XEngine

System Design and Implementation

Author: Martin Ecker
Project Website: http://xengine.sourceforge.net
Last Modified: 5. May 2004 22.49

Please note that this document is a bit out of date, however the basic design of XEngine hasn't changed much from what is presented here. In particular, there is now a new component called XEngineUtil and the Mesh class has been moved there. For a current structural design overview of XEngine please see the API reference documentation.

1.1 Introduction

One of the main goals in developing XEngine was to have a clean and object-oriented design. Many modern and acknowledged, object-oriented design patterns and guidelines [Land95] have been used in the design of the engine. The next couple of sections will describe XEngine's design in detail with the use of UML diagrams [Hitz99]. First the high-level component design is presented. Then the class design of the three main components, XEngineMath, XEngineCore, and XEngineSceneGraph, will be discussed. Finally, the subsequent section 1.6 reflects on the design patterns used.
In addition to a modern design, the engine was also implemented by using modern C++ techniques, such as templates and exception handling, as will be discussed in section 1.7.

I would like to note at this point that XEngineSceneGraph is currently not publicly available because it is in a rather messy and unfinished state. Also more features need to be added and performance optimizations need to be done before the scene graph can be released. I still decided to describe the current design here because some people might find it interesting. I'm also grateful for any further design ideas or improvements for the scene graph.

1.2 Component Design

In its current state, XEngine consists of five main components where each component is built as a shared library or, to be specific, as a DLL under Windows and as a shared object under Linux. The five components are shown in the UML component diagram in figure 1.1. User applications only have direct access to three of these components, XEngineMath, XEngineCore, and XEngineSceneGraph. The other two components are renderer libraries for a particular 3D graphics API used only internally by the core render system.

XEngineMath contains all the mathematical routines and classes for handling vectors, matrices, quaternions, and a number of two-dimensional and three-dimensional geometric objects. XEngineCore is the core render system and can be seen as a wrapper of an underlying 3D graphics API, such as OpenGL or Direct3D. From the point of view of the client application, XEngineCore actually *is* a 3D graphics API. XEngineCore defines an abstract interface that needs to be implemented by renderer libraries wrapping an underlying 3D graphics API. Currently, renderer libraries for two graphics APIs are available, XEngineRendererOGL13 for OpenGL 1.3 and XEngineRendererDX81 for DirectX 8.1. Finally, XEngineSceneGraph contains a flexible scene graph with efficient visibility culling, a BSP tree, and a scene loader for Quake 3 BSP files. Note that applications are not required to use this scene graph, but can use their own scene management system if so desired.

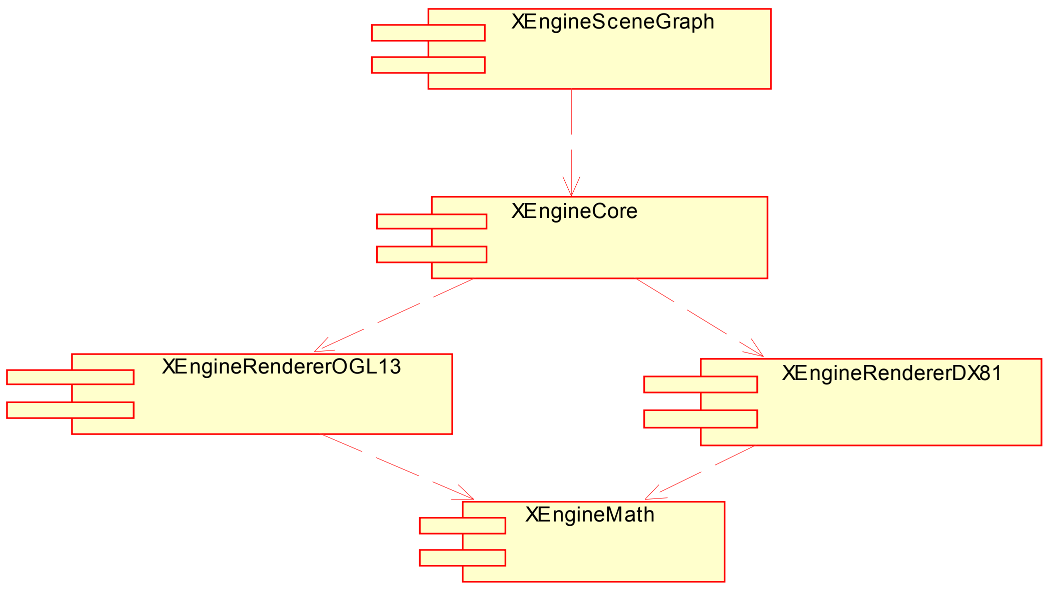

Figure 1.1: UML Component Diagram of the Component Design

The subsequent sections discuss the design of the three main components of the engine in more detail.

1.3 Mathematics Library Design

XEngineMath is the mathematics library of XEngine. It offers a number of stand-alone classes for handling vectors, matrices, and quaternions, and a class hierarchy of two-dimensional and three-dimensional geometry objects. All the other components of XEngine heavily use XEngineMath, and it can be used independently of the other engine components.

1.3.1 Stand-Alone Classes

Figure 1.2 shows a high-level UML class diagram of the available stand-alone classes in the mathematics library. The vector classes offer all expected operations, such as addition, subtraction, multiplication, division, dot product, cross product, normalization, and many more. Similarly, the matrix classes have a number of useful operations, such as matrix-vector multiplication, matrix-matrix multiplication, inversion, and a number of others. Since mostly homogeneous four-by-four matrices are used in 3D graphics, the Matrix4x4 class has a variety of additional Set-methods, such as SetTranslation, SetRotation, or SetPerspectiveProj, that create special matrices. Note that the engine uses row-major matrices by convention. Finally, the Math class is a special class that only contains static methods for various often required mathematics operations, such as computing the absolute value or calculating the square root of a value.

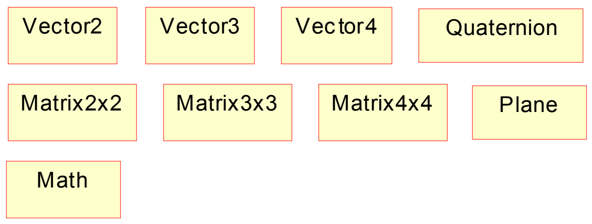

Figure 1.2: UML Class Diagram of the Stand-Alone Mathematics Classes

Currently, none of these classes are optimized with the use of specialized CPU instruction sets, such as the SIMD extensions of Intel processors. However, the design of the mathematics library easily allows the integration of specialized versions of these classes.

1.3.2 Two-Dimensional Geometric Objects

In addition to the stand-alone classes presented in the last section, the mathematics library of XEngine offers a variety of classes for dealing with geometric objects. Figure 1.3 shows the two-dimensional geometric object classes of XEngineMath. Using a template mix-in class, all the geometric classes offer methods to compute the minimum distance between each other, including the two points of both objects that have the shortest distance. Additionally, all objects can be transformed by a three-by-three matrix. The individual geometric object classes offer further useful methods. All the used mathematical algorithms are documented with detailed descriptions in the XEngine API reference documentation.

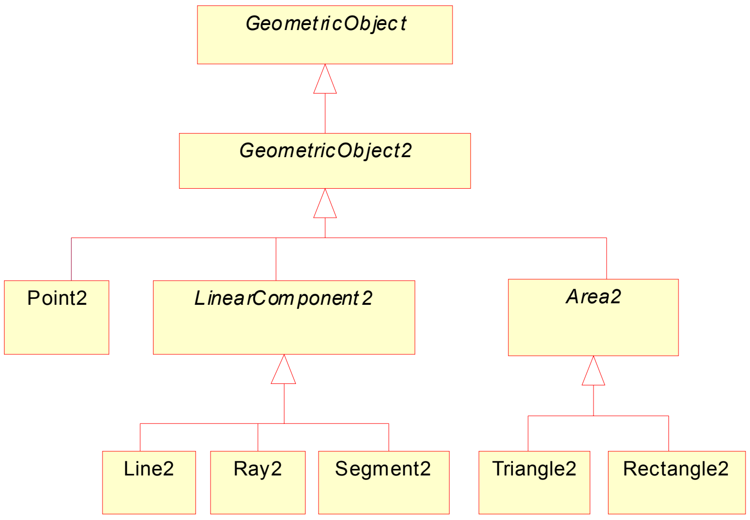

Figure 1.3: UML Class Diagram of Two-Dimensional Geometric Objects

Following object-oriented design patterns, all geometric objects are derived from the abstract base class GeometricObject, including the hierarchy of three-dimensional geometric object classes presented in the next section. All two-dimensional geometric objects are then derived from the base class GeometricObject2. Finally, the classes representing linear components, Line2, Ray2, and Segment2, are derived from a common base class called LinearComponent2, and the classes representing an area, Triangle2 and Rectangle2, are subclasses of Area2.

These two-dimensional classes are currently only rarely used in other parts of the engine, but can be useful to user applications, for example for developing a two-dimensional user interface that is layered on top of a 3D scene, such as a head-up display, using XEngine.

1.3.3 Three-Dimensional Geometric Objects

All of the two-dimensional geometric object classes offered by XEngineMath have a corresponding three-dimensional version. Additionally, a number of volume classes are added to the class hierarchy, some of which are also used as bounding volumes in the scene graph component. The additional volume classes are Box, which represents an oriented box, AxisAlignedBox, Sphere, and Frustum. The latter is used to represent the view frustum of the camera in the scene graph component. Figure 1.4 shows the class hierarchy of three-dimensional geometric object classes in XEngineMath.

