CRC Press
Taylor & Francis Group
AN A K PETERS BOOK

3D Engine Design for Virtual Globes

Patrick Cozzi • Kevin Ring

3D Engine Design for Virtual Globes

3D Engine Design for Virtual Globes

Patrick Cozzi
Kevin Ring

Cover design by Francis X. Kelly.
3D cover art by Jason K. Martin.

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed in the United States of America on acid-free paper
Version Date: 20110505

International Standard Book Number: 978-1-56881-711-8 (Hardback)

Library of Congress Cataloging-in-Publication Data

Cozzi, Patrick.
3D Engine design for virtual globes / Patrick Cozzi, Kevin Ring.
p. cm. -- (An A K Peters book)
Includes bibliographical references and index.
ISBN 978-1-56881-711-8 (hardback)
1. Globes--Computer-assisted instruction. 2. Digital mapping. 3. Texture mapping. 4. Computer graphics. I. Ring, Kevin. II. Title.

G3170.C69 2011
912.0285'6693--dc22 2011010460

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

To my parents, who bought me my first computer in 1994. Honestly, I just wanted to play games; I didn't think anything productive would come of it.

When I was seven years old, I declared that I wanted to make my own computer games. This book is dedicated to my mom and dad, who thought that was a neat idea.

Contents

Foreword

Do not let the title of this book fool you. What the title tells you is that if you have an interest in learning about high-performance and robust terrain rendering for games, this book is for you. If you are impressed by the features and performance of mapping programs such as NASA World Wind or Google Earth and you want to know how to write software of this type, this book is for you.

Some authors write computer books that promise to tell you everything you need to know about a topic, yet all that is delivered is a smattering of high-level descriptions but no low-level details that are essential to help you bridge the gap between theoretical understanding and practical source code. This is not one of those books. You are given a quality tutorial about globe and terrain rendering; the details about real-time 3D rendering of high-precision data, including actual source code to work with; and the mathematical foundations needed to be an expert in this field. Moreover, you will read about state-of-the-art topics such as geometry clipmapping and other level-of-detail algorithms that deal efficiently with massive terrain datasets. The book's bibliography is extensive, allowing you to investigate the large body of research on which globe rendering is built.

What the title of the book does not tell you is that there are many more chapters and sections about computing with modern hardware in order to exploit parallelism. Included are discussions about multithreaded engine design, out-of-core rendering, task-level parallelism, and the basics necessary to deal with concurrency, synchronization, and shared resources. Although necessary and useful for globe rendering, this material is invaluable for any application that involves scientific computing or visualization and processing of a large amount of data. Effectively, the authors are providing you with two books for the price of one. I prefer to keep only a small number of technical books at my office, opting for books with large information-per-page density. This book is now one of those.

—Dave Eberly

Preface

Planet rendering has a long history in computer graphics. Some of the earliest work was done by Jim Blinn at NASA's Jet Propulsion Laboratory (JPL) in the late 1970s and 80s to create animations of space missions. Perhaps the most famous animations are the flybys of Jupiter, Saturn, Uranus, and Neptune from the Voyager mission.

Today, planet rendering is not just in the hands of NASA. It is at the center of a number of games, such as *Spore* and *EVE Online*. Even non-planet-centric games use globes in creative ways; for example, *Mario Kart Wii* uses a globe to show player locations in online play.

The popularity of virtual globes such as Google Earth, NASA World Wind, Microsoft Bing Maps 3D, and Esri ArcGIS Explorer has also brought significant attention to globe rendering. These applications enable viewing massive real-world datasets for terrain, imagery, vector data, and more.

Given the widespread use of globe rendering, it is surprising that no single book covers the topic. We hope this book fills the gap by providing an in-depth treatment of rendering algorithms utilized by virtual globes. Our focus is on accurately rendering real-world datasets by presenting the core rendering algorithms for globes, terrain, imagery, and vector data.

Our knowledge in this area comes from our experience developing Analytical Graphics, Inc.'s (AGI) Satellite Tool Kit (STK) and Insight3D. STK is a modeling and analysis application for space, defense, and intelligence systems that has incorporated a virtual globe since 1993 (admittedly, we were not working on it back then). Insight3D is a 3D visualization component for aerospace and geographic information systems (GIS) applications. We hope our real-world experience has resulted in a pragmatic discussion of virtual globe rendering.

Intended Audience

This book is written for graphics developers interested in rendering algorithms and engine design for virtual globes, GIS, planets, terrain, and

massive worlds. The content is diverse enough that it will appeal to a wide audience: practitioners, researchers, students, and hobbyists. We hope that our survey-style explanations satisfy those looking for an overview or a more theoretical treatment, and our tutorial-style code examples suit those seeking hands-on "in the trenches" coverage.

No background in virtual globes or terrain is required. Our treatment includes both fundamental topics, like rendering ellipsoids and terrain representations, and more advanced topics, such as depth buffer precision and multithreading.

You should have a basic knowledge of computer graphics, including vectors and matrices; experience with a graphics API, such as OpenGL or Direct3D; and some exposure to a shading language. If you understand how to implement a basic shader for per-fragment lighting, you are well equipped. If you are new to graphics—welcome! Our website contains links to resources to get you up to speed: http://www.virtualglobebook.com/.

This is also the place to go for the example code and latest book-related news.

Finally, you should have working knowledge of an object-oriented programming language like C++, C#, or Java.

Acknowledgments

The time and energy of many people went into the making of this book. Without the help of others, the manuscript would not have the same content and quality.

We knew writing a book of this scope would not be an easy task. We owe much of our success to our incredibly understanding and supportive employer, Analytical Graphics, Inc. We thank Paul Graziani, Frank Linsalata, Jimmy Tucholski, Shashank Narayan, and Dave Vallado for their initial support of the project. We also thank Deron Ohlarik, Mike Bartholomew, Tom Fili, Brett Gilbert, Frank Stoner, and Jim Woodburn for their involvement, including reviewing chapters and tirelessly answering questions. In particular, Deron played an instrumental role in the initial phases of our project, and the derivations in Chapter 2 are largely thanks to Jim. We thank Francis Kelly, Jason Martin, and Glenn Warrington for their fantastic work on the cover.

This book may have never been proposed if it were not for the encouragement of our friends at the University of Pennsylvania, namely Norm Badler, Steve Lane, and Joe Kider. Norm initially encouraged the idea and suggested A K Peters as a publisher, who we also owe a great deal of thanks to. In particular, Sarah Cutler, Kara Ebrahim, and Alice and Klaus Peters helped us through the entire process. Eric Haines (Autodesk) also provided a great deal of input to get us started in the right direction.

We're fortunate to have worked with a great group of chapter reviewers, whose feedback helped us make countless improvements. In alphabetical order, they are Quarup Barreirinhas (Google), Eric Bruneton (Laboratoire Jean Kuntzmann), Christian Dick (Technische Universität München), Hugues Hoppe (Microsoft Research), Jukka Jylänki (University of Oulu), Dave Kasik (Boeing), Brano Kemen (Outerra), Anton Frühstück Malischew (Greentube Internet Entertainment Solutions), Emil Persson (Avalanche Studios), Aras Pranckevičius (Unity Technologies), Christophe Riccio (Imagination Technologies), Ian Romanick (Intel), Chris Thorne (VRshed), Jan Paul Van Waveren (id Software), and Mattias Widmark (DICE).

Two reviewers deserve special thanks. Dave Eberly (Geometric Tools, LLC), who has been with us since the start, reviewed several chapters multiple times and always provided encouraging and constructive feedback. Aleksandar Dimitrijević (University of Niš) promptly reviewed many chapters; his enthusiasm for the field is energizing.

Last but not least, we wish to thank our family and friends who have missed us during many nights, weekends, and holidays. (Case-in-point: we are writing this section on Christmas Eve.) In particular, we thank Kristen Ring, Peg Cozzi, Margie Cozzi, Anthony Cozzi, Judy MacIver, David Ring, Christy Rowe, and Kota Chrome.

Dataset Acknowledgments

Virtual globes are fascinating because they provide access to a seemingly limitless amount of GIS data, including terrain, imagery, and vector data. Thankfully, many of these datasets are freely available. We graciously acknowledge the providers of datasets used in this book.

Natural Earth

Natural Earth (http://www.naturalearthdata.com/) provides public domain raster and vector datasets at 1 : 10, 1 : 50, and 1 : 110 million scales. We use the image in Figure 1 and Natural Earth's vector data throughout this book.

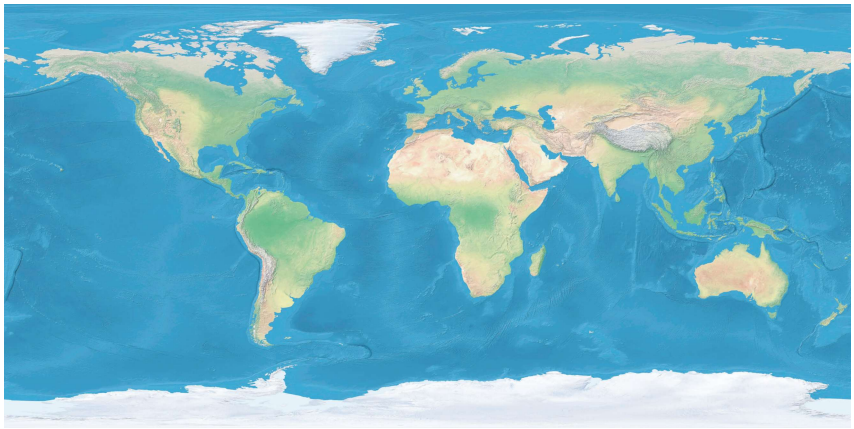


Figure 1. Satellite-derived land imagery with shaded relief and water from Natural Earth.

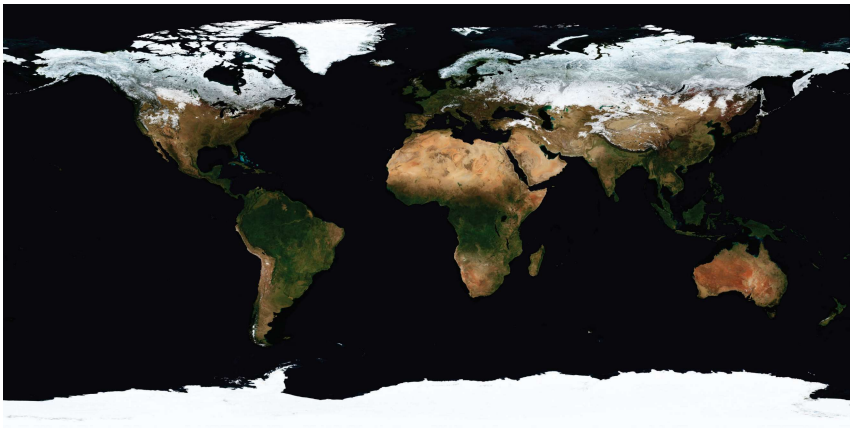
NASA Visible Earth

NASA Visible Earth (http://visibleearth.nasa.gov/) provides a wide array of satellite images. We use the images shown in Figure 2 throughout this book. The images in Figure 2(a) and 2(b) are part of NASA's Blue Marble collection and are credited to Reto Stockli, NASA Earth Observatory. The city lights image in Figure 2(c) is by Craig Mayhew and Robert Simmon, NASA GSFC. The data for this image are courtesy of Marc Imhoff, NASA GSFC, and Christopher Elvidge, NOAA NGDC.

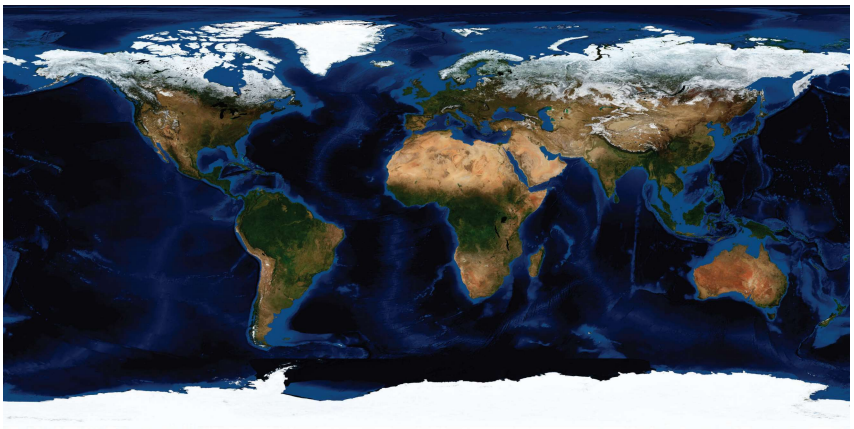
NASA World Wind

We use NASA World Wind's `mergedElevations` terrain dataset (http://worldwindcentral.com/wiki/World_Wind_Data_Sources) in our terrain implementation. This dataset has 10 m resolution terrain for most of the United States, and 90 m resolution data for other parts of the world. It is derived from three sources: the Shuttle Radar Topography Mission (SRTM) from NASA's Jet Propulsion Laboratory;[1] the National Elevation

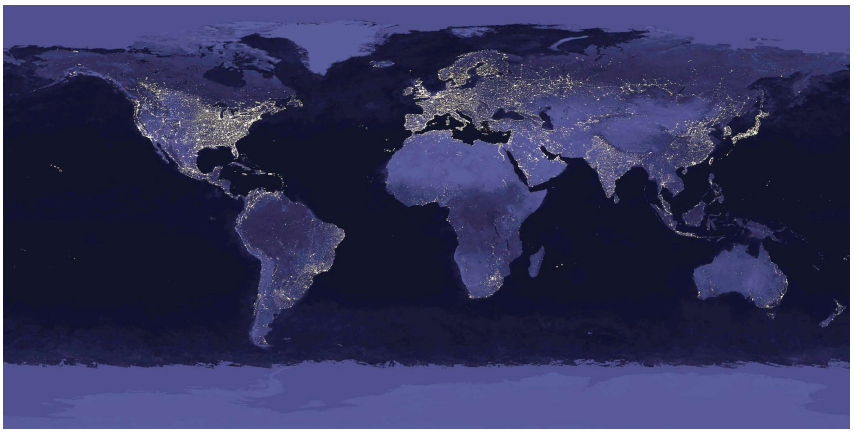
[1] http://www2.jpl.nasa.gov/srtm/



(a)



(b)



(c)

Figure 2. Images from NASA Visible Earth.

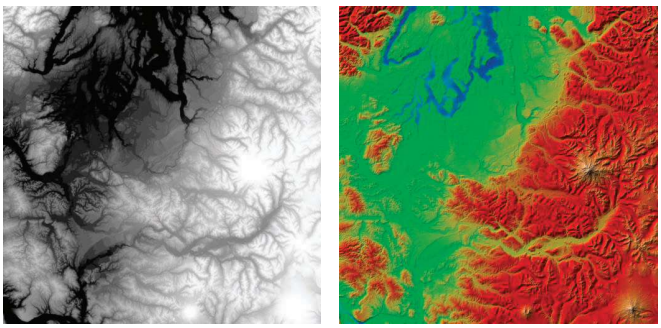
Dataset (NED) from the United States Geological Survey (USGS);[2] and SRTM30_PLUS: SRTM30, coastal and ridge multibeam, estimated topography, from the Institute of Geophysics and Planetary Physics, Scripps Institution of Oceanography, University of California San Diego.[3]

National Atlas of the United States of America

The National Atlas of the United States of America (http://www.nationalatlas.gov/atlasftp.html) provides a plethora of map data at no cost. In our discussion of vector data rendering, we use their airport and Amtrak terminal datasets. We acknowledge the Administration's Research and Innovative Technology Administration/Bureau of Transportation Statistics (RITA/BTS) National Transportation Atlas Databases (NTAD) 2005 for the latter dataset.

Georgia Institute of Technology

Like many developers working on terrain algorithms, we've used the terrain dataset for Puget Sound in Washington state, shown in Figure 3. These data are part of the Large Geometric Models Archive at the Georgia Institute of Technology (http://www.cc.gatech.edu/projects/large_models/ps.html). The original dataset[4] was obtained from the USGS and made available by the University of Washington. This subset was extracted by Peter Lindstrom and Valerio Pascucci.



(a) (b)

Figure 3. (a) A height map and (b) color map (texture) of Puget Sound from the Large Geometric Models Archive at the Georgia Institute of Technology.

[2] http://ned.usgs.gov/
[3] http://topex.ucsd.edu/WWW_html/srtm30_plus.html
[4] http://rocky.ess.washington.edu/data/raster/tenmeter/onebytwo10/

Yusuke Kamiyamane

The icons used in our discussion of vector data rendering were created by Yusuke Kamiyamane, who provides a large icon collection under the Creative Commons Attribution 3.0 license (http://p.yusukekamiyamane.com/).

Feedback

You are encouraged to email us with feedback, suggestions, or corrections at authors@virtualglobebook.com.

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.

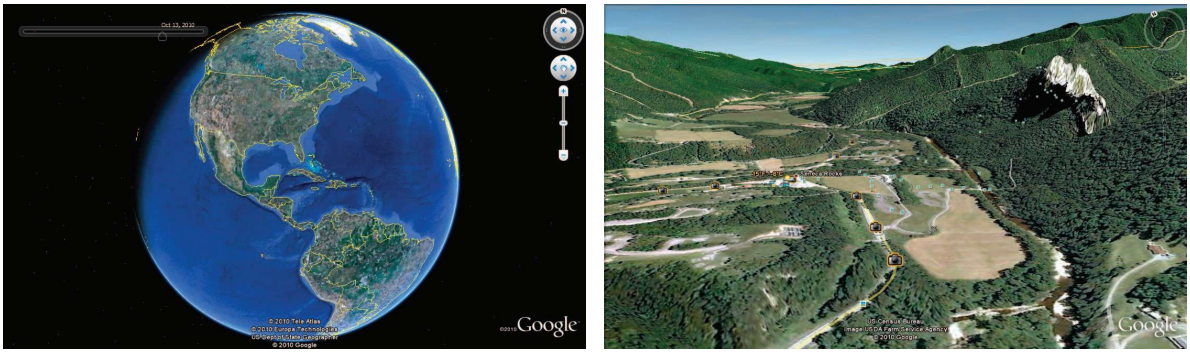
Obviously, implementing a 3D engine for virtual globes requires careful management of these datasets. Storing the entire world in memory and brute force rendering are certainly out of the question. Virtual globes, though, face additional rendering challenges beyond massive data management. This chapter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



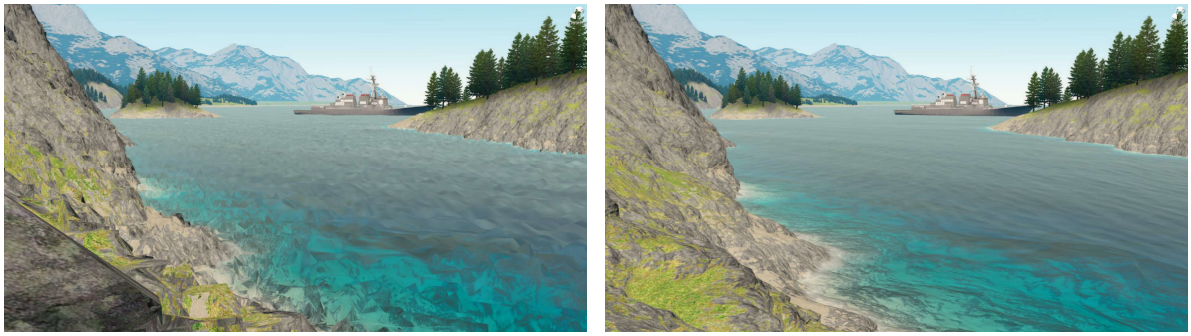
(a) (b)

(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.

