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

Contents

1 Introduction

Virtual globes are known for their ability to render massive real-world terrain, imagery, and vector datasets. The servers providing data to virtual globes such as Google Earth and NASA World Wind host datasets measuring in the terabytes. In fact, in 2006, approximately 70 terabytes of compressed imagery were stored in Bigtable to serve Google Earth and Google Maps [24]. No doubt, that number is significantly higher today.

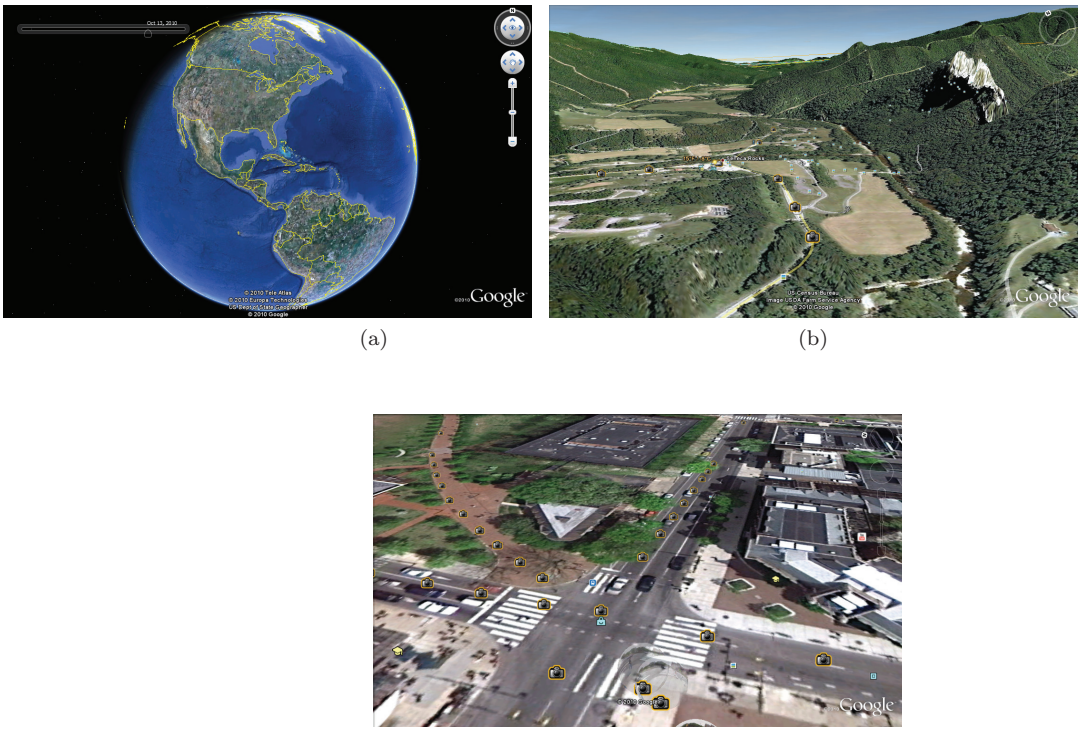
Obviously, implementing a 3D engine for virtual globes requires careful management of these datasets. Storing the entire world in memory and brute force rendering are certainly out of the question. Virtual globes, though, face additional rendering challenges beyond massive data management. This chapter presents these unique challenges and paves the way forward.

1.1 Rendering Challenges in Virtual Globes

In a virtual globe, one moment the viewer may be viewing Earth from a distance (see Figure 1.1(a)); the next moment, the viewer may zoom in to a hilly valley (see Figure 1.1(b)) or to street level in a city (see Figure 1.1(c)). All the while, real-world data appropriate for the given view are paged in and precisely rendered.

The freedom of exploration and the ability to visualize incredible amounts of data give virtual globes their appeal. These factors also lead to a number of interesting and unique rendering challenges:

- *Precision.* Given the sheer size of Earth and the ability for users to view the globe as a whole or zoom in to street level, virtual globes require a large view distance and large world coordinates. Trying to render a massive scene by naïvely using a very close near plane; very



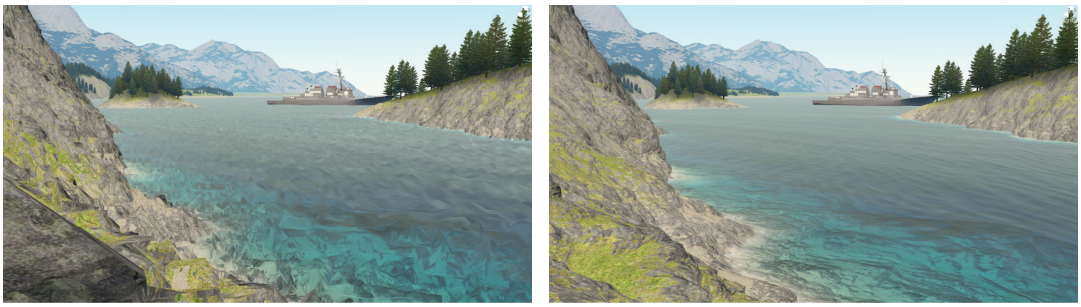
(a) (b)

(c)

Figure 1.1. Virtual globes allow viewing at varying scales: from (a) the entire globe to (b) and (c) street level. (a) © 2010 Tele Atlas; (b) © 2010 Europa Technologies, US Dept of State Geographer; (c) © 2010 Google, US Census Bureau, Image USDA Farm Service Agency. (Images taken using Google Earth.)

distant far plane; and large, single-precision, floating-point coordinates leads to z-fighting artifacts and jittering, as shown in Figures 1.2 and 1.3. Both artifacts are even more noticeable as the viewer moves. Strategies for eliminating these artifacts are presented in Part II.

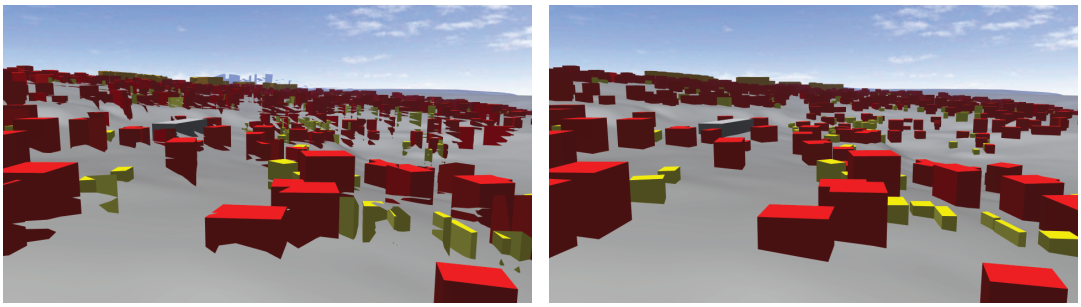
- *Accuracy.* In addition to eliminating rendering artifacts caused by precision errors, virtual globes should also model Earth accurately. Assuming Earth is a perfect sphere allows for many simplifications, but Earth is actually about 21 km longer at the equator than at the poles. Failing to take this into account introduces errors when positioning air and space assets. Chapter 2 describes the related mathematics.



(a) (b)

Figure 1.2. (a) Jitter artifacts caused by precision errors in large worlds. Insufficient precision in 32-bit floating-point numbers creates incorrect vertex positions. (b) Without jittering. (Images courtesy of Brano Kemen, Outerra.)