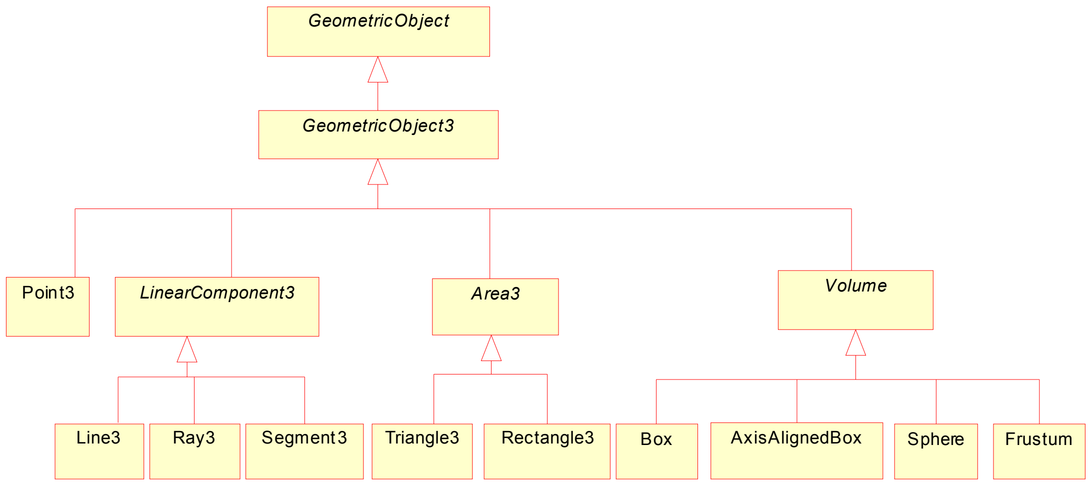

Figure 1.4: UML Class Diagram of Three-Dimensional Geometric Objects

All three-dimensional geometric object classes are derived from the same abstract base class, GeometricObject, as the two-dimensional counterparts shown in figure 1.3, and additionally from a common base class called GeometricObject3. The linear component and area classes also have common base classes, as do the volume classes. Three-dimensional geometric objects can be transformed by a four-by-four matrix. All Volume-derived classes have a method GetVolume to compute the volume of the object. The Box, AxisAlignedBox, and Frustum classes offer additional methods to retrieve their eight corner vertices, the six planes representing the six sides of the volume in the form of six Plane objects, and the six rectangles in the form of six Rectangle3 objects that make up the volume.

1.4 Render System Design

The render system component, called XEngineCore, implements the render system of the engine and provides API-independence from the underlying 3D graphics APIs using renderer libraries that are dynamically loaded at runtime. XEngineCore's public interface can itself be seen as an object-oriented 3D graphics API.

Just like XEngineMath, XEngineCore can be used independently of the higher-level components of the engine, specifically the scene graph component described in section 1.5. This can be useful for applications that do not want to use the scene graph management offered by XEngine (maybe because the application wants to implement its own scene graph) and just want to take advantage of the benefits of platform- and rendering API-independence.

1.4.1 Render System Definition

The render system acts as an abstraction layer between the application and the underlying rendering API, such as OpenGL, DirectX, or possibly even a software renderer. It maps the functionality offered by the underlying rendering API to a well-defined, object-oriented interface that the application uses to render scenes. The application only works with the interfaces offered by the render system and usually does not have to concern itself much with the peculiarities of the underlying 3D graphics API. Therefore, XEngineCore is the component responsible for providing the application with API-independence. To achieve this API-independence the render system defines an abstract interface that gets implemented by the specific renderer libraries. The renderer libraries then map the functionality of the common application interface of XEngineCore to the underlying rendering API.

The main functionality of the render system comprises routines to create render contexts and render targets, to begin and end a scene, to swap back with front buffers, to render primitives, to create vertex and index buffers, to manage render states and texture sampler stages, and to offer support for vertex and fragment shaders using low-level and high-level shading languages.

The render system resides in its own shared library called XEngineCore with all the API-dependent code again residing in separate shared libraries. The DirectX 8.1 renderer, for example, resides in a shared library called XEngineRendererDX81 (see figure 1.1). The engine provides a special dialog that can be used to let the user choose a renderer at runtime. An application can, of course, also decide which renderer library to use depending on other criteria. Once a renderer has been chosen, the corresponding shared library is loaded, and the renderer can be used to render a scene. More than one renderer can be used by an application at the same time, which allows using different 3D graphics APIs to render into multiple windows.

1.4.2 Render System Conventions

3D graphics APIs all have their own conventions regarding coordinate systems, vector and matrix forms, and so on. It is therefore imperative for a 3D engine to have well-defined conventions that are used throughout. This section outlines the conventions used in XEngine.

XEngine uses a right-handed coordinate system where the x-axis points to the right, the y-axis points up, and the z-axis points out of the screen. The vertices of front faces are specified in counterclockwise winding. Therefore, backface culling will usually be set to cull clockwise-oriented faces.

Just like Direct3D, but unlike OpenGL, XEngine uses row-major matrices and vectors. When addressing an element of a matrix M, M[x][y] refers to the *x*-th row and the *y*-th column of the matrix. This is also the way that matrices are stored in the programming language C, which is one of the reasons why the row-major form was chosen for the engine. A homogeneous transformation M is applied to a vector v by pre-multiplying the vector with the matrix, so that the resulting vector is $v' = v \cdot M$. A nice by-product of this convention is that, when constructing a combined transformation matrix that is the product of various transformation matrices, the multiplications are done from left to right. Or in other words, transformations are applied in the order they are multiplied together and not in reverse order, as would be the case with column-major matrices and vectors. This is another reason why the author prefers row-major matrices.

Given the homogeneous vertex $\begin{bmatrix} x & y & z & w \end{bmatrix}$ in clip coordinates, the clipping volume used in XEngine is defined as the axis-aligned space given by the following inequalities

$$-w < x < w$$
$$-w < y < w$$
$$0 < z < w$$

Finally, the origin of the texture coordinate system is the left-most corner for one-dimensional texture maps, the top-left corner for two-dimensional texture maps and for each of the six two-dimensional faces of a cube map, and the top-left-front corner for three-dimensional texture maps. The faces of a cube map texture are arranged as shown in figure 1.5.

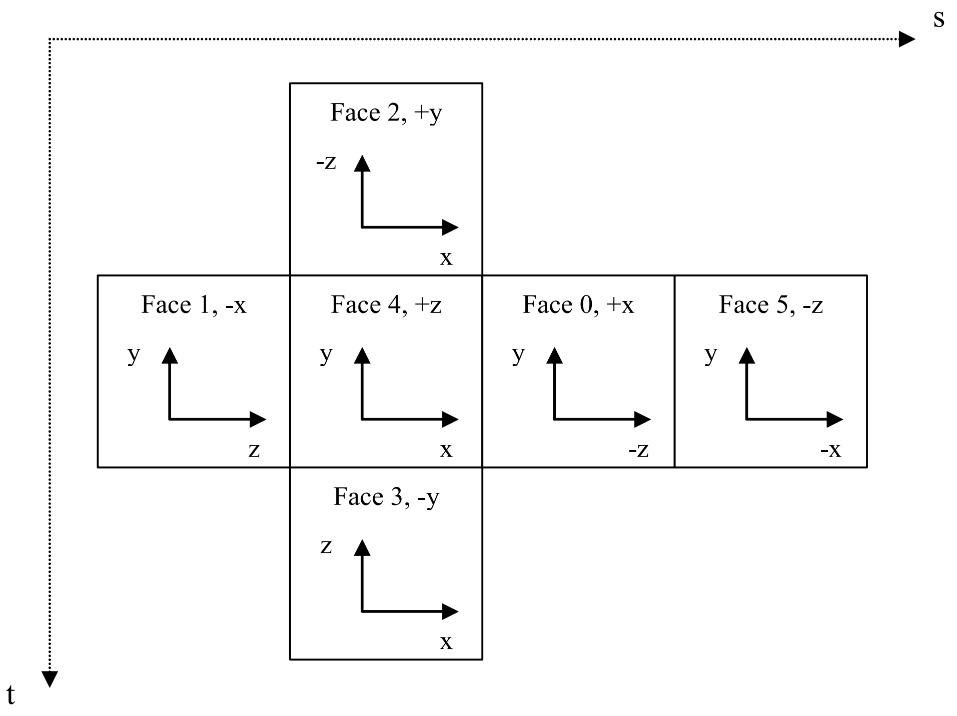

Figure 1.5: Cube Map Texture Conventions

The engine accepts texture coordinates with one, two, three, or four components that are named s, t, r, and q, respectively. If less than four values are specified, the rest of the components are filled up with the value 0, except for the fourth component, for which the value 1 is used. When using the fixed-function pipeline, the s, t, and r coordinates are always divided by q by the rasterizer to allow projective texture mapping.

1.4.3 Render System High-Level UML Class Diagram

The following UML class diagram shows a high-level view of the most important class hierarchies in the render system, in particular the hierarchy of render resource and render target classes. The subsequent sections describe the most important classes in more detail. The diagram only shows the classes that an application built with the engine uses. For this reason certain classes like Renderer or RenderWindow are only shown as abstract base classes. In the specific renderer libraries, classes that implement the concrete versions for a particular graphics API are derived from these abstract base classes.