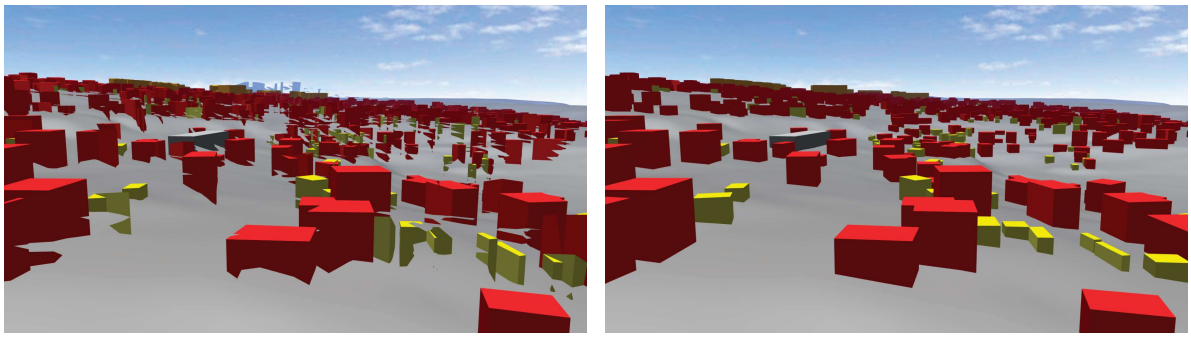
- *Accuracy.* In addition to eliminating rendering artifacts caused by precision errors, virtual globes should also model Earth accurately. Assuming Earth is a perfect sphere allows for many simplifications, but Earth is actually about 21 km longer at the equator than at the poles. Failing to take this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

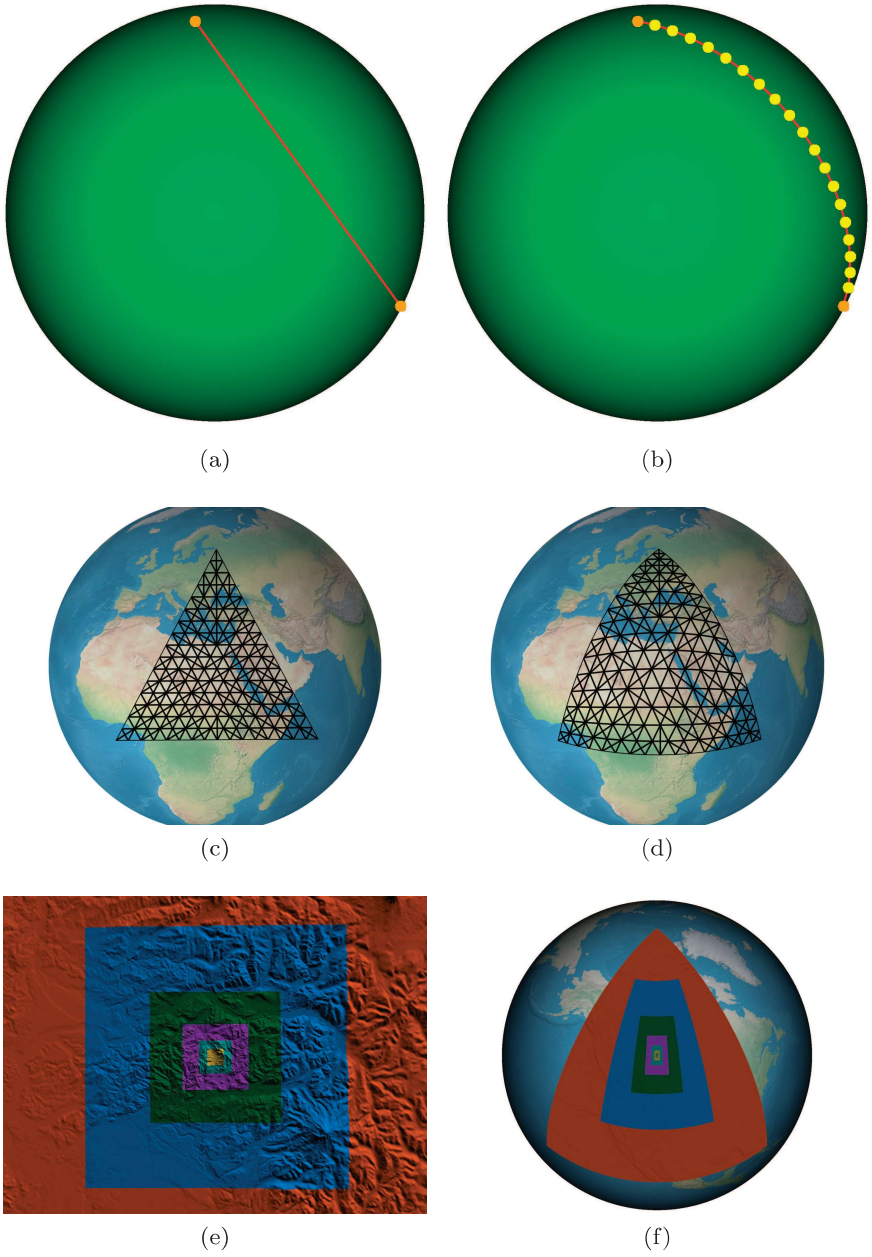


Figure 1.4. (a) Lines connecting surface points cut underneath a globe; instead, (b) points should be connected with a curve. Likewise, (c) polygons composed of triangles cut under a globe unless (d) curvature is taken into account. Mapping flat-world algorithms, (e) like geometry clipmapping terrain,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

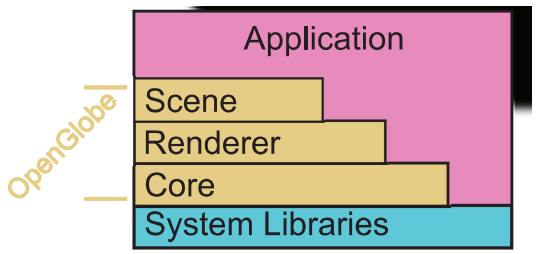


Figure 1.5. The stack of OpenGlobe assemblies.

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/

Part I

Fundamentals

2 Math Foundations

At the heart of an accurately rendered virtual globe is an ellipsoidal representation of Earth. This chapter introduces the motivation and mathematics for such a representation, with a focus on building a reusable Ellipsoid class containing functions for computing surface normals, converting between coordinate systems, computing curves on an ellipsoid surface, and more.

This chapter is unique among the others in that it contains a significant amount of math and derivations, whereas the rest of the book covers more pragmatic engine design and rendering algorithms. You don't need to memorize the derivations in this chapter to implement a virtual globe; rather, aim to come away with a high-level understanding and appreciation of the math and knowledge of how to use the presented Ellipsoid methods.

Let's begin by looking at the most common coordinate systems used in virtual globes.

2.1 Virtual Globe Coordinate Systems

All graphics engines work with one or more coordinate systems, and virtual globes are no exception. Virtual globes focus on two coordinate systems: geographic coordinates for specifying positions on or relative to a globe and Cartesian coordinates for rendering.

2.1.1 Geographic Coordinates

A geographic coordinate system defines each position on the globe by a $(longitude, latitude, height)$-tuple, much like a spherical coordinate system defines each position by an $(azimuth, inclination, radius)$-tuple. Intuitively, longitude is an angular measure west to east, latitude is an angular

measure south to north, and height is a linear distance above or below the surface. In Section 2.2.3, we more precisely define latitude and height.

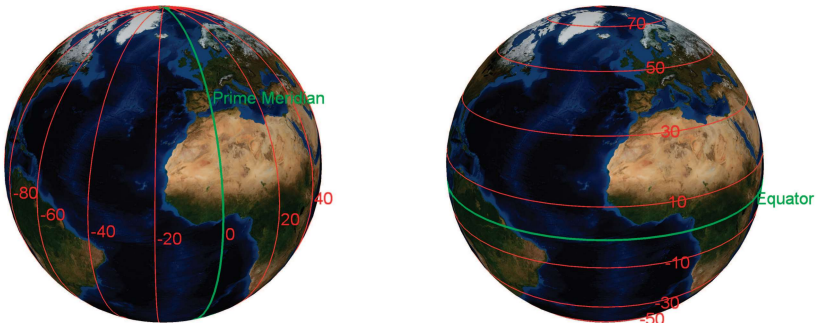
Geographic coordinates are widely used; most vector data are defined in geographic coordinates (see Part III). Even outside virtual globes, geographic coordinates are used for things such as the global positioning systems (GPS).

We adopt the commonly used convention of defining longitude in the range $[-180°, 180°]$. As shown in Figure 2.1(a), longitude is zero at the prime meridian, where the western hemisphere meets the eastern. Increasing longitude moves to the east, and decreasing longitude moves to the west; longitude is positive in the eastern hemisphere and negative in the western. Longitude increases or decreases until the antimeridian, $\pm 180°$ longitude, which forms the basis for the International Date Line (IDL) in the Pacific Ocean. Although the IDL turns in places to avoid land, for our purposes, it is approximated as $\pm 180°$. Many algorithms need special consideration for the IDL.

Longitude is sometimes defined in the range $[0°, 360°]$, where it is zero at the prime meridian and increases to the east through the IDL. To convert longitude from $[0°, 360°]$ to $[-180°, 180°]$, simply subtract $360°$ if longitude is greater than $180°$.

Latitude, the angular measure south to north, is in the range $[-90°, 90°]$. As shown in Figure 2.1(b), latitude is zero at the equator and increases from south to north. It is positive in the northern hemisphere and negative in the southern.

Longitude and latitude should not be treated as 2D x and y Cartesian coordinates. As latitude approaches the poles, lines of constant longitude converge. For example, the extent with southwest corner $(0°, 0°)$ and northwest corner $(10°, 10°)$ has much more surface area than the extent from



(a) (b)

Figure 2.1. Longitude and latitude in geographic coordinates. (a) Longitude spanning west to east. (b) Latitude spanning south to north.

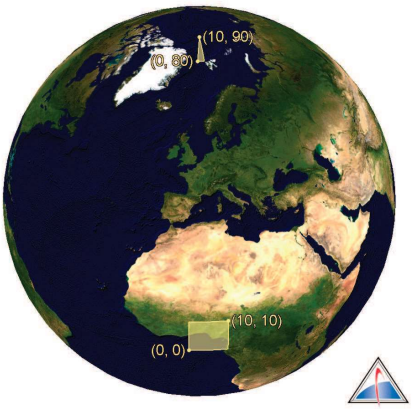


Figure 2.2. Extents with the same square number of degrees do not necessarily have the same surface area. (Image taken using STK. The Blue Marble imagery is from NASA Visible Earth.)

$(0°, 80°)$ to $(10°, 90°)$, even though they are both one square degree (see Figure 2.2). Therefore, algorithms based on a uniform longitude/latitude grid oversample near the poles, such as in the geographic grid tessellation in Section 4.1.4.

As mentioned in the Introduction, we use degrees for longitude and latitude, except in code examples, where they are in radians because C# and GLSL functions expect radians. Conversion between the two is straightforward: there are 2π rad or $360°$ in a circle, so one radian is $\frac{\pi}{180}°$, and one degree is $\frac{180}{\pi}$ rad. Although not used in this book, longitude and latitude are sometimes measured in *arc minutes* and *arc seconds*. There are 60 arc minutes in a degree and 60 arc seconds in an arc minute.

In OpenGlobe, geographic coordinates are represented using Geodetic2D and Geodetic3D, the difference being the former does not include height, implying the position is on the surface. A static class, Trig, provides `ToRadians` and `ToDegrees` conversion functions. Simple examples for these types are shown in Listing 2.1.

```
Geodetic3D p = Trig.ToRadians(new Geodetic3D(180.0, 0.0, 5.0));

Console.WriteLine(p.Longitude); // 3.14159...
Console.WriteLine(p.Latitude);  // 0.0
Console.WriteLine(p.Height);    // 5.0

Geodetic2D g = Trig.ToRadians(new Geodetic2D(180.0, 0.0));
Geodetic3D p2 = new Geodetic3D(g, 5.0);

Console.WriteLine(p == p2);     // True
```

Listing 2.1. Geodetic2D and Geodetic3D examples.

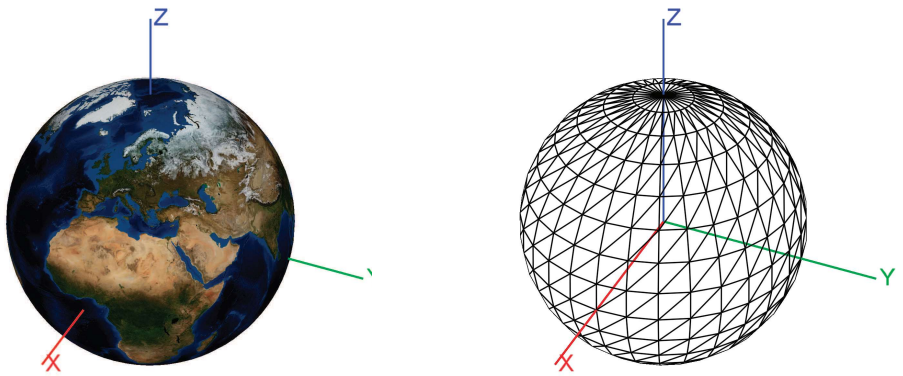
2.1.2 WGS84 Coordinate System

Geographic coordinates are useful because they are intuitive—intuitive to humans at least. OpenGL doesn't know what to make of them; OpenGL uses Cartesian coordinates for 3D rendering. We handle this by converting geographic coordinates to Cartesian coordinates for rendering.

The Cartesian system used in this book is called the World Geodetic System 1984 (WGS84) coordinate system [118]. This coordinate system is fixed to Earth; as Earth rotates, the system also rotates, and objects defined in WGS84 remain fixed relative to Earth. As shown in Figure 2.3, the origin is at Earth's center of mass; the x-axis points towards geographic $(0°, 0°)$, the y-axis points towards $(90°, 0°)$, and the z-axis points towards the north pole. The equator lies in the xy-plane. This is a right-handed coordinate system, hence $x \times y = z$, where x, y, and z are unit vectors along their respective axis.

In OpenGlobe, Cartesian coordinates are most commonly represented using Vector3D, whose interface surely looks similar to other vector types you've seen. Example code for common operations like normalize, dot product, and cross product is shown in Listing 2.2.

The only thing that may be unfamiliar is that a Vector3D's `X`, `Y`, and `Z` components are doubles, indicated by the `D` suffix, instead of floats, which are standard in most graphics applications. The large values used in virtual globes, especially those of WGS84 coordinates, are best represented by



(a) (b)

Figure 2.3. WGS84 coordinate system. (a) WGS84 coordinate system shown with a textured globe. (b) A wireframe globe shows the WGS84 coordinate system origin.

```
Vector3D x = new Vector3D(1.0, 0.0, 0.0);
// (Same as Vector3D.UnitX)
Vector3D y = new Vector3D(0.0, 1.0, 0.0);
// (Same as Vector3D.UnitY)

double s = x.X + x.Y + x.Z;          // 1.0
Vector3D n = (y - x).Normalize();    // (1.0 / Sqrt(2.0),
                                     // -1.0 / Sqrt(2.0), 0.0)
double p = n.Dot(y);                 // 1.0 / Sqrt(2.0)
Vector3D z = x.Cross(y);             // (0.0, 0.0, 1.0)
```

Listing 2.2. Fundamental Vector3D operations.

double precision as explained in Chapter 5. OpenGlobe also contains vector types for 2D and 4D vectors and float, Half (16-bit floating point), int, and bool data types.[1]

We use meters for units in Cartesian coordinates and for height in geodetic coordinates, which is common in virtual globes.

Let's turn our attention to ellipsoids, which will allow us to more precisely define geographic coordinates, and ultimately discuss one of the most common operations in virtual globes: conversion between geographic and WGS84 coordinates.

2.2 Ellipsoid Basics

A sphere is defined in 3-space by a center, c, and a radius, r. The set of points r units away from c define the sphere's surface. For convenience, the sphere is commonly centered at the origin, making its implicit equation:

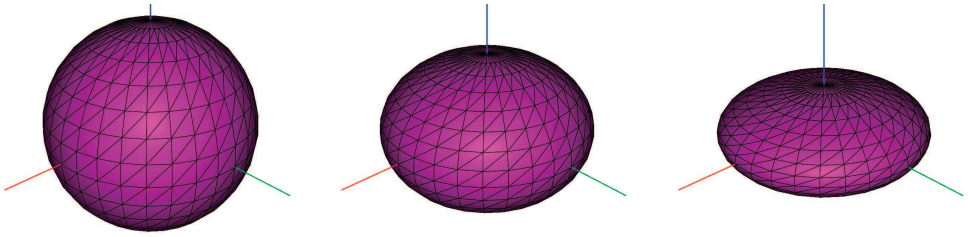
$$x_s^2 + y_s^2 + z_s^2 = r^2. \tag{2.1}$$

A point (x_s, y_s, z_s) that satisfies Equation (2.1) is on the sphere's surface. We use the subscript s to denote that the point is on the surface, as opposed to an arbitrary point (x, y, z), which may or may not be on the surface.

In some cases, it is reasonable to model a globe as a sphere, but as we shall see in the next section, an ellipsoid provides more precision and flexibility. An ellipsoid centered at $(0, 0, 0)$ is defined by three radii (a, b, c) along the x-, y-, and z-axes, respectively. A point (x_s, y_s, z_s) lies on the surface of an ellipsoid if it satisfies Equation (2.2):

$$\frac{x_s^2}{a^2} + \frac{y_s^2}{b^2} + \frac{z_s^2}{c^2} = 1. \tag{2.2}$$

[1] In C++, templates eliminate the need for different vector classes for each data type. Unfortunately, C# generics do not allow math operations on generic types.



(a) (b) (c)

Figure 2.4. Oblate spheroids with different semiminor axes. All of these oblate spheroids have a semimajor axis = 1 and a semiminor axis along the z-direction (blue). (a) Semiminor axis = 1. The oblate spheroid is a sphere. (b) Semiminor axis = 0.7. (c) Semiminor axis = 0.4.

When $a = b = c$, Equation (2.2) simplifies to Equation (2.1), hence the ellipsoid is a sphere. An *oblate spheroid* is a type of ellipsoid that is particularly useful for modeling Earth. An oblate spheroid has two radii of equal length (e.g., $a = b$) and a smaller third radius (e.g., $c < a$, $c < b$). The larger radii of equal length are called the *semimajor axes* and the smaller radius is called the *semiminor axis*. Figure 2.4 shows oblate spheroids with varying semiminor axes. The smaller the semiminor axis compared to the semimajor axis, the more *oblate* the spheroid.

2.2.1 WGS84 Ellipsoid

For many applications, particularly games, it is acceptable to represent Earth or a planet using a sphere. In fact some celestial bodies, such as the Moon, with a semimajor axis of 1,738.1 km at its equator and a semiminor axis of 1,736 km at its poles, are almost spherical [180]. Other celestial bodies are not even close to spherical, such as Phobos, one of Mars's moons, with radii of $27 \times 22 \times 18$ km [117].

Although not as oddly shaped as Phobos, Earth is not a perfect sphere. It is best represented as an oblate spheroid with an equatorial radius of 6,378,137 m, defining its semimajor axis, and a polar radius of 6,356,752.3142 m, defining its semiminor axis, making Earth about 21,384 m longer at the equator than at the poles.

This ellipsoid representation of Earth is called the WGS84 ellipsoid [118]. It is the National Geospatial-Intelligence Agency's (NGA) latest model of Earth as of this writing (it originated in 1984 and was last updated in 2004).

```
public class Ellipsoid
{
  public static readonly Ellipsoid Wgs84 =
      new Ellipsoid(6378137.0, 6378137.0, 6356752.314245);
  public static readonly Ellipsoid UnitSphere =
      new Ellipsoid(1.0, 1.0, 1.0);

  public Ellipsoid(double x, double y, double z) { /* ... */ }
  public Ellipsoid(Vector3D radii)  { /* ... */ }

  public Vector3D Radii
  {
    get { return _radii; }
  }

  private readonly Vector3D _radii;
}
```

Listing 2.3. Partial Ellipsoid implementation.

The WGS84 ellipsoid is widely used; we use it in STK and Insight3D, as do many virtual globes. Even some games use it, such as Microsoft's Flight Simulator [163].

The most flexible approach for handling globe shapes in code is to use a generic ellipsoid class constructed with user-defined radii. This allows code that supports the WGS84 ellipsoid and also supports other ellipsoids, such as those for the Moon, Mars, etc. In OpenGlobe, Ellipsoid is such a class (see Listing 2.3).

2.2.2 Ellipsoid Surface Normals

Computing the outward-pointing surface normal for a point on the surface of an ellipsoid has many uses, including shading calculations and precisely defining height in geographic coordinates. For a point on a sphere, the surface normal is found by simply treating the point as a vector and normalizing it. Doing the same for a point on an ellipsoid yields a *geocentric surface normal*. It is called geocentric because it is the normalized vector from the center of the ellipsoid through the point. If the ellipsoid is not a perfect sphere, this vector is not actually normal to the surface for most points.

On the other hand, a *geodetic surface normal* is the actual surface normal to a point on an ellipsoid. Imagine a plane tangent to the ellipsoid at the point. The geodetic surface normal is normal to this plane, as shown in Figure 2.5. For a sphere, the geocentric and geodetic surface normals are equivalent. For more oblate ellipsoids, like the ones shown in Figures 2.5(b) and 2.5(c), the geocentric normal significantly diverges from the geodetic normal for most surface points.

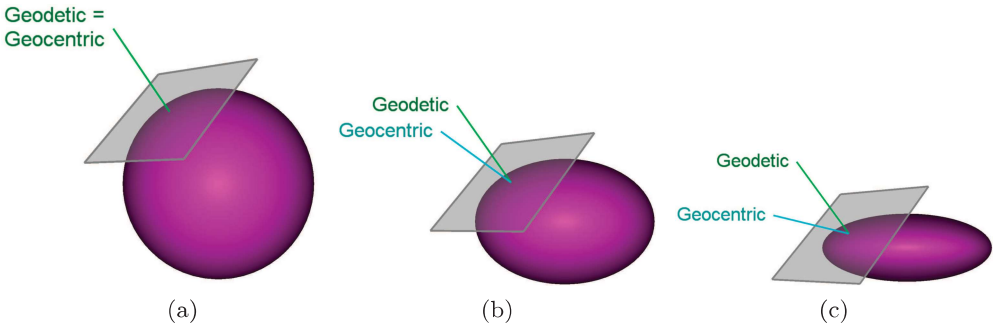


Figure 2.5. Geodetic versus geocentric surface normals. The geocentric normal diverges from the geodetic normal as the ellipsoid becomes more oblate. All figures have a semimajor axis = 1. (a) Semiminor axis = 1. (b) Semiminor axis = 0.7. (c) Semiminor axis = 0.4.

The geodetic surface normal is only slightly more expensive to compute than its geocentric counterpart

$$\mathbf{m} = \left(\frac{x_s}{a^2}, \frac{y_s}{b^2}, \frac{z_s}{c^2}\right),$$
$$\hat{\mathbf{n}}_s = \frac{\mathbf{m}}{\|\mathbf{m}\|},$$

where (a, b, c) are the ellipsoid's radii, (x_s, y_s, z_s) is the surface point, and $\hat{\mathbf{n}}_s$ is the resulting surface normal.

In practice, $\left(\frac{1}{a^2}, \frac{1}{b^2}, \frac{1}{c^2}\right)$ is precomputed and stored with the ellipsoid. Computing the geodetic surface normal simply becomes a component-wise multiplication of this precomputed value and the surface point, followed by normalization, as shown in Ellipsoid.`GeodeticSurfaceNormal` in Listing 2.4.

Listing 2.5 shows a very similar GLSL function. The value passed to `oneOverEllipsoidRadiiSquared` is provided to the shader by a uniform, so it is precomputed on the CPU once and used for many computations on the GPU. In general, we look for ways to precompute values to improve performance, especially when there is little memory overhead like here.

```
public Ellipsoid(Vector3D radii)
{
  // ...
  _oneOverRadiiSquared = new Vector3D(
      1.0 / (radii.X * radii.X),
      1.0 / (radii.Y * radii.Y),
      1.0 / (radii.Z * radii.Z));
}
```

```
public Vector3D GeodeticSurfaceNormal(Vector3D p)
{
  Vector3D normal = p.MultiplyComponents(_oneOverRadiiSquared);
  return normal.Normalize();
}

// ...
private readonly Vector3D _oneOverRadiiSquared;
```

Listing 2.4. Computing an ellipsoid's geodetic surface normal.

```
vec3 GeodeticSurfaceNormal(vec3 p,
                           vec3 oneOverEllipsoidRadiiSquared)
{
  return normalize(p * oneOverEllipsoidRadiiSquared);
}
```

Listing 2.5. Computing an ellipsoid's geodetic surface normal in GLSL.

