3D Engine Design for Virtual Globes

Patrick Cozzi and Kevin Ring

Editorial, Sales, and Customer Service Office

A K Peters, Ltd.
5 Commonwealth Road, Suite 2C
Natick, MA 01760
www.akpeters.com

Library of Congress Cataloging-in-Publication Data

To be determined

Printed in the United States of America
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

3 Renderer Design

Some graphics applications start off life with OpenGL or Direct3D calls sprinkled throughout. For small projects, this is manageable, but as projects grow in size, developers start asking, How can we cleanly manage OpenGL state?; How can we make sure everyone is using OpenGL best practices?; or even, How do we go about supporting both OpenGL and Direct3D?

The first step to answering these questions is abstraction—more specifically, abstracting the underlying rendering API such as OpenGL or Direct3D using interfaces that make most of the application's code API-agnostic. We call such interfaces and their implementation a renderer. This chapter describes the design behind the renderer in OpenGlobe. First, we pragmatically consider the motivation for a renderer, then we look at the major components of our renderer: state management, shaders, vertex data, textures, and framebuffers. Finally, we look at a simple example that renders a triangle using our renderer.

If you have experience using a renderer, you may just want to skim this chapter and move on to the meat of virtual globe rendering. Examples in later chapters build on our renderer, so some familiarity with it is required.

This chapter is not a tutorial on OpenGL or Direct3D, so you need some background in one API. Nor is this a description of how to wrap every OpenGL call in an object-oriented wrapper. We are doing much more than wrapping functions; we are raising the level of abstraction.

Our renderer contains quite a bit of code. To keep the discussion focused, we only include the most important and relevant code snippets in these pages. Refer to the code in the OpenGlobe.Renderer project for the full implementation. In this chapter, we are focused on the organization of the public interfaces and the design trade-offs that went into them; we are not concerned with minute implementation details.

Throughout this chapter, when we refer to GL, we mean OpenGL 3.3 core profile specifically. Likewise, when we refer to D3D, we mean Direct3D

11 specifically. Also, we define client code as code that calls the renderer, for example, application code that uses the renderer to issue draw commands.

Finally, software design is often subjective, and there is rarely a single best solution. We want you to view our design as something that has worked well for us in virtual globes, but as only one of a myriad approaches to renderer design.

3.1 The Need for a Renderer

Given that APIs such as OpenGL and Direct3D are already an abstraction, it is natural to ask, why build a renderer layer in our engine at all? Aren't these APIs sufficiently high level enough to use directly throughout our engine?

Many small projects do scatter API calls throughout their code, but as projects get larger, it is important that they properly abstract the underlying API for many reasons:

- *Ease of development.* Using a renderer is almost always easier and more concise than calling the underlying API directly. For example, in GL, the process of compiling and linking a shader and retrieving its uniforms can be simplified to a single constructor call on a shader program abstraction.

  Since most engines are written in object-oriented languages, a renderer allows us to present the procedural GL API using object-oriented constructs. For example, constructors invoke `glCreate*` or `glGen*`, and destructors invoke `glDelete*`, allowing the C# garbage collector to handle resource lifetime management, freeing client code from having to explicitly delete renderer resources.[1]

  Besides conciseness and object orientation, a renderer can simplify development by minimizing or completely eliminating error-prone global states, such as the depth and stencil tests. A renderer can group these states into a coarse-grained render state, which is provided per draw call, eliminating the need for global state.

  When using Direct3D 9, a renderer can also hide the details of handling a "lost device," where the GPU resources are lost due to a user changing the window to or from full screen, a laptop's cover opening/closing, etc. The renderer implementation can shadow a copy of

[1] In C++, similar lifetime management can be achieved with smart pointers. Our renderer abstractions implement IDisposable, which gives client code the option to explicitly free an object's resources in a deterministic manner instead of relying on the garbage collector.

GPU resources in system memory, so it can restore them in response to a lost device, without any interaction from client code.