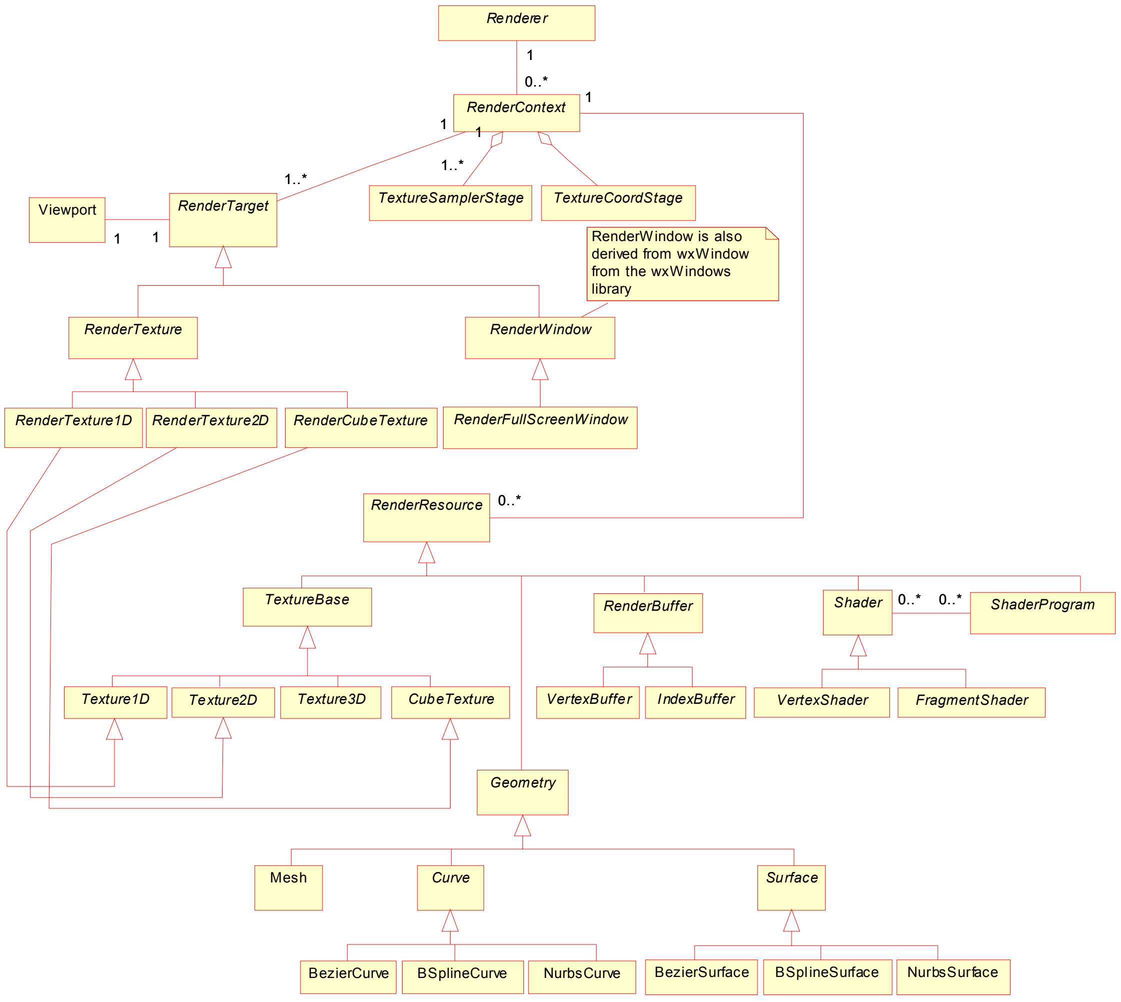

Figure 1.6: Render System High-Level UML Class Diagram

1.4.4 The Renderer and RenderContext Classes

The Renderer and RenderContext classes are the main interfaces for an application to the render system. The Renderer class is the first interface the client acquires by calling the static member function Renderer::CreateRenderer of the XEngineCore shared library. An instance of the Renderer class represents one of XEngine's renderer libraries that implement the core functionality for a specific 3D graphics API. Using a Renderer object, the application can then create a RenderContext object which is logically composed of all the available render states and the available texture sampler stages and texture coordinate stages and their associated states. A render context is always bound to a specific graphics adapter in the system and should only be used from the thread that created it. Additionally, a render context has a number of associated render resources and render targets that the application can create. When a render context is created, at least one render target, a primary render window, must be created. A render window is an instance of the class RenderWindow or any class derived from it. Render targets will be discussed in more detail in section 1.4.6.

With the RenderContext interface an application can begin and end a scene, swap back with front buffers, create render targets (different types of windows and textures) and render resources, draw primitives and meshes, and set render states. There are also routines to set the world, view and projection matrices and a whole range of other methods that usually directly map to the underlying rendering API. One of the most frequently called methods of RenderContext are the two RenderContext::Draw methods used to render primitives to the currently active render target. Primitives supported by XEngine are point lists, line lists, line strips, triangle lists, triangle strips, and triangle fans. The vertices that compose primitives are stored in one or more vertex buffers. The attributes of a vertex can therefore be split up and stored separately in multiple buffers. The first of the two Draw methods of the RenderContext takes a vertex buffer stream mapping as input parameter and renders the geometry contained in the vertex buffers defined by the mapping to the current render target. The second Draw method additionally takes an index buffer as input parameter and renders indexed primitives to the current render target. It is advisable to use indexed primitives whenever possible, because it reduces the amount of memory sent to the graphics hardware. With indexed primitives, multiple primitives can use the same vertex by reusing its index in the index buffer without having to duplicate all the vertex attributes of that vertex in the vertex buffer.

Both the Renderer and the RenderContext classes are actually abstract base classes from which classes are derived in the renderer libraries that implement most of the functionality using the underlying rendering API. Most of the time, the deriving classes have to map the routines to the corresponding routines of the rendering API and not actually implement much. Therefore, implementing a renderer for a new 3D graphics API is usually a task that is tedious, because of the high number of routines to implement, but uncomplicated, since it mostly only involves mapping functions to functions.

1.4.5 States

XEngine differentiates between a variety of state types. A *render state* is a state that affects the entire render context. A *texture state* only affects texture objects, or in other words it represents per-texture object state. A *texture sampler state* affects a texture sampler stage and correspondingly a *texture coordinate state* affects a texture coordinate stage. Examples for render states are the state of the depth buffer or stencil buffer. Examples for texture states are the used filters or addressing modes. Currently XEngine does not offer any texture sampler states. Examples for texture coordinate states are the texture matrix and texture coordinate generation modes.

Render states, texture states, texture sampler states, and texture coordinate states in XEngine are grouped together in classes. For example, all the depth buffer-related states are grouped together in a class called RSZBuffer. If an application wants to set a particular depth buffer state, it first creates an instance of RSZBuffer, sets the required state variable in that object, and then actually sets the state by passing the object as parameter to the method RenderContext::SetState. This design choice was made for better object-oriented encapsulation and ease of extensibility with new render states without breaking the binary interface of the RenderContext class. It also reduces the interface complexity of the RenderContext class, since only one method is required for setting render states and that method can handle all the available render states. Similar to render states, the texture sampler states and texture coordinate states are grouped together in various classes. Each texture sampler stage and texture coordinate stage, respectively, has a distinct set of states. These states are set by passing an instance of any of the state classes to the methods TextureSamplerStage::SetState and TextureCoordStage::SetState, respectively. Texture states are also grouped together in classes. Texture states are set by calling TextureBase::SetState.

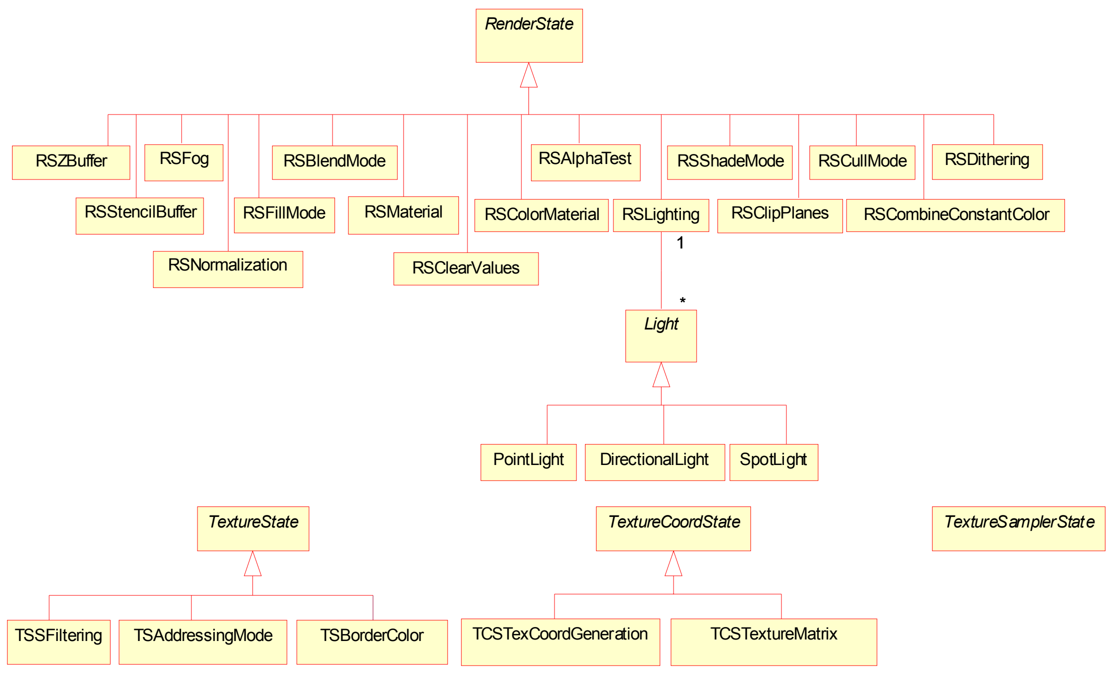

Figure 1.7: UML Class Diagram of the State Class Hierarchy

Figure 1.7 shows the class hierarchy of states in XEngine. All render states are derived from an abstract base class RenderState, all texture states are derived from an abstract base class TextureState, all texture sampler states are derived from an abstract base class TextureSamplerState, and eventually all texture coordinate states are derived from an abstract base class TextureCoordState. Additionally a template mix-in class[1] called StateBase is used, which defines the abstract interface of all types of states. It would also have been possible to directly derive the texture sampler states and texture coordinate

1. For details on template mix-in classes, see section 1.7.3.

states from the render states which would, however, require that all textures sampler states and texture coordinate states have a reference to one of the corresponding stages. Since a state should be independent of a specific stage, this option was dropped. Note that, due to the nature of the programmable pipeline, a lot of these states only apply when the fixed-function pipeline is used (for details, see my other paper titled Programmable Graphics Pipeline Architectures).

Note that, unlike many other engines, the concepts of materials and lights used in the fixed-function pipeline of OpenGL and Direct3D are also considered to be regular render states in XEngine. Another design variation would have been to have separate classes, Light and Material, and corresponding methods, RenderContext::SetMaterial for materials and RenderContext::SetLight or RenderContext::AddLight for lights. But since materials and lights can be considered to be general render states, a decision has been made to include materials and lights in the render state class hierarchy.In addition to the state classes presented so far, the engine offers a StateSet class, which is composed of all available render states, all texture sampler states for all texture sampler stages, and all texture coordinate states for all texture coordinate stages. In addition it contains a reference to a shader program and a reference to a texture object for each texture sampler stage. So a state set can be considered to represent the entire state of the graphics pipeline. Various operations can be performed on state sets, in particular merging two state sets, computing the difference or intersection of two state sets, and applying an entire state set to the render context that created it. For details see the XEngine API reference documentation.

1.4.6 The RenderTarget Class Hierarchy

A render target is a target for render operations initiated by the two RenderContext::Draw methods. XEngine supports two main types of render targets, windows and textures. Rendering a scene to a texture allows powerful techniques, such as dynamic environment mapping, and is the basis for a lot of other techniques involving floating point buffers, since these types of buffers are only supported in the form of textures on current graphics hardware. Each render context can have any number of associated render targets, restricted only by the capabilities of the graphics hardware. However, at any time, only one target can be set as the active target to be rendered to.

Render targets cannot be created directly by using a constructor, but are created by a special constructor function called RenderContext::CreateRenderTarget, which receives a data object describing the type and format of the to-be-created render target. This technique is necessary to give the renderer libraries a chance to return renderer-specific render targets. A render target is made active by calling RenderContext::SetRenderTarget passing in the desired render target as parameter. Each render target has an associated viewport which defines the visible rectangular area of the target that is used for rendering. All primitives are clipped against this viewport. Therefore, primitives rendered outside the viewport are discarded.