Run Chapter02EllipsoidSurfaceNormals and increase and decrease the oblateness of the ellipsoid. The more oblate the ellipsoid, the larger the difference between the geodetic and geocentric normals.

Try This

2.2.3 Geodetic Latitude and Height

Given our understanding of geodetic surface normals, latitude and height in geographic coordinates can be precisely defined. *Geodetic latitude* is the

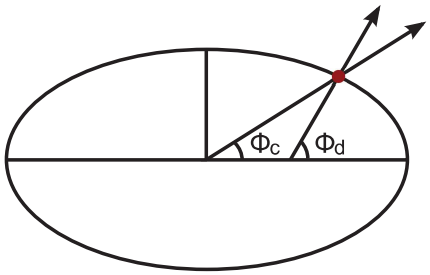


Figure 2.6. Comparison of geodetic latitude, ϕ_d, and geocentric latitude, ϕ_c.

angle between the equatorial plane (e.g., the xy-plane in WGS84 coordinates) and a point's geodetic surface normal. On the other hand, geocentric latitude is the angle between the equatorial plane and a vector from the origin to the point. At most points on Earth, geodetic latitude is different from geocentric latitude, as shown in Figure 2.6. Unless stated otherwise in this book, latitude always refers to geodetic latitude.

Height should be measured along a point's geodetic surface normal. Measuring along the geocentric normal introduces error, especially at higher heights, like those of space assets [62]. The larger the angular difference between geocentric and geodetic normals, the higher the error. The angular difference is dependent on latitude; on the WGS84 ellipsoid, the maximum angular difference between geodetic and geocentric normals is at $\approx 45°$ latitude.

2.3 Coordinate Transformations

Given that so much virtual globe data are stored in geographic coordinates but are rendered in WGS84 coordinates, the ability to convert from geographic to WGS84 coordinates is essential. Likewise, the ability to convert in the opposite direction, from WGS84 to geographic coordinates, is also useful.

Although we are most interested in the Earth's oblate spheroid, the conversions presented here work for all ellipsoid types, including a *triaxial ellipsoid*, that is, an ellipsoid where each radius is a different length ($a \neq b \neq c$).

In the following discussion, longitude is denoted by λ, geodetic latitude by ϕ, and height by h, so a $(longitude, latitude, height)$-tuple is denoted by (λ, ϕ, h). As before, an arbitrary point in Cartesian coordinates is denoted by (x, y, z), and a point on the ellipsoid surface is denoted by (x_s, y_s, z_s). All surface normals are assumed to be geodetic surface normals.

2.3.1 Geographic to WGS84

Fortunately, converting from geographic to WGS84 coordinates is a straightforward and closed form. The conversion is the same regardless of whether the point is above, below, or on the surface, but a small optimization can be made for surface points by omitting the final step.

Given a geographic point (λ, ϕ, h) and an ellipsoid (a, b, c) centered at the origin, determine the point's WGS84 coordinate, $r = (x, y, z)$.

The conversion takes advantage of a convenient property of the surface normal, $\hat{\mathbf{n}}_s$, to compute the location of the point on the surface, r_s; then, the height vector, $\mathbf{h}$, is computed directly and added to the surface point to produce the final position, r, as shown in Figure 2.7.

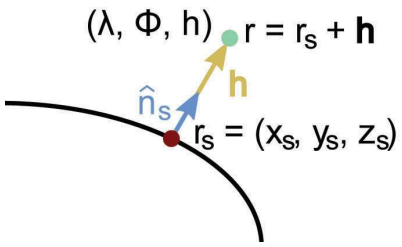


Figure 2.7. The geographic point (λ, ϕ, h) is converted to WGS84 coordinates by using the surface normal, $\hat{\mathbf{n}}_s$, to compute the surface point, r_s, which is offset by the height vector, $\mathbf{h}$, to produce the final point, r.

Given the surface position (λ, ϕ), the surface normal, $\hat{\mathbf{n}}_s$, is defined as

$$\hat{\mathbf{n}}_s = \cos\phi\cos\lambda\hat{\mathbf{i}} + \cos\phi\sin\lambda\hat{\mathbf{j}} + \sin\phi\hat{\mathbf{k}}. \tag{2.3}$$

Given the surface point $r_s = (x_s, y_s, z_s)$, the unnormalized surface normal, $\mathbf{n}_s$, is

$$\mathbf{n}_s = \frac{x_s}{a^2}\hat{\mathbf{i}} + \frac{y_s}{b^2}\hat{\mathbf{j}} + \frac{z_s}{c^2}\hat{\mathbf{k}}. \tag{2.4}$$

We are not given r_s but can determine it by relating $\hat{\mathbf{n}}_s$ and $\mathbf{n}_s$, which have the same direction but likely different magnitudes:

$$\hat{\mathbf{n}}_s = \gamma\mathbf{n}_s. \tag{2.5}$$

By substituting Equation (2.4) into $\mathbf{n}_s$ in Equation (2.5), we can rewrite $\hat{\mathbf{n}}_s$ as

$$\hat{\mathbf{n}}_s = \gamma\left(\frac{x_s}{a^2}\hat{\mathbf{i}} + \frac{y_s}{b^2}\hat{\mathbf{j}} + \frac{z_s}{c^2}\hat{\mathbf{k}}\right). \tag{2.6}$$

We know $\hat{\mathbf{n}}_s$, a^2, b^2, and c^2 but do not know γ, x_s, y_s, and z_s. Let's rewrite Equation (2.6) as three scalar equations:

$$\begin{aligned} \hat{n}_x &= \frac{\gamma x_s}{a^2}, \\ \hat{n}_y &= \frac{\gamma y_s}{b^2}, \\ \hat{n}_z &= \frac{\gamma z_s}{c^2}. \end{aligned} \tag{2.7}$$

Ultimately, we are interested in determining (x_s, y_s, z_s), so let's rearrange

Equation (2.7) to solve for x_s, y_s, and z_s:

$$\begin{aligned} x_s &= \frac{a^2 \hat{n}_x}{\gamma}, \\ y_s &= \frac{b^2 \hat{n}_y}{\gamma}, \\ z_s &= \frac{c^2 \hat{n}_z}{\gamma}. \end{aligned} \tag{2.8}$$

The only unknown on the right-hand side is γ; if we compute γ, we can solve for x_s, y_s, and z_s. Recall from the implicit equation of an ellipsoid in Equation (2.2) in Section 2.2 that a point is on the surface if it satisfies

$$\frac{x_s^2}{a^2} + \frac{y_s^2}{b^2} + \frac{z_s^2}{c^2} = 1.$$

We can use this to solve for γ by substituting Equation (2.8) into this equation, then isolating γ:

$$\begin{aligned} \frac{(\frac{a^2 \hat{n}_x}{\gamma})^2}{a^2} + \frac{(\frac{b^2 \hat{n}_y}{\gamma})^2}{b^2} + \frac{(\frac{c^2 \hat{n}_z}{\gamma})^2}{c^2} &= 1 \\ a^2 \hat{n}_x^2 + b^2 \hat{n}_y^2 + c^2 \hat{n}_z^2 &= \gamma^2 \\ \gamma &= \sqrt{a^2 \hat{n}_x^2 + b^2 \hat{n}_y^2 + c^2 \hat{n}_z^2}. \end{aligned} \tag{2.9}$$

Since γ is now written in terms of values we know, we can solve for x_s, y_s, and z_s using Equation (2.8). If the original geographic point is on the surface (i.e., $h = 0$) then the conversion is complete. For the more general case when the point may be above or below the surface, we compute a height vector, $\mathbf{h}$, with the direction of the surface normal and the magnitude of the point's height:

$$\mathbf{h} = h\hat{\mathbf{n}}_s.$$

```
public class Ellipsoid
{
  public Ellipsoid(Vector3D radii)
  {
    // ...
    _radiiSquared = new Vector3D(
        radii.X * radii.X,
        radii.Y * radii.Y,
        radii.Z * radii.Z);
  }

  public Vector3D GeodeticSurfaceNormal(Geodetic3D geodetic)
  {
    double cosLatitude = Math.Cos(geodetic.Latitude);
```

```
    return new Vector3D(
        cosLatitude * Math.Cos(geodetic.Longitude),
        cosLatitude * Math.Sin(geodetic.Longitude),
        Math.Sin(geodetic.Latitude));
  }

  public Vector3D ToVector3D(Geodetic3D geodetic)
  {
    Vector3D n = GeodeticSurfaceNormal(geodetic);
    Vector3D k = _radiiSquared.MultiplyComponents(n);
    double gamma = Math.Sqrt(
        k.X * n.X +
        k.Y * n.Y +
        k.Z * n.Z);

    Vector3D rSurface = k / gamma;
    return rSurface + (geodetic.Height * n);
  }

  // ...
  private readonly Vector3D _radiiSquared;
}
```

Listing 2.6. Converting from geographic to WGS84 coordinates.

The final WGS84 point is computed by offsetting the surface point, $r_s = (x_s, y_s, z_s)$, by $\mathbf{h}$:

$$r = r_s + \mathbf{h}. \tag{2.10}$$

The geographic to WGS84 conversion is implemented in Ellip soid.`ToVector3D`, shown in Listing 2.6. First, the surface normal is computed using Equation (2.3), then γ is computed using Equation (2.9). The converted WGS84 point is finally computed using Equations (2.8) and (2.10).

2.3.2 WGS84 to Geographic

Converting from WGS84 to geographic coordinates in the general case is more involved than conversion in the opposite direction, so we break it into multiple steps, each of which is also a useful function on its own.

First, we present the simple, closed form conversion for points on the ellipsoid surface. Then, we consider scaling an arbitrary WGS84 point to the surface using both a geocentric and geodetic surface normal. Finally, we combine the conversion for surface points with scaling along the geodetic surface normal to create a conversion for arbitrary WGS84 points.

The algorithm presented here uses only two inverse trigonometric functions and converges quickly, especially for Earth's oblate spheroid.

WGS84 surface points to geographic. Given a WGS84 point (x_s, y_s, z_s) on the surface of an ellipsoid (a, b, c) centered at the origin, the geographic point (λ, ϕ) is straightforward to compute.

```
public class Ellipsoid
{
  public Vector3D GeodeticSurfaceNormal(Vector3D p)
  {
    Vector3D normal = p.MultiplyComponents(_oneOverRadiiSquared);
    return normal.Normalize();
  }

  public Geodetic2D ToGeodetic2D(Vector3D p)
  {
    Vector3D n = GeodeticSurfaceNormal(p);
    return new Geodetic2D(
        Math.Atan2(n.Y, n.X),
        Math.Asin(n.Z / n.Magnitude));
  }

  // ...
}
```

Listing 2.7. Converting surface points from WGS84 to geographic coordinates.

Recall from Equation (2.4) that we can determine the unnormalized surface normal, $\mathbf{n}_s$, given the surface point:

$$\mathbf{n}_s = \frac{x_s}{a^2}\hat{\mathbf{i}} + \frac{y_s}{b^2}\hat{\mathbf{j}} + \frac{z_s}{c^2}\hat{\mathbf{k}}$$

The normalized surface normal, $\hat{\mathbf{n}}_s$, is simply computed by normalizing $\mathbf{n}_s$:

$$\hat{\mathbf{n}}_s = \frac{\mathbf{n}_s}{\|\mathbf{n_s}\|}.$$

Given $\hat{\mathbf{n}}_s$, longitude and latitude are computed using inverse trigonometric functions:

$$\lambda = \arctan\frac{\hat{\mathbf{n}}_y}{\hat{\mathbf{n}}_x},$$

$$\phi = \arcsin\frac{\hat{\mathbf{n}}_z}{\|\mathbf{n_s}\|}.$$

This is implemented in Ellipsoid.`ToGeodetic2D`, shown in Listing 2.7.

Scaling WGS84 points to the geocentric surface. Given an arbitrary WGS84 point, $r = (x, y, z)$, and an ellipsoid, (a, b, c), centered at the origin, we wish to determine the surface point, $r_s = (x_s, y_s, z_s)$, along the point's geocentric surface normal, as shown in Figure 2.8(a).

This is useful for computing curves on an ellipsoid (see Section 2.4) and is a building block for determining the surface point using the geodetic

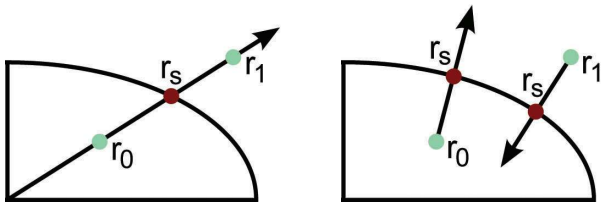


Figure 2.8. Scaling two points, r_0 and r_1, to the surface. (a) When scaling along the geocentric normal, a vector from the center of the ellipsoid is intersected with the ellipsoid to determine the surface point. (b) When scaling along the geodetic normal, an iterative process is used.

normal, as shown in Figure 2.8(b). Ultimately, we want to convert arbitrary WGS84 points to geographic coordinates by first scaling the arbitrary point to the geodetic surface and then converting the surface point to geographic coordinates and adjusting the height.

Let the position vector, $\mathbf{r}$, equal $r - 0$. The geocentric surface point, r_s, will be along this vector; that is

$$r_s = \beta \mathbf{r},$$

where r_s represents the intersection of the vector $\mathbf{r}$ and the ellipsoid. The variable β determines the position along the vector and is computed as

$$\beta = \frac{1}{\sqrt{\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2}}}. \tag{2.11}$$

```
public class Ellipsoid
{
  public Vector3D ScaleToGeocentricSurface(Vector3D p)
  {
    double beta = 1.0 / Math.Sqrt(
        (p.X * p.X) * _oneOverRadiiSquared.X +
        (p.Y * p.Y) * _oneOverRadiiSquared.Y +
        (p.Z * p.Z) * _oneOverRadiiSquared.Z);
    return beta * position;
  }

  // ...
}
```

Listing 2.8. Scaling a point to the surface along the geocentric surface normal.

Therefore, r_s is determined with

$$\begin{aligned} x_s &= \beta x, \\ y_s &= \beta y, \\ z_s &= \beta z. \end{aligned} \tag{2.12}$$

Equations (2.11) and (2.12) are used to implement Ellipsoid.`ScaleToGeocentricSurface` shown in Listing 2.8.

Scaling to the geodetic surface. Using the geocentric normal to determine a surface point doesn't have the accuracy required for WGS84 to geographic conversion. Instead, we seek the surface point whose *geodetic normal* points towards the arbitrary point, or in the opposite direction for points below the surface.

More precisely, given an arbitrary WGS84 point, $r = (x, y, z)$, and an ellipsoid, (a, b, c), centered at the origin, we wish to determine the surface point, $r_s = (x_s, y_s, z_s)$, whose geodetic surface normal points towards r, or in the opposite direction.

We form r_s in terms of a single unknown and use the Newton-Raphson method to iteratively approach the solution. This method converges quickly for Earth's oblate spheroid and doesn't require any trigonometric functions, making it efficient. It will not work for points very close to the center of the ellipsoid, where multiple solutions are possible, but these cases are rare in practice.

Let's begin by considering the three vectors in Figure 2.9: the arbitrary point vector, $\mathbf{r} = r - 0$; the surface point vector, $\mathbf{r}_s = r_s - 0$; and the height vector, $\mathbf{h}$. From the figure,

$$\mathbf{r} = \mathbf{r}_s + \mathbf{h}. \tag{2.13}$$

Recall that we can compute the unnormalized normal, $\mathbf{n}_s$, for a surface point

$$\mathbf{n}_s = \frac{x_s}{a^2}\hat{\mathbf{i}} + \frac{y_s}{b^2}\hat{\mathbf{j}} + \frac{z_s}{c^2}\hat{\mathbf{k}}. \tag{2.14}$$

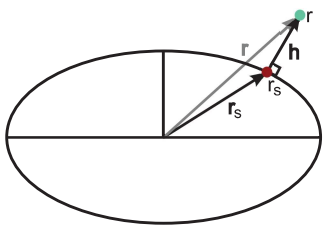


Figure 2.9. Computing r_s given r using the geodetic normal.

Observe that $\mathbf{h}$ has the same direction as $\mathbf{n}_s$, but likely a different magnitude. Let's relate them:

$$\mathbf{h} = \alpha \mathbf{n}_s.$$

We can substitute $\alpha \mathbf{n}_s$ for $\mathbf{h}$ in Equation (2.13):

$$\mathbf{r} = \mathbf{r}_s + \alpha \mathbf{n}_s.$$

Let's rewrite this as three scalar equations and substitute Equation (2.14) in for $\mathbf{n}_s$:

$$\begin{aligned} x &= x_s + \alpha \frac{x_s}{a^2}, \\ y &= y_s + \alpha \frac{y_s}{b^2}, \\ z &= z_s + \alpha \frac{z_s}{c^2}. \end{aligned}$$

Next, factor out x_s, y_s, and z_s:

$$\begin{aligned} x &= x_s(1 + \frac{\alpha}{a^2}), \\ y &= y_s(1 + \frac{\alpha}{b^2}), \\ z &= z_s(1 + \frac{\alpha}{c^2}). \end{aligned}$$

Finally, rearrange to solve for x_s, y_s, and z_s:

$$\begin{aligned} x_s &= \frac{x}{1 + \frac{\alpha}{a^2}}, \\ y_s &= \frac{y}{1 + \frac{\alpha}{b^2}}, \\ z_s &= \frac{z}{1 + \frac{\alpha}{c^2}}. \end{aligned} \tag{2.15}$$

We now have $r_s = (x_s, y_s, z_s)$ written in terms of the known point, $r = (x, y, z)$; the ellipsoid radii, (a, b, c); and a single unknown, α. In order to determine α, recall the implicit equation of an ellipsoid, which we can write in the form $F(x) = 0$:

$$S = \frac{x_s^2}{a^2} + \frac{y_s^2}{b^2} + \frac{z_s^2}{c^2} - 1 = 0. \tag{2.16}$$

Substitute the expressions for x_s, y_s, and z_s in Equation (2.15) into Equation (2.16):

$$S = \frac{x^2}{a^2 \left(1 + \frac{\alpha}{a^2}\right)^2} + \frac{y^2}{b^2 \left(1 + \frac{\alpha}{b^2}\right)^2} + \frac{z^2}{c^2 \left(1 + \frac{\alpha}{c^2}\right)^2} - 1 = 0. \tag{2.17}$$

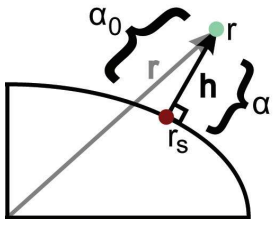


Figure 2.10. The Newton-Raphson method is used to find α from the initial guess, α_0.

Since this equation is no longer written in terms of the unknowns x_s, y_s, and z_s, we only have one unknown, α. Solving for α will allow us to use Equation (2.15) to find r_s.

We solve for α using the Newton-Raphson method for root finding; we are trying to find the root for S because when $S = 0$, the point lies on the ellipsoid surface. Initially, we guess a value, α_0, for α, then iterate until we are sufficiently close to the solution.

We initially guess r_s is the geocentric r_s computed in the previous section. Recall β from Equation (2.11):

$$\beta = \frac{1}{\sqrt{\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2}}}.$$

For a geocentric r_s, $r_s = \beta \mathbf{r}$, so our initial guess is

$$\begin{aligned} x_s &= \beta x, \\ y_s &= \beta y, \\ z_s &= \beta z. \end{aligned}$$

The surface normal for this point is

$$\begin{aligned} \mathbf{m} &= \left(\frac{x_s}{a^2}, \frac{y_s}{b^2}, \frac{z_s}{c^2} \right), \\ \hat{\mathbf{n}}_s &= \frac{\mathbf{m}}{\|\mathbf{m}\|}. \end{aligned}$$

Given our guess for r_s and $\hat{\mathbf{n}}_s$, we can now determine α_0. The unknown α scales $\hat{\mathbf{n}}_s$ to produce the height vector, $\mathbf{h}$. Our initial guess, α_0, simply scales $\hat{\mathbf{n}}_s$ to represent the distance between the ellipsoid surface and r as measured along the arbitrary point vector, $\mathbf{r}$, as shown in Figure 2.10. Therefore,

$$\alpha_0 = (1 - \beta) \frac{\|\mathbf{r}\|}{\|\mathbf{n_s}\|}.$$

We can now set $\alpha = \alpha_0$ and begin to iterate using the Newton-Raphson method. To do so, we need the function S from Equation (2.17) and its derivative with respect to α:

$$S = \frac{x^2}{a^2\left(1+\frac{\alpha}{a^2}\right)^2} + \frac{y^2}{b^2\left(1+\frac{\alpha}{b^2}\right)^2} + \frac{z^2}{c^2\left(1+\frac{\alpha}{c^2}\right)^2} - 1 = 0,$$

$$\frac{\partial S}{\partial \alpha} = -2\left[\frac{x^2}{a^4(1+\frac{\alpha}{a^2})^3} + \frac{y^2}{b^4(1+\frac{\alpha}{b^2})^3} + \frac{z^2}{c^4(1+\frac{\alpha}{c^2})^3}\right].$$

We iterate to find α by evaluating S and $\frac{\partial S}{\partial \alpha}$. If S is sufficiently close to zero (i.e., within a given epsilon) iteration stops and α is found. Otherwise, a new α is computed:

$$\alpha = \alpha - \frac{S}{\frac{\partial S}{\partial \alpha}}.$$

Iteration continues until S is sufficiently close to zero. Given α, r_s is computed using Equation (2.15).

The whole process for scaling an arbitrary point to the geodetic surface is implemented using Ellipsoid `.ScaleToGeodeticSurface`, shown in Listing 2.9.