- *Portability.* A renderer greatly reduces, but does not eliminate, the burden of supporting multiple APIs. For example, an engine may want to use Direct3D on Windows, OpenGL on Linux, OpenGL ES on mobile devices,[2] and LibGCM on PlayStation 3. To support different APIs on different platforms, a different renderer implementation can be swapped in, while the majority of the engine code remains unchanged. Some renderer implementations, such as a GL renderer and GL ES renderer or a GL 3.x renderer and GL 4.x renderer, may even share a good bit of code.

  A renderer also makes it easier to migrate to new versions of an API or new GL extensions. For example, when all GL calls are isolated, it is generally straightforward to replace global-state selectors with direct-state access (EXT_direct_state_access [91]). Likewise, the renderer can decide if an extension is available or not and take appropriate action. Supporting some new features or extension requires exposing new or different public interfaces; for example, consider how one would migrate from GL uniforms to uniform buffers.

- *Flexibility.* A renderer allows a great deal of flexibility since a renderer's implementation can be changed largely independent of client code. For example, if it is more efficient to use GL display lists[3] than vertex buffer objects (VBOs) on certain hardware, that optimization can be made in a single location. Likewise, if a bug is found that was caused by a misunderstanding of a GL call or by a driver bug, the fix can be made in one location, usually without impacting client code.

  A renderer helps new code plug into an engine. Without a renderer, a call to a virtual method may leave GL in an unknown state. The implementor of such a method may not even work for the same company that developed the engine and may not be aware of the GL conventions used. This problem can be avoided by passing a renderer to the method, which is used for all rendering activities. A renderer enables the flexibility of engine "plug-ins" that are as seamless as core engine code.

- *Robustness.* A renderer can improve an engine's robustness by providing statistics and debugging aids. In particular, it is easy to count

[2] With ARB_ES2_compatibility, OpenGL 3.x is now a superset of OpenGL ES 2.0 [19]. This simplifies porting and sharing code between desktop OpenGL and OpenGL ES.

[3] Display lists were deprecated in OpenGL 3, although they are still available through the compatibility profile.

the number of draw calls and triangles drawn per frame when GL commands are isolated in a renderer. It can also be worthwhile to have an option to log underlying GL calls for later debugging. Likewise, a renderer can easily save the contents of the framebuffer and textures, or show the GL state at any point in time. When run in debug mode, each GL call in the renderer can be followed by a call to `glGetError` to get immediate feedback on errors.[4]

Many of these debugging aids are also available through third-party tools, such as BuGLe,[5] GLIntercept,[6] and gDEBugger.[7] These tools track GL calls to provide debugging and performance information, similar to what can be done in a renderer.

- *Performance.* At first glance, it may seem that a renderer layer can hurt performance. It does add a lot of virtual methods calls. However, considering the amount of work the driver is likely to do, virtual call overhead is almost never a concern. If it were, virtual calls aren't even required to implement a renderer unless the engine supports changing rendering APIs at runtime, an unlikely requirement. Therefore, a renderer can be implemented with plain or inline methods if desired.

  A renderer can actually help performance by allowing optimizations to be made in a single location. Client code doesn't need to be aware of GL best practices; only the renderer implementation does. The renderer can shadow GL state to eliminate redundant state changes and avoid expensive calls to `glGet*`. Depending on the level of abstraction chosen for the renderer, it can also optimize vertex and index buffers for the GPU's caches and select optimal vertex formats with proper alignment for the target hardware. Renderer abstractions can also make it easier to sort by state, a commonly used optimization.

  For engines written in a managed language like Java or C#, such as OpenGlobe, a renderer can improve performance by minimizing the managed-to-native-code round trip overhead. Instead of calling into native GL code for every fine-grained call, such as changing a uniform or a single state, a single coarse-grained call can pass a large amount of state to a native C++ component that does several GL calls.