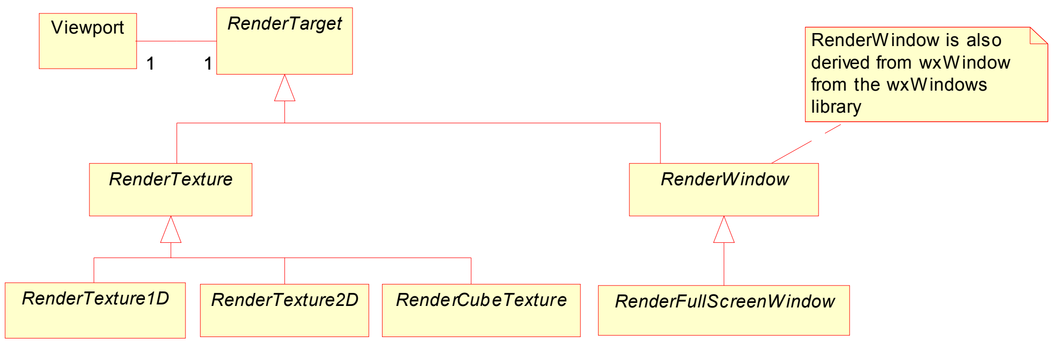

Figure 1.8: UML Class Diagram of the Render Target Class Hierarchy

Figure 1.8 depicts the class hierarchy of render targets supported by XEngine. Corresponding to the two basic render target types, windows and textures, figure 1.8 shows two base classes RenderWindow and RenderTexture, both derived from the superclass RenderTarget. The engine currently does not yet support rendering to depth textures, which would allow shadow mapping and similar techniques, but this will be implemented in the near future. Note that all the classes are abstract bases and their actual rendering API-dependent implementation is hidden in the renderer libraries.

1.4.6.1 Render Windows

An instance of the RenderWindow class represents a rectangular window, just like any other window in a typical desktop application. RenderFullScreenWindow is a specialized window class that represents a full screen window, a window that covers the entire area of the graphics adapter bound to the render context that created the window. Applications for visual simulation, CAD-like programs, or level editors for games will typically use one or more RenderWindow objects as render targets, whereas games typically tend to use full screen windows. Sometimes it might be desirable for an application to switch between windowed and full screen mode at runtime. This can be achieved in XEngine by creating two render targets, a regular window and a full screen window. Using RenderContext::SetRenderTarget and hiding the inactive window, the application can alternate between the two windows at any time.

Note that the RenderWindow class also inherits from wxWindow, which is the base class for all kinds of windows in the platform-independent windowing framework wxWindows [wxWi03]. Therefore, a render window of the engine can be used just like any other wxWindows window. It is, for example, possible to use an XEngine render window in a regular dialog, just like other windows, such as buttons, edit fields, or combo boxes. XEngine is tightly coupled with the wxWindows library and mostly depends on it to provide cross-platform services, thus making the engine itself platform-independent. wxWindows will be discussed in more detail in section 1.7.2.1.

1.4.6.2 Render Textures

XEngine offers three different types of texture render targets, one-dimensional, two-dimensional, and cube map render textures. The corresponding classes, such as the RenderTexture2D class, are not only derived from the base class for render textures, RenderTexture, as shown in figure 1.8, but also from the class that represents the corresponding render resource using multiple inheritance (see figure 1.6). RenderTexture2D, for example, inherits from RenderTexture and also from Texture2D. Therefore, a RenderTexture2D object can be used just like any other two-dimensional texture, but can additionally be passed as parameter to RenderContext::SetRenderTarget to be set as the active render target. The design decision for this easy-to-use solution for render-to-texture was not an easy one, considering the way render-to-texture is implemented in OpenGL using so-called pbuffers, which does not lend itself easily to such a design. However, ease-of-use of the API was one of the main design goals for XEngine, which is why the current approach has been chosen, incurring more implementation complexity in the OpenGL renderer library.

1.4.7 The RenderResource Class Hierarchy

A render resource represents any entity used by the graphics hardware. In particular render resources are textures, index and vertex buffers, shaders and shader programs, and geometric objects, such as meshes, curves, and patches. A resource can either be stored in system memory, in local video memory or non-local video memory (AGP memory), which is usually decided by the graphics driver or the underlying rendering API. Using flags specified during the creation of a render resource, an application can influence the storage location of a render resource to a certain degree without any guarantee that the specified type of memory will be used. Furthermore, each resource has an associated priority that indicates the importance of storing it in high-performance memory. The higher the priority the less the chance that the resource will get removed from video memory.

Figure 1.9 shows the class hierarchy of render resources. RenderResource is the abstract base class for all render resources in XEngine. Just like render targets, resources can never be created directly by using the constructor, but only through certain constructor functions in the RenderContext class, such as RenderContext::CreateVertexBuffer or RenderContext::CreateTexture2D.

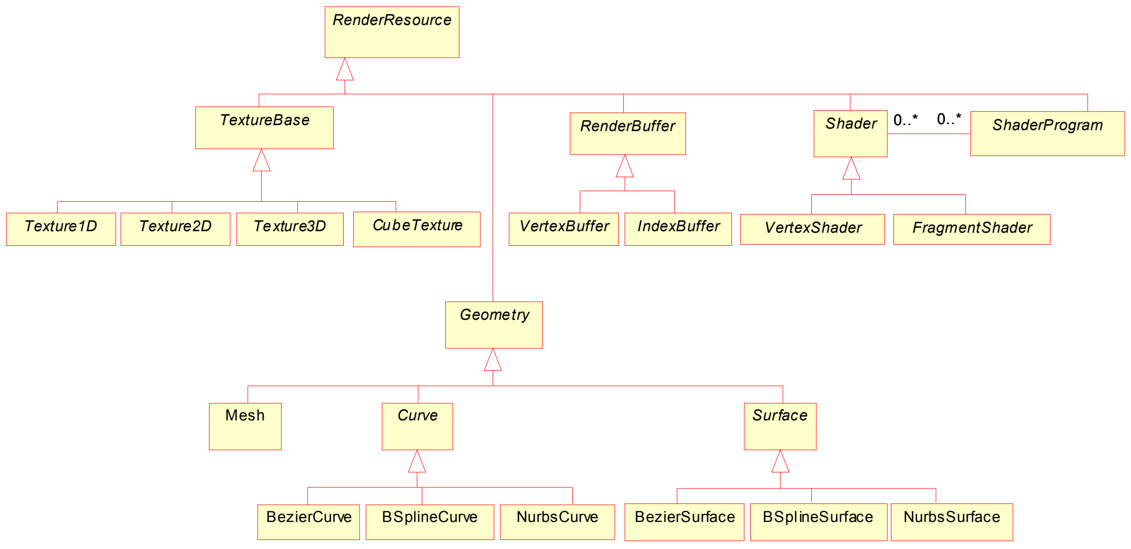

Figure 1.9: UML Class Diagram of the Render Resource Class Hierarchy

Render resources are reference counted so that ownership of a single resource can be shared by multiple objects. For example, a shader program consists of one or more shader objects, and none of its associated shaders must be destroyed while it is the active shader program. Therefore a shader program has shared ownership of all its shaders so that the application cannot delete a shader while it is being actively used. In order to destroy a render resource, a special method named Release needs to be called to indicate that the reference to the resource is no longer needed. Once all references to a resource have been released, the resource destroys itself. Additionally, the render context that was used to create a render resource is also responsible for managing the associated memory of a resource. If any render resources have not yet been destroyed when the context itself is deleted, the render context will delete any resources it created. In other words, a render context has weak references to all resources it creates. It is also important to note that by design resources cannot be directly copied (there is no assignment operator or copy constructor) because they typically occupy scarce video memory. If a copy of a render resource is required nevertheless, the application needs to manually recreate an equivalent resource.

1.4.7.1 Textures

The engine supports all standard types of textures, specifically one-dimensional, two-dimensional, three-dimensional, and cube map textures. The dimensions for each of these texture types must be a power of two, but need not be square. Rectangular textures where the dimensions can be of any size, as provided by the NV_texture_rectangle or EXT_texture_rectangle OpenGL extensions, are not yet supported. Nearest neighbour, bilinear, trilinear, and anisotropic filtering is available for textures. Additionally a texture object can have a chain of mipmaps. A mipmap is a sequence of texture images, each of which is a progressively lower resolution representation of the same image. The height and width of each level in the mipmap is a power of two smaller than the previous level. If desired, the engine can automatically and, if possible, in hardware generate mipmap levels for the entire mipmap chain. A texture is set active in one of the texture sampler stages by calling the method TextureSamplerStage::SetTexture on the desired stage.

1.4.7.2 Vertex and Index Buffers

All geometry data submitted to the render system of XEngine must be stored in vertex and index buffers. A vertex buffer stores vertices where a vertex is a number of vertex attributes, such as position, normal, colour, and so on. An index buffer stores indices into a vertex buffer. To actually draw the geometry data, one of the two RenderContext::Draw methods must be called. Both Draw methods take the primitive type, such as line list or triangle strip, and the number of primitives that are to be rendered as input parameters. The first Draw method additionally requires a vertex buffer stream mapping as parameter which specifies the vertex buffers containing all the vertices that define the to-be-drawn primitives. The vertices in the buffers are used one after the other. If, for example, the given primitive type is triangle list, every three vertices of the vertex buffers will be used to render a triangle.
The second Draw method also takes a vertex buffer stream mapping as input parameter, and additionally an index buffer which contains indices into the vertex buffers and adds a level of indirection. If the specified primitive type is again triangle list, every three indices in the index buffer will be used to retrieve three vertices from the vertex buffers that are then used to render a triangle. Since both vertex and index buffers are render resources, they can potentially be stored in video memory.

An important concept in combination with vertex buffers is the concept of *vertex formats*. Each vertex buffer must have an associated vertex format specified using a data class called VertexFormat, which describes the format, or layout, of the vertex type of the vertices stored in the buffer. A vertex format defines the number of vertex attributes that together compose a vertex or parts of a vertex and their data types. For the fixed-function pipeline the meaning, order, and type of the vertex attributes is fixed and cannot be modified. For vertex formats that describe vertex types used with the programmable pipeline the order and data types of the vertex attributes are not pre-defined and can be freely defined by the application.