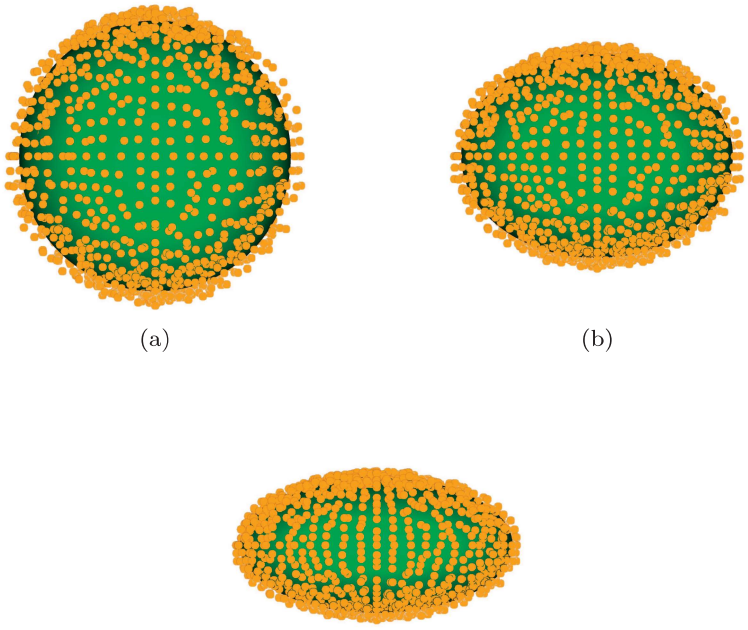
```
public class Ellipsoid
{
  public Ellipsoid(Vector3D radii)
  {
    // ...
    _radiiToTheFourth = new Vector3D(
        _radiiSquared.X * _radiiSquared.X,
        _radiiSquared.Y * _radiiSquared.Y,
        _radiiSquared.Z * _radiiSquared.Z);
  }

  public Vector3D ScaleToGeodeticSurface(Vector3D p)
  {
    double beta = 1.0 / Math.Sqrt(
        (p.X * p.X) * _oneOverRadiiSquared.X +
        (p.Y * p.Y) * _oneOverRadiiSquared.Y +
        (p.Z * p.Z) * _oneOverRadiiSquared.Z);
    double n = new Vector3D(
        beta * p.X * _oneOverRadiiSquared.X,
        beta * p.Y * _oneOverRadiiSquared.Y,
        beta * p.Z * _oneOverRadiiSquared.Z).Magnitude;
    double alpha = (1.0 - beta) * (p.Magnitude / n);

    double x2 = p.X * p.X;
    double y2 = p.Y * p.Y;
    double z2 = p.Z * p.Z;

    double da = 0.0;
    double db = 0.0;
    double dc = 0.0;
```

```
    double s = 0.0;
    double dSdA = 1.0;

    do
    {
      alpha -= (s / dSdA);

      da = 1.0 + (alpha * _oneOverRadiiSquared.X);
      db = 1.0 + (alpha * _oneOverRadiiSquared.Y);
      dc = 1.0 + (alpha * _oneOverRadiiSquared.Z);

      double da2 = da * da;
      double db2 = db * db;
      double dc2 = dc * dc;

      double da3 = da * da2;
      double db3 = db * db2;
      double dc3 = dc * dc2;

      s = x2 / (_radiiSquared.X * da2) +
          y2 / (_radiiSquared.Y * db2) +
          z2 / (_radiiSquared.Z * dc2) - 1.0;

      dSdA = -2.0 *
          (x2 / (_radiiToTheFourth.X * da3) +
           y2 / (_radiiToTheFourth.Y * db3) +
           z2 / (_radiiToTheFourth.Z * dc3));
    }
    while (Math.Abs(s) > 1e-10);

    return new Vector3D(
        p.X / da,
        p.Y / db,
        p.Z / dc);
  }

  // ...
  private readonly Vector3D _radiiToTheFourth;
}
```

Listing 2.9. Scaling a point to the surface along the geodetic surface normal.

Patrick Says ○○◯◯

I was assured this method converges quickly, but, for curiosity's sake, I ran a few tests to see how quickly. I created a 256×128 grid of points, each with a random height up to 10% of the semiminor axis above or below the surface. Then, I tested how many iterations `ScaleToGeodeticSurface` took to converge on three different ellipsoids, shown in Figure 2.11, tracking the minimum, maximum, and average number of iterations. The results, shown in Table 2.1, are encouraging. For Earth, all points converted in one or two iterations. As the ellipsoid becomes more oblate, more iterations are generally necessary, but never an impractical number.



(a) (b)

(c)

Figure 2.11. Ellipsoids used for `ScaleToGeodeticSurface` testing. (a) An ellipsoid with the oblateness of Earth. (b) Semimajor axis = 1 and semiminor axis = 0.75. (c) Semimajor axis = 1 and semiminor axis = 0.5. The images do not show the full resolution 256×128 point grid.

Ellipsoid	Minimum	Maximum	Average
Earth	1	2	1.9903
Semimajor axis: 1 Semiminor axis: 0.75	1	4	3.1143
Semimajor axis: 1 Semiminor axis: 0.5	1	4	3.6156

Table 2.1. Minimum, maximum, and average number of `ScaleToGeodetic Surface` iterations for ellipsoids of different oblateness.

Arbitrary WGS84 points to geographic. Using our function to scale a point to the geodetic surface and the function to convert a surface point from WGS84 to geographic coordinates, it is straightforward to convert an arbitrary point from WGS84 to geographic coordinates.

```
public Geodetic3D ToGeodetic3D(Vector3D position)
{
  Vector3D p = ScaleToGeodeticSurface(position);
  Vector3D h = position - p;
  double height = Math.Sign(h.Dot(position)) * h.Magnitude;
  return new Geodetic3D(ToGeodetic2D(p), height);
}
```

Listing 2.10. Converting a WGS84 point to geographic coordinates.

Given $r = (x, y, z)$, first, we scale it to the geodetic surface, producing r_s. A height vector, $\mathbf{h}$, is then computed:

$$\mathbf{h} = r - r_s.$$

The height of r above or below the ellipsoid is

$$h = \text{sign}\left(\mathbf{h} \cdot \mathbf{r} - \mathbf{0}\right) \|\mathbf{h}\|.$$

Finally, the surface point, r_s, is converted to geographic coordinates as done previously, and the resulting longitude and latitude are paired with h to create the final geographic coordinate, (λ, ϕ, h).

Given our implementations of `ScaleToGeodeticSurface` and `ToGeodetic2D`, this only requires a few lines of code, as shown in Listing 2.10.

2.4 Curves on an Ellipsoid

Many times in virtual globes, we have two endpoints on the surface of an ellipsoid that we wish to connect. Simply connecting the endpoints with a line results in a path that cuts under the ellipsoid, as shown in Figure 2.12(a). Instead, we wish to connect the endpoints with a path that approximately follows the curvature of the ellipsoid by subsampling points, as shown in Figures 2.12(b) and 2.12(c).

There are a variety of path types, each with different properties. A *geodesic curve* is the shortest path connecting two points. *Rhumb lines* are paths of constant bearing; although they are not the shortest path, they are widely used in navigation because of the simplicity of following a constant bearing.

Let's consider a simple way to compute a curve based on intersecting a plane and the ellipsoid surface, as shown in Figure 2.13(a). On a sphere, a *great circle* is the intersection of a plane containing the sphere's center and the sphere's surface. The plane cuts the sphere into two equal halves. When a plane containing the two endpoints and the sphere's center is intersected with the sphere's surface, two *great arcs* are formed: a *minor arc*, which is

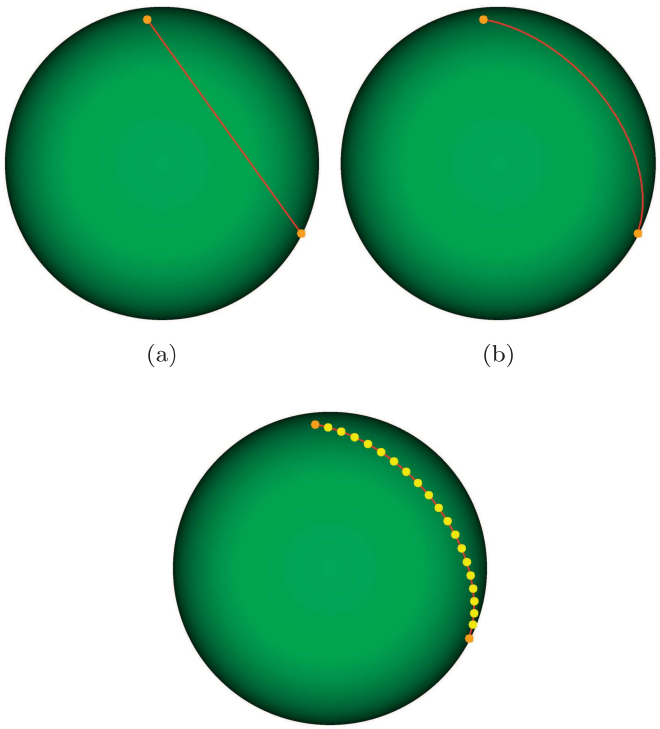


Figure 2.12. (a) Connecting two endpoints on the ellipsoid surface with a line results in a path that travels under the ellipsoid. The line is drawn here without depth testing. (b) Subsampling points along the surface approximates a path between the endpoints. (c) Subsampled points shown in yellow.

the shortest path on the sphere between the points, and a *major arc*, the longer path along the great circle.

Using the same plane-intersection technique on an ellipsoid does not result in a geodesic curve, but for near-spherical ellipsoids, such as Earth's oblate spheroid, it produces a reasonable path for rendering purposes, especially when the endpoints are close together.

The algorithm for computing such a curve is straightforward, efficient, and easy to code. Given two endpoints, p and q, and a granularity, γ, we wish to compute a path on an ellipsoid, (a, b, c), centered at the origin, by subsampling points with γ angular separation.

As shown in Figures 2.13(c) to 2.13(e), as γ decreases, the subsampled points become closer together, and thus better approximate the curve. Except in the limit when γ approaches zero, the line segments will always

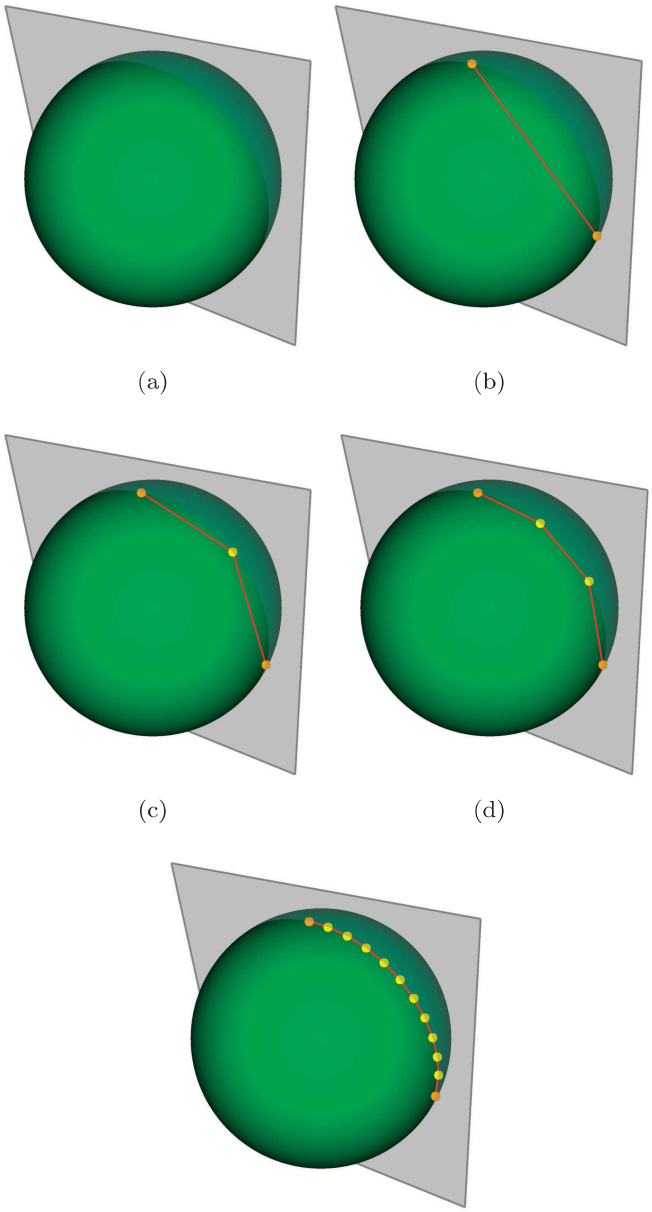


Figure 2.13. (a) Our curves are determined by the intersection of a plane and the ellipsoid. (b) The plane contains the ellipsoid's center and the line endpoints. (c)–(e) As the granularity decreases, the subsampled points better approximate a curve on the ellipsoid surface.

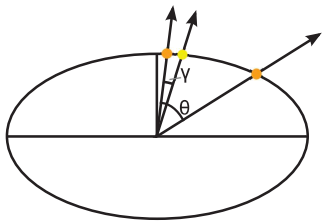


Figure 2.14. θ is the angle between the endpoints. The granularity, γ, is the angular separation between points on the curve.

cut under the ellipsoid; in Section 7.2, we present a rendering strategy that allows the line segments to still win the depth test against the ellipsoid.

To compute the curve between p and q, first create vectors from the ellipsoid center to the endpoints and take the cross product to determine the plane's normal, $\hat{\mathbf{n}}$:

$$\mathbf{p} = p - 0, \mathbf{q} = q - 0, \mathbf{m} = \mathbf{p} \times \mathbf{q}, \hat{\mathbf{n}} = \frac{\mathbf{m}}{\|\mathbf{m}\|}.$$

Next, determine the angle between vectors $\mathbf{p}$ and $\mathbf{q}$, θ, shown in Figure 2.14:

$$\hat{\mathbf{p}} = \frac{\mathbf{p}}{\|\mathbf{p}\|},$$
$$\hat{\mathbf{q}} = \frac{\mathbf{q}}{\|\mathbf{q}\|},$$
$$\theta = \arccos \hat{\mathbf{p}} \cdot \hat{\mathbf{q}}.$$

The number of subsampled points, s, is based on the granularity and the angle between endpoints:

$$n = \lfloor \frac{\theta}{\gamma} \rfloor - 1,$$
$$s = \max(\mathrm{n}, 0)$$

Each subsampled point is computed by rotating $\mathbf{p}$ around the plane normal, $\hat{\mathbf{n}}$, and scaling the resulting vector to the geocentric surface as described in Section 2.3.2. Using geocentric scaling, as opposed to geodetic scaling, keeps the point in the plane, and thus on the desired curve.

As shown in the implementation of Ellipsoid `.ComputeCurve` in Listing 2.11, a for loop is used to compute each subsampled point using an angle, $\phi = i\gamma$, where i is the index of the point.

```
public class Ellipsoid
{
  public IList<Vector3D> ComputeCurve(
      Vector3D p,
      Vector3D q,
      double granularity)
  {
    Vector3D normal = p.Cross(q).Normalize();
    double theta = p.AngleBetween(q);
    int s = Math.Max((int)(theta / granularity) - 1, 0);
    List<Vector3D> positions = new List<Vector3D>(2 + s);

    positions.Add(p);
    for (int i = 1; i <= s; ++i)
    {
      double phi = (i * granularity);
      Vector3D rotated = p.RotateAroundAxis(normal, phi);
      positions.Add(ScaleToGeocentricSurface(rotated));
    }
    positions.Add(q);

    return positions;
  }

  // ...
}
```

Listing 2.11. Computing a curve on the ellipsoid between two surface points.

Try This ○○◯◯

Run Chapter02Curves and experiment with the ellipsoid oblateness and curve granularity. How can we implement LOD for a curve?

Try This ○○◯◯

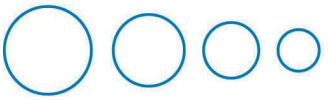
Modify `Ellipsoid.ComputeCurve` to create curves at a constant height above or below the ellipsoid.

Question ○○◯◯

Instead of using an angular granularity and rotating around the plane's normal, an implementation could simply use linear interpolation to subsample points along the line connecting the endpoints, then call `ScaleToGeocentricSurface` for each point. What are the advantages and disadvantages of this approach?

2.5 Resources

Vallado and McClain cover WGS84 and an incredible number of other topics in depth [171]. Precision concerns in virtual globes, such as those described in Section 2.2.3, with a focus on visualizing space assets, are described by Giesecke [62]. Luebke et al. cover georeferencing in their excellent book on LOD [107].



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

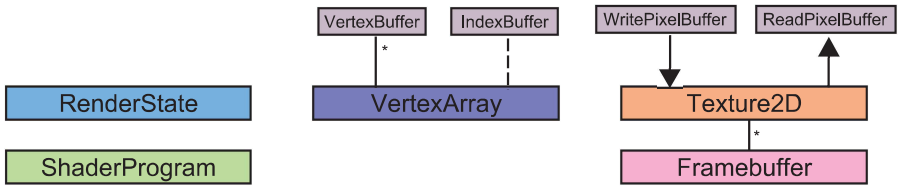


Figure 3.1. Our renderer's major components include render states; shader programs; vertex arrays that reference vertex and index buffers; 2D textures, whose data are manipulated with pixel buffers; and framebuffers, which contain textures.

one or more vertex buffers. An optional index buffer indexes into the vertex buffers to select vertices for drawing.

Two-dimensional textures represent images in driver-controlled memory that, when combined with a sampler that describes filtering and wrapping behavior, are accessible to shaders. Data are transferred to and from textures via write and read pixel buffers, respectively. Pixel buffers are channels for transferring data; ultimately, image data are stored in the texture itself.

Finally, framebuffers are lightweight containers of textures that enable render to texture techniques. A framebuffer can contain multiple textures as color attachments and a texture as a depth or depth/stencil attachment.

A static class named Device is used to create the render objects shown in Figure 3.2. It is helpful to think of Device as a factory for creating objects, but not for issuing rendering commands. A Context is used for issuing rendering commands. This distinction is similar to ID3D11Device and ID3D11DeviceContext in Direct3D.

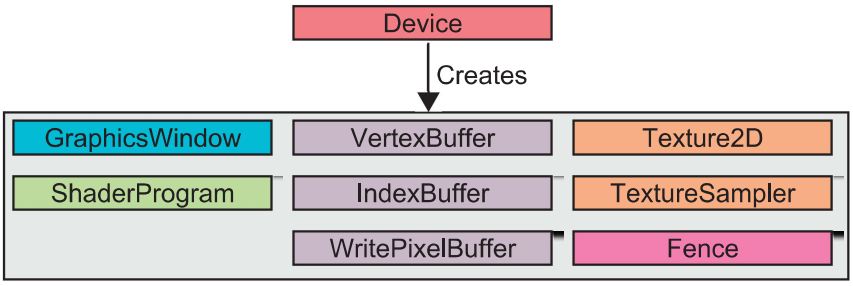


Figure 3.2. Device is a static class that creates renderer objects that are shared between contexts, except for GraphicsWindow, which contains a context.

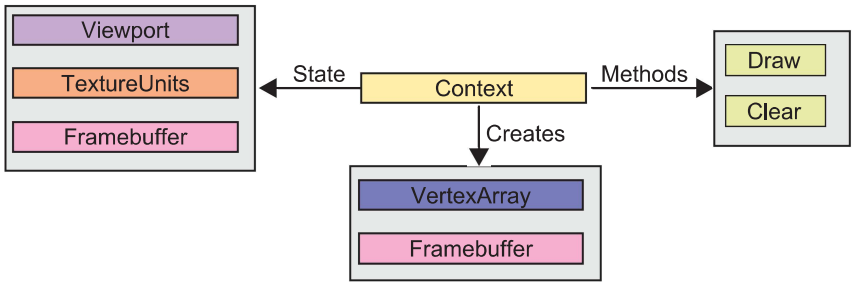


Figure 3.3. Context is used to issue rendering commands, `Draw` and `Clear` and create objects that are not shared between contexts. Rendering methods rely on context-level state, such as the framebuffer, and arguments passed to the methods describing the render state, etc.

A Context is part of a GraphicsWindow, which is created using Device `.CreateWindow`. A graphics window represents the canvas for drawing and contains the context and events for rendering, window resizing, and input handling. The objects created using Device can be shared among multiple contexts, except, of course, GraphicsWindow. For example, client code can create two different windows that use the same shaders, vertex/index buffers, textures, etc. This chapter describes these types in detail, with the exception of fences, which are described in Section 10.4 along with OpenGL multithreading.

As shown in Figure 3.3, in addition to issuing render commands, a context is used to create certain renderer objects and contains some state. Objects created by a Context can only be used in that context, unlike objects created by the Device, which can be used in any context. Only two objects fall into this nonshareable category: vertex arrays and framebuffers, both lightweight containers.

Patrick Says ○○◯◯

As much as an interface designer tries not to let implementation details (e.g., the underlying API) influence public interfaces, here is a case where it does. Vertex arrays and framebuffers cannot be shared among contexts in our renderer because GL does not allow them to be shared. Since these are lightweight containers, the GL overhead for making them shareable might be significant relative to the object itself.

A context contains some state: the viewport transformation, the framebuffer to render to, and the textures and samplers to render with. All the

other information needed for rendering, such as a shader program and vertex array, are passed to Context.Draw. Our design uses only context-level state where appropriate to simplify state management for client code. This organization is much different from the state machine used by OpenGL.

The relevant subsections of the Device and Context interfaces are shown in Listings 3.1 and 3.2, respectively.[11]

```
public static class Device
{
  public static GraphicsWindow CreateWindow(int width,
                                             int height);
  public static ShaderProgram CreateShaderProgram(
      string vertexShaderSource,
      string geometryShaderSource,
      string fragmentShaderSource);
  public static VertexBuffer CreateVertexBuffer(
      BufferHint usageHint, int sizeInBytes);
  public static IndexBuffer CreateIndexBuffer(
      BufferHint usageHint, int sizeInBytes);
  public static WritePixelBuffer CreateWritePixelBuffer(
      PixelBufferHint usageHint, int sizeInBytes);
  public static Texture2D CreateTexture2D(
      Texture2DDescription description);
  public static TextureSampler CreateTexture2DSampler(/* ... */);
  public static Fence CreateFence();
  // ...
}
```

Listing 3.1. Device interface.

```
public abstract class Context
{
  public abstract VertexArray CreateVertexArray();
  public abstract Framebuffer CreateFramebuffer();

  public abstract TextureUnits TextureUnits { get; }
  public abstract Rectangle Viewport { get; set; }
  public abstract Framebuffer Framebuffer { get; set; }

  public abstract void Clear(ClearState clearState);
  public abstract void Draw(PrimitiveType primitiveType,
      int offset, int count, DrawState drawState,
      SceneState sceneState);
  // ...
}
```

Listing 3.2. Context interface.

[11] Some members of Context use C# properties. These are signified with { get ; } after the member's name for read-only properties or { get; set; } for read/write properties. In C++ and Java, properties would be implemented with explicit get and/or set methods.

