripts are output by a 3D modeling tool exporter, and Ogre recognizes them by their `.material` extension.
- **GPU program**: High-level GPU programs (HLSL, GLSL, Cg) are recognized by their `.program` extension. Low-level ASM programs are recognized by an `.asm` extension. Ogre will parse (but not compile) these files prior to parsing any `.material` files, so that the programs defined within the `.program` files are available before being referenced in a material.
- **Texture**: 2D texture data can exist in any format supported by Ogre (actually, by the OpenIL image library, which means an extremely wide variety of image formats). These files are recognized by their particular extensions.
- **Compositor**: Ogre's Compositor framework uses Compositor scripts the same way that the material system uses `.material` files; the difference is that Compositor scripts use the `.compositor` extension.
- **Font**: Ogre uses font definition files to define the fonts it uses in overlays. These files use a `.fontdef` extension.

Each of these types of resources has its own particular **ResourceManager** (for example, **MaterialManager**, **FontManager**, and so on), but unless you are writing new plug-ins or adding new types of resources to Ogre's resource management system, you will not need to deal with **ResourceManager** at all.

The **ResourceGroupManager** is responsible for finding your resources when you ask for them by name. It does not perform the actual memory management tasks required of an actual resource manager (such as unloading old resources to make room for new ones when needed); that is handled by the **ResourceManager** base class. The **ResourceGroupManager** instead allows you to load and unload groups of resources by their group name (such as unloading all **Font** resources to free up some memory).

By default, Ogre expects its resources to exist as disk files. However, certain types of resources can be manually managed; currently only the mesh and font resource types have manual resource loader implementations in Ogre, but if you have a need to create manual resource loaders for a particular type of resource, the framework is there to do so.

Scene Management

The scene graph design discussed earlier is part of a larger concept in Ogre known as the *scene manager*. All scene graph implementations are derivations of the **SceneManager** class. You will interact quite often with the active **SceneManager** in your application. Actually, you might interact with the active "scene managers," since Ogre supports multiple simultaneous active scene managers. However, the vast majority of applications will create and use only a single scene manager at a time.

Your scene manager is the source for your **SceneNode** objects. *Scene nodes* are the structural element in the Ogre scene graph design; they are what you actually move around in the scene. They can also be related hierarchically (you can have parent and child nodes, in other words); therefore you can translate them, scale them, and rotate them in world, parent, or local (object) space. Scene nodes can exist independent of the scene graph; one simple means of preventing the rendering of content in your scene is simply to detach a part of the scene graph hierarchy: the contents are unaffected, and you can reattach it at will.

Your content is, in turn, attached to these scene nodes. Almost all of your content will exist in the form of **Entity** instances, which are implementations of **MovableObject**, and also created by the scene manager. Once you have a valid entity, you can attach it to an existing scene node. An entity most often is loaded from disk, where it exists as a binary `.mesh` file. However, it is possible to create "manual" content objects, as well as procedural objects such as a movable plane (the only intrinsic procedural object supported currently in Ogre). Since your content is attached to a scene node, it is the node that is moved around the scene, and not the content.

You can also attach other noncontent objects to scene nodes. For example, you might have a reason to want to attach a camera to a scene node. You can also attach lights to scene nodes if you wish.

Render Systems and Render Targets

You typically will not need to interact directly with a render system. **RenderSystem** is a generalization of the interface between Ogre and the underlying hardware API (OpenGL or Direct3D). You will, however, likely interact, at least somewhat, with an object created by the render system: the **RenderTarget**. **RenderTarget** is a generalization of two important objects in Ogre: the render window and the render texture. The former is what nearly every Ogre application will use; render textures are a more specialized (yet still commonly used) object for performing more advanced rendering magic.

The render window in Ogre is your application's main window (among others; multiple render windows are supported). In some cases, the render window can be embedded within another window (useful for creating Ogre-based 3D tools), but in nearly all cases, if you want to see your scene rendered to the screen, you will need at least one render window. Exceptions to this rule would be applications that render to offscreen render targets and then display the results via another mechanism; this could be useful for a 3D tool that wanted a non–real time render preview using the 3D accelerated graphic pipeline.

The render window can be created automatically (the easy way) through the **Root** object façade, or more manually through **Root** or via **RenderSystem**. Manual creation obviously allows more customization of the render window properties, but not all applications need a great deal of customization; for those applications, automatic render window creation is more than enough.