The attributes for vertex types used with the fixed-function pipeline must be specified in the following order with the mentioned data types:

- Position in local object space, three 32-bit IEEE floating point values.
- Normal in local object space, three 32-bit IEEE floating point values.
- Primary colour, one 32-bit unsigned integer value in ARGB order.
- Secondary colour, one 32-bit unsigned integer value in ARGB order. The alpha component is ignored and should be zero.
- A maximum of *n* sets of texture coordinates, where each set can consist of one, two, three, or four 32-bit IEEE floating point values. *n* is the number of available texture sampler stages. Texture coordinate sets must be specified in ascending order of the

corresponding texture sampler stage number. It is not necessary to specify a set of texture coordinates for each texture sampler stage since some stages could use automatic texture coordinate generation.

Of course, it is possible to omit certain attributes if they are not used. In fact, the attributes secondary colour and normal are mutually exclusive from a logical point of view because, if a vertex normal is specified, it will be used for computing fixed-function vertex lighting and any specified secondary colour will be ignored in this case.

To give vertex attributes a meaning, that is, to associate them with vertex shader inputs or the inputs to the fixed-function pipeline, a so-called *vertex attribute binding* must be defined by using the VertexAttribBinding class. A vertex attribute binding binds an attribute, specified via its zero-based index in a vertex format, to an input variable name, specified as string, used in the vertex shader or in the fixed-function pipeline. In other words, a vertex attribute binding represents and defines the connection between the vertex data stored in a vertex buffer and the graphics pipeline. It gives the various vertex attributes a semantic meaning, which is implicitly defined by the input variable name the vertex attribute is bound to. A binding does not necessarily have to bind all the vertex attributes described by a vertex format. The not bound attributes will simply be ignored and cannot be used by the vertex shader. This allows reuse of vertex buffers with different vertex shaders, which is especially important for multi-pass rendering.

Vertex attribute bindings for the programmable pipeline can be freely defined by the application and must generally match the required input variables used by the vertex shader, since the vertex shader is the first programmable processing stage of the pipeline. Just as with vertex formats, vertex attribute bindings for the fixed-function graphics pipeline must adhere to stricter rules because of the hardwired logic of the fixed-function pipeline. Firstly, the vertex attributes need to be specified in a certain order and need to be packed tightly. So it is not permissible to not bind an attribute. Secondly, the data type for each attribute is pre-determined and cannot be changed by the application. Thirdly, there are default variable names to which the vertex attributes must be bound to. The default bindings for the fixed-function pipeline have the following names and they cannot be bound in any other order than specified here:

- Position
- Normal
- PrimaryColor
- SecondaryColor
- TexCoord[n]

TexCoord[n] specifies *n* sets of texture coordinates, where *n* is a zero-based texture sampler stage index and cannot be greater than the number of available stages. The texture coordinate sets do not have to be ordered by stage number. So this would be a perfectly legal binding for the fixed-function pipeline

Position - Normal - TexCoord[5] - TexCoord[0] - TexCoord[2] but the following would not because of the wrong order of the normal and the primary colour

Position - PrimaryColor - Normal

The concepts of vertex formats and vertex attribute bindings in XEngine form a very powerful shading language-independent mechanism to specify input data to the programmable pipeline using any current or future shading language. Since the engine forces applications to use vertex formats and vertex attribute bindings for the fixed-function pipeline as well, it should be noted that the engine treats the fixed-function pipeline just as a particular shader program and not something separate, as in many other 3D engines. This will become more evident when the engine's shader system is discussed in section 1.4.7.4. Additionally, to use fixed-function texture blending XEngine defines a small fixed-function fragment shader language that is used to configure the fixed-function texture blending stages. See the XEngine API reference documentation for details.

1.4.7.3 Geometry Classes

The hierarchy of geometry classes offers various classes to store geometry data in a more advanced way than using vertex and index buffers. The Mesh class represents a generalized mesh that can store any number of primitive groups, where each group can have an associated state set which is to be applied when the primitive group is rendered. Additionally, the design provides for various high-order curves and surface classes that are implemented completely in software in XEngineCore, but can also be inherited from by renderer libraries to provide hardware-accelerated versions of these classes. High-order curves and surfaces get tessellated at runtime where the application can choose from a number of subdivision algorithms, such as uniform sampling, subdivision by arc length, and various other non-uniform subdivision algorithms. Even though the design includes them, the current implementation of XEngine does not yet provide curves and surfaces.

1.4.7.4 Shader System

The engine's shader system supports a variety of shading languages, the most important being the high-level shading language Cg. Most of the supported low-level languages can even be cross-compiled, making it possible to use shaders written in the Direct3D languages with various languages only available for the OpenGL API and vice versa. This unique feature of the engine will be discussed in more detail in section 1.7.5. By design, before any geometry can be rendered with the engine, a shader program for the graphics pipeline must be created and set. This even applies to the fixed-function pipeline which is considered to be a special shader program by the engine.

Shaders and Shader Programs

XEngine differentiates between the notion of *shader* and the notion of *shader program*, and it is important to understand these two concepts. Shaders represent programs for a certain stage of the graphics pipeline. Currently, programmable graphics hardware supports two kinds of programmable stages, the vertex processing stage and the fragment processing stage. Correspondingly, as shown in figure 1.9, the engine supports two types of shaders represented by the classes VertexShader and FragmentShader, respectively. Note that these two Shader-derived classes only represent the current state of graphics hardware. In the future, additional classes might be required representing new programmable stages of the pipeline. Future shading languages might combine vertex and fragment shaders (in fact, the Stanford Shading Language already does so), or offer support for other computational frequencies, such as per-primitive. By design, it is relatively easy to extend the engine to support new shader types by simply deriving a new class from the Shader class.

When creating a shader via RenderContext::CreateShader, the application specifies the shader type (per-vertex or per-fragment) and the shading language the shader is going to use, then sets the source code of the shader in the form of a regular character string, and finally compiles the shader. Shaders by themselves cannot be used directly in a render context, as they only represent a program piece for a particular stage of the graphics pipeline. Instead a shader program, an object of the ShaderProgram class, must be created, which represents a program for the entire graphics pipeline and can be set as the active shader program for a render context by calling RenderContext::SetShaderProgram. Creating a shader program involves attaching a number of shaders to it, which need not necessarily use the same shading language, and, when all necessary shaders have been attached and compiled, linking the shader program. This mechanism is similar to the way a regular program for a CPU is built, where first the separate translation units are compiled and then linked together to form an executable program. Similarly, in XEngine the shaders are first compiled separately and then linked together to form a shader program, an executable program for the graphics pipeline.

In theory, a shader program can contain any number of shaders for any shader type. In practice, because of the limitations of current shading languages and shader execution environments, currently only one shader for each shader type can be attached to a shader program. It is, however, possible to attach the same shader object to multiple shader programs. If no shader for a particular shader type is attached to a shader program, the shader program uses the fixed-function pipeline for the processing stage represented by that shader type. So in order to create a shader program that exclusively uses the fixed-function pipeline, a shader program needs to be created and linked without attaching any shader objects to it. Another way of creating a fixed-function shader program or a program where certain potentially programmable stages use fixed functionality is to manually attach special shader objects to the shader program that use a special shading language called FixedFunction. A shader created with the language FixedFunction always represents the fixed-function processing stage described by the shader type of the created shader. For example, creating a vertex shader object via RenderContext::CreateShader passing in the shading language FixedFunction will create a shader representing the fixed-function transform and lighting stage. If an application wants to configure the fixed-function texture blending stages the latter approach must be chosen and a special fragment shader with the FixedFunction language must be created and attached to the shader program. See the API reference documentation for the EBNF grammar of XEngine's fixed-function fragment shader language.

To sum up, for XEngine a shader is a piece of code that is used to program a certain part of the graphics pipeline. A shader program is a collection of multiple shaders (this includes fixed function shaders and possibly even multiple shaders with the same computational frequency) that are linked together to form one single program for the entire graphics pipeline.

Shader Parameters

Most shader programs also require parameters that remain constant per-primitive, so-called uniform parameters, to be set by the application. This is done by using the method ShaderProgram::SetParameter which accepts the variable name of the uniform parameter, as used in the shader, in the form of a character string as input parameter and a value to set the parameter to. The SetParameter method is overloaded for multiple data types, such as floating point values or various vector and matrix types, to guarantee type safety. SetParameter can be called anytime, even if the shader program has not yet been linked or there are no attached shaders. However, the validation of the given parameter name and type might not happen immediately, depending on when ShaderProgram::SetParameter is called. The validation also depends on the shading language and shader execution environment used. There are a couple of possibilities when the parameter name and type are validated, in particular:

- immediately when ShaderProgram::SetParameter is called,
- when a shader gets attached to the shader program,
- when an attached shader gets compiled,
- when the shader program is linked,
- when the shader program gets set as active shader program.

Applications should be prepared to catch validation errors in all of these situations. Parameter values set with SetParameter are stored internally by a shader program object, so that they can be reset when a shader program gets activated after having been set inactive for a while.

A unique feature of the shader system in XEngine is the fact that it supports automatic state tracking for every shading language. Automatic state tracking is, similar to the state tracking offered by ARB_vertex_program, the automatic update of various uniform shader variables with the values of a certain render state whenever that render state changes. Typical render states that a shader might want to have automatically tracked in a shader variable include the world, view, projection, and combined world-view-projection matrix, the lighting render states, such as a light’s ambient, diffuse, or specular colour, and the fog parameters. Automatic state tracking for a parameter is activated by calling ShaderProgram::TrackParameter, passing in the variable name and the to-be-tracked state in the form of a character string. Examples of strings and their corresponding trackable state are "matrix.world" for the world matrix, "light[0].ambient" for the ambient colour of the first light, or "fog.color" for the fog colour value. Using character strings instead of, for example, an enumeration type for the trackable states was an important design decision which allows for easy extensibility with new trackable states and also integrates well with future scripting languages used with the engine. The drawback is that parsing the string incurs a slight performance penalty.

1.5 Scene Graph Design

To provide applications with a high-level scene management system the engine comes with a scene graph component called XEngineSceneGraph that sits on top of XEngine-Core.

1.5.1 Scene Graph Definition

The scene graph component of the engine provides a tree-like scene graph where each node can have at most one parent node. General branch nodes can have any number of children. A design decision has been made to not support nodes with multiple parents, as this would make caching of accumulated attributes in the nodes impossible, or at least very hard and thus inefficient to implement. Sharing of memory-intensive resource data is, however, still possible. Unlike many other engines, applications are not forced to use the scene graph and can provide their own scene management system or be written by only using the core render system of XEngine. The scene graph of the engine strictly distinguishes between the data structure of the scene graph and the algorithms performed on it. This is achieved by decoupling scene graph traversals from the scene graph itself, using so-called *traversal handler chains*. A traversal handler chain is a list of traversal handler objects which contain the operations to be performed on a particular node when a specific scene graph traversal is initiated. Each node of the scene graph has an attached traversal handler chain.