3.2.1 Code Organization

Our renderer is implemented in the OpenGlobe.Renderer.dll assembly. All publicly exposed types are in the `OpenGlobe.Renderer` namespace. All types for the OpenGL 3.3 implementation are in the `OpenGlobe.Renderer.GL3x` namespace and also have a `GL3x` appended to their name. Since these types are an implementation detail, they are defined with internal scope so they are not accessible outside of the renderer assembly.[12] For example, Context is a public type in the `OpenGlobe.Renderer` namespace, and ContextGL3x is an internal type in the `OpenGlobe.Renderer.GL3x` namespace.

Some types in `OpenGlobe.Renderer` do not rely on the underlying API, so they do not have a GL3x counterpart. For example, RenderState is just a container of render-state parameters and does not require any GL calls. Types that do rely on the underlying API are defined as abstract classes with one or more virtual or abstract members.[13] The corresponding GL3x type inherits from this class and overrides the abstract and possibly virtual methods.

3.3 State Management

A major design decision for a renderer is how render state is managed. Render state configures the fixed-function components of the pipeline for rendering and affects draw calls. It includes things like the scissor test, depth test, stencil test, and blending.

3.3.1 Global State

In raw GL, render states are global states that can be set any time, as long as a valid context is current in the calling thread. For example, the depth test can be enabled with a call like:

```
GL.Enable(EnableCap.DepthTest);
```

The simplest way to expose this through a renderer would be to mirror the GL design: provide `Enable` and `Disable` methods and an enumeration defining states that can be enabled and disabled. This design still has the fundamental problem of global state. At any given point in time, is the depth test enabled or disabled? What if a virtual method is called? How did it leave the depth test state? I don't know. Do you?

[12] In C++, this is similar to not exporting the class from the dll.
[13] An abstract method in C# is equivalent to a pure virtual function in C++.

One approach to managing global state is to guarantee a set of states before a method is called, and require the called method to restore any states it changed. For example, our convention could be that the depth test is always enabled, so if a method wants the depth test disabled, it must do something like the following:

```
public virtual void Render()
{
  GL.Disable(EnableCap.DepthTest);
  // ... Draw call
  GL.Enable(EnableCap.DepthTest);
}
```

The obvious problem here is that this can lead to state thrashing, where every method sets and restores the same state. For example, if the above method was called 10 times, it would disable and enable the depth test 10 times, when it only needed to do it once. The driver may optimize away state changes if the state doesn't actually change from draw call to draw call, but there is still some driver overhead associated with calling into GL. This is peanuts compared to the real problem: this design requires the person implementing the method to know what the incoming state is and remember to restore it.

Let's pretend that the developer is able to remember what the incoming state is. We can surround calls to their method with push and pop attribute GL calls, like the following:

```
// ... Set initial states
GL.PushAttrib(AttribMask.AllAttribBits);
GL.PushClientAttrib(ClientAttribMask.ClientAllAttribBits);
Render();
GL.PopClientAttrib();
GL.PopAttrib();

// ...

public virtual void Render()
{
  GL.Disable(EnableCap.DepthTest);
  // ... Draw call
  // No need to restore
}
```

With this design, `Render` is not required to restore any states because the push and pop attribute saved and restored the states. Since it is not known what states `Render` will change, all attributes are pushed and popped, which is likely to be overkill. To add to the fun, these methods were deprecated

in OpenGL 3. Instead of pushing and popping state, we can "restore" the state by explicitly setting the entire state before each call to `Render`, for example:

```
GL.Enable(EnableCap.DepthTest);
// ... Set other states
Render();
```

As before, this still leads to redundant state changes and requires the person implementing `Render` to know what the incoming state is.

Patrick Says ○○◯◯

I've used this approach where the incoming state is well defined and must be preserved. It was sometimes painful because I always had to double check what the incoming state was before writing new code, for example, "This object is translucent so depth writes are disabled, or are they?"

The final evil that comes from global render state is developers can write seemingly harmless code like this:

```
public virtual void Render()
{
  GL.ColorMask(false, false, false, false);
  GL.DepthMask(false);
  // ... Draw call

  GL.ColorMask(true, true, true, true);
  // ... Another draw call
}
```

At first glance, it appears obvious that depth writes are disabled for the second draw call. How obvious would it be if several other state changes were added to this function? What if a developer comments out the call to GL.`DepthMask`? Did they intend that for just the first draw call or every draw call? Whoops. How much easier would it be if, instead, the render state were completely defined for each draw call?

3.3.2 Defining Render State

Just because the GL API uses global state doesn't mean that a renderer has to expose global state. Global state is an implementation detail. A

renderer can present render state using any abstraction it pleases. One approach that eliminates global state and enables sorting by state is to group all render states into a single object that is passed to a draw call. The draw call then does the actual GL calls to set the states before issuing the GL draw call. In this design, there is never any question about what the current state is; there is no current state. Each draw call has its own render state, which may or may not change from call to call.

We define a RenderState class with the following properties:

```
public class RenderState
{
  public PrimitiveRestart PrimitiveRestart { get; set; }
  public FacetCulling FacetCulling { get; set; }
  public RasterizationMode RasterizationMode { get; set; }
  public ScissorTest ScissorTest { get; set; }
  public StencilTest StencilTest { get; set; }
  public DepthTest DepthTest { get; set; }
  public DepthRange DepthRange { get; set; }
  public Blending Blending { get; set; }
  public ColorMask ColorMask { get; set; }
  public bool DepthMask { get; set; }
}
```

Listing 3.3. RenderState Properties.

The types for these properties are all API-agnostic and are defined in `OpenGlobe.Renderer`. They are exactly what you would expect them to be

```
public enum DepthTestFunction
{
  Never,
  Less,
  Equal,
  LessThanOrEqual,
  Greater,
  NotEqual,
  GreaterThanOrEqual,
  Always
}

public class DepthTest
{
  public DepthTest()
  {
    Enabled = true;
    Function = DepthTestFunction.Less;
  }

  public bool Enabled { get; set; }
  public DepthTestFunction Function { get; set; }
}
```

Listing 3.4. DepthTest state.

given their names. For example, the DepthTest consists of an `Enabled` boolean property and a `DepthTestFunction` enumeration that defines the comparison function used when the depth test is enabled (see Listing 3.4).

When default constructed, RenderState properties match the default GL states, with two exceptions. The depth test, DepthTest, is enabled since that is the common case in our engine, and facet culling, RenderState.`FacetCulling`.`Enabled`, is enabled instead of disabled.

All other RenderState properties are similar to DepthTest. It is worth mentioning that the stencil test, RenderState.`StencilTest`, has separate front and back facing state, and blending, RenderState.`Blending`, has separate RGB and alpha blend factors. These objects are just containers; they store state but do not actually set the GL global state.

Client code simply allocates a RenderState and then sets any properties that have inappropriate defaults. For example, the following code defines the render state for billboards (see Chapter 9):

```
RenderState renderState = new RenderState();
renderState.FacetCulling.Enabled = false;
renderState.Blending.Enabled = true;
renderState.Blending.SourceRGBFactor =
    SourceBlendingFactor.SourceAlpha;
renderState.Blending.SourceAlphaFactor =
    SourceBlendingFactor.SourceAlpha;
renderState.Blending.DestinationRGBFactor =
    DestinationBlendingFactor.OneMinusSourceAlpha;
renderState.Blending.DestinationAlphaFactor =
    DestinationBlendingFactor.OneMinusSourceAlpha;
```

Our RenderState and related types are similar to D3D state objects, which group state into coarse-grained objects: ID3D11BlendState, ID3D11DepthStencilState, and ID3D11RasterizerState. The main difference is that the D3D types are immutable and, therefore, cannot be changed once they are created. Immutable types allow some optimizations, but they also reduce the flexibility to client code. For example, with a mutable render state, an object can determine its depth write property right before rendering based on its alpha value. With an immutable render state, the object either needs to create a new render state if its depth write property changes or keep two render states that are selected based on its alpha.

The other difference between D3D state objects and our RenderState is that D3D state objects are still assigned to global state using methods such as ID3D11DeviceContext::`OMSetBlendState`, whereas we completely eliminate global render state by passing objects directly to draw calls.

When I first designed render states for Insight3D, I used an immutable type that was created based on a "template," which defined the actual states. The template was passed to a global factory, which either created a new render state or returned one from its cache. The benefit was that since all render states were known, it was possible to use bucket sort to sort by state (see Section 3.3.6). The problem was, client code was always releasing render states and asking for new ones. I ultimately decided the flexibility of a mutable render state outweighed the performance gains of an immutable one and switched the implementation to use a mutable render state that is sorted at draw time with a comparison sort (e.g., `std::sort`). Ericson describes a similar flexibility versus performance trade-off for state sorting in God of War III [47].

Patrick Says

3.3.3 Syncing GL State with Render State

So far, we've defined RenderState as a container object for render states that affect the fixed-function configuration of the pipeline during draw calls. In order to apply a RenderState, it is passed to a draw call, like the following:

```
RenderState renderState = new RenderState();
// ... Set states
context.Draw(PrimitiveType.Triangles, renderState, /* ... */);
```

Of course, a RenderState does not need to be allocated before every draw call. An object can allocate one or more RenderStates at construction time and set their properties when needed. The same RenderState can also be used with different contexts.

The implementation of ContextGL3x.`Draw` is exactly what you would expect. It uses fine-grained GL calls to sync the GL state with the state passed in. The context keeps a "shadowed" copy of the current state, `_renderState`, so it can avoid GL calls for state that does not need to change. For example, to set the depth test state, Context.`Draw` calls `ApplyDepthTest`:

```
private void ApplyDepthTest(DepthTest depthTest)
{
  if (_renderState.DepthTest.Enabled != depthTest.Enabled)
  {
    Enable(EnableCap.DepthTest, depthTest.Enabled);
    _renderState.DepthTest.Enabled = depthTest.Enabled;
  }
```

```
  if (depthTest.Enabled)
  {
    if (_renderState.DepthTest.Function != depthTest.Function)
    {
      GL.DepthFunc(TypeConverterGL3x.To(depthTest.Function));
      _renderState.DepthTest.Function = depthTest.Function;
    }
  }
}

protected static void Enable(EnableCap enableCap, bool enable)
{
  if (enable)
  {
    GL.Enable(enableCap);
  }
  else
  {
    GL.Disable(enableCap);
  }
}
```

To set the depth function, our renderer's enumeration is converted to the GL value using TypeConverterGL3x.`To`. This can be implemented with a series of if ... else statements, a switch statement, or a table lookup. A robust implementation should verify the enumeration and assert or throw an exception if appropriate. Unless a renderer's enumeration values match the GL values, an enumeration cannot be cast directly to the GL type.

The majority of the code in `OpenGlobe.Renderer.GL3x`.ContextGL3x sets GL states just like the above snippet. When shadowing state, it is important that the shadowed copy never becomes out of sync with the GL state. When the context is constructed, the GL state should be synced with the shadowed state (see ContextGL3x.`ForceApplyRenderState`), and the shadowed state should always be updated when a GL state change is made.

Try This ○○◯◯

If several draw calls are made with the same render state, as done when rendering terrain with geometry clipmapping in Chapter 13, it may be worth avoiding all the fine-grained if statements that compare individual properties of the shadowed state with the render state passed in. Implement a quick accept, coarse-grained check by remembering the last render state instance used and its "version," which is just an integer that is incremented every time a property of the render state changes. ContextGL3x.`Draw` can skip all fine-grained checks if the passed-in render state reference equals the shadowed render state reference and their versions match, indicating that the same render state was used in the last draw call.

3.3.4 Draw State

Although we've described Context.`Draw` as taking a RenderState, it actually takes a higher-level container object called DrawState. In order to issue a draw call, the render state is needed to configure the fixed-function pipeline but another state is required to configure other parts of the pipeline. Notably, a shader program that is executed in response to the draw called is required (see Section 3.4), as is a vertex array referencing vertex buffers and an optional index buffer (see Section 3.5).

In GL, these are global states; the shader program is specified using `glUseProgram`,[14] and the vertex array is specified using `glBindVertexArray`, both before issuing a draw call. In D3D, these are also global state, bound using ID3D11DeviceContext methods; shader stages are bound using `*SetShader` methods such as `VSSetShader` and `PSSetShader`, and vertex-array state is bound using `IA*` methods such as `IASetVertexBuffers` and `IASetIndexBuffer`.

These global states lead to the same problems as global render states. The solution is the same: group them into a container that is passed to a draw call. DrawState is such a container; it includes the render state, shader program, and vertex array used for drawing, as shown in Listing 3.5.

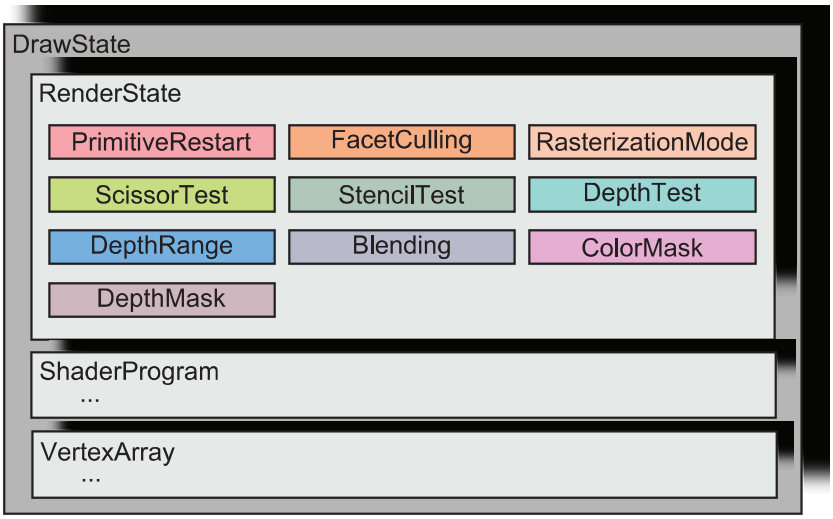


Figure 3.4. DrawState and RenderState properties. A DrawState object is passed to a draw call, which eliminates the need for global render, shader, and vertex-array states.

[14] When ARB_separate_shader_objects is supported, `glBindProgramPipeline` can be used to bind a pipeline program instead [87].

```
public class DrawState
{
  // ... Constructors
  public RenderState RenderState { get; set; }
  public ShaderProgram ShaderProgram { get; set; }
  public VertexArray VertexArray { get; set; }
}
```

Listing 3.5. DrawState properties.

This also enables sorting by shader, as explained in Section 3.3.6. A diagram showing DrawState properties is shown in Figure 3.4. A draw call does more than just call `glUseProgram` for the shader program and `glBindVertexArray` for the vertex array, but let's save that discussion until Sections 3.4 and 3.5, respectively.

Patrick Says ○○◯◯

When I first designed Context.`Draw`, it did take just a RenderState object. Client code was responsible for "binding" a shader program and vertex array to the context before calling `Draw`. Then I realized these were unnecessary global states, with the same problems associated with global render states, so I combined them with RenderState into a higher-level container named DrawState. When abstracting any API, it is important to remember that you don't need to let that API's implementation details show through in your design. This is a great ideal to strive for, but sometimes easier said than done.

3.3.5 Clear State

A number of states that affect draw calls also affect clearing the framebuffer. Clearing is different than drawing though, because geometry does not need to go through the pipeline and a shader program is not executed. Although one could clear the framebuffer by rendering a full screen quad, this is not a good practice because it doesn't take advantage of fast clears, which properly initialize buffers used for compression and hierarchical z-culling [136].

In GL, clears are affected by a number of states, including the scissor test and color, depth, and stencil masks. Clears also rely on color, depth, and stencil clear-value states set with `glClearColor`, `glClearDepth`, and `glClearStencil`, respectively. Once the states are configured, one or more buffers are cleared by calling `glClear`.

D3D has a different design that does not rely on global state; ID3D11DeviceContext::`ClearRenderTargetView` clears a render target (e.g., a color

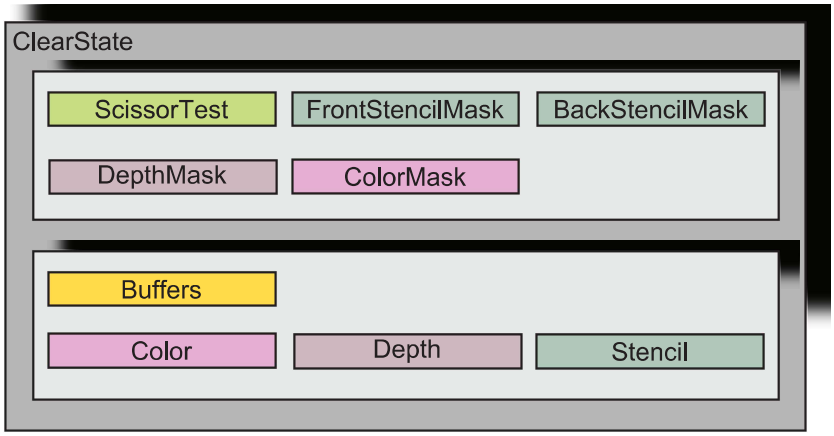


Figure 3.5. ClearState properties. Similar to how DrawState is used for draw calls, a ClearState object is passed to clear calls, which eliminates the need for related global states.

buffer) and ID3D11DeviceContext::`ClearDepthStencilView` clears a depth and/or stencil buffer. The clear values are passed as arguments instead of global state. Separate clear methods for the depth and stencil buffers are not provided; clearing them together is more efficient because they are usually stored in the same buffer [136].

Our renderer design for clears is similar to D3D, even though it is implemented using GL. It uses a container object, ClearState, shown in Listing 3.6 and Figure 3.5, to encapsulate the state needed for clearing. One of its members, `ClearBuffers`, is a bitmask that defines which buffers are cleared. To clear one or more buffers in the framebuffer, a ClearState object is created and passed to Context.`Clear`. For example, the following client code clears only the depth and stencil buffers:

```
ClearState clearState = new ClearState();
clearState.Buffers = ClearBuffers.DepthBuffer |
                     ClearBuffers.StencilBuffer;
context.Clear(clearState);
```

The ContextGL3x.`Clear` implementation is a series of straightforward GL calls based on the ClearState passed in. Similar to shadowing render state, the color, depth, and stencil clear values are shadowed to avoid unnecessary GL calls.

```
[Flags]
public enum ClearBuffers
{
  ColorBuffer = 1,
  DepthBuffer = 2,
  StencilBuffer = 4,
  ColorAndDepthBuffer = ColorBuffer | DepthBuffer,
  All = ColorBuffer | DepthBuffer | StencilBuffer
}

public class ClearState
{
  // ... Constructor

  public ScissorTest ScissorTest { get; set; }
  public ColorMask ColorMask { get; set; }
  public bool DepthMask { get; set; }
  public int FrontStencilMask { get; set; }
  public int BackStencilMask { get; set; }

  public ClearBuffers Buffers { get; set; }
  public Color Color { get; set; }
  public float Depth { get; set; }
  public int Stencil { get; set; }
}
```

Listing 3.6. ClearState properties.

It is important to note that the scissor test does not affect clears in D3D, but it does affect clears in GL. As mentioned in Section 3.1, designing a renderer for portability has some challenges. In this case, if both GL and D3D are supported, the scissor test may be removed from ClearState and always disabled for clears. Alternatively, the D3D implementation could assert or throw an exception if the scissor test is enabled, although this forces client code to know which renderer it is using.

3.3.6 Sorting by State

A common optimization is to sort by state. Doing so avoids wasted CPU driver overhead and helps exploit the parallelism of the GPU (see Section 10.1.2) by minimizing pipeline stalls [52]. The most expensive state changes are those that require large-scale reconfiguration of the pipeline, such as changing shaders, depth test state, or blending state.

State sorting is not done by our renderer itself but, instead, can be implemented on top of it by sorting by DrawState and issuing Context `.Draw` calls in an order that minimizes changing expensive states. It is also common to sort by texture or combine multiple textures in a single texture, as discussed in Section 9.2.

One approach to rendering a scene with state sorting is to use three passes:

1. First, hierarchical culling determines a list of visible objects.
2. Next, the visible objects are sorted by state.
3. Finally, the visible objects are drawn in sorted order.

Of course, there are many variations. If all the states are known before traversing the scene, the first two passes can be combined, and state sorting basically comes for free. During initialization, a list of sorted buckets is allocated with one bucket for each possible state. Each frame is rendered in two passes:

1. Hierarchical culling determines the visible objects and drops each into a bucket based on its state. Mapping an object to a bucket based on its state can be done in $\mathcal{O}(1)$ time if each state is given a unique index.
2. Next, the already sorted buckets are traversed and objects in non-empty buckets are drawn. After a bucket is rendered, it is cleared in preparation for the next frame.

This design assumes all possible states are known at initialization time and that the number of possible states is on the order of the number of objects. That is, this design is not ideal if there are 100,000 possible states and only ten objects.

For some scenes, hierarchical culling is not required. If the set of states is known in advance and objects do not change state, the scene can be rendered in a single pass that simply draws the objects in sorted order, culling or not culling individually along the way.

Regardless of when state sorting occurs, a sort order needs to be defined. This can easily be done with our DrawState object using a comparison method such as `CompareDrawStates` shown in Listing 3.7. This methods returns -1 when `left` $<$ `right`, 1 when `left` $>$ `right`, and 0 when `left` $=$ `right`. To sort by state, `CompareDrawStates` can be provided to List <T>.`Sort`, similar to how a function object is passed to `std::sort` in C++.[15]

A comparison method should compare the most expensive state first, then compare less expensive states until an ordering can be determined. The method `CompareDrawStates` sorts first by shader, then by depth test enabled. It should then move on, comparing other render states such as the depth compare function and blending states. The problem with `CompareDraw States`'s implementation is it gets long as more states are compared, and the large number of branches may hurt performance when used for sorting every frame.

[15] To be exact, the function object used by `std::sort` requires only a strict weak ordering, whereas the comparison method shown here is a full ordering.

```
private static int CompareDrawStates(DrawState left,
                                     DrawState right)
{
  // Sort by shader first
  int leftShader = left.ShaderProgram.GetHashCode();
  int rightShader = right.ShaderProgram.GetHashCode();

  if (leftShader < rightShader)
  {
    return -1;
  }
  else if (leftShader > rightShader)
  {
    return 1;
  }

  // Shaders are equal, compare depth test enabled
  int leftEnabled =
      Convert.ToInt32(left.RenderState.DepthTest.Enabled);
  int rightEnabled =
      Convert.ToInt32(right.RenderState.DepthTest.Enabled);

  if (leftEnabled < rightEnabled)
  {
    return -1;
  }
  else if (rightEnabled > leftEnabled)
  {
    return 1;
  }

  // ... Continue comparing other states in order of most to ↩
      least expensive...
  return 0;
}
```

Listing 3.7. A comparison method for sorting by DrawState. First it sorts by shader, then by depth test enabled. A complete implementation would sort by additional states.