- *Additional functionality.* A renderer layer is the ideal place to add functionality that isn't in the underlying API. For example, Sec-

[4] If ARB_debug_output is supported, calls to `glGetError` can be replaced with a callback function [93].
[5] http://sourceforge.net/projects/bugle/
[6] http://glintercept.nutty.org/
[7] http://www.gremedy.com/

tion 3.4.1 introduces additional built-in GLSL constants that are not part of the GLSL language, and Section 3.4.5 introduces GLSL uniforms that are not built into GLSL but are still set automatically at draw time by the renderer. A renderer doesn't just wrap the underlying API; it raises the level of abstraction and provides additional functionality.

Even with all the benefits of a renderer, there is an important pitfall to watch for:

- *A false sense of portability.* Although a renderer eases supporting multiple APIs, it does not completely eliminate the pain. David Eberly explains his experience with Wild Magic: "After years of maintaining an abstract rendering API that hides DirectX, OpenGL, and software rendering, the conclusion is that each underlying API suffers to some extent from the abstraction" [45]. No renderer is a "one size fits all" solution. We freely admit that the renderer described in this chapter is biased to OpenGL as we've not implemented it with Direct3D yet.

  One prominent concern is that having both GL and D3D implementations of the renderer requires us to maintain two versions of all shaders: a GLSL version for the GL renderer and an HLSL version for the D3D renderer. Given that shader languages are so similar, it is possible to use a tool to convert between languages, even at runtime. For example, HLSL2GLSL,[8] a tool from AMD, converts D3D9 HLSL shaders to GLSL. A modified version of this tool, HLSL2GLSLFork,[9] maintained by Aras Pranckevičius, is used in Unity 3.0. The Google ANGLE[10] project translates in the opposite direction, from GLSL to D3D9 HLSL.

  To avoid conversions, shaders can be written in NVIDIA's Cg, which supports both GL and D3D. A downside is that the Cg runtime is not available for mobile platforms at the time of this writing.

  Ideally, using a renderer would avoid the need for multiple code paths in client code. Unfortunately, this is not always possible. In particular, if different generations of hardware are supported with different renderers, client code may also need multiple code paths. For example, consider rendering to a cube map. If a renderer is implemented using GL 3, geometry shaders will be available, so the cube map can be rendered in a single pass. If the renderer is implemented with

[8] http://sourceforge.net/projects/hlsl2glsl/
[9] http://code.google.com/p/hlsl2glslfork/
[10] http://code.google.com/p/angleproject/

an older GL version, each cube-map face needs to be rendered in a separate pass.

A renderer is such an important piece of an engine that most game engines include a renderer of some sort, as do applications like Google Earth. It is fair to ask, if a renderer is so important, why does everyone roll their own? Why isn't there one renderer that is in widespread use? Because different engines prefer different renderer designs. Some engines want low-level, nearly one-to-one mappings between renderer calls and GL calls, while other engines want very high-level abstractions, such as effects. A renderer's performance and features tend to be tuned for the application it is designed for.

Patrick Says

When writing an engine, consider using a renderer from the start. In my experience, taking an existing engine with GL calls scattered throughout and refactoring it to use a renderer is a difficult and error-prone endeavor. When we started on Insight3D, one of my first tasks was to replace many of the GL calls in the existing codebase we were leveraging with calls to a new renderer. Even with all the debug code I included to validate GL state, I injected my fair share of bugs.

Although developing software by starting with solid foundations and building on top is much easier than retrofitting a large codebase later, do not fall into the trap of doing architecture for architecture's sake. A renderer's design should be driven by actual use cases.

3.2 Bird's-Eye View

A renderer is used to create and manipulate GPU resources and issue rendering commands. Figure 3.1 shows our renderer's major components. A small amount of render state configures the fixed-function components of the pipeline for rendering. Given that we are using a fully shader-based design, there isn't much render state, just things like depth and stencil testing. The render state doesn't include legacy fixed-function states that can be implemented in shaders like per-vertex lighting and texture environments.

Shader programs describe the vertex, geometry, and fragment shaders used to execute draw calls. Our renderer also includes types for communicating with shaders using vertex attributes and uniforms.

A vertex array is a lightweight container object that describes vertex attributes used for drawing. It receives data for these attributes through