1.5.2 Scene Graph Node Class Hierarchy

Figure 1.10 shows the class hierarchy of the scene graph node classes. In its current state, the scene graph only offers basic node classes that will be further extended with more advanced nodes, for example for character animation, in the future. All node classes inherit from the abstract base class Node, and then from one of the two abstract classes Leaf or Branch depending on the type of the node. Also the list of traversal handlers owned by each node is depicted in figure 1.10. The following paragraphs discuss the rest of the classes shown in the UML diagram in more detail.

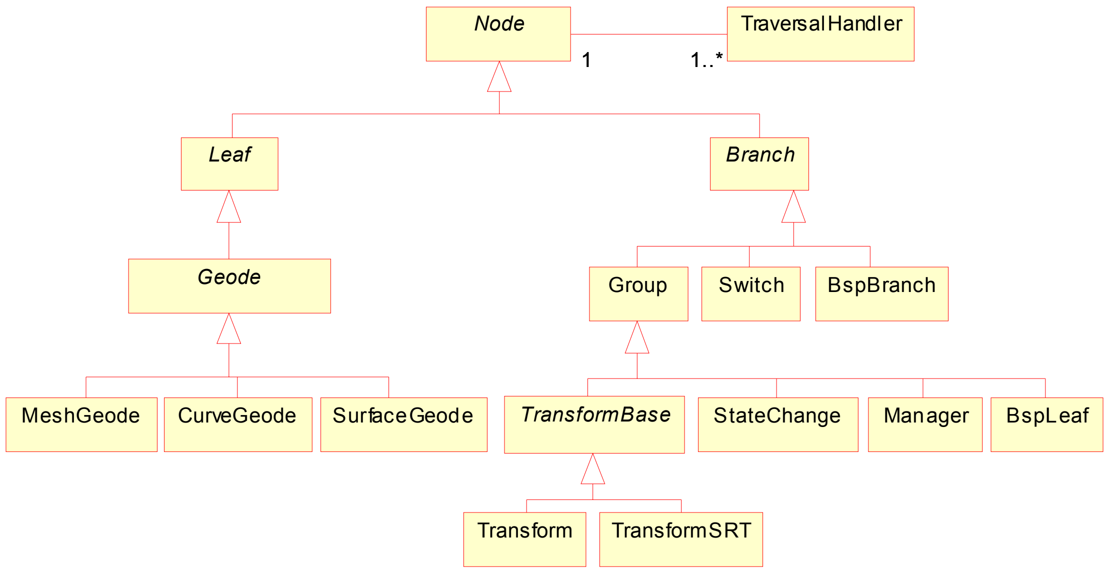

Figure 1.10: UML Class Diagram of the Node Class Hierarchy

The Group class represents a general branch node that can have an arbitrary number of attached child nodes. During a scene graph traversal all children of a group are traversed. The Switch class can also have any number of attached children, but only one of these will be traversed during a scene graph traversal. Which child node is traversed is decided by the application. With this node type discrete level-of-detail can be implemented.

A number of classes are derived from the Group class that have additional properties. The Transform and TransformSRT classes represent some kind of transformation, such as a rotation or translation, that changes the spatial position and orientation of all the children. The Transform node stores the transformation information in the form of a homogeneous four-by-four matrix, whereas the TransformSRT class stores a translation vector, a quaternion for the rotation, and three scale factors. The StateChange class represents a change of any of the render states that will affect all of the node's children. For example, a new texture could be set or depth buffer writes could be disabled by using a node of this type. The Manager class is a special group node that is responsible for keeping the entire subgraph rooted at it up-to-date. A render traversal of a scene graph or parts of it can only be initiated at a manager node. Whenever one of the subnodes of a manager node changes its state, so that it requires an update, it delivers an update request to its manager node, which in turn makes sure that the subnode gets updated in the next update traversal. The manager node tracks all update requests of its subnodes and determines the minimum required update passes for the next update traversal.

Finally, the BspBranch and BspLeaf classes together form a Quake 3-style binary space partitioning tree. A BspBranch node stores a splitting plane and can have at most two children which both must either be another BspBranch node or a BspLeaf node. The first of the two children represents the half-space on the front side of the splitting plane, and the second child represents the half-space on the back side. Note that a BspLeaf node is actually a group node because it by itself only represents the leaf of the BSP tree, but not the leaf of the scene graph. In addition, it does not contain any geometry data, which is instead stored in the child nodes of the BSP leaf node in form of regular Geode-type nodes.

The leaf classes offered by XEngineSceneGraph are used to store actual geometry data that gets rendered when a render traversal of the scene graph determines that the leaf is visible (that is, in the view frustum). Currently, all the Geode-derived classes, MeshGeode, CurveGeode, and SurfaceGeode, internally store references to the corresponding render resources of XEngineCore. However, this must not necessarily be the case, and geometry nodes could even procedurally generate geometry data, such as a terrain, but at the moment no such advanced node classes are provided.

1.5.3 Scene Graph Traversals

A scene graph traversal is initiated by calling Node::Traverse, passing in an initialized object of the Traversal class or one of its subclasses. That traversal object is then responsible for actually performing the scene graph traversal according to the data stored in it. There are three types of traversals an application can perform on a scene graph, depth-first down traversals, up traversals, and so-called one-shot traversals that only traverse a single node. Depending on what type of traversal it is, the traversal object traverses the scene graph in a particular order, visiting the nodes one after the other. Whenever a new node is visited, its traversal handler chain is processed. Each traversal handler function then performs whatever it has to do (for example, update the cached world transformation matrix in Transform nodes) and also has the opportunity to stop either the entire traversal or the current traversal handler chain, which will cause the traversal to skip all remaining traversal handlers of the currently visited node and continue to the traversal with the next node. In addition to driving the traversal, traversal objects collect various interesting information that can be used by the traversal handler functions, such as the Transform or StateChange node visited last. Traversal objects can also optionally perform culling using an application-specified camera object, an object of the Camera class, which represents a view frustum and is also used to render the scene.

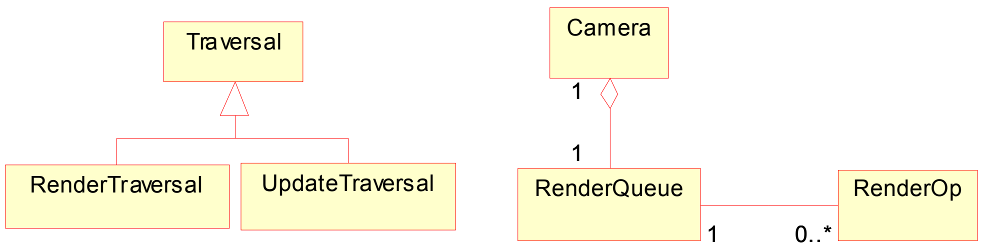

Figure 1.11: UML Class Diagram of the Traversal and Camera Classes

Currently, XEngineSceneGraph supports two types of traversals, update and render traversals. Figure 1.11 shows the corresponding class diagram. Applications are free to add new types of traversals by deriving new classes from the Traversal class. Traversals of both traversal types are executed when a scene graph or parts of a scene graph are rendered. In rendering a scene graph the afore-mentioned Camera class is used, which in addition to representing a camera and its parameters, such as position, orientation, or the field-of-view, also has an associated render target to which the scene captured by the camera is rendered. To render a scene graph, Camera::Render must be called a manager node being passed in. As was mentioned before, only parts of a scene graph rooted at a manager node can be rendered. Before the scene graph is rendered, however, an update traversal is started which is responsible for updating all the nodes. For example, if the application has changed the transformation matrix of a Transform node in the scene graph, the cached world transformation matrices and cached bounding volumes in world coordinates will require an update. All the updates are done in the corresponding traversal handler functions of the nodes that are responsible for update traversals. Therefore, it is possible for the application to perform additional processing during an update traversal, or any traversal for that matter, by adding an additional traversal handler to the traversal handler chain of the desired nodes. Additionally, an application could remove the default traversal handler from the chain, completely disabling the default update traversal, which generally is not a good idea, however.

After the update traversal has completed successfully, the render traversal is started which first performs view frustum culling and, whenever it reaches a visible geometry leaf node, renders the geometry data stored in the leaf. This is again done in the corresponding traversal handler function that reacts to render traversals. Actually, the geometry is not rendered immediately, but instead the necessary render operations (that is, the primitives that need to be rendered and their associated render states) are added to the render queue that is part of the camera object (see figure 1.11). Each time a new render operation is inserted into it, the render queue is automatically sorted according to the most expensive state changes, such as shader program or texture changes. Additionally transparent objects are always sorted in at the end of the queue in a furthest-away-first fashion among themselves, since they should be drawn last and in correct depth order. Once the render traversal is done, the render queue contains all render operations necessary to render the currently visible objects. Since the render queue is already sorted, the final step to render the scene is to iterate through the queue and perform all the stored render operations on the render target associated with the camera.

Since the default render traversal also uses traversal handler functions to perform its duty, an application can add additional traversal handlers to particular nodes to do additional, application-specific operations, as required. It is also possible to completely replace the default behaviour of the render traversal handlers, which, as outlined above, is to add the necessary render operations to the render queue. For example, the geometry data could be rendered immediately or, instead of rendering anything, some kind of debug output could be written to a file. Note that changing the traversal handler chain of a node is possbile at runtime. The traversal handler system makes the traversal scheme of XEngineSceneGraph flexible, highly customizable, and powerful, and, most importantly, separates the scene graph data structure from the algorithms performed on it.

1.6 Used Design Patterns

This section examines which acknowledged, object-oriented design patterns have been used in the design of the engine. Most of these patterns originate from the infamous "Gang of Four" book [Gamm94] about object-oriented design patterns. Design patterns offer solutions to common design problems in object-oriented systems. It is good practice to identify situations in the design phase where the use of patterns is appropriate and use them accordingly. Design patterns also facilitate the communication about a design of a system between many developers because common knowledge of design patterns can be assumed and complex design issues can be easily understood just by naming the applied design pattern.