In order to write a more concise and efficient comparison method, RenderState can be changed to store all the sortable states in one or more bitmasks. Each bitmask stores the most expensive states in the most significant bits and the less expensive states in the least significant bits. The large number of individual render-state comparisons in the comparison method is then replaced with a few comparisons using the bitmasks. This can also significantly reduce the amount of memory used by RenderState, which helps cache performance.

Try This ○○◯◯

Modify RenderState so a bitmask can be used for state sorting.

The render state used in Insight3D uses a bitmask to store state with a bit ordering such that states are sorted according to Forsyth's advice [52]. This has the advantage of using very little memory and enabling a simple and efficient functor for sorting. Debugging can be tricky, though, because it is not obvious what the current render state is given that individual states are jammed into a bitmask. When using a design like this, it is worth writing debug code to easily visualize the states in the bitmask.

Patrick Says

It is also common to sort by things other than state. For example, all opaque objects are usually rendered before all translucent objects. Opaque objects may be sorted by depth near to far to take advantage of z-buffer optimizations, as discussed in Section 12.4.5, and translucent objects may be sorted far to near for proper blending. State sorting is also not fully compatible with multipass rendering because each pass depends on the results of the previous pass, making it hard to reorder draw calls.

3.4 Shaders

In our renderer, shader support involves more interfaces and code than any of the other major renderer components, and rightfully so: shaders are at the center of modern 3D engine design. Even mobile devices supporting OpenGL ES 2.0 have very capable vertex and fragment shaders. Today's desktops have a highly programmable pipeline, including programmable vertex, tessellation control, tessellation evaluation, geometry, and fragment-shading stages. In D3D, the tessellation stages are called hull and domain, respectively, and a fragment shader is called a pixel shader. We use the GL terminology. This section focuses on abstracting the programmable stages supported by OpenGL 3.x and Direct3D 10 class hardware: vertex, geometry, and fragment stages.

Add support for tessellation control and evaluation shaders to the renderer. How should client code react if tessellation is not supported by the hardware?

Try This

3.4.1 Compiling and Linking Shaders

To begin, let's consider compiling and linking shaders with a simple example:

```
string vs = // ...
string fs = // ...
ShaderProgram sp = Device.CreateShaderProgram(vs, fs);
```

Two strings, `vs` and `fs`, contain the vertex and fragment-shader source code. A geometry shader is optional and not shown here. The source could be a hard-coded string, a procedurally generated string from different code snippets, or a string read from disk. Most of our examples store shader source in `.glsl` files, separate from C# source files. These files are included in the C# project and marked as an embedded resource so they are compiled into the assembly.[16] At design time, shaders are in individual files, but at runtime, they are embedded inside the `.dll` or `.exe` file. A shader's source is retrieved at runtime with a helper function that retrieves a string given a resource's name:

```
string vs = EmbeddedResources.GetText(
    "OpenGlobe.Scene.Globes.RayCasted.Shaders.GlobeVS.glsl");
```

Patrick Says ○○◯◯

I've organized shaders many ways over the years, and I am convinced that embedded resources is the most convenient. Since shaders are part of the assembly, this approach does not require additional files at runtime but still provides the convenience of individual source files at design time. Unlike hard-coded strings and shaders generated procedurally at runtime, shaders embedded as resources can easily be authored in third party tools. There is still something to be said for the potential performance advantages of procedurally generated shaders though, which I've used in a few places in Insight3D. For hardware that supports ARB_shader_subroutine, there should be less of a need to procedurally generate shaders since different shader subroutines can be selected at runtime, similar to virtual calls [20].

[16] In C++ on Windows, shader source files can be embedded as user-defined resources and accessed with `FindResource` and related functions.

Strings containing shader source are passed to Device.`CreateShaderProgram` to create a shader program, fully constructed and ready to be used for rendering (of course, the user may want to set uniforms first). As mentioned in Section 3.3.4, a shader program is part of DrawState. It does not need to be bound to the context like in vanilla GL or D3D;[17] instead, a shader program is specified to each draw call, as part of DrawState, to avoid global state. Note that clearing the framebuffer does not invoke a shader, so a shader program is not part of ClearState.

Our GL implementation of a ShaderProgram is in ShaderProgramGL3x, which uses a helper class for shader objects, ShaderObjectGL3x. A shader object represents one programmable stage of the pipeline and is a GL renderer implementation detail. It is not part of the renderer abstraction, which represents all stages together in one shader program. It can be useful to expose interfaces for individual pipeline stages, especially when using ARB_separate_shader_objects, but we did not have the need.

The ShaderObjectGL3x constructor calls `glCreateShader`, `glShaderSource`, `glCompileShader`, and `glGetShader` to create and compile a shader object. If compilation fails, the renderer throws a custom CouldNotCreateVideoCardResourceException exception. This is one of the benefits of a renderer abstraction layer: it can convert error return codes into object-oriented exceptions.

The ShaderProgramGL3x constructor creates shader objects, which are then linked into a shader program using `glCreateProgram`, `glAttachShader`, `glLinkProgram`, and `glGetProgram`. Similar to a compiler error, a link error is converted into an exception.

All of these GL calls show another benefit of a renderer: conciseness. A single call to Device.`CreateShaderProgram` in client code constructs a ShaderProgramGL3x, which makes many GL calls, including proper error handling, on the client's behalf.

Besides adding object-oriented constructs and making client code more concise, a renderer allows adding useful building blocks for shader authors. Our renderer has built-in constants, which are constants available to all shaders but not defined in GLSL by default. These constants are always prefixed with `og_`. A few built-in constants are shown in Listing 3.8. The values for these GLSL constants are computed using C#'s Math constants and constants in OpenGlobe's static Trig class.

Shader authors can simply use these constants, without explicitly declaring them. This is implemented in ShaderObjectGL3x by providing an array of two strings to `glShaderSource`: one with the built-in constants and another with the actual shader source. In GLSL, a #version directive must

[17] In D3D, "shader programs" do not exist; instead, individual shader stages are bound to the context. This is also possible in GL, with ARB_separate_shader_objects.

```
float og_pi = 3.14159265358979;
float og_oneOverPi = 0.318309886183791;
float og_twoPi = 6.28318530717959;
float og_halfPi = 1.5707963267949;
```

Listing 3.8. Selected GLSL constants built into our renderer.

be the first source line and cannot be repeated, so we include #version 330 as the first line of the built-in constants and comment out the #version line if it was provided in the actual source.

Try This ○○◯◯

In D3D, shader linking is not required. Instead, shaders are compiled (potentially offline) with `D3D10CompileShader`, and objects for individual stages are created with ID3D11DeviceContext calls like `CreateVertexShader` and `CreatePixelShader`, then bound to individual stages for rendering with calls such as `VSSetShader` and `PSSetShader`. This is also possible in GL with ARB_get_program_binary [103] and ARB_separate_shader_objects.

Use these extensions to modify the renderer to support offline compilation and to treat shader stages as individual objects and properties of DrawState. In OpenGL, the binary shader may be rejected if it was compiled for different hardware or even compiled using a different driver version, so a fallback method must be provided.

Try This ○○◯◯

Since 3D engines are designed for use by multiple applications, it would be useful for applications to define their own built-in constants that are available for all shaders. Modify the renderer to support this.

Try This ○○◯◯

Built-in constants are useful, but built-in reusable functions are even more useful. How would you add support for this to the renderer? Would it work the same way as built-in constants? Or is a #include mechanism preferred? What are the trade-offs?

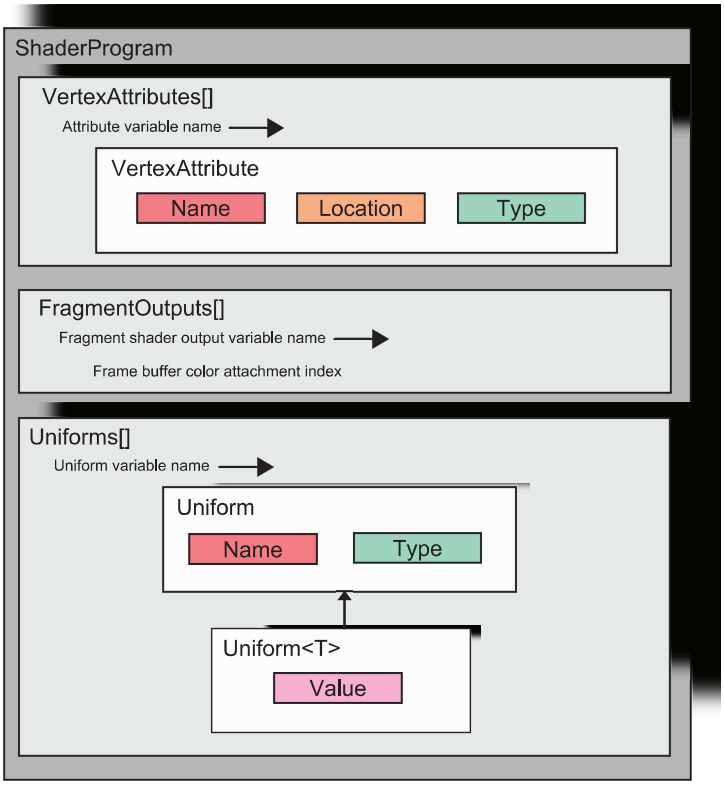


Figure 3.6. ShaderProgram properties. A shader program contains a collection of attribute variables, a collection of fragment-shader output variables, and a collection of uniforms, all indexable by name.

After compiling and linking, a few collections, shown in Listing 3.9 and Figure 3.6, are populated based on the shader program's inputs and outputs. We consider these collections separately in the following sections.

```
public abstract class ShaderProgram : Disposable
{
  public abstract ShaderVertexAttributeCollection
      VertexAttributes { get; }
  public abstract FragmentOutputs FragmentOutputs  { get; }
  public abstract UniformCollection Uniforms { get; }

  // ...
}
```

Listing 3.9. Common ShaderProgram properties.

3.4.2 Vertex Attributes

Client code needs to be able to assign vertex buffers to named vertex attributes in GLSL vertex shaders. This is how attributes receive data. Each attribute has a numeric location used to make the connection. After linking, ShaderProgramGL3x queries its active attributes using `glGetProgram`, `glGetActiveAttrib`, and `glGetAttribLocation` and populates its publicly exposed collection of attributes used in the vertex shader: `VertexAttributes` in Listing 3.9 and Figure 3.6. Each item in the collection includes an attribute's name, numeric location, and type (for example, Float, FloatVector4), as shown below:

```
public class ShaderVertexAttribute
{
  // ... Constructor
  public string Name { get; }
  public int Location { get; }
  public ShaderVertexAttributeType DataType { get; }
}
```

For example, a vertex shader may have two active attributes:

```
in vec4 position;
in vec3 normal;
```

The associated shader program would find and expose two attributes:

	Name	Location	Type
sp.VertexAttributes["position"]	"position"	0	FloatVector4
sp.VertexAttributes["normal"]	"normal"	1	FloatVector3

This collection of attributes is passed to Context.`CreateVertexArray` (see Section 3.5.3) to connect vertex buffers to attributes. Client code can rely on the shader program to create this collection, but sometimes, it is convenient to not have to create a shader program before creating a vertex array.

To achieve this, the attributes, including their location, need to be known in advance. Client code can use this information to explicitly create a ShaderVertexAttributeCollection instead of relying on the one provided by a shader program.

Our renderer includes built-in constants, shown in Listing 3.10, that can be used to explicitly specify a vertex attribute's location directly in the

```
#define og_positionVertexLocation           0
#define og_normalVertexLocation             1
#define og_textureCoordinateVertexLocation  2
#define og_colorVertexLocation              3
```

Listing 3.10. Built-in constants for specifying vertex attribute locations.

vertex-shader source. The attribute declarations in the vertex shader would look like the following:

```
layout(location = og_positionVertexLocation) in vec4 position;
layout(location = og_normalVertexLocation) in vec3 normal;
```

Instead of relying on a shader program to construct a collection of vertex attributes, client code can do it with code like the following:

```
ShaderVertexAttributeCollection vertexAttributes =
    new ShaderVertexAttributeCollection();
vertexAttributes.Add(new ShaderVertexAttribute(
    "position", VertexLocations.Position,
    ShaderVertexAttributeType.FloatVector4, 1));
vertexAttributes.Add(new ShaderVertexAttribute(
    "normal", VertexLocations.Normal,
    ShaderVertexAttributeType.FloatVector3, 1));
```

Here, a static class VertexLocations contains the constants for the vertex location for position and normal. The constants in VertexLocations are the C# equivalent of the GLSL constants; in fact, VertexLocations is used to procedurally create the GLSL constants.

The client code still needs to know about the attributes used in a shader program, but it doesn't have to create the shader program first. By specifying locations as built-in constants, locations for certain vertex attributes, such as position or normals, can be consistent across an application.

In Section 3.5.3, we will see how exactly the attribute collection is used to connect vertex buffers to attributes.

Our renderer does not support array vertex attributes. Add support for them.

○○○○ Try This

3.4.3 Fragment Outputs

Similar to how vertex attributes defined in a vertex shader need to be connected to vertex buffers in a vertex array, fragment-shader output variables need to be attached to framebuffer color attachments when multiple color attachments are used. Thankfully, the process is even simpler than that for vertex attributes.

Most of our fragment shaders just have a single output variable, usually declared as out vec3 `fragmentColor`; or out vec4 `fragmentColor`;, depending on if an alpha value is written. Usually, we rely on the fixed function to write depth, but some fragment shaders write it explicitly by assigning to gl_FragDepth, such as the shader for GPU ray casting a globe in Section 4.3.

When a fragment shader has multiple output variables, they need to be connected to the appropriate framebuffer color attachments. Similar to vertex attribute locations, client code can do this by asking a shader program for its output variable locations or by explicitly assigning locations in the fragment-shader source. For the former, ShaderProgram exposes a FragmentOutputs collection, shown earlier in Listing 3.9 and Figure 3.6. This maps an output variable's name to its numeric location. It is implemented in FragmentOutputsGL3x using `glGetFragDataLocation`.

For example, consider a fragment shader with two output variables:

```
out vec4 dayColor;
out vec4 nightColor;
```

Client code can simply ask the shader program for the location of an output variable given its name and assign a texture to that color attachment location:

```
framebuffer.ColorAttachments[sp.FragmentOutputs("dayColor")] = ↩
    dayTexture;
```

Section 3.7 goes into more detail on framebuffers and their color attachments. Alternatively, a fragment shader can explicitly declare its output variable locations with the same syntax used to declare vertex attribute locations:

```
layout(location = 0) out vec4 dayColor;
layout(location = 1) out vec4 nightColor;
```

Now client code can know an output variable's location before creating a shader program.

3.4.4 Uniforms

Vertex buffers provide vertex shaders with data that can vary every vertex. Typical vertex components include positions, normals, texture coordinates, and colors. These types of data usually vary per vertex, but other types of data vary less frequently and would be wasteful to store per vertex. Uniform variables are a way to provide data to any shader stage that vary only up to every primitive. Since uniforms are set before a draw call, completely separate from vertex data, they are typically the same for many primitives. Common uses for uniforms include the model-view-projection transformation; light and material properties, such as diffuse and specular components; and application-specific values, such as the position of the sun or the viewer's altitude.

A shader program presents a collection of active uniforms called `Uniforms`, shown in Listing 3.9 and Figure 3.6, so client code can inspect uniforms and change their values. A shader-authoring tool may want to know what uniforms a shader program uses, so it can present text boxes, color selectors, etc., to modify their values. Most of our example code already knows what uniforms are used and simply needs to set their values, either once after creating a ShaderProgram or before every draw call. This section describes how a shader program in our renderer presents uniforms and makes GL calls to modify them. The following section discusses uniforms that are set automatically by the renderer.

Our renderer supports uniform data types defined by UniformType, which include the following:

- float, int, and bool scalars.
- float, int, and bool 2D, 3D, and 4D vectors. In GLSL, these are types like vec4, ivec4, and bvec4, which correspond to Vector4F, Vector4I, and Vector4B, respectively, in OpenGlobe.
- $2\times2, 3\times3, 4\times4, 2\times3, 2\times4, 3\times2, 3\times4, 4\times2$, and 4×3 float matrices, where an n x m matrix has n columns and m rows. In GLSL, these are types like mat4 and mat2x3, which correspond to Matrix4F and Matrix23, respectively.
- Sampler uniforms used to refer to textures (see Section 3.6), each of which is really just a scalar int.

Our most commonly used uniforms are 4×4 matrices, floating-point vectors, and samplers.

```
public class Uniform
{
  // ... Protected constructor

  public string Name { get; }
  public UniformType DataType { get; }
}

public abstract class Uniform<T> : Uniform
{
  protected Uniform(string name, UniformType type)
    : base(name, type)
  {
  }

  public abstract T Value { set; get; }
}
```

Listing 3.11. Each uniform is accessed by client code using an instance of Uniform<T>.

Try This ○○◯◯

OpenGL also includes unsigned ints and array uniforms. Add support to the renderer for them.

Try This ○○◯◯

OpenGL 4.1 includes support for uniform types with double precision. Double precision is a "killer feature" for virtual globes, as described in Chapter 5. Add support to the renderer for double-precision uniforms. Since OpenGL 4.x requires different hardware than 3.x, how should this be designed?

After a shader program is linked, ShaderProgramGL3x uses `glGetProgram`, `glGetActiveUniform`, `glGetActiveUniforms`, and `glGetUniformLocation` to populate its collection of uniforms. Each uniform in the collection is an instance of Uniform<T> shown in Listing 3.11 and Figure 3.7.

Let's consider a shader with three uniforms:

```
uniform mat4 modelViewPerspectiveMatrix;
uniform vec3 sunPosition;
uniform sampler2D diffuseTexture;
```

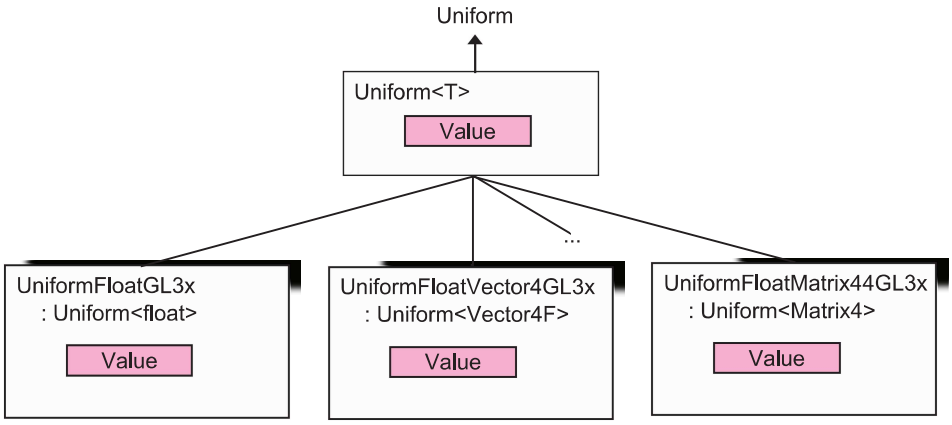


Figure 3.7. Client code sets uniform values using Uniform<T> since different types of uniforms need different data types for their `Value` properties. For each possible `T`, there is a GL implementation that calls the right GL function (e.g., `glUniform1f`, `glUniform4f`).

Once client code creates a ShaderProgram for a shader with these uniforms, the `Uniforms` collection will contain three Uniform objects:

	Name	Type (UniformType)
Uniforms["modelView PerspectiveMatrix"]	"modelView PerspectiveMatrix"	FloatVector4
Uniforms["sunPosition"]	"sunPosition"	FloatVector3
Uniforms["diffuseTexture"]	"diffuseTexture"	Sampler2D

The `Uniforms` collection presents uniforms through their base class, Uniform, so client code needs to cast a uniform to the appropriate Uniform <T> type to access its `Value` property. For example, the three uniforms in the above example would be assigned values with code like the following:

```
((Uniform<Matrix4F>)sp.Uniforms["modelViewPerspectiveMatrix"]).
    Value = new Matrix4F(/* ... */);
((Uniform<Vector4F>)sp.Uniforms["sunPosition"]).Value =
    new Vector4F(/* ... */);
// Texture unit zero
((Uniform<int>)sp.Uniforms["diffuseTexture"]).Value = 0;
```

If the uniform is set before every draw call, the string lookup and cast should be done just once and the Uniform<T> cached in client code for use before every draw call:

```
Uniform<Matrix4F> u =
    (Uniform<Matrix4F>)sp.Uniforms["modelViewPerspectiveMatrix"];

// ...

while (/* ... */)
{
  u.Value = // ...
  context.Draw(/* ... */);
}
```

If the client code does not know what Uniform<T> to cast to, it can check the `Type` property and then make the appropriate cast.

Our GL implementation implements each Uniform<T> in a different class. The inheritance relationship between Uniform<T> and a few GL classes is shown in Figure 3.7. The naming convention is straightforward: Uniform<float> is implemented in UniformFloatGL3x, and so on. The code ShaderProgram GL3x.`CreateUniform` uses the uniform type returned by `glGetActiveUniform` to create the appropriate Uniform<T>-derived type.

Each GL class stores a copy of the uniform's value and makes the appropriate `glUniform*` call to sync the value with GL. When a user assigns a value to the `Value` property, the GL call is not made immediately. Instead, a delayed technique is used [32]. The code ShaderProgramGL3x keeps a collection of all of its uniforms and a separate list of "dirty" uniforms. When client code sets a uniform's `Value` property, the uniform saves the value and adds itself to the shader program's "dirty list" if it wasn't already added.

When a draw call is made, the shader program that is part of the DrawState is bound to the context with `glUseProgram`, then the program is "cleaned." That is, a `glUniform*` call is made for each uniform in the dirty list, and then the dirty list is cleared. If most of a program's uniforms change from draw call to draw call, this implementation could be changed to not use a dirty list and, instead, just iterate through the collection of all uniforms, making a call to `glUniform*` for each. A highly tuned engine could do timings at runtime and use the more efficient approach.