- *Curvature.* The curvature of Earth, whether modeled with a sphere or a more accurate representation, presents additional challenges compared to many graphics applications where the world is extruded from a plane (see Figure 1.4): lines in a planar world are curves on Earth, oversampling can occur as latitude approaches 90° and −90°, a singularity exists at the poles, and special care is often needed to handle the International Date Line. These concerns are addressed throughout this book, including in our discussion of globe rendering in Chapter 4, polygons in Chapter 8, and mapping geometry clipmapping to a globe in Chapter 13.



(a) (b)

Figure 1.3. (a) Z-fighting and jittering artifacts caused by precision errors in large worlds. In z-fighting, fragments from different objects map to the same depth value, causing tearing artifacts. (b) Without z-fighting and jittering. (Images courtesy of Aleksandar Dimitrijević, University of Niš.)

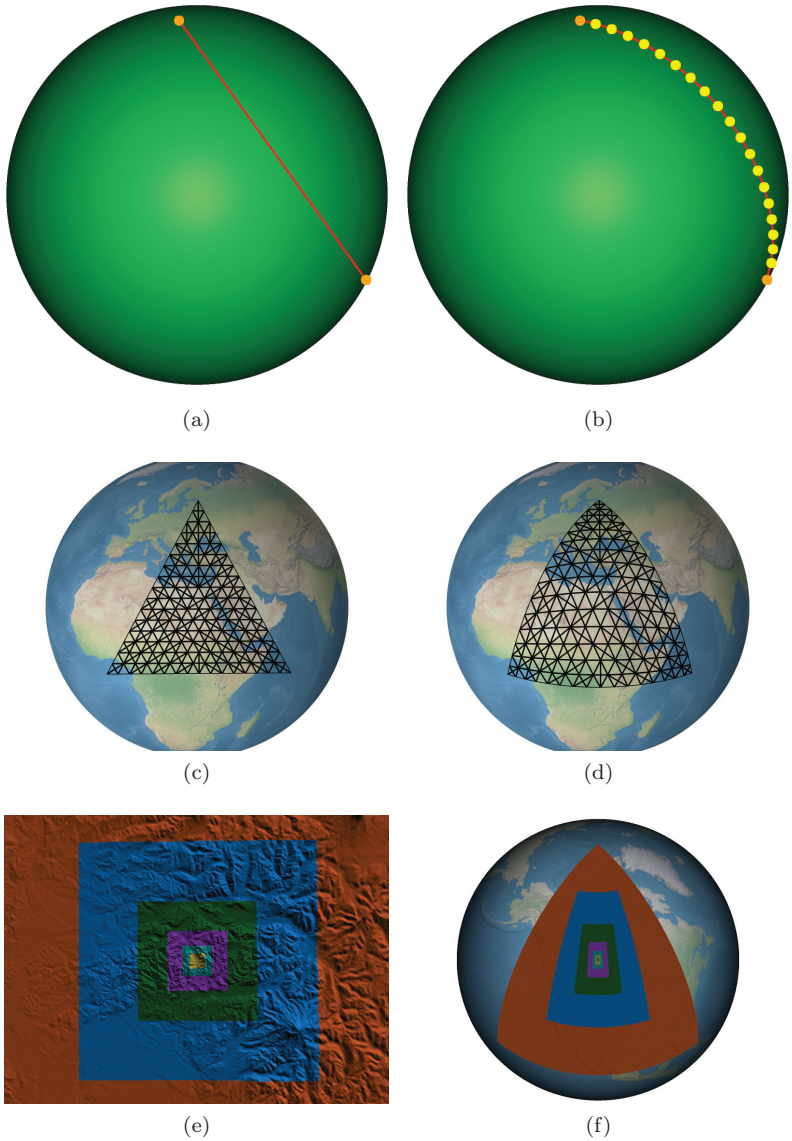


Figure 1.4. (a) Lines connecting surface points cut underneath a globe; instead, (b) points should be connected with a curve. Likewise, (c) polygons composed of triangles cut under a globe unless (d) curvature is taken into account. Mapping flat-world algorithms, (e) like geometry clipmapping terrain, to a globe can lead to (f) oversampling near the poles. (a) and (c) are shown without depth testing. (b) and (d) use the depth-testing technique presented in Chapter 7 to avoid z-fighting with the globe.

- *Massive datasets.* Real-world data have significant storage requirements. Typical datasets will not fit into GPU memory, system memory, or a local hard disk. Instead, virtual globes rely on server-side data that are paged in based on view parameters using a technique called *out-of-core rendering*, which is discussed in the context of terrain in Chapter 12 and throughout Part IV.

- *Multithreading.* In many applications, multithreading is considered to be only a performance enhancement. In virtual globes, it is an essential part of the 3D engine. As the viewer moves, virtual globes are constantly paging in data and processing it for rendering. Doing so in the rendering thread causes severe stalls, making the application unusable. Instead, virtual globe resources are loaded and processed in one or more separate threads, as discussed in Chapter 10.

- *Few simplifying assumptions.* Given their unrestrictive nature, virtual globes cannot take advantage of many of the simplifying assumptions that other graphics applications can.

  A viewer may zoom from a global view to a local view or vice versa in an instant. This challenges techniques that rely on controlling the viewer's speed or viewable area. For example, flight simulators know the plane's top speed and first-person shooters know the player's maximum running speed. This knowledge can be used to prefetch data from secondary storage. With the freedom of virtual globes, these techniques become more difficult.

  Using real-world data also makes procedural techniques less applicable. The realism in virtual globes comes from higher-resolution data, which generally cannot be synthesized at runtime. For example, procedurally generating terrains or clouds can still be done, but virtual globe users are most often interested in *real* terrains and clouds.

This book address these rendering challenges and more.

1.2 Contents Overview

The remaining chapters are divided into four parts: fundamentals, precision, vector data, and terrain.

1.2.1 Fundamentals

The fundamentals part contains chapters on low-level virtual globe components and basic globe rendering algorithms.

- *Chapter 2: Math Foundations.* This chapter introduces useful math for virtual globes, including ellipsoids, common virtual globe coordinate systems, and conversions between coordinate systems.

- *Chapter 3: Renderer Design.* Many 3D engines, including virtual globes, do not call rendering APIs such as OpenGL directly, and instead use an abstraction layer. This chapter details the design rationale behind the renderer in our example code.

- *Chapter 4: Globe Rendering.* This chapter presents several fundamental algorithms for tessellating and shading an ellipsoidal globe.

1.2.2 Precision

Given the massive scale of Earth, virtual globes are susceptible to rendering artifacts caused by precision errors that many other 3D applications are not. This part details the causes and solutions to these precision problems.

- *Chapter 5: Vertex Transform Precision.* The 32-bit precision on most of today's GPUs can cause objects in massive worlds to jitter, that is, literally bounce around in a jerky manner as the viewer moves. This chapter surveys several solutions to this problem.

- *Chapter 6: Depth Buffer Precision.* Since virtual globes call for a close near plane and a distant far plane, extra care needs to be taken to avoid z-fighting due to the nonlinear nature of the depth buffer. This chapter presents a wide range of techniques for eliminating this artifact.

1.2.3 Vector Data

Vector data, such as political boundaries and city locations, give virtual globes much of their richness. This part presents algorithms for rendering vector data and multithreading techniques to relieve the rendering thread of preparing vector data, or resources in general.

- *Chapter 7: Vector Data and Polylines.* This chapter includes a brief introduction to vector data and geometry-shader-based algorithms for rendering polylines.

- *Chapter 8: Polygons.* This chapter presents algorithms for rendering filled polygons on an ellipsoid using a traditional tessellation and subdivision approach and rendering filled polygons on terrain using shadow volumes.

- *Chapter 9: Billboards.* Billboards are used in virtual globes to display text and highlight places of interest. This chapter covers geometry-shader-based billboards and texture atlas creation and usage.