The RenderContext class of XEngineCore follows the *Abstract Factory* [Gamm94] design pattern. Its interface provides factory methods for creating render resource and render target objects without specifying their concrete classes. The specific RenderContext-derived classes in each renderer library represent specific factories that create concrete render resource and render target objects specific to the renderer library. Figure 1.12 depicts an example of this design pattern. RenderContext::CreateVertexBuffer is the factory method to create vertex buffer objects. The specific render context, RenderContextDX81, overrides the factory method and creates a concrete vertex buffer object that is returned to the client as reference to the abstract vertex buffer class. Since the *Factory Method* pattern is related to the Abstract Factory pattern and the factory methods are also used internally by the RenderContext class, the Factory Method design pattern [Gamm94] also applies to render contexts.
Furthermore, the *Template Method* pattern [Gamm94] is used throughout the RenderContext class, for example for creating render resource objects or setting render states. The Template Method pattern lets subclasses redefine certain tasks by overriding special hook methods provided by the base class. Additionally, the RenderContext class can be considered to use the *Collection Object* design pattern [Nobl96] because it is a collection for texture sampler stages, texture coordinate stages, render targets, and render resources.

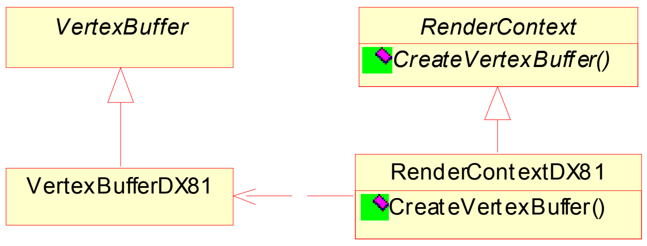

Figure 1.12: Abstract Factory Design Pattern Example

All the specific classes derived from the RenderResource and RenderTarget classes of XEngineCore in the renderer libraries, together with their base classes, represent the *Objectifier* pattern [Copl95], where an abstract class provides a common interface for specific implementations for variable behaviour. For example, the execution of the tasks that can be performed on a vertex buffer varies depending on the used renderer library. However, the client only keeps references to the abstract vertex buffer base class and need not concern itself with the specifics of the concrete vertex buffer classes in each renderer library. Thus the different behaviours of the specific vertex buffer classes is well encapsulated, which results in a highly configurable design required to provide rendering API-independence in XEngine.

All the mesh and scene loaders used in XEngineCore and XEngineSceneGraph, respectively, correspond to the *Builder* pattern [Gamm94]. The construction of complex mesh objects or even entire scene graphs, in the case of scene loaders, is separated from its representation, so that the same construction process can create different representations. The advantage of using this pattern for loading and constructing meshes and scene graphs is that the code for construction and the code for representation are isolated from one another. Figure 1.13 shows an example of the Builder design pattern where the MeshLoader3DS class is a builder to load and construct mesh objects from Autodesk 3D Studio files.

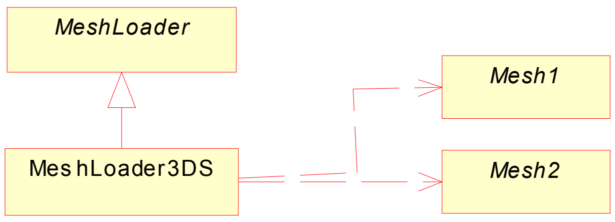

Figure 1.13: Builder Design Pattern Example

The Mesh class of XEngineCore uses the *Iterator* design pattern [Gamm94]. Mesh objects are complex, aggregate objects that consist of multiple primitive groups. Using the Iterator pattern, the Mesh class offers a special iterator object that provides a way for clients to iterate over the primitive groups without exposing the more complex underlying representation.

The entire scene graph data structure of XEngineSceneGraph follows the *Composite* pattern [Gamm94] which is used to compose complex objects into tree hierarchies. Clients can treat objects and compositions uniformly. This is exactly how the tree-like scene graph in XEngine is designed. Each branch node of the scene graph can be considered a composite of multiple child nodes that in turn can again be composites (that is, branch nodes). Additionally, because nodes can be cloned, XEngine's scene graph also follows the *Prototype* design pattern [Gamm94]. Each node can be considered a prototypical instance, and new instances can be created by copying this prototype. This design patterns is used because it is typically easier to simply make a deep copy of an entire scene graph hierarchy than to rebuild it from scratch if a clone of it is required.

The traversal handler chains used in the scene graph are designed according to the *Chain of Responsibility* pattern [Gamm94] which avoids coupling the scene graph data structure to the algorithms performed upon it. Also multiple handlers get a chance to perform tasks when the scene graph is traversed. However, the actual design slightly differs from the original Chain of Responsibility design pattern in that each traversal handler can handle a traversal, if desired, and the processing of the traversal handler chain of a node is not stopped when one of the traversal handler functions handled the traversal. In the original pattern, as soon as a request is handled, the processing of the chain is stopped.

Additionally, the traversal handler functions, which can be seen as receivers of traversal events with the currently visited node being the sender, follow the *Sender Argument* pattern [Nobl96] because they receive a reference to the currently visited node during a traversal as argument.

The Traversal class and its subclasses represent the *Strategy* design pattern [Gamm94] because these classes encapsulate the scene graph traversal algorithms and each algorithm can vary independently from the clients that use it. Each Traversal-derived class represents a different traversal algorithm. This pattern allows easy reuse of families of algorithms. Inheritance is used to factor out common functionality into the base class (see figure 1.11).

Finally, the RenderQueue and RenderOperation classes represent a simplified form of the *Command* design pattern [Gamm94]. A command, in this case the render operation, is encapsulated into a parameterized object. In theory, different kinds of render operations could be used in the render queue. In the current implementation, however, only one type of render operation which consists of a primitive group that is to be rendered and an associated state set is used.

1.7 Implementation

The previous sections outlined the design of the engine. This section shortly discusses the implementation of the engine, such as the third-party libraries it depends on, and various implementation details of interest.

1.7.1 Implementation Language and Required Tools

XEngine is implemented in ISO/ANSI C++ and uses various advanced features of the language, such as templates, runtime type information, and exception handling. Also the C++ Standard Template Library is used extensively. Since only in recent years C++ compilers have managed to fulfil the ISO/ANSI C++ standard to a high degree, a relatively new compiler is required to compile XEngine, for example Microsoft Visual C++ 6.0 or 7.0 or gcc 3.0 and up.

In addition to a C++ compiler, the lexer generator flex++ and the parser generator bison++ are required to rebuild the lexers and parsers used by the engine for the shading language cross-compilers. Both tools are C++ versions of the standard C flex and bison tools found on most Linux systems. Both generators take definition files as input which describe the to-be-generated lexer or parser, respectively. Based on templates defined via special skeleton C++ header and implementation files, a lexer and parser class that can be used in a C++ program are then generated. The skeleton files used in XEngine are heavily modified versions of the files that come with the tools to better suit the needs of the engine.

1.7.2 Depending Libraries

Since the author strongly believes in the open source idea and software reuse (Why reinvent the wheel when somebody else already did a better job than you could ever do?), XEngine depends on a rather large number of third-party, open source libraries. The use of some libraries can be prevented with configuration options of the engine's build system, effectively reducing the dependencies on other libraries.

1.7.2.1 wxWindows

wxWindows [wxWi03] is a mature, open source, cross-platform C++ framework, mostly specializing in the development of GUI applications, that is available on a number of platforms. Currently ports for all versions of Microsoft Windows, a number of Unix systems, such as Linux, Irix, and Solaris, MacOS 9 and MacOS X, and OS/2 are available. Additionally a number of language bindings other than C++ can be used, the most prominent being the Pyhton binding called wxPython. Other language bindings that are readily available or are actively being developed include bindings for Basic, Perl, Eiffel, JavaScript, all .NET languages, and Java. Apart from a variety of GUI-related classes for windows, dialogs, buttons, and other controls, wxWindows offers a large range of useful classes for cross-platform development, for example for file and stream handling, threads and thread synchronization, memory management, image processing, database access, and networking.

wxWindows is the layer that provides XEngine with platform-independence. Due to the fact that the engine tightly integrates with wxWindows, mostly because a render window can be used as a regular wxWindows window, it is very easy for regular wxWindows GUI applications to use the engine, allowing for quick prototyping and development of visualization applications, level editors, CAD-like programs, or other types of applications that use 3D graphics. Although it would theoretically be possible to completely decouple XEngine from wxWindows and use other cross-platform GUI frameworks such as Qt, it was never the intention to provide support for any other framework. The decision to use wxWindows as basis for the engine was made after a rather intensive analysis phase of freely available GUI frameworks, where wxWindows emerged as superior candidate in every respect. It is mature, feature-rich, license and royalty free, and has excellent documentation. Furthermore, it is still being actively maintained and developed, and has a large and helpful user community.

1.7.2.2 STLport

STLport [STLp03] is an open source implementation of the ISO/ANSI-standardized C++ Standard Template Library (or STL, for short) that is available on a number of platforms. It features thread and exception safety, a debug mode with rigorous runtime validity checking, and important, non-standard extensions, such as hash tables and singly-linked lists. The main reason for using STLport with XEngine, instead of the STL implementations that come with specific compilers, was to have a consistent, solid implementation for all compilers that are supported by the engine. Furthermore, XEngine makes heavy use of the hash tables provided by STLport (which originate from the SGI STL implementation that was made publicly and freely available by SGI in 1996).

1.7.2.3 Boost

Boost [Boos03] is a collection of free, peer-reviewed, cross-platform C++ libraries that emphasize good integration with the C++ Standard Template Library. One goal of the Boost project is to establish good practices and provide reference implementations which are suitable for eventual standardization and inclusion in an upcoming version of the C++ Standard Library by the ISO C++ Standards Committee. XEngine uses only a few of the Boost libraries, mostly libraries for automatic memory management that help to make code more exception safe. However, future versions of the engine might use the Boost signal and slots library and possibly the Python-binding library which provides a convenient way to create a Python language binding.

1.7.2.4 DevIL

DevIL, the *Developer's Image Library* [Wood02], is an open source image loading, saving, and processing library that can handle a large variety of image formats. The use of DevIL with XEngine is optional and can be turned off by using a configuration option of the engine's build system. When DevIL is used, texture objects can be directly created from image files in all the image formats supported by DevIL. Note that even if the use of DevIL is prevented, textures can still be created and loaded from image files directly. However, only the image formats supported by wxWindows are available.

1.7.2.5 lib3ds

lib3ds, the *3D Studio File Format Library* [Hoff02], is an open source library to load geometry data from files in the widespread Autodesk 3D Studio format. The mesh and scene loaders of the engine use this library to load geometry data from 3DS files and transform it into engine-internal data structures, such as mesh objects. The use of lib3ds is optional and can be turned off via a configuration option in the build system, in which case the engine can no longer load 3DS files directly.