The benefit of this delayed technique is that no redundant calls to `glUniform*` are made if client code redundantly sets a uniform's `Value` property. A context doesn't even need to be current when setting the property. It also simplifies GL state management since a call to `glUniform*` requires the corresponding shader program be bound.[18]

Direct3D 9 has constants that are similar to GL uniforms. A shader's constants can be queried and set using ID3DXConstantTable. D3D 10 replaced constants with constant buffers, which are buffers containing con-

[18] Using ARB_separate_shader_objects also simplifies state management since the program is passed directly to `glProgramUniform*` instead of needing to be bound.

stants that should be grouped together based on their expected update frequency. Buffers can improve performance by reducing per-constant overhead. OpenGL introduced a similar feature called uniform buffers.

Try This

Our renderer includes support for uniform buffers. See UniformBlock* and their implementations in UniformBlock*GL3x, as well as ShaderProgramGL3x.`FindUniformBlocks`. When these were originally coded, drivers were not stable enough to allow their use for examples throughout the book. Now that stable drivers are available, how would you organize uniforms into uniform buffers? Given that D3D 10 dropped support for constants, should our renderer drop support for uniforms and only expose uniform buffers?

3.4.5 Automatic Uniforms

Uniforms are a convenient way to provide shaders with variables that are constant across one or more draw calls. Thus far, our renderer requires client code to explicitly assign every active uniform of every shader using Uniform<`T`>. Some uniforms, like the model-view matrix and the camera position, will be the same across many draw calls or even the entire frame. It is inconvenient for client code to have to explicitly set these uniforms for every shader, or even at all. Wouldn't it be great if the renderer defined a collection of uniforms that are automatically set before a draw call? Wouldn't it be even better if client code could add new uniforms to this collection? After all, one of the reasons to use a renderer in the first place is to add functionality above the underlying API (see Section 3.1).

Our renderer defines a number of these uniforms and a framework for adding new ones. We call these automatic uniforms; they go by different names in various engines: preset uniforms, engine uniforms, engine variables, engine parameters, etc. The idea is always the same: a shader author simply declares a uniform (e.g., mat4 `og_modelViewMatrix`) and it is automatically set, without the need to write any client code.

Similar to the built-in GLSL constants of Section 3.4.5, our automatic uniform names are prefixed with `og_`. Our implementation is similar to Cozzi's [31], although we divide automatic uniforms into two types:

- *Link automatic.* Uniforms that are set just once after a shader is compiled and linked.
- *Draw automatic.* Uniforms that need to be set as often as every draw call.

The vast majority of our automatic uniforms are draw automatic. Link automatics are obviously less useful but have less overhead since they do not need to be set per draw call. An automatic uniform implements either LinkAutomaticUniform or DrawAutomaticUniform and DrawAutomaticUniformFactory. These abstract classes are shown in Listing 3.12.

A link automatic simply needs to provide its name and know how to assign its value to a Uniform. The Device stores a collection of link automatics. After a shader is compiled and linked, ShaderProgram.`InitializeAutomaticUniforms` iterates through the shader's uniforms, and if any uniforms match the name of a link automatic, the abstract `Set` method is called to assign a value to the uniform.

The only link automatic in our renderer is `og_textureN`, which assigns a sampler uniform to texture unit N. For example, sampler2D `og_texture1` defines a 2D sampler that is automatically set to texture unit 1. Listing 3.13 shows how this would be implemented.

Draw automatics are slightly more involved than link automatics. Similar to link automatics, the Device stores a collection of all draw automatic factories. Unlike with link automatics, when ShaderProgram.`InitializeAutomatic Uniforms` comes across a draw automatic, it cannot simply assign a value because the value can vary from draw call to draw call. Instead, the uniform factory creates the actual draw automatic uniform, which is stored in a collection of draw automatics for the particular shader. Before each draw call, the shader is "cleaned"; it iterates through its draw automatic uniforms and calls the abstract `Set` method to assign their values.

Automatic uniforms are implemented independently of the GL renderer, except that ShaderProgramGL3x needs to call the protected `Initial`

```
public abstract class LinkAutomaticUniform
{
  public abstract string Name { get; }
  public abstract void Set(Uniform uniform);
}

public abstract class DrawAutomaticUniformFactory
{
  public abstract string Name { get; }
  public abstract DrawAutomaticUniform Create(Uniform uniform);
}

public abstract class DrawAutomaticUniform
{
  public abstract void Set(Context context, DrawState drawState,
      SceneState sceneState);
}
```

Listing 3.12. Abstract classes for automatic uniforms.

```
internal class TextureUniform1 : LinkAutomaticUniform
{
  public override string Name
  {
    get { return "og_texture1"; }
  }

  public override void Set(Uniform uniform)
  {
    ((Uniform<int>)uniform).Value = 1;
  }
}
```

Listing 3.13. An example link automatic uniform.

`izeAutomaticUniforms` after compiling and linking and `SetDrawAuto maticUniforms` when it is cleaned. Individual automatic uniforms, such as those in Listings 3.13 and 3.15, are implemented without any knowledge of the underlying API.

Given the signature of DrawAutomaticUniform.`Set` in Listing 3.12, a draw automatic can use the Context and the DrawState to determine the value to assign to its uniform. But how does this help with uniforms like the model-view matrix and the camera's position? It doesn't, of course. The third argument to `Set` is a new type called SceneState that encapsulates scene-level state like transformations and the camera, as shown in Listing 3.14. Similar to DrawState, a SceneState is passed to a draw call, which ultimately makes its way to an automatic uniform's `Set` method. An application can use one SceneState for all draw calls or use different SceneStates as appropriate.

```
public class Camera
{
  // ...

  public Vector3D Eye { get; set; }
  public Vector3D Target { get; set; }
  public Vector3D Up { get; set; }
  public Vector3D Forward { get; }
  public Vector3D Right { get; }

  public double FieldOfViewX { get; }
  public double FieldOfViewY { get; set; }
  public double AspectRatio { get; set; }

  public double PerspectiveNearPlaneDistance { get; set; }
  public double PerspectiveFarPlaneDistance { get; set; }

  public double Altitude(Ellipsoid shape);
}

public class SceneState
{
```

```
  // ...

  public float DiffuseIntensity { get; set; }
  public float SpecularIntensity { get; set; }
  public float AmbientIntensity { get; set; }
  public float Shininess { get; set; }

  public Camera Camera { get; set; }
  public Vector3D CameraLightPosition { get; }

  public Matrix4D ComputeViewportTransformationMatrix(Rectangle
      viewport, double nearDepthRange, double farDepthRange);
  public static Matrix4D ComputeViewportOrthographicMatrix(
      Rectangle viewport);
  public Matrix4D OrthographicMatrix { get; }
  public Matrix4D PerspectiveMatrix { get; }
  public Matrix4D ViewMatrix { get; }
  public Matrix4D ModelMatrix { get; set; }
  public Matrix4D ModelViewPerspectiveMatrix { get; }
  public Matrix4D ModelViewOrthographicMatrix { get; }
  public Matrix42 ModelZToClipCoordinates { get; }
}
```

Listing 3.14. Selected Camera and SceneState members used to implement draw automatic uniforms.

Let's firm up our understanding of draw automatics by looking at the implementation for `og_wgs84Height` in Listing 3.15. This floating-point uniform is the height of the camera above the WGS84 ellipsoid (see Section 2.2.1), basically the viewer's altitude, neglecting terrain. One use for this is to fade in/out layers of vector data.

The factory implementation in Listing 3.15 is straightforward, as is the implementation for all draw automatic factories; one property simply returns the uniform's name, and the other method actually creates the draw automatic uniform. The automatic uniform itself, Wgs84Height Uniform, asks the scene state's camera for the height above the WGS84 ellipsoid and assigns it to the uniform.

```
internal class Wgs84HeightUniformFactory :
    DrawAutomaticUniformFactory
{
  public override string Name
  {
    get { return "og_wgs84Height"; }
  }

  public override DrawAutomaticUniform Create(Uniform uniform)
  {
    return new Wgs84HeightUniform(uniform);
  }
}

internal class Wgs84HeightUniform : DrawAutomaticUniform
{
```

```
  public Wgs84HeightUniform(Uniform uniform)
  {
    _uniform = (Uniform<float>)uniform;
  }

  public override void Set(Context context, DrawState drawState,
                           SceneState sceneState)
  {
    _uniform.Value =
        (float)sceneState.Camera.Height(Ellipsoid.Wgs84);
  }

  private Uniform<float> _uniform;
}
```

Listing 3.15. Implementation for `og_wgs84Height` draw automatic uniform.

Since automatic uniforms are found based on name only, it is possible that a shader author may declare an automatic uniform with the wrong data type (e.g., int `og_wgs84Height` instead of float `og_wgs84Height`), resulting in an exception at runtime when a cast to Uniform<T> fails. Add more robust error handling that reports the data type mismatch or only detects an automatic uniform if its name and data type match.

Try This ○○◯◯

Without caching camera and scene-state values, automatic uniforms can introduce CPU overhead by asking the camera or scene state to redundantly compute things. For example, in Wgs84HeightUniform, the call to Camera.`Height` is likely to be redundant for each shader using the `og_wgs84Height` automatic uniform. Add caching to SceneState and Camera to avoid redundant computations.

Try This ○○◯◯

How can uniform buffers, UniformBlock*, simplify automatic uniforms?

Try This ○○◯◯

Table 3.1 lists the most common automatic uniforms used in examples throughout this book. We've only included uniforms that we actually use in our examples, but many others are useful in the general sense. Draw automatic uniforms don't even have to set a uniform's value before every

GLSL Uniform Name	Source Renderer Property/Method	Description
vec3 og_cameraEye	Camera.Eye	The camera's position in world coordinates.
vec3 og_cameraLightPosition	SceneState.CameraLightPosition	The position of a light attached to the camera in world coordinates.
vec4 og_diffuseSpecularAmbientShininess	DiffuseIntensity, SpecularIntensity, AmbientIntensity, and Shininess SceneState properties	Lighting equation coefficients.
mat4 og_modelViewMatrix	SceneState.ModelViewMatrix	The model-view matrix, which transforms model coordinates to eye coordinates.
mat4 og_modelViewOrthographicMatrix	SceneState.ModelViewOrthographicMatrix	The model-view-orthographic projection matrix, which transforms model coordinates to clip coordinates using an orthographic projection.
mat4 og_modelViewPerspectiveMatrix	SceneState.ModelViewPerspectiveMatrix	The model-view-perspective projection matrix, which transforms model coordinates to clip coordinates using a perspective projection.
mat4x2 og_modelZToClipCoordinates	SceneState.ModelZToClipCoordinates	A 4×2 matrix that transforms a Z component in model coordinates to clip coordinates, which is useful in GPU ray casting (see Section 4.3).
float og_perspectiveFarPlaneDistance	Camera.PerspectiveFarPlaneDistance	The distance from the camera to the far plane. Used for a logarithmic depth buffer (see Section 6.4).
float og_perspectiveNearPlaneDistance	Camera.PerspectiveNearPlaneDistance	The distance from the camera to the far plane, which is useful for clipping against the near plane when rendering wide lines (see Section 7.3.3).
mat4 og_perspectiveMatrix	SceneState.PerspectiveMatrix	The perspective projection matrix, which transforms eye coordinates to clip coordinates.
mat4 og_viewportOrthographicMatrix	SceneState.ComputeViewportOrthographicMatrix()	The orthographic projection matrix for the entire viewport.
mat4 og_viewportTransformationMatrix	SceneState.ComputeViewportTransformationMatrix()	The viewport transformation matrix, which transforms normalized device coordinates to window coordinates, which is used for rendering wide lines and billboards.
mat4 og_viewport	Context.Viewport	The viewport's left, bottom, width, and height values.
float og_wgs84Height	Camera.Altitude()	The camera's altitude above the WGS84 ellipsoid.
sampler* og_texture$N$	n/a	A texture bound to texture unit N, $0 \le N <$ Context.TextureUnits.Count.

Table 3.1. Selected renderer automatic uniforms. Many of these replace the deprecated OpenGL built-in uniforms.

draw call. In Insight3D, some draw automatic uniforms set a sampler uniform once and actually bind a texture (see Section 3.6) (e.g., the terrain's depth or silhouette texture) before each draw call. A similar technique can be done with noise textures.

3.4.6 Caching Shaders

Section 3.3.6 discussed the performance benefits of sorting `context.Draw` calls by DrawState. In Listing 3.7, the following code was used to sort by shader:

```
private static int CompareDrawStates(DrawState left,
                                     DrawState right)
{
  int leftShader = left.ShaderProgram.GetHashCode();
  int rightShader = right.ShaderProgram.GetHashCode();

  if (leftShader < rightShader)
  {
    return -1;
  }
  else if (leftShader > rightShader)
  {
    return 1;
  }
  // ... If shaders are the same, sort by other state.
}
```

In order to sort by shader, we need to be able to compare shaders. We are not concerned with the final sorted order, just that shaders that are the same wind up next to each other, in order to minimize the number of actual shader changes required (i.e., calls to `glUseProgram` in the GL implementation). Above, the C# method `GetHashCode` determines the ordering, similar to how a C++ implementation might use the shader's memory address for sorting. The key to sorting by shader is that shaders that are the same need to be the same ShaderProgram instance.

How many unique instances of ShaderPrograms does the following create?

```
string vs = // ...
string fs = // ...
ShaderProgram sp = Device.CreateShaderProgram(vs, fs);
ShaderProgram sp2 = Device.CreateShaderProgram(vs, fs);
```

Even though both shaders are created with the same source, this creates two distinct ShaderProgram instances, just like making similar GL calls would.

```
public class ShaderCache
{
  public ShaderProgram FindOrAdd(
      string key,
      string vertexShaderSource,
      string fragmentShaderSource);
  public ShaderProgram FindOrAdd(
      string key,
      string vertexShaderSource,
      string geometryShaderSource,
      string fragmentShaderSource);
  public ShaderProgram Find(string key);
  public void Release(string key);
}
```

Listing 3.16. ShaderCache interface.

For sorting by shader, we want both shaders to be the same instance so the compare method can group them next to each other.

This calls for a shader cache, which is implemented in ShaderCache. A shader cache simply maps a unique key to a ShaderProgram instance. We use user specified strings as keys, but one could also use integers or even the shader source itself.

The shader cache is part of the renderer, but it is not dependent on a particular API; it only deals with ShaderPrograms. As shown in Listing 3.16, the shader cache supports adding, finding, and releasing shaders.

Reference counting is used to track the number of clients with references to the same shader. When a shader is added to the cache, its reference count is one. Each time it is retrieved from the cache, either via `FindOrAdd` or `Find`, its count is increased. When a client is finished with a cached shader, it calls `Release`, which decreases the count. When the count reaches zero, the shader is removed from the cache.

Our example that created two distinct ShaderPrograms with the same shader source can be rewritten to use a shader cache, so only one ShaderProgram instance is created, making state sorting work as expected:

```
string vs = // ...
string fs = // ...
ShaderCache cache = new ShaderCache();
ShaderProgram sp = cache.FindOrAdd("example key", vs, fs);
ShaderProgram sp2 = cache.Find("example key");
// sp and sp2 are the same instance
cache.Release("example key");
cache.Release("example key");
```

The call to `Find` could be replaced with `FindOrAdd`. The difference is that `Find` does not require having the shader's source code and will return null if the key is not found, whereas `FindOrAdd` will create the shader if the key

is not found. When procedurally generating shaders, `Find` is useful because the cache can be checked without procedurally generating the entire source.

Client code will usually only want one shader cache, but nothing is stopping clients from creating several. The implementation of ShaderCache is a straightforward use of a Dictionary[19] that maps a string to a ShaderProgram and its reference count. Since multiple threads may want to access a shader cache, each method is protected with a coarse-grained lock to serialize access to the shared cache. See Chapter 10 for more information on parallelism and threads.

One could argue that a coarse-grained lock hurts parallelism here. In particular, since the lock is held while a shader is compiled and linked (e.g., it calls Device.`CreateShaderProgram`), only one shader can be compiled/linked at a time, regardless of how many threads are used. In practice, is this really a problem? How would you change the locking strategy to allow multiple threads to compile/link in parallel, assuming the driver itself is not holding a coarse-grained lock preventing this?

◯◯◯◯ Try This

Why not build caching directly into Device.`CreateShaderProgram`? What are the trade-offs between doing so and our design?

◯◯◯◯ Question

When I first implemented a shader cache in Insight3D, I created a cache for both shader programs and shader objects. I found the cache for shader programs useful to enable sorting by shader, but the cache for shader objects wasn't all that useful, except for minimizing the number of GL shader objects created. In our renderer, this is not a concern because the concept of shader object doesn't exist; only entire shader programs are exposed.

◯◯◯◯ Patrick Says

[19] In C++, a `std::map` or `std::hash_map` could be used.

3.5 Vertex Data

A primary source of vertex-shader input is attributes that vary from vertex to vertex. Attributes are generally used for values such as positions, normals, and texture coordinates. For example, a vertex shader may declare the following attributes:

```
in vec3 position;
in vec3 normal;
in vec2 textureCoordinate;
```

To clear up some terminology, a vertex is composed of attributes. Above, a vertex is composed of a position, normal, and texture-coordinate attribute. An attribute is composed of components. Above, the position attribute is composed of three floating-point components, and the texture-coordinate attribute is composed of two (see Figure 3.8).

In our renderer, vertex buffers store data used for attributes, and index buffers store indices used to select vertices for rendering. Both types of buffers are shown in Figure 3.9.

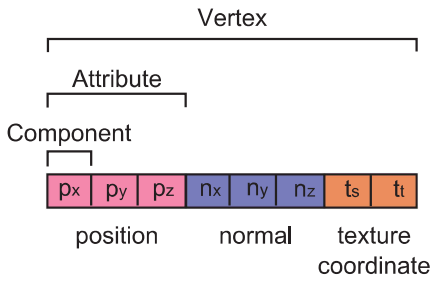


Figure 3.8. A vertex is composed of one or more attributes, which are composed of one or more components.

3.5.1 Vertex Buffers

Vertex buffers are raw, untyped buffers that store attributes in driver-controlled memory, which could be GPU memory, system memory, or both. Clients can copy data from arrays in system memory to a vertex buffer's driver-controlled memory or vice versa. The most common thing our examples do is compute positions on the CPU, store them in an array, and then copy them to a vertex buffer.

Vertex buffers are represented using the abstract class VertexBuffer, whose GL implementation is VertexBufferGL3x. A vertex buffer is

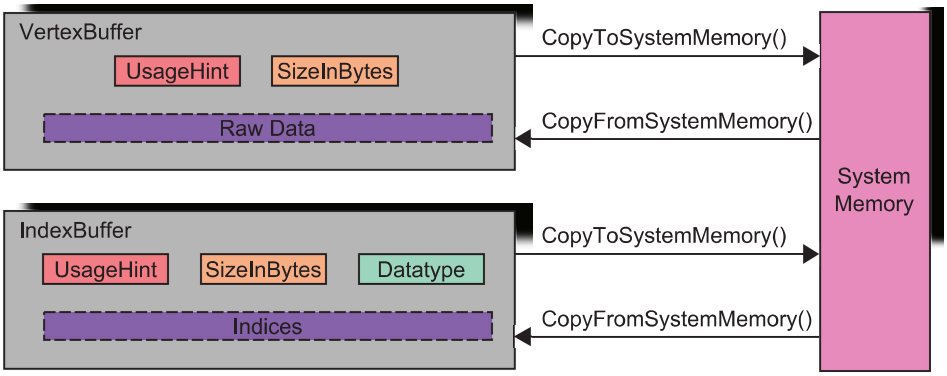


Figure 3.9. Vertex and index buffers. A vertex buffer is an untyped buffer, containing one or more attributes, that is interpreted once assigned to a vertex array. An index buffer is fully typed, having an explicit `Datatype` property. Both buffers have methods to copy their contents to and from system memory.

created using Device.`CreateVertexBuffer`, whose implementation is shown in Listing 3.17. The client only needs to provide two arguments to create a vertex buffer: a usage hint indicating how they intend to copy data to the buffer and the size of the buffer in bytes.

The usage hint is defined by the BufferHint enumeration, which can have three values:

- `StaticDraw`. The client will copy data to the buffer once and draw with it many times. Many vertex buffers in virtual globes should use this hint, which is likely to make the driver store the buffer in GPU memory.[20] Vertex buffers for things like terrain and stationary vector data are usually used for many frames and will benefit significantly from not being sent over the system bus every frame.
- `StreamDraw`. The client will copy data to the buffer once and draw with it at most a few times (e.g., think of a streaming video, where each video frame is used just once). In virtual globes, this hint is useful for real-time data. For example, billboards may be rendered to display the position of every commercial flight in the United States. If position updates are received frequently enough, the vertex buffer storing positions should use this hint.
- `DynamicDraw`. The client will repeatedly copy data to the buffer and draw with it. This does not imply the entire vertex buffer will change

[20] An OpenGL driver is also likely to shadow a copy in system memory to handle what D3D calls a lost device where the GPU resources need to be restored.

```
public static class Device
{
  // ...

  public static VertexBuffer CreateVertexBuffer(
      BufferHint usageHint, int sizeInBytes)
  {
    return new VertexBufferGL3x(usageHint, sizeInBytes);
  }
}
```

Listing 3.17. `CreateVertexBuffer` implementation.

```
public abstract class VertexBuffer : Disposable
{
  public virtual void CopyFromSystemMemory<T>(
      T[] bufferInSystemMemory) where T : struct;
  public virtual void CopyFromSystemMemory<T>(
      T[] bufferInSystemMemory,
      int destinationOffsetInBytes) where T : struct;
  public abstract void CopyFromSystemMemory<T>(
      T[] bufferInSystemMemory,
      int destinationOffsetInBytes,
      int lengthInBytes) where T : struct;

  public virtual T[] CopyToSystemMemory<T>() where T : ↩
      struct;
  public abstract T[] CopyToSystemMemory<T>(
      int offsetInBytes, int sizeInBytes) where T : struct;

  public abstract int SizeInBytes { get; }
  public abstract BufferHint UsageHint { get; }
}
```

Listing 3.18. VertexBuffer interface.

repeatedly; the client may only update a subset each frame. In the previous example, this hint is useful if only a subset of aircraft positions change each update.