Ogre Managers

A *manager* in Ogre is simply a class that manages access to or lifetimes of other related types of objects, hence the name. For example, the **ArchiveManager** in Ogre manages the creation and registry of **Archive** implementations, as well as access to registered **Archive** implementation instances. Each of the managers, including **Root** (which can be said to be a manager itself, the "Ogre Operations Center" if you will), exists as stand-alone "singleton" objects. One of the side-effects of the creation of **Root** is the initial instancing of all of Ogre's manager objects.

■**Note** The Singleton design pattern is commonly used for classes designed to have only a single existence throughout an application. For this reason, **Manager** classes are commonly implemented as singletons, since they typically are responsible for managing access to specific types of application data and resources. The Singleton pattern allows access to the **Manager** classes' single instance from anywhere in the global namespace of an application, a property often used to avoid having to pass around pointers to their instances, but mostly the Singleton pattern allows control over the lifetime of the class instance. Singletons are widely subclassed by Ogre managers.

I will give a brief description here of what each of those managers is responsible for managing. The more detailed discussion of each of these managers is what the rest of this book contains.

- **LogManager**: Sends logging messages to output streams for Ogre as well as for any code that wishes to use it.
- **ControllerManager**: Manages *controllers*, which are classes that produce state values for other classes based on various inputs; most commonly used for animating textures or materials.
- **DynLibManager**: Manages dynamic link libraries (DLLs on Windows, shared objects on Linux), which makes this class central to the plug in–based design of Ogre. Will also cleanly unload loaded libraries at shutdown.
- **PlatformManager**: Provides abstract access to details of the underlying hardware and operating system, such as timers and windowing system specifics (such as the Ogre configuration and error dialogs).
- **CompositorManager**: Provides access to, and management of, the Compositor framework, which in turn supports typical 2D composition and postprocessing tasks in screen space.
- **ArchiveManager**: Provides to the resource management system the correct type of class to handle file "containers" such as ZIP files or file system directories.
- **ParticleSystemManager**: Manages the details and implementations of various particle systems, emitters, and affectors.
- **MaterialManager**: Maintains all loaded **Material** instances in the application, allowing reuse of **Material** objects of the same name.

- **SkeletonManager**: Maintains all loaded **Skeleton** instances in the application, allowing reuse of **Skeleton** objects of the same name.
- **MeshManager**: Maintains all loaded **Mesh** instances in the application, allowing reuse of **Mesh** objects of the same name.
- **HighLevelGpuProgramManager**: Maintains, loads, and compiles all high-level GPU shader and vertex programs used in the application (i.e., GPU programs written in HLSL, GLSL, or Cg).
- **GpuProgramManager**: Maintains and loads low-level GPU programs (i.e., those written in assembler), as well as high-level GPU programs previously compiled down to assembler.
- **ExternalTextureSourceManager**: Manages external texture source class instances, such as those that implement video streaming.
- **FontManager**: Manages and loads defined and available fonts for use in Overlay text rendering.
- **ResourceGroupManager**: Serves as the main "point of contact" for loading and lifetime management of all registered application resources, such as mesh and material.
- **OverlayManager**: Manages loading and creation of 2D Overlay class instances, used typically for HUD, GUI, or other 2D content that is rendered on top of a scene.
- **HardwareBufferManager**: Manages lifetime of and access to shared hardware buffers (pixel buffers, vertex buffers, index buffers, and so on).
- **TextureManager**: Manages lifetime of and access to all textures referenced, loaded, and used in the application.

As you can see, there are few stones left unturned in the Ogre class design, and this is just the top-level class list. Each of these manager classes allows access to (or provides access to, in the case of custom implementations) dozens more classes that do the actual work in Ogre.

Conclusion

This chapter was not intended to cover everything about Ogre. At this point, you should be familiar with the most common Ogre objects, as well as have a passing familiarity with some of the less common ones as well. The rest of the book will cover each major area of Ogre functionality in much greater detail, but at least now you have a working base of knowledge about Ogre on which you can build as you work through the book.

If you just want to dive in and make graphics on the screen, you should carry straight on to the next chapter. However, if you are less reckless and want to get to know Ogre a bit better before becoming so intimate, you can skip Chapter 4 and come back to it when you are ready. Either way, it will be a fun ride!