1.7.3 Template Mix-In Classes

In the implementation of the renderer libraries one particular advanced C++ implementation technique is of special interest, so-called template mix-in classes. Template mix-ins are special template classes responsible for implementing a common interface which is shared by multiple classes. These classes, however, have nothing in common except for that interface. To achieve this, the template mix-in inherits from the to-be-implemented interface and receives a base class as template parameter from which the template inherits as well by multiple inheritance. This way multiple classes that all derive from different base classes can use the template mix-in by inheriting from it to derive from and at the same time implement the interface. In some cases, the template mix-in also immediately defines the interface it implements itself, which gets rid of the multiple inheritance. In the implementation of the renderer libraries template mix-in classes have proven to be a good way of preventing code doubling on several occasions. Figure 1.14 shows an exemplary class hierarchy in the DirectX renderer that uses template mix-ins.

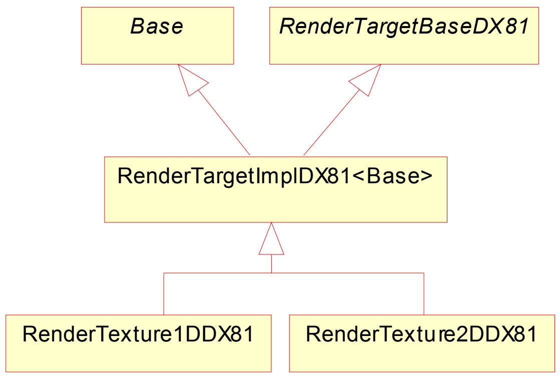

Figure 1.14: Echxemplary Class Hierarchy Using a Template Mix-In Class

In this example, RenderTargetImplDX81 is the template mix-in class that implements the RenderTargetBaseDX81 interface, which must be implemented by all RenderTarget-derived classes in the DirectX renderer. The abstract base class that the template mix-in also derives from, called Base in the figure, can be any RenderTarget-derived class of XEngineCore (see figure 1.8). Furthermore, the diagram depicts two exemplary concrete classes, RenderTexture1DDX81 and RenderTexture2DDX81 whi use the template mix-in class. The first instantiates the template using the RenderTexture1D abstract base class for the Base template parameter and the second instantiates the template with RenderTexture2D.

1.7.4 Cg Runtime

Implementing the shader system with its support for multiple shading languages across various graphics APIs was one of the key points of the implementation. The most important shading language supported by XEngine is NVIDIA's high-level language Cg, because it is, just like XEngine itself, API-independent and the Cg compiler can compile to various OpenGL-specific and the Direct3D low-level shading languages.

Because the initial releases of NVIDIA's Cg runtime, responsible for compiling and managing Cg shaders, were not very well designed and quite buggy, XEngine has its own internal Cg runtime that only requires the command-line version of the Cg compiler to work. The internal runtime uses various means, such as environment variables, registry settings, and some common system paths, to find the Cg compiler executable which is then executed with the given Cg shader source code to create object code using one of the supported Cg profiles. Unless the application explicitly specifies a profile that is to be used, the runtime tries to use the best profile in terms of feature-richness available. When compiling a Cg shader with the Cg compiler, the compiler not only outputs the shader object code in a low-level shading language but also various information originally intended for NVIDIA's Cg runtime that describes the variables used in the shader, including their type and binding to hardware registers. XEngine's internal Cg runtime parses this information and stores it in internal data structures that can later be used to verify the types specified by the application when it sets shader parameters using ShaderProgram::SetParameter or specifies vertex attribute bindings when a shader program is created.

Only recently, in December 2002, NVIDIA managed to release a stable version of the Cg runtime, which underwent a complete redesign and was incompatible with older releases. At present XEngine supports both its own internal Cg runtime and NVIDIA's Cg runtime, and applications can choose which is to be used. The advantage of using NVIDIA's runtime is that even new Cg profiles that might be introduced in future releases of the Cg compiler can be used, whereas for the internal Cg runtime support for new profiles must be explicitly implemented. The disadvantage is that the DLLs of NVIDIA's Cg runtime must be distributed with the application and the engine additionally depends on the Cg runtime libraries which it must be linked to. Also due to a small deficiency in NVIDIA's runtime, the ColorARGB data type for vertex attributes which is used for colours in ARGB format packed into a 32-bit word is only supported with XEngine's internal Cg runtime.

1.7.5 Shading Language Cross-Compilers

XEngine was designed to not only support Cg as shading language. Instead the engine is language-open, which means that it can potentially work together with any shading language, sometimes at the cost of losing 3D graphics API-independence. In its current state XEngine supports most of the wide-spread low-level shading languages, such as the Direct3D 8 low-level languages and a number of OpenGL low-level languages. A unique feature of the engine is the possibility of transparently cross-compiling the low-level languages from one graphics API to another. This allows the use of, for example, a shader written in the Direct3D vertex shader assembler with the OpenGL API, or the use of the same shader under Linux where the Direct3D API is not even available since it is Windows-only. The ulterior motive to develop such cross-compilers was the idea to be able to reuse old shaders with newer shading languages and different graphics APIs. For example, since Direct3D 8 was the first API to have low-level shading languages, a lot of shaders written in its low-level shading languages were readily available when similar OpenGL extension began to crop up. Due to the syntactical differences between the languages, the shaders had to be rewritten from scratch, which is a rather tedious and error-prone task. With XEngine, a shader need not be rewritten and can simply be reused as it is as long as a cross-compiler is available.

The shading language cross-compilers are implemented by using the lexer generator flex++ and the parser generator bison++. At present XEngine has the following integrated cross-compilers:

- Direct3D 8 vertex shader assembler versions 1.0 and 1.1 to NV_vertex_program
- Direct3D 8 vertex shader assembler versions 1.0 and 1.1 to ARB_vertex_program
- NV_vertex_program and NV_vertex_program1_1 to
  Direct3D 8 vertex shader assembler version 1.1
- ARB_vertex_program to Direct3D 8 vertex shader assembler version 1.1

- NV_vertex_program and NV_vertex_program1_1 to ARB_vertex_program
- Direct3D 8 pixel shader assembler versions 1.0, 1.1, 1.2, and 1.3 to NV_register_combiners/NV_texture_shader

The development of the parsers for the Direct3D 8 low-level shading languages was especially time-consuming due to the fact that the language grammars are not publicly available and the semantic restrictions on various shader instructions are not well documented. Therefore, owing to reverse engineering methods and trial and error the grammars for the vertex and pixel shader assembly languages had to be devised from scratch. In particular, determining all the semantic restrictions imposed on pixel shaders was hard, simply because of the large number of restrictions.

1.7.6 Fixed-Function Shaders

As mentioned a few times before, for XEngine the fixed-function pipeline is nothing more than a special shader program that consists of a number of shaders, one for each programmable pipeline stage/shader type. Each shader uses the special shading language FixedFunction. This is not only a characteristic of the design, but is also evident in the implementation in that all renderer libraries contain special Shader-derived classes representing the fixed-function shaders. Figure 1.15 shows a UML class diagram that depicts the fixed-function shader classes for both the Direct3D 8 and the OpenGL renderer.

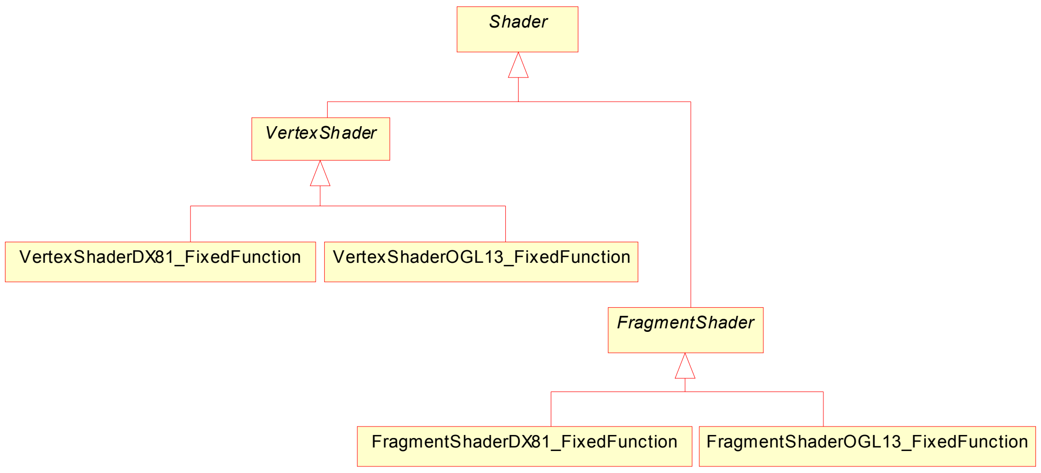
Shader
VertexShader
VertexShaderDX81_FixedFunction
VertexShaderOGL13_FixedFunction
FragmentShader
FragmentShaderDX81_FixedFunction
FragmentShaderOGL13_FixedFunction

Figure 1.15: UML Class Diagram of the Fixed-Function Shader Hierarchy

In earlier versions of XEngine the concrete ShaderProgram-derived classes in the renderer libraries took care of fixed-function pipeline functionality, but this caused a lot of special cases in the implementation. After a restructuring process the current fixed-function shader classes were introduced which simplified the implementation of the shader system in the renderer libraries to a large extent.

Bibliography

[Boos03] *Boost C++ Libraries*, http://www.boost.org

[Copl95] Jim O. Coplien, James O. Coplien, Douglas C. Schmidt: *Pattern Languages of Program Design 1*, Addison-Wesley, 1995

[Gamm94] Erich Gamma, Richard Helm, John Vlissides, Ralph Johnson: *Design Patterns. Elements of Reusable Object Oriented Software*, Addison-Wesley, 1994

[Hitz99] Martin Hitz, Gerti Kappel: *UML@Work: Von der Analyse zur Realisierung*, dpunkt-Verlag, Heidelberg, 1999

[Hoff02] J. E. Hoffmann: *The 3D Studio File Format Library Homepage*, http://lib3ds.sourceforge.net

[Land95] Niklas Landin, Axel Niklasson: *Development of Object-Oriented Frameworks*, Department of Communication Systems, Lund Institute of Technology, Lund University, 1995

[Nobl96] James Noble: *Some Patterns for Relationships*, Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS Pacific 21), Melbourne, 1996

[STLp03] *STLport Standard Library Project*, http://www.stlport.org

[Wood02] Denton Woods: *DevIL, Developer's Image Library Manual*, http://openil.sourceforge.net, 2002

[wxWi03] *wxWindows, Cross-Platform GUI Framework*, http://www.wxwindows.org