- *Chapter 10: Exploiting Parallelism in Resource Preparation.* Given the large datasets used by virtual globes, multithreading is a must. This chapter reviews parallelism in computer architecture, presents software architectures for multithreading in virtual globes, and demystifies multithreading in OpenGL.

1.2.4 Terrain

At the heart of a virtual globe is a terrain engine capable of rendering massive terrains. This final part starts with terrain fundamentals, then moves on to rendering real-world terrain datasets using level of detail (LOD) and out-of-core techniques.

- *Chapter 11: Terrain Basics.* This chapter introduces height-map-based terrain with a discussion of rendering algorithms, normal computations, and shading, both texture-based and procedural.

- *Chapter 12: Massive-Terrain Rendering.* Rendering real-world terrain accurately mapped to an ellipsoid requires the techniques discussed in this chapter, including LOD, culling, and out-of-core rendering. The next two chapters build on this material with specific LOD algorithms.

- *Chapter 13: Geometry Clipmapping.* Geometry clipmapping is an LOD technique based on nested, regular grids. This chapter details its implementation, as well as out-of-core and ellipsoid extensions.

- *Chapter 14: Chunked LOD.* Chunked LOD is a popular terrain LOD technique that uses hierarchical levels of detail. This chapter discusses its implementation and extensions.

There is also an appendix on implementing a message queue for communicating between threads.

We've ordered the parts and chapters such that the book flows from start to finish. You don't have to read the chapters in order though; we certainly didn't write them in order. Just ensure you are familiar with the terms and high level-concepts in Chapters 2 and 3, then jump to the chapter that interests you most. The text contains cross-references so you know where to go for more information.

There are *Patrick Says* and *Kevin Says* boxes throughout the text. These are the voices of the individual authors and are used to tell a story,

usually an implementation war story, or to inject an opinion without clouding the main text. We hope these lighten up the text and provide deeper insight into our experiences.

The text also includes *Question* and *Try This* boxes that provide questions to think about and modifications or enhancements to make to the example code.

1.3 OpenGlobe Architecture

A large amount of example code accompanies this book. These examples were written from scratch, specifically for this book. In fact, just as much effort went into the example code as went into the book you hold in your hands. As such, treat the examples as an essential part of your learning—take the time to run them and experiment. Tweaking code and observing the result is time well spent.

Together, the examples form a solid foundation for a 3D engine designed for virtual globes. As such, we've named the example code *OpenGlobe* and provide it under the liberal MIT License. Use it as is in your commercial products or select bits and pieces for your personal projects. Download it from our website: http://www.virtualglobebook.com/.

The code is written in C# using OpenGL[1] and GLSL. C#'s clean syntax and semantics allow us to focus on the graphics algorithms without getting bogged down in language minutiae. We've avoided lesser-known C# language features, so if your background is in another object-oriented language, you will have no problem following the examples. Likewise, we've favored clean, concise, readable code over micro-optimizations.

Given that the OpenGL 3.3 core profile is used, we are taking a modern, fully shader-based approach. In Chapter 3, we build an abstract renderer implemented with OpenGL. Later chapters use this renderer, nicely tucking away the OpenGL API details so we can focus on virtual globe and terrain specifics.

OpenGlobe includes implementations for many of the presented algorithms, making the codebase reasonably large. Using the conservative metric of counting only the number of semicolons, it contains over 16,000 lines of C# code in over 400 files, and over 1,800 lines of GLSL code in over 80 files. We strongly encourage you to build, run, and experiment with the code. As such, we provide a brief overview of the engine's organization to help guide you.

OpenGlobe is organized into three assemblies:[2] OpenGlobe.Core.dll, OpenGlobe.Renderer.dll, and OpenGlobe.Scene.dll. As shown in Figure 1.5,

[1] OpenGL is accessed from C# using OpenTK: http://www.opentk.com/.

[2] *Assembly* is the .NET term for a compiled code library (i.e., an .exe or .dll file).

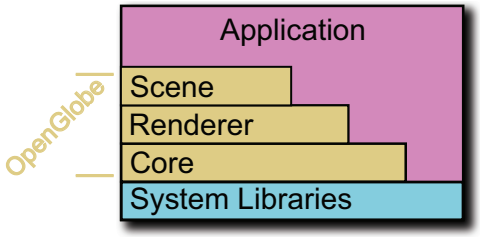


Figure 1.5. The stack of OpenGlobe assemblies.

these assemblies are layered such that Renderer depends on Core, and Scene depends on Renderer and Core. All three assemblies depend on the .NET system libraries, similar to how an application written in C depends on the C standard library.

Each OpenGlobe assembly has types that build on its dependent assemblies:

- *Core.* The Core assembly exposes fundamental types such as vectors, matrices, geographic positions, and the Ellipsoid class discussed in Chapter 2. This assembly also contains geometric algorithms, including the tessellation algorithms presented in Chapters 4 and 8, and engine infrastructure, such as the message queue discussed in Appendix A.

- *Renderer.* The Renderer assembly contains types that present an abstraction for managing GPU resources and issuing draw calls. Its design is discussed in depth in Chapter 3. Instead of calling OpenGL directly, an application built using OpenGlobe uses types in this assembly.

- *Scene.* The Scene assembly contains types that implement rendering algorithms using the Renderer assembly. This includes algorithms for globes (see Chapter 4), vector data (see Chapters 7–9), terrain shading (see Chapter 11), and geometry clipmapping (see Chapter 13).

Each assembly exposes types in a namespace corresponding to the assembly's filename. Therefore, there are three public namespaces: `OpenGlobe.Core`, `OpenGlobe.Renderer`, and `OpenGlobe.Scene`.

An application may depend on one, two, or all three assemblies. For example, a command line tool for geometric processing may depend just on Core, an application that implements its own rendering algorithms may depend on Core and Renderer, and an application that uses high-level objects like globes and terrain would depend on all three assemblies.

The example applications generally fall into the last category and usually consist of one main .cs file with a simple `OnRenderFrame` implementation that clears the framebuffer and issues `Render` for a few objects created from the Scene assembly.

OpenGlobe requires a video card supporting OpenGL 3.3, or equivalently, Shader Model 4. These cards came out in 2006 and are now very reasonably priced. This includes the NVIDIA GeForce 8 series or later and ATI Radeon 2000 series or later GPUs. Make sure to upgrade to the most recent drivers.

All examples compile and run on Windows and Linux. On Windows, we recommend building with any version of Visual C# 2010, including the free Express Edition.[3] On Linux, we recommend MonoDevelop.[4] We have tested on Windows XP, Vista, and 7, as well as Ubuntu 10.04 and 10.10 with Mono 2.4.4 and 2.6.7, respectively. At the time of this writing, OpenGL 3.3 drivers were not available on OS X. Please check our website for the most up-to-date list of supported platforms and integrated development environments (IDEs).

To build and run, simply open `Source\OpenGlobe.sln` in your .NET development environment, build the entire solution, then select an example to run.

We are committed to filling these pages with descriptive text, figures, and tables, not verbose code listing upon listing. Therefore, we've tried to provide relevant, concise code listings that supplement the core content. To keep listings concise, some error checking may be omitted, and #version 330 is always omitted in GLSL code. The code on our website includes full error checking and #version directives.

1.4 Conventions

This book uses a few conventions. Scalars and points are lowercase and italicized (e.g., s and p), vectors are bold (e.g., $\mathbf{v}$), normalized vectors also have a hat over them (e.g., $\hat{\mathbf{n}}$), and matrices are uppercase and bold (e.g., $\mathbf{M}$).

Unless otherwise noted, units in Cartesian coordinates are in meters (m). In text, angles, such as longitude and latitude, are in degrees (°). In code examples, angles are in radians because C# and GLSL functions expect radians.

[3] http://www.microsoft.com/express/Windows/
[4] http://monodevelop.com/

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

Try This

Run the night-lights example with a frame-rate utility. Note the frame rate when viewing just the daytime side of the globe and just the nighttime side. Why is the frame rate higher for the nighttime side? Because the night-lights texture is a lower resolution than the daytime texture and does not require any lighting computations. This shows using dynamic branching to improve performance.

Virtual globe applications use real-world data like this night-light texture derived from satellite imagery. On the other hand, video games generally focus on creating a wide array of convincing artificial data using very little memory. For example, EVE Online takes an interesting approach to rendering night lights for their planets [109]. Instead of relying on a night-light texture whose texels are directly looked up, a texture atlas of night lights is used. The spherically mapped texture coordinates are used to look up surrogate texture coordinates, which map into the texture atlas. This allows a lot of variation from a single texture atlas because sections can be rotated and mirrored.

Rendering night lights is one of many uses for multitexturing in globe rendering. Other uses include cloud textures and gloss maps to show specular highlights on bodies of waters [57,147]. Before the multitexturing hardware was available, effects like these required multiple rendering passes. STK, being one of the first products to implement night lights, uses a multiple-pass approach.

4.3 GPU Ray Casting

GPUs are built to rasterize triangles at very rapid rates. The purpose of ellipsoid-tessellation algorithms is to create triangles that approximate the shape of a globe. These triangles are fed to the GPU, which rapidly rasterizes them into shaded pixels, creating an interactive visualization of the globe. This process is very fast because it is embarrassingly parallel; individual triangles and fragments are processed independently, in a massively parallel fashion. Since tessellation is required, rendering a globe this way is not without its flaws:

- No single tessellation is perfect; each has different strengths and weaknesses.
- Under-tessellation leads to a coarse triangle mesh that does not approximate the surface well, and over-tessellation creates too many

triangles, negatively affecting performance and memory usage. View-dependent level-of-detail algorithms are required for most applications to strike a balance.

- Although GPUs exploit the parallelism of rasterization, memories are not keeping pace with the increasing computation power, so a large number of triangles can negatively impact performance. This is especially true of some level-of-detail algorithms where new meshes are frequently sent over the system bus.

Ray tracing is an alternative to rasterization. Rasterization starts with triangles and ends with pixels. Ray tracing takes the opposite approach: it starts with pixels and asks what triangle(s), or objects in general, contribute to the color of this pixel. For perspective views, a ray is cast from the eye through each pixel into the scene. In the simplest case, called *ray casting*, the first object intersecting each ray is found, and lighting computations are performed to produce the final image.

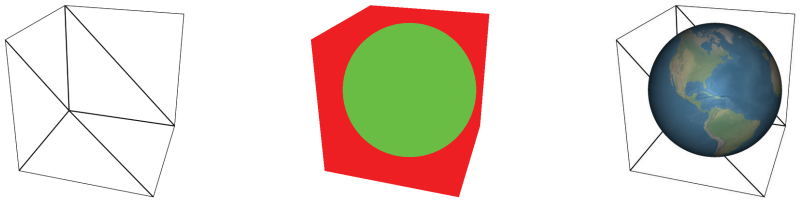
A strength of ray casting is that objects do not need to be tessellated into triangles for rendering. If we can figure out how to intersect a ray with an object, then we can render it. Therefore, no tessellation is required to render a globe represented by an ellipsoid because there is a well-known equation for intersecting a ray with an ellipsoid's implicit surface. The benefits of ray casting a globe include the following:

- The ellipsoid is automatically rendered with an infinite level of detail. For example, as the viewer zooms in, the underlying triangle mesh does not become apparent because there is no triangle mesh; intersecting a ray with an ellipsoid produces an infinitely smooth surface.

- Since there are no triangles, there is no concern about creating thin triangles, triangles crossing the poles, or triangles crossing the IDL. Many of the weaknesses of tessellation algorithms go away.

- Significantly less memory is required since a triangle mesh is not stored or sent across the system bus. This is particularly important in a world where size is speed.

Since current GPUs are built for rasterization, you may wonder how to efficiently ray cast a globe. In a naïve CPU implementation, a nested `for` loop iterates over each pixel in the scene and performs a ray/ellipsoid intersection. Like rasterization, ray casting is embarrassingly parallel. Therefore, a wide array of optimizations are possible on today's CPUs, including casting each ray in a separate thread and utilizing single instruction multiple data (SIMD) instructions. Even with these optimizations, CPUs



(a) (b) (c)

Figure 4.17. In GPU ray casting, (a) a box is rendered to (b) invoke a ray-casting fragment shader that finds the ellipsoid's visible surface. When an intersection is found, (c) the geodetic surface normal is used for shading.

do not support the massive parallelism of GPUs. Since GPUs are built for rasterization, the question is how do we use them for efficient ray casting?

Fragment shaders provide the perfect vehicle for ray casting on the GPU. Instead of tessellating an ellipsoid, create geometry for a bounding box around the ellipsoid. Then, render this box using normal rasterization and cast a ray from the eye to each fragment created by the box. If the ray intersects the inscribed ellipsoid, shade the fragment; otherwise, discard it.

The box is rendered with front-face culling, as shown in Figure 4.17(a). Front-facing culling is used instead of back-face culling so the globe still appears when the viewer is inside the box.

This is the only geometry that needs to be processed to render the ellipsoid, a constant vertex load of 12 triangles. With front-face culling, fragments for six of the triangles are processed for most views. The result is that a fragment shader is run for each fragment we want to cast a ray through. Since the fragment shader can access the camera's world-space position through a uniform, and the vertex shader can pass the vertex's interpolated world-space position to the fragment shader, a ray can be constructed from the eye through each fragment's position.[4] The ray simply has an origin of `og_cameraEye` and a direction of normalize(`worldPosition` − `og_cameraEye`).

The fragment shader also needs access to the ellipsoid's center and radii. Since it is assumed that the ellipsoid is centered at the origin, the fragment shader just needs a uniform for the ellipsoid's radii. In practice, intersecting a ray with an ellipsoid requires $\frac{1}{\text{radii}^2}$, so that should be precomputed once on the CPU and passed to the fragment shader as a uniform. Given the

[4]In this case, a ray is cast in world coordinates with the ellipsoid's center at the origin. It is also common to perform ray casting in eye coordinates, where the ray's origin is the coordinate system's origin. What really matters is that the ray and object are in the same coordinate system.

ray and ellipsoid information, Listing 4.16 shows a fragment shader that colors fragments green if a ray through the fragment intersects the ellipsoid or red if the ray does not intersect, as shown in Figure 4.17(b).

This shader has two shortcomings. First, it does not do any actual shading. Fortunately, given the position and surface normal of the ray intersection, shading can utilize the same techniques used throughout this chapter, namely `LightIntensity`() and `ComputeTextureCoordinates`(). Listing 4.17 adds shading by computing the position of the intersection along the ray using `i.Time` and shading as usual. If the ray does not intersect the ellipsoid, the fragment is discarded. Unfortunately, using discard has the adverse effect of disabling GPU depth buffer optimizations, including fine-grained early-z and coarse-grained z-cull, as discussed in Section 12.4.5.

```
in vec3 worldPosition;
out vec3 fragmentColor;
uniform vec3 og_cameraEye;
uniform vec3 u_globeOneOverRadiiSquared;

struct Intersection
{
  bool Intersects;
  float Time;       // Time of intersection along ray
};

Intersection RayIntersectEllipsoid(vec3 rayOrigin,
    vec3 rayDirection, vec3 oneOverEllipsoidRadiiSquared)
{ // ... }

void main()
{
  vec3 rayDirection = normalize(worldPosition - og_cameraEye);
  Intersection i = RayIntersectEllipsoid(og_cameraEye,
      rayDirection, u_globeOneOverRadiiSquared);
  fragmentColor = vec3(i.Intersects, !i.Intersects, 0.0);
}
```

Listing 4.16. Base GLSL fragment shader for ray casting.

```
// ...
vec3 GeodeticSurfaceNormal(vec3 positionOnEllipsoid,
                           vec3 oneOverEllipsoidRadiiSquared)
{
  return normalize(positionOnEllipsoid *
                   oneOverEllipsoidRadiiSquared);
}

void main()
{
  vec3 rayDirection = normalize(worldPosition - og_cameraEye);
  Intersection i = RayIntersectEllipsoid(og_cameraEye,
      rayDirection, u_globeOneOverRadiiSquared);
  if (i.Intersects)
  {
```

```
    vec3 position = og_cameraEye + (i.Time * rayDirection);
    vec3 normal = GeodeticSurfaceNormal(position,
        u_globeOneOverRadiiSquared);

    vec3 toLight = normalize(og_cameraLightPosition - position);
    vec3 toEye = normalize(og_cameraEye - position);
    float intensity = LightIntensity(normal, toLight, toEye,
        og_diffuseSpecularAmbientShininess);

    fragmentColor = intensity * texture(og_texture0,
        ComputeTextureCoordinates(normal)).rgb;
  }
  else
  {
    discard;
  }
}
```

Listing 4.17. Shading or discarding a fragment based on a ray cast.

```
float ComputeWorldPositionDepth(vec3 position)
{
  vec4 v = og_modelViewPerspectiveMatrix * vec4(position, 1);
  v.z /= v.w;
  v.z = (v.z + 1.0) * 0.5;
  return v.z;
}
```

Listing 4.18. Computing depth for a world-space position.

The remaining shortcoming, which may not be obvious until other objects are rendered in the scene, is that incorrect depth values are written. When an intersection occurs, the box's depth is written instead of the ellipsoid's depth. This can be corrected by computing the ellipsoid's depth, as shown in Listing 4.18, and writing it to gl_FragDepth. Depth is computed by transforming the world-space positions of the intersection into clip coordinates, then transforming this z-value into normalized device coordinates and, finally, into window coordinates. The final result of GPU ray casting, with shading and correct depth, is shown in Figure 4.17(c).

Since this algorithm doesn't have any overdraw, all the red pixels in Figure 4.17(b) are wasted fragment shading. A tessellated ellipsoid rendered with back-face culling does not have wasted fragments. On most GPUs, this is not as bad as it seems since the dynamic branch will avoid the shading computations [135, 144, 168], including the expensive inverse trigonometry for texture-coordinate generation. Furthermore, since the branches are coherent, that is, adjacent fragments in screen space are likely to take the same branch, except around the ellipsoid's silhouette, the GPU's parallelism is used well [168].

To reduce the number of rays that miss the ellipsoid, a viewport-aligned convex polygon bounding the ellipsoid from the viewer's perspective can be used instead of a bounding box [30]. The number of points in the bounding polygon determine how tight the fit is and, thus, how many rays miss the ellipsoid. This creates a trade-off between vertex and fragment processing.

GPU ray casting an ellipsoid fits seamlessly into the rasterization pipeline, making it an attractive alternative to rendering a tessellated approximation. In the general case, GPU ray casting, and full ray tracing in particular, is difficult. Not all objects have an efficient ray intersection test like an ellipsoid, and large scenes require hierarchical spatial data structures for quickly finding which objects a ray may intersect. These types of linked data structures are difficult to implement on today's GPUs, especially for dynamic scenes. Furthermore, in ray tracing, the number of rays quickly explodes with effects like soft shadows and antialiasing. Nonetheless, GPU ray tracing is a promising, active area of research [134, 178].

4.4 Resources

A detailed description of computing a polygonal approximation to a sphere using subdivision surfaces, aimed towards introductory graphics students, is provided by Angel [7]. The book is an excellent introduction to computer graphics in general. A survey of subdivision-surface algorithms is presented in *Real-Time Rendering* [3]. The book itself is an indispensable survey of real-time rendering. See "The Orange Book" for more information on using multitexturing in fragment shaders to render the Earth [147]. The book is generally useful as it thoroughly covers GLSL and provides a wide range of example shaders.

An ellipsoid tessellation based on the honeycomb [39], a figure derived from a soccer ball, may prove advantageous over subdividing platonic solids, which leads to a nonuniform tessellation. Another alternative to the tessellation algorithms discussed in this chapter is the HEALPix [65].

A series on procedurally generating 3D planets covers many relevant topics, including cube-map tessellation, level of detail, and shading [182]. An experimental globe-tessellation algorithm for NASA World Wind is described by Miller and Gaskins [116].

The entire field of real-time ray tracing is discussed by Wald [178], including GPU approaches. A high-level discussion on ray tracing virtual globes, with a focus on improving visual quality, is presented by Christen [26].

12

Massive-Terrain Rendering

Virtual globes visualize massive quantities of terrain and imagery. Imagine a single rectangular image that covers the entire world with a sufficient resolution such that each square meter is represented by a pixel. The circumference of Earth at the equator and around the poles is roughly 40 million meters, so such an image would contain almost a quadrillion (1×10^{15}) pixels. If each pixel is a 24-bit color, it would require over 2 million gigabytes of storage—approximately 2 petabytes! Lossy compression reduces this substantially, but nowhere near enough to fit into local storage, never mind on main or GPU memory, on today's or tomorrow's computers. Consider that popular virtual globe applications offer imagery at resolution higher than one meter per pixel in some areas of the globe, and it quickly becomes obvious that such a naïve approach is unworkable.

Terrain and imagery datasets of this size must be managed with specialized techniques, which are an active area of research. The basic idea, of course, is to use a limited storage and processing budget where it provides the most benefit. As a simple example, many applications do not need detailed imagery for the approximately 70% of Earth covered by oceans; it makes little sense to provide one-meter resolution imagery there. So our terrain- and imagery-rendering technique must be able to cope with data with varying levels of detail in different areas. In addition, when high-resolution data are available for a wide area, more triangles should be used to render nearby features and sharp peaks, and more texels should be used where they map to more pixels on the screen. This basic goal has been pursued from a number of angles over the years. With the explosive growth in GPU performance in recent years, the emphasis has shifted from

minimizing the number of triangles drawn, usually by doing substantial computations on the CPU, to maximizing the GPU's triangle throughout.

We consider the problem of rendering planet-sized terrains with the following characteristics:

- They consist of far too many triangles to render with just the brute-force approaches introduced in Chapter 11.
- They are much larger than available system memory.

The first characteristic motivates the use of terrain LOD. We are most concerned with using LOD techniques to reduce the complexity of the geometry being rendered; other LOD techniques include reducing shading costs. In addition, we use culling techniques to eliminate triangles in parts of the terrain that are not visible.

The second characteristic motivates the use of out-of-core rendering algorithms. In out-of-core rendering, only a small subset of a dataset is kept in system memory. The rest resides in secondary storage, such as a local hard disk or on a network server. Based on view parameters, new portions of the dataset are brought into system memory, and old portions are removed, ideally without stuttering rendering.

Beautifully rendering immense terrain and imagery datasets using proven algorithms is pretty easy if you're a natural at spatial reasoning, have never made an off-by-one coding error, and scoff at those who consider themselves "big picture" people because you yourself live for the details. For the rest of us, terrain and imagery rendering takes some patience and attention to detail. It is immensely rewarding, though, combining diverse areas of computer science and computer graphics to bring a world to life on your computer screen.

Presenting all of the current research in terrain rendering could fill several books. Instead, this chapter presents a high-level overview of the most important concepts, techniques, and strategies for rendering massive terrains, with an emphasis on pointing you toward useful resources from which you can learn more about any given area.

In Chapters 13 and 14, we dive into two specific terrain algorithms: *geometry clipmapping* and *chunked LOD*. These two algorithms, which take quite different approaches to rendering massive terrains, serve to illustrate many of the concepts in this chapter.

We hope that you will come away from these chapters with lots of ideas for how massive-terrain rendering can be implemented in your specific application. We also hope that you will acquire a solid foundation for understanding and evaluating the latest terrain-rendering research in the years to come.

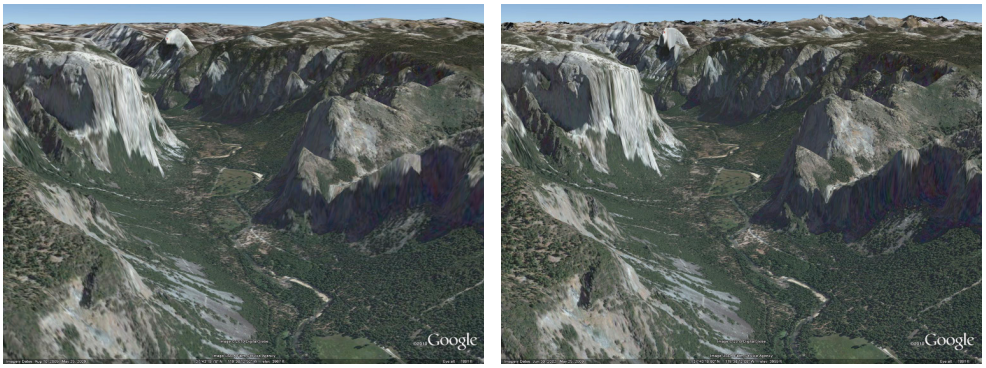
12.1 Level of Detail

Terrain LOD is typically managed using algorithms that are tuned to the unique characteristics of terrain. This is especially true when the terrain is represented as a height map; the regular structure allows techniques that are not applicable to arbitrary models. Even so, it is helpful to consider terrain LOD among the larger discipline of LOD algorithms.

LOD algorithms reduce an object's complexity when it contributes less to the scene. For example, an object in the distance may be rendered with less geometry and lower resolution textures than the same object if it were close to the viewer. Figure 12.1 shows the same view of Yosemite Valley, El Capitan, and Half Dome at different geometric levels of detail.

LOD algorithms consist of three major parts [3]:

- *Generation* creates different versions of a model. A simpler model usually uses fewer triangles to approximate the shape of the original model. Simpler models can also be rendered with less-complex shaders, smaller textures, fewer passes, etc.
- *Selection* chooses the version of the model to render based on some criteria, such as distance to the object, its bounding volume's estimated pixel size, or estimated number of nonoccluded pixels.
- *Switching* changes from one version of a model to another. A primary goal is to avoid *popping*: a noticeable, abrupt switch from one LOD to another.



(a) (b)

Figure 12.1. The same view of Yosemite Valley, El Capitan, and Half Dome at (a) low detail and (b) high detail. The differences are most noticeable in the shapes of the peaks in the distance. Image USDA Farm Service Agency, Image (C) 2010 DigitalGlobe. (Figures taken using Google Earth.)

Furthermore, we can group LOD algorithms into three broad categories: *discrete*, *continuous*, and *hierarchical*.

12.1.1 Discrete Level of Detail

Discrete LOD is perhaps the simplest LOD approach. Several independent versions of a model with different levels of detail are created. The models may be created manually by an artist or automatically by a polygonal simplification algorithm such as vertex clustering [146].

Applied to terrain, discrete LOD would imply that the entire terrain dataset has several discrete levels of detail and that one of them is selected for rendering at each frame. This is unsuitable for rendering terrain in virtual globes because terrain is usually both "near" and "far" at the same time.

The portion of terrain that is right in front of the viewer is nearby and requires a high level of detail for accurate rendering. If this high level of detail is used for the entire terrain, the hills in the distance will be rendered with far too many triangles. On the other hand, if we select the LOD based on the distant hills, the nearby terrain will have insufficient detail.

12.1.2 Continuous Level of Detail

In continuous LOD (CLOD), a model is represented in such a way that the detail used to display it can be precisely selected. Typically, the model is represented as a base mesh plus a sequence of transformations that make the mesh more or less detailed as each is applied. Thus, each successive version of the mesh differs from the previous one by only a few triangles.

At runtime, a precise level of detail for the model is created by selecting and applying the desired mesh transformations. For example, the mesh might be encoded as a series of *edge collapses*, each of which simplifies the mesh by removing two triangles. The opposite operation, called a *vertex split*, adds detail by creating two triangles. The two operations are shown in Figure 12.2.

CLOD is appealing because it allows a mesh to be selected that has a minimal number of triangles for a required visual fidelity given the viewpoint or other simplification criteria. In days gone by, CLOD was the best way to interactively render terrain. Many historically popular terrain-rendering algorithms use a CLOD approach, including Lindstrom et al.'s CLOD for height fields [102], Duchaineau et al.'s real-time optimally adapting mesh (ROAM) [41], and Hoppe's view-dependent progressive meshes [74]. Luebke et al. have excellent coverage of these techniques [107].

Today, however, these have largely fallen out of favor for use as runtime rendering techniques. CLOD generally requires traversing a CLOD data

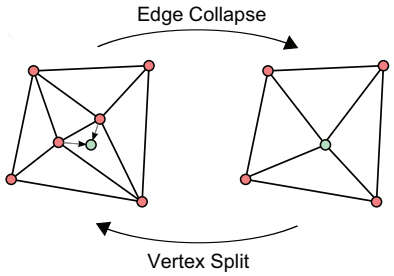


Figure 12.2. For an edge interior to a mesh, edge collapse removes two triangles and vertex split creates two triangles.

structure on the CPU and touching each vertex or edge in the current LOD. On older generations of hardware, this trade-off made a lot of sense; triangle throughput was quite low, so it was important to make every triangle count. In addition, it was worthwhile to spend extra time refining a mesh on the CPU in order to have the GPU process fewer triangles.

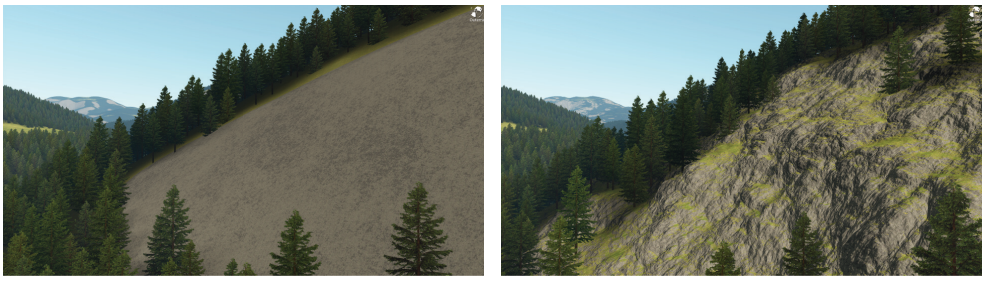
Today's GPUs have truly impressive triangle throughput and are, in fact, significantly faster than CPUs for many tasks. It is no longer a worthwhile trade-off to spend, for example, 50% more time on the CPU in order to reduce the triangle count by 50%. For that reason, CLOD-based terrain-rendering algorithms are inappropriate for use on today's hardware.

Today, these CLOD techniques, if they're used at all, are instead used to preprocess terrain into view-independent blocks for use with hierarchical LOD algorithms such as chunked LOD. These blocks are static; the CPU does not modify them at runtime, so CPU time is minimized. The GPU can easily handle the additional triangles that it is required to render as a result.

A special form of CLOD is known as infinite level of detail. In an infinite LOD scheme, we start with a surface that is defined by a mathematical function (e.g., an implicit surface). Thus, there is no limit to the number of triangles we can use to tessellate the surface. We saw an example of this in Section 4.3, where the implicit surface of an ellipsoid was used to render a pixel-perfect representation of a globe without tessellation.

Some terrain engines, such as the one in Outerra,[1] use fractal algorithms to procedurally generate fine terrain details (see Figure 12.3). This is a form of infinite LOD. Representing an entire real-world terrain as an implicit surface, however, is not feasible now or in the foreseeable future. For that reason, infinite LOD has only limited applications to terrain rendering in virtual globes.

[1] http://www.outerra.com



(a) (b)

Figure 12.3. Fractal detail can turn basic terrain into a beautiful landscape. (a) Original 76 m terrain data. (b) With fractal detail. (Images courtesy of Brano Kemen, Outerra.)

12.1.3 Hierarchical Level of Detail

Instead of reducing triangle counts using CLOD, today's terrain-rendering algorithms focus on two things:

- Reducing the amount of processing by the CPU.
- Reducing the quantity of data sent over the system bus to the GPU.

The LOD algorithms that best achieve these goals generally fall into the category of hierarchical LOD (HLOD) algorithms.

HLOD algorithms operate on *chunks* of triangles, sometimes called *patches* or *tiles*, to approximate the view-dependent simplification achieved by CLOD. It some ways, HLOD is a hybrid of discrete LOD and CLOD. The model is partitioned and stored in a multiresolution spatial data structure, such as an octree or quadtree (shown in Figure 12.4), with a drastically simplified version of the model at the root of the tree. A node contains one chunk of triangles. Each child node contains a subset of its parent, where each subset is more detailed than its parent but is spatially smaller. The union of all nodes at any level of the tree is a version of the full model. The node at level 0 (i.e., the root) is the most simplified version. The union of the nodes at maximum depth represents the model at full resolution.

If a given node has sufficient detail for the scene, it is rendered. Otherwise, the node is refined, meaning that its children are considered for rendering instead. This process continues recursively until the entire scene is rendered at an appropriate level of detail. Erikson et al. describe the major strategies for HLOD rendering [48].

HLOD algorithms are appropriate for modern GPUs because they help achieve both of the reductions identified at the beginning of this section.

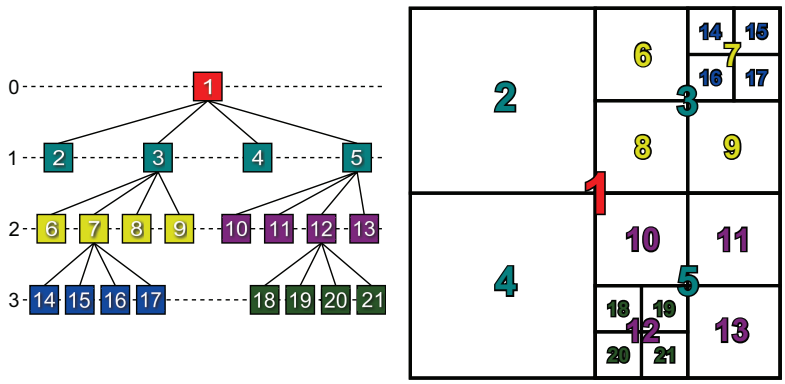


Figure 12.4. In HLOD algorithms, a model is partitioned and stored in a tree. The root contains a drastically simplified version of the model. Each child node contains a more detailed version of a subset of its parent.

In HLOD algorithms, the CPU only needs to consider each chunk rather than considering individual triangles, as is required in CLOD algorithms. In this way, the amount of processing that needs to be done by the CPU is greatly reduced.

HLOD algorithms can also reduce the quantity of data sent to the GPU over the system bus. At first glance, this is somewhat counterintuitive. After all, rendering with HLOD rather than CLOD generally means more triangles in the scene for the same visual fidelity, and triangles are, of course, data that need to be sent over the system bus.

HLOD, however, unlike CLOD, does not require that new data be sent to the GPU every time the viewer position changes. Instead, chunks are cached on the GPU using static vertex buffers that are applicable to a range of views. HLOD sends a smaller number of larger updates to the GPU, while CLOD sends a larger number of smaller updates.

Another strength of HLOD is that it integrates naturally with out-of-core rendering (see Section 12.3). The nodes in the spatial data structure are a convenient unit for loading data into memory, and the spatial ordering is useful for load ordering, replacement, and prefetching. In addition, the hierarchical organization offers an easy way to optimize culling, including hardware occlusion queries (see Section 12.4.4).

12.1.4 Screen-Space Error

No matter the LOD algorithm we use, we must choose which of several possible LODs to use for a given object in a given scene. Typically, the

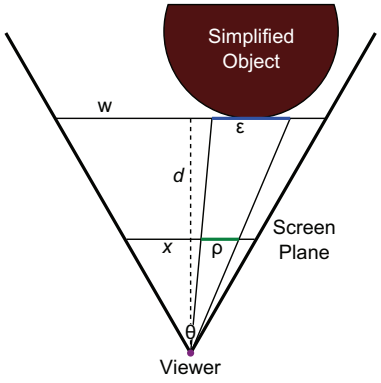


Figure 12.5. The screen-space error, ρ, of an object is estimated from the distance between the object and the viewer, the parameters of the view, and the geometric error, ϵ, of the object.

goal is to render with the simplest LOD possible while still rendering a scene that looks good. But how do we determine whether an LOD will provide a scene that looks good?

A useful objective measure of quality is the number of pixels of difference, or screen-space error, that would result by rendering a lower-detail version of an object rather than a higher-detail version. Computing this precisely is usually challenging, but it can be estimated effectively. By estimating it conservatively, we can arrive at a guaranteed error bound; that is, we can be sure that the screen-space error introduced by using a lower-detail version of a model is less than or equal to a computed value [107].

In Figure 12.5, we are considering the LOD to use for an object a distance d from the viewer in the view direction, where the view frustum has a width of w. In addition, the display has a resolution of x pixels and a field of view angle θ. A simplified version of the object has a geometric error ϵ; that is, each vertex in the full-detail object diverges from the closest corresponding point on the reduced-detail model by no more than ϵ units. What is the screen-space error, ρ, that would result if we were to render this simplified version of the object?

From the figure, we can see that ρ and ϵ are proportional and can solve for ρ:

$$\frac{\epsilon}{w} = \frac{\rho}{x},$$
$$\rho = \frac{\epsilon x}{w}.$$

The view-frustum width, w, at distance d is easily determined and substituted into our equation for ρ:

$$w = 2d \tan \frac{\theta}{2},$$

$$\rho = \frac{\epsilon x}{2d \tan \frac{\theta}{2}}. \tag{12.1}$$

Technically, this equation is only accurate for objects in the center of the viewport. For an object at the sides, it slightly underestimates the true screen-space error. This is generally considered acceptable, however, because other quantities are chosen conservatively. For example, the distance from the viewer to the object is actually larger than d when the object is not in the center of the viewport. In addition, the equation assumes that the greatest geometric error occurs at the point on the object that is closest to the viewer.

Kevin Says

For a bounding sphere centered at c and with radius r, the distance d to the closest point of the sphere in the direction of the view, $\mathbf{v}$, is given by

$$d = (c - \text{viewer}) \cdot \mathbf{v} - r.$$

By comparing the computed screen-space error for an LOD against the desired maximum screen-space error, we can determine if the LOD is accurate enough for our needs. If not, we refine.

12.1.5 Artifacts

While the LOD techniques used to render terrain are quite varied, there's a surprising amount of commonality in the artifacts that show up in the process.

Cracking. *Cracking* is an artifact that occurs where two different levels of detail meet. As shown in Figure 12.6, cracking occurs because a vertex in a higher-detail region does not lie on the corresponding edge of a lower-detail region. The resulting mesh is not watertight.

The most straightforward solution to cracking is to drop vertical skirts from the outside edges of each LOD. The major problem with skirts is that they introduce short vertical cliffs in the terrain surface that lead to texture stretching. In addition, care must be taken in computing the normals of the skirt vertices so that they aren't visible as mysterious dark or light

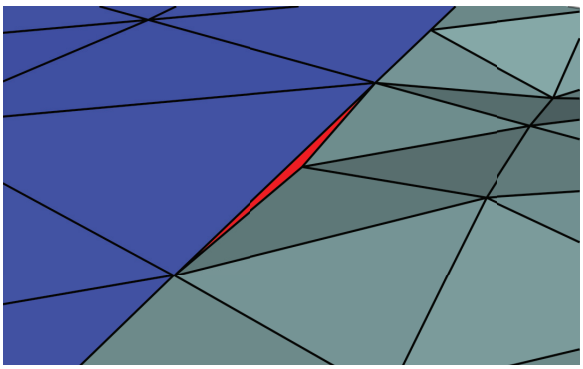


Figure 12.6. Cracking occurs when an edge shared by two adjacent LODs is divided by an additional vertex in one LOD but not the other.

lines around an LOD region. Chunked LOD uses skirts to avoid cracking, as will be discussed in Section 14.3.

Another possibility is to introduce extra vertices around the perimeter of the lower LOD region to match the adjacent higher LODs. This is effective when only a small number of different LODs are available or when there are reasonable bounds on the different LODs that are allowed to be adjacent to each other. In the worst case, the coarsest LOD would require an incredible number of vertices at its perimeter to account for the possibility that it is surrounded by regions of the finest LOD. Even in the best cases, however, this approach requires extra vertices in coarse LODs.

A similar approach is to force the heights of the vertices in the finer LOD to lie on the edges of the coarser LOD. The geometry-clipmapping terrain LOD algorithm (see Chapter 13) uses this technique effectively. A danger, however, is that this technique leads to a new problem: *T-junctions*.

T-junctions. T-junctions are similar to cracking, but more insidious. Whereas cracking occurs when a vertex in a higher-detail region does not lie on the corresponding edge of a lower-detail region, T-junctions occur because the high-detail vertex *does* lie on the low-detail edge, forming a T shape. Small differences in floating-point rounding during rasterization of the adjacent triangles lead to very tiny pinholes in the terrain surface. These pinholes are distracting because the background is visible through them.

Ideally, these T-junctions are eliminated by subdividing the triangle in the coarser mesh so that it, too, has a vertex at the same location as the vertex in the finer mesh. If the T-junctions were introduced in the first place in an attempt to eliminate cracking, however, this is a less than satisfactory solution.

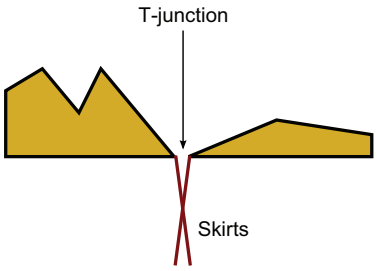


Figure 12.7. A side view of an exaggerated T-junction between two adjacent LODs. Skirts can hide T-junctions if they are angled slightly outward.

Another possibility is to fill the T-junctions with degenerate triangles. Even though these degenerate triangles mathematically have no area and thus should produce no fragments, the same rounding errors that cause the tiny T-junction holes to appear in the first place also cause a few fragments to be produced from the degenerate triangles, and those fragments fill the holes.

A final possibility, which is effective when cracks are filled with skirts, is to make the skirts of adjacent LODs overlap slightly, as shown in Figure 12.7.

Popping. As the viewer moves, the level of detail of various objects in the scene is adjusted. When the LOD of a given object is abruptly changed from a coarser one to a finer one, or vice versa, the user may notice a "pop" as vertices and edges change position.

This may be acceptable. To a large extent, virtual globe users tend to be more accepting of popping artifacts than, say, people playing a game. A virtual globe is like a web browser. Users instinctively understand that a web browser combines a whole lot of content loaded from remote servers. No one complains when a web browser shows the text of a page first and then "pops" in the images once they have been downloaded from the web server. This is much better than the alternative: showing nothing until all of the content is available.

Similarly, virtual globe users are not surprised that data are often streamed incrementally from a remote server and, therefore, are also not surprised when it suddenly pops into existence. As virtual globe developers, we can take advantage of this user expectation even in situations where it is not strictly necessary, such as in the transition between two LODs, both cached on the GPU.

In many cases, however, popping can be prevented.

One way to prevent popping is to follow what Bloom refers to as the "mantra of LOD": a level of detail should only switch when that switch

will be imperceptible to the user [18]. Depending on the specific LOD algorithm in use and the capabilities of the hardware, this may or may not be a reasonable goal.

Another possibility is to blend between different levels of detail instead of switching them abruptly. The specifics of how this blending is done are tied closely to the terrain-rendering algorithm, so we cover two specific examples in Sections 13.4 and 14.3.

12.2 Preprocessing

Rendering a planet-sized terrain dataset at interactive frame rates requires that the terrain dataset be preprocessed. As much as we wish it were not, this is an inescapable fact. Whatever format is used to store the terrain data in secondary storage, such as a disk or network server, must allow lower-detail versions of the terrain dataset to be obtained efficiently.

As described in Chapter 11, terrain data in virtual globes are most commonly represented as a height map. Consider a height map with 1 trillion posts covering the entire Earth. This would give us approximately 40 m between posts at the equator, which is relatively modest by virtual globe standards. If this height map is stored as a giant image, it will have 1 million texels on each side.

Now consider a view of Earth from orbit such that the entire Earth is visible. How can we render such a scene? It's unnecessary, even if it were possible, to render all of the half a trillion visible posts. After all, half a trillion posts is orders of magnitude more posts than there are pixels on even a high-resolution display.

Terrain-rendering algorithms strive to be *output sensitive*. That is, the runtime should be dependent on the number of pixels shaded, not on the size or complexity of the dataset.

Perhaps we'd like to just fill a $1{,}024 \times 1{,}024$ texture with the most relevant posts and render it using the vertex-shader displacement-mapping technique described in Section 11.2.2. How do we obtain such a texture from our giant height map? Figure 12.8 illustrates what is required.

First we would read one post. Then we would seek past about 1,000 posts before reading another post. This process would repeat until we had scanned through nearly the entire file. Seeking through a file of this size and reading just a couple of bytes at a time would take a substantial amount of time, even from local storage.

Worse, reading just one post out of every thousand posts would result in aliasing artifacts. A much better approach would be to find the average or maximum of the $1{,}000 \times 1{,}000$ "skipped" post heights to produce one