It is worth experimenting with BufferHint to see which yields the best performance for your scenario. Keep in mind that these are just hints passed to the driver, although many drivers take these hints seriously, and thus, they affect performance. In our GL renderer implementation, BufferHint corresponds to the `usage` argument passed to `glBufferData`. See NVIDIA's white paper for more information on setting it appropriately [120].

The VertexBuffer interface is shown in Listing 3.18. Client code copies data from an array or subset of an array to the vertex buffer with a `CopyFromSystemMemory` overload and copies to an array using a `Copy`

`ToSystemMemory` overload. The implementor (e.g., VertexBufferGL3x) only has to implement one version of each method, as the virtual overloads delegate to the abstract overload.

Add a `CopyFromSystemMemory` overload that includes a `sourceOffset InBytes` argument, so copying doesn't have to begin at the start of the array. When is this overload useful?

Try This

Even though a vertex buffer is typeless, that is, it contains raw data and does not interpret them as any particular data type, the copy methods use C# generics[21] to allow client code to copy to and from arrays without casting. This is just syntactic sugar; the buffer itself should be considered to be raw bytes until it is interpreted using a vertex array (see Section 3.5.3).

A vertex buffer may store just a single attribute type, such as positions, or it may store multiple attributes, such as positions and normals, either interleaving each position and normal or storing all positions followed by all normals. These approaches are shown in Figure 3.10 and are described in the following three sections.

Separate buffers. One approach is to use a separate vertex buffer for each attribute type, as shown in Figure 3.10(a). For example, if a vertex shader requires positions, normals, and texture coordinates, three different vertex buffers are created. The position vertex buffer would be created and populated with code like the following:

```
Vector3F[] positions = new Vector3F[]
{
  new Vector3F(1, 0, 0),
  new Vector3F(0, 1, 0)
};

int sizeInBytes = ArraySizeInBytes.Size(positions);
VertexBuffer positionBuffer = Device.CreateVertexBuffer(
    BufferHint.StaticDraw, sizeInBytes);
positionBuffer.CopyFromSystemMemory(positions);
```

An array of 3D floating point-vectors is copied to a vertex buffer storing positions. Similar code would be used to copy normal and texture-coordinate arrays into their vertex buffers.

[21] C# generics are similar to C++ templates.

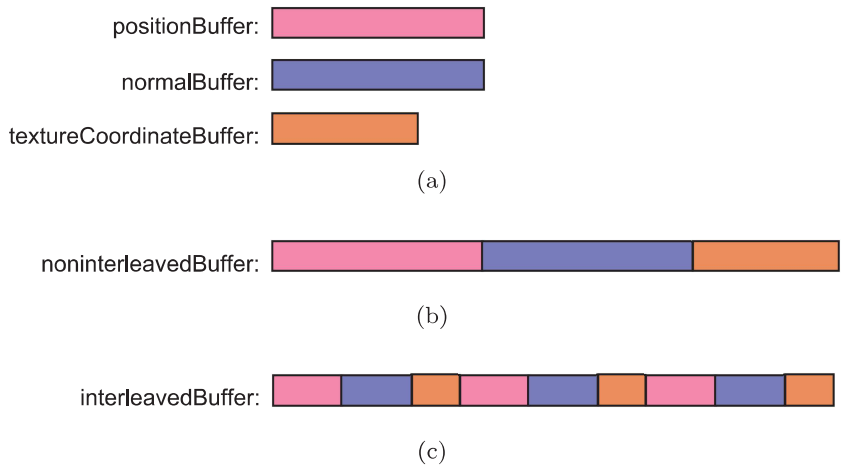


Figure 3.10. (a) Separate buffers for positions, normals, and texture coordinates. Storing each attribute in a separate buffer is the most flexible approach. (b) A single noninterleaved buffer stores positions, followed by normals, and then texture coordinates, requiring only one buffer for multiple attributes. (c) A single interleaved buffer stores all three components, generally resulting in the best performance for static data [14].

The strength of this approach is flexibility. Each buffer can be created with a different BufferHint. For example, if we are rendering billboards whose positions change frequently, we may create a vertex buffer for positions using `StreamDraw` and a vertex buffer for texture coordinates, which do not change, using `StaticDraw`. The flexibility of separate buffers also allows us to reuse a vertex buffer across multiple batches (e.g., multiple batches of billboards may use different vertex buffers for positions but the same vertex buffer for texture coordinates).

Noninterleaved buffers. Instead of storing one attribute per vertex buffer, all attributes can be stored in a single vertex buffer, as shown in Figure 3.10(b). Creating fewer vertex buffers helps reduce per-buffer overhead.

Visualize the vertex buffer as an array of raw bytes that we can copy any data into, since, for example, the data type for positions and textures coordinates may be different. The `CopyFromSystemMemory` overload taking an offset in bytes that describes where to start copying into the vertex buffer should be used to concatenate arrays of attributes in the vertex buffer:

```
Vector3F[] positions = // ...
Vector3F[] normals = // ...
Vector2H[] textureCoordinates = // ...
```

```
VertexBuffer vertexBuffer = Device.CreateVertexBuffer(
    BufferHint.StaticDraw,
    ArraySizeInBytes.Size(positions) +
    ArraySizeInBytes.Size(normals) +
    ArraySizeInBytes.Size(textureCoordinates));

int normalsOffset = ArraySizeInBytes.Size(positions);
int textureCoordinatesOffset =
    normalsOffset + ArraySizeInBytes.Size(normals);

vertexBuffer.CopyFromSystemMemory(positions);
vertexBuffer.CopyFromSystemMemory(normals, normalsOffset);
vertexBuffer.CopyFromSystemMemory(textureCoordinates,
                                  textureCoordinatesOffset);
```

Although noninterleaved attributes use only a single buffer, memory coherence may suffer because a single vertex composed of multiple attributes will need to be fetched from different parts of memory. We have not used this approach in practice, and instead, interleave attributes in a single buffer.

Interleaved buffers. As shown in Figure 3.10(c), a single vertex buffer can store multiple attributes by interleaving each attribute. That is, if the buffer contains positions, normals, and texture coordinates, a single position is stored, followed by a normal and then a texture coordinate. The interleaved pattern continues for each vertex.

A common way to achieve this is to store each component for a vertex in a struct laid out sequentially in memory:

```
[StructLayout(LayoutKind.Sequential)]
public struct InterleavedVertex
{
  public Vector3F Position { get; set; }
  public Vector3F Normal { get; set; }
  public Vector2H TextureCoordinate { get; set; }
}
```

Then, create an array of these structs:

```
InterleavedVertex[] vertices = new InterleavedVertex[]
{
   new InterleavedVertex()
   {
      Position = new Vector3F(1, 0, 0),
      Normal = new Vector3F(1, 0, 0),
      TextureCoordinate = new Vector2H(0, 0),
   },
   // ...
};
```

Finally, create a vertex buffer that is the same size as the array, in bytes, and copy the entire array to the vertex buffer:

```
VertexBuffer vertexBuffer = Device.CreateVertexBuffer(
    BufferHint.StaticDraw, ArraySizeInBytes.Size(vertices));
vertexBuffer.CopyFromSystemMemory(vertices);
```

When rendering large, static meshes, interleaved buffers outperform their counterparts [14]. A hybrid technique can be used to get the performance advantages of interleaved buffers and the flexibility of separate buffers. Since multiple vertex buffers can be pulled from in the same draw call, a static interleaved buffer can be used for attributes that do not change, and a dynamic or stream vertex buffer can be used for attributes that change often.

GL renderer implementation. Our GL vertex buffer implementation in VertexBufferGL3x makes straightforward use of GL's buffer functions with a target of `GL_ARRAY_BUFFER`. The constructor creates the name for the buffer object with `glGenBuffers`. When the object is disposed, `glDeleteBuffers` deletes the buffer object's name.

The constructor also allocates memory for the buffer with `glBufferData` by passing it a null pointer. The object `CopyFromSystemMemory` calls `glBufferSubData` to copy data from the input array to the buffer object. It is likely that the driver is implemented such that the initial call to `glBufferData` does not have much overhead, and instead, the allocation occurs at the first call to `glBufferSubData`. Alternatively, our implementation could delay calling `glBufferData` until the first call to `CopyFromSystemMemory`, at the cost of some additional bookkeeping.

A call to `glBufferData` and `glBufferSubData` is always preceded with calls to `glBindBuffer`, to ensure that the correct GL buffer object is modified, and `glBindVertexArray`(0), to ensure a GL vertex array is not accidentally modified.

Try This ○○◯◯

Instead of updating a GL buffer object with `glBufferData` and `glBufferSubData`, the buffer or a subset of it can be mapped into the application's address space using `glMapBuffer` or `glMapBufferRange`. It can then be modified using a pointer like any other memory. Add this feature to the renderer. What are the advantages of this approach? What challenges does using a managed language like C# impose?

3.5.2 Index Buffers

Index buffers are represented using the abstract class IndexBuffer, whose GL implementation is IndexBufferGL3x. The interface and implementation are nearly identical to vertex buffers. Client code usually looks similar, like the following example, which copies indices for a single triangle into a newly created index buffer:

```
ushort[] indices = new ushort[] { 0, 1, 2 };

IndexBuffer indexBuffer = Device.CreateIndexBuffer(
    BufferHint.StaticDraw, indices.Length * sizeof(ushort));
indexBuffer.CopyFromSystemMemory(indices);
```

Unlike vertex buffers, index buffers are fully typed. Indices can be either unsigned shorts or unsigned ints, as defined by the IndexBufferData type enumeration. Client code does not need to explicitly declare the data type for indices in an index buffer. Instead, the generic parameter `T` of `CopyFromSystemMemory` is used to determine the data type.

What are the design trade-offs between having client code explicitly declare a data type for the index buffer compared to implicitly determining the data type when `CopyFromSystemMemory` is called?

○○○○ Question

Clients should strive to use the data type that uses the least amount of memory but still allows indexing fully into a vertex buffer. If 64 K or fewer vertices are used, unsigned short indices suffice, otherwise unsigned int indices are called for. In practice, we have not noticed performance differences between unsigned shorts and unsigned ints, although "size is speed" in many cases, and using less memory, especially GPU memory, is always a good practice.

A useful renderer feature is to trim indices down to the smallest sufficient data type. This allows client code to always work with unsigned ints, while the renderer only allocates unsigned shorts if possible. Design and implement this feature. Under what circumstances is this not desirable?

○○○○ Try This

The only noteworthy GL implementation difference between our index and vertex buffers is that the index buffer uses the `GL_ELEMENT_ARRAY_BUFFER` target instead of `GL_ARRAY_BUFFER`.

3.5.3 Vertex Arrays

Vertex and index buffers simply store data in driver-controlled memory; they are just buffers. A vertex array defines the actual components making up a vertex, which are pulled from one or more vertex buffers. A vertex array also references an optional index buffer that indexes into these vertex buffers. Vertex arrays are represented using the abstract class VertexArray, shown in Listing 3.19 and Figure 3.11.

Each attribute in a vertex array is defined by a VertexBufferAttribute, shown in Listing 3.20. Attributes are accessed using a zero-

```
public abstract class VertexArray : Disposable
{
  public virtual VertexBufferAttributes Attributes { get; }
  public virtual IndexBuffer IndexBuffer { get; set; }
}
```

Listing 3.19. VertexArray interface.

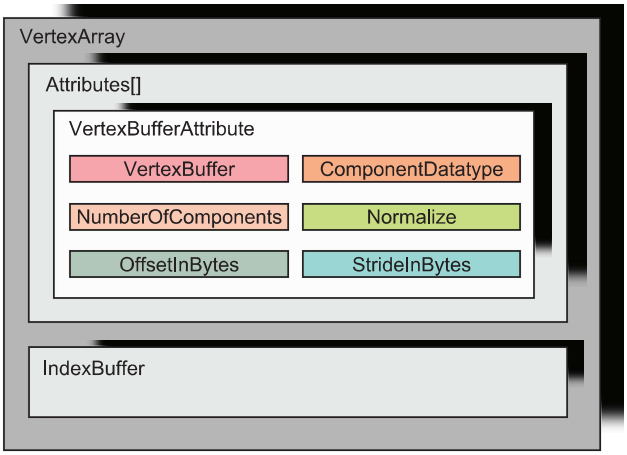


Figure 3.11. A vertex array contains one or more vertex attributes (e.g., position, normal) and an optional index buffer. The `VertexArray` member of DrawState and the arguments passed to Context.`Draw` determine the vertices used for rendering.

```
public enum ComponentDatatype
{
  Byte,
  UnsignedByte,
  Short,
  UnsignedShort,
  Int,
  UnsignedInt,
  Float,
  HalfFloat,
}

public class VertexBufferAttribute
{
  // ... Constructors
  public VertexBuffer VertexBuffer { get; }
  public ComponentDatatype ComponentDatatype { get; }
  public int NumberOfComponents { get; }
  public bool Normalize { get; }
  public int OffsetInBytes { get; }
  public int StrideInBytes { get; }
}
```

Listing 3.20. VertexBufferAttribute interface.

based index that corresponds to the shader's attribute location (see Section 3.4.2). Each vertex array supports up to Device.`MaximumNumberOfVertexAttributes` attributes.

An attribute is defined by the vertex buffer containing the data, its component's data type (ComponentDatatype), and its number of components. Unlike indices, which only support unsigned shorts and unsigned ints, a vertex component supports these signed and unsigned integral types, as well as bytes, floats, and half-floats. The latter are useful for saving memory for texture coordinates in the range $[0, 1]$ compared to floats.

If each attribute type is in a separate vertex buffer, it is straightforward to create a vertex array referencing each component. A VertexBuffer Attribute referencing the appropriate vertex buffer is created for each component as shown below:

```
VertexArray va = window.Context.CreateVertexArray();
va.Attributes[0] = new VertexBufferAttribute(
    positionBuffer, ComponentDatatype.Float, 3);
va.Attributes[1] = new VertexBufferAttribute(
    normalBuffer, ComponentDatatype.Float, 3);
va.Attributes[2] = new VertexBufferAttribute(
    textureCoordinatesBuffer, ComponentDatatype.HalfFloat, 2);
```

Positions and normals are each composed of three floating-point components (e.g., x, y, and z), and texture coordinates are composed of two

floating-point components (e.g., s and t). If a single vertex buffer contains multiple attribute types, the `OffsetInBytes` and `StrideInBytes` VertexBufferAttribute properties can be used to select attributes from the appropriate parts of the vertex buffer. For example, if positions, normals, and texture coordinates are stored noninterleaved in a single vertex buffer, the `OffsetInBytes` property can be used to select the starting point for pulling from a vertex buffer:

```
Vector3F[] positions = // ...
Vector3F[] normals = // ...
Vector2H[] textureCoordinates = // ...

int normalsOffset = ArraySizeInBytes.Size(positions);
int textureCoordinatesOffset =
    normalsOffset + ArraySizeInBytes.Size(normals);

// ...

va.Attributes[0] = new VertexBufferAttribute(positionBuffer,
    ComponentDatatype.Float, 3);
va.Attributes[1] = new VertexBufferAttribute(normalBuffer,
    ComponentDatatype.Float, 3, false, normalsOffset, 0);
va.Attributes[2] = new VertexBufferAttribute(
    textureCoordinatesBuffer, ComponentDatatype.HalfFloat, 2,
    false, textureCoordinatesOffset, 0);
```

Finally, if attributes are interleaved in the same vertex buffer, the `StrideInBytes` property is also used to set the stride between each attribute since attributes are no longer adjacent to each other:

```
int normalsOffset = SizeInBytes<Vector3F>.Value;
int textureCoordinatesOffset =
    normalsOffset + SizeInBytes<Vector3F>.Value;

va.Attributes[0] = new VertexBufferAttribute(
    positionBuffer, ComponentDatatype.Float, 3,
    false, SizeInBytes<InterleavedVertex>.Value);
va.Attributes[1] = new VertexBufferAttribute(
    normalBuffer, ComponentDatatype.Float, 3,
    false, normalsOffset, SizeInBytes<InterleavedVertex>.Value);
va.Attributes[2] = new VertexBufferAttribute(
    textureCoordinatesBuffer, ComponentDatatype.HalfFloat, 2,
    false, textureCoordinatesOffset,
    SizeInBytes<InterleavedVertex>.Value);
```

As shown earlier in Listing 3.5 and Figure 3.4, DrawState has a VertexArray member that provides vertices for rendering. The renderer's public interface has no global state for the "currently bound vertex array"; instead, client code provides a vertex array to every draw call.

3.5.4 GL Renderer Implementation

The GL calls that configure a vertex array are in VertexArrayGL3x and VertexBufferAttributeGL3x. A GL name for the vertex array is created with `glGenVertexArrays` and, of course, eventually deleted with `glDeleteVertexArrays`. Assigning a component or index buffer to a vertex array does not immediately result in any GL calls. Instead the calls are delayed until the next draw call that uses the vertex array, to simplify state management [32]. When a vertex array is modified, it is marked as dirty. When a draw call is made, the GL vertex array is bound with `glBindVertexArray`. If it is dirty, its dirty components are cleaned by calling `glDisableVertexAttribArray` or calling `glEnableVertexAttribArray`, `glBindBuffer`, and `glVertexAttribPointer` to actually modify the GL vertex array. Likewise, a call to `glBindBuffer` with `GL_ELEMENT_ARRAY_BUFFER` is used to clean the vertex array's index buffer.

In ContextGL3x, the actual draw call is issued with `glDrawRangeElements` if an index buffer is used or with `glDrawArrays` if no index buffer is present.

3.5.5 Vertex Data in Direct3D

Our renderer abstractions for vertex buffers, index buffers, and vertex arrays also map nicely to D3D. In D3D, vertex and index buffers are created with ID3D11Device::`CreateBuffer`, similar to calling GL's `glBufferData`. The function `CreateBuffer` has a `D3D11_BUFFER_DESC` argument that describes the type of buffer to create, its size in bytes, and a usage hint, among other things. The data for the buffer itself (e.g., the array passed to our `CopyFromSystemMemory` methods) are passed to D3D's `CreateBuffer` as part of the `D3D11_SUBRESOURCE_DATA` argument.

A buffer can be updated using ID3D11DeviceContext::`UpdateSubresource`, similar to GL's `glBufferSubData`, or by mapping a pointer to the buffer using ID3D11DeviceContext::`Map`, similar to GL's `glMapBuffer`.

Although D3D does not have an identical object to GL's vertex array object, our VertexArray interface can still be implemented through D3D methods for binding an input layout, vertex buffer(s), and an optional index buffer to the input-assembler stage, similar to what `glBindVertexArray` achieves in GL. An input-layout object is created using ID3D11Device::`CreateInputLayout`, which describes the components in a vertex buffer. This object is bound to the input-assembler stage before rendering using ID3D11DeviceContext::`IASetInputLayout`. One or more vertex buffers are bound using ID3D11DeviceContext::`IASetVertexBuffers`, and an optional index buffer is bound using ID3D11DeviceContext::`IASetIndexBuffer`. The

index data type is specified as an argument, which can be either a 16-bit or 32-bit unsigned integer.

Draw calls are issued using methods on the ID3D11DeviceContext interface. In addition to binding the input layout, vertex buffer(s), and index buffer, the primitive type is also bound using `IASetPrimitiveTopology`. Finally, a draw call is issued. A call to `Draw` would be used in place of GL's `glDrawArrays`, and a call to `DrawIndexed` would be used in place of `glDrawRangeElements`.

3.5.6 Meshes

Even with our renderer abstractions, creating vertex buffers, index buffers, and vertex arrays requires quite a bit of bookkeeping. In many cases, in particular those for rendering static meshes, client code shouldn't have to concern itself with how many bytes to allocate, how to organize attributes into one or more vertex buffers, or which vertex-array attributes correspond to which shader attribute locations. Luckily, the renderer allows us to raise the level of abstraction and simplify this process. Low-level vertex and index buffers are available to clients, but clients can also use higher-level types when ease of use is more important than fine-grained control.

OpenGlobe.Core contains a Mesh class shown in Listing 3.21 and Figure 3.12. A mesh describes geometry. It may represent the ellipsoidal surface of the globe, a tile of terrain, geometry for billboards, a model for a building, etc. The Mesh class is similar to VertexArray in that it contains vertex attributes and optional indices. A key difference is Mesh has strongly typed vertex attributes, as opposed to the renderer's VertexBuffer, which requires a VertexBufferAttribute to interpret its attributes.

The Mesh class is defined in OpenGlobe.Core because it is just a container. It is not used directly for rendering, it does not depend on any renderer types, and it does not create any GL objects under the hood, so it doesn't belong in OpenGlobe.Renderer. It simply contains data. An overload of Context.`CreateVertexArray` takes a mesh as input and creates a vertex array containing vertex and index buffers based on the mesh. This

```
public class Mesh
{
  public VertexAttributeCollection Attributes { get; }
  public IndicesBase Indices { get; set; }

  public PrimitiveType PrimitiveType { get; set; }
  public WindingOrder FrontFaceWindingOrder { get; set; }
}
```

Listing 3.21. Mesh interface.

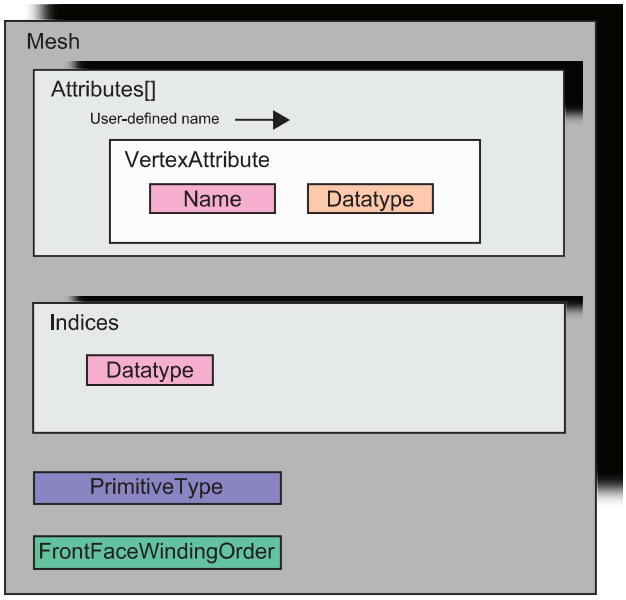


Figure 3.12. A Mesh contains a collