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

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.

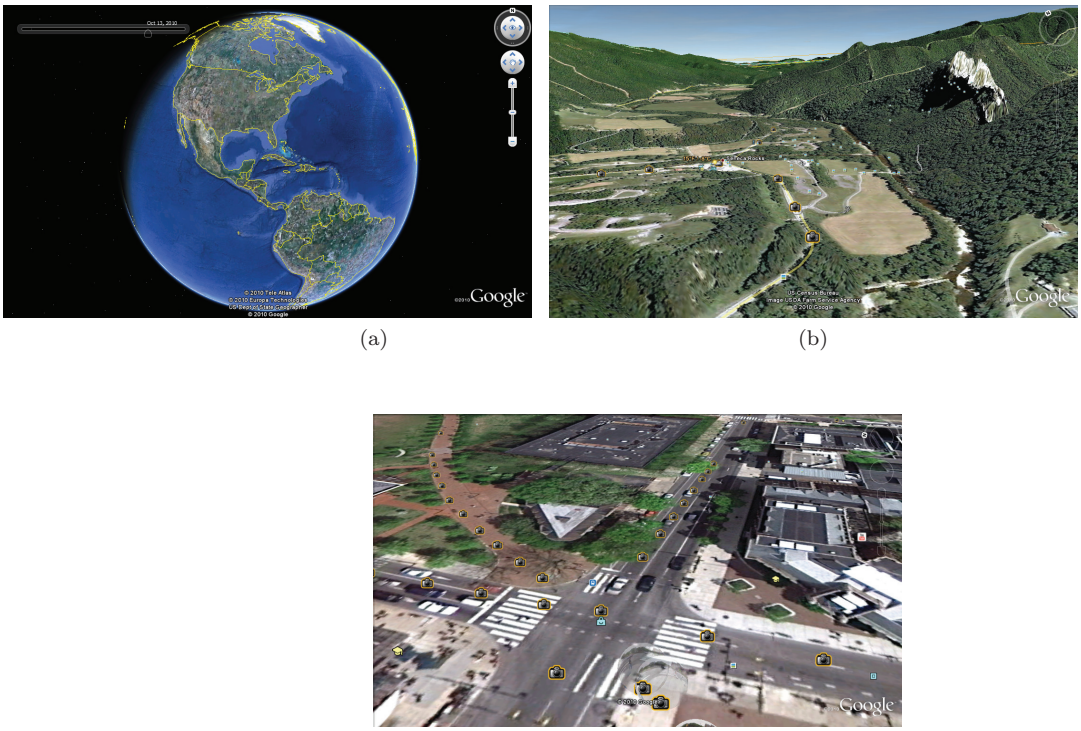
Obviously, implementing a 3D engine for virtual globes requires careful management of these datasets. Storing the entire world in memory and brute force rendering are certainly out of the question. Virtual globes, though, face additional rendering challenges beyond massive data management. This chapter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



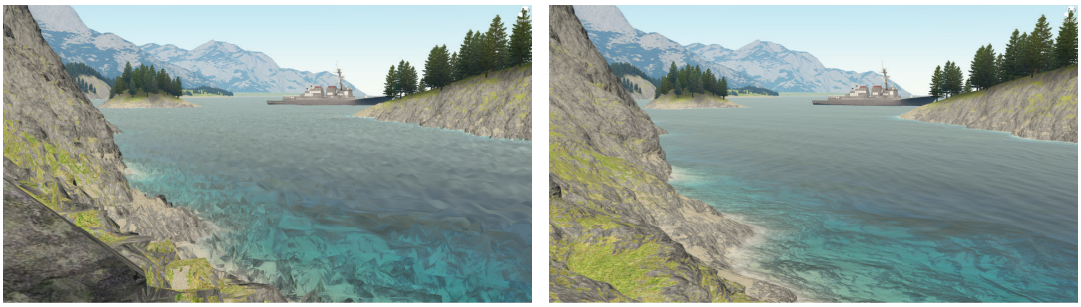
(a) (b)

(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.

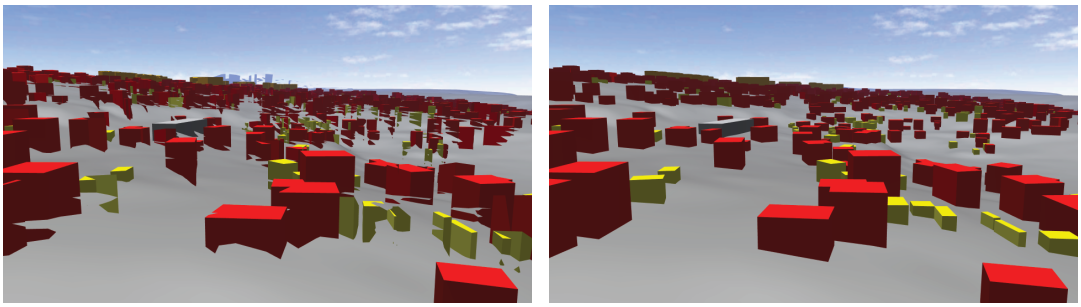
- *Accuracy.* In addition to eliminating rendering artifacts caused by precision errors, virtual globes should also model Earth accurately. Assuming Earth is a perfect sphere allows for many simplifications, but Earth is actually about 21 km longer at the equator than at the poles. Failing to take this 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.)

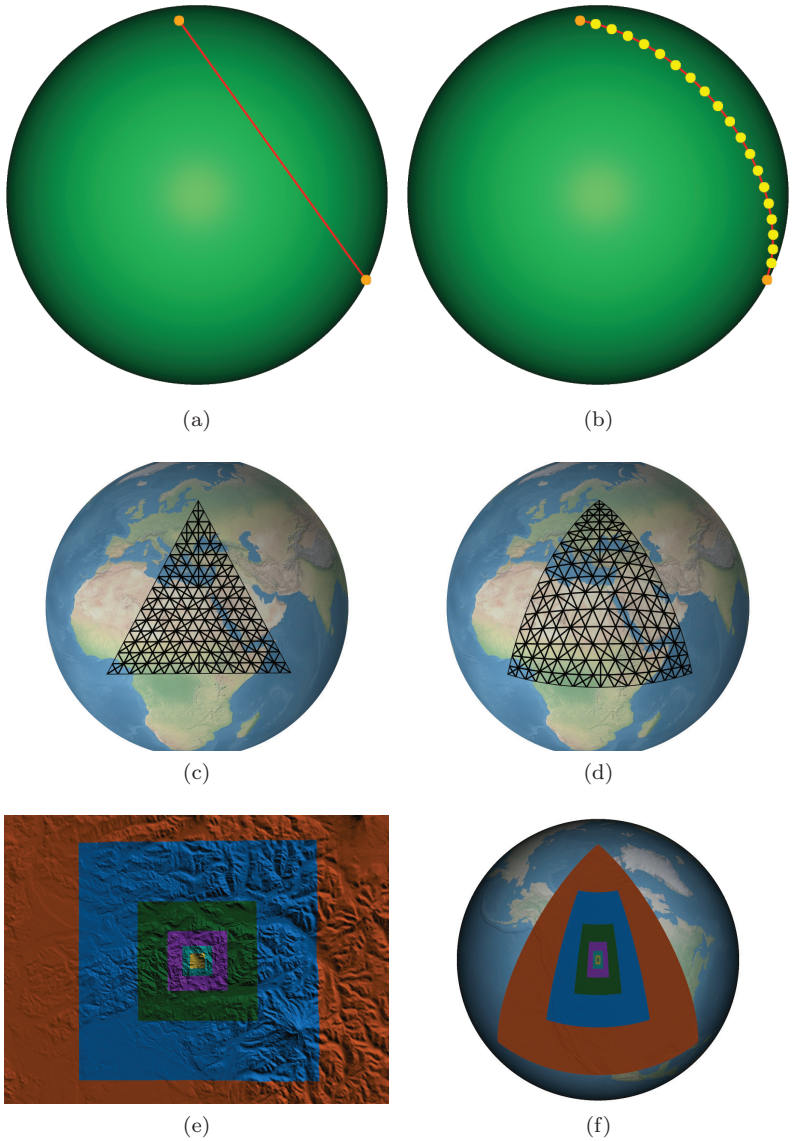


Figure 1.4. (a) Lines connecting surface points cut underneath a globe; instead, (b) points should be connected with a curve. Likewise, (c) polygons composed of triangles cut under a globe unless (d) curvature is taken into account. Mapping flat-world algorithms, (e) like geometry clipmapping terrain, 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).

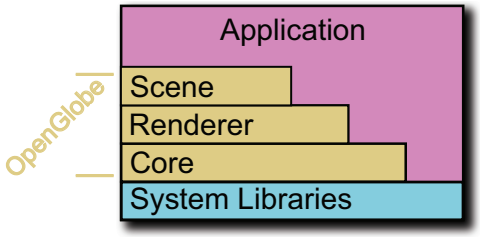


Figure 1.5. The stack of OpenGlobe assemblies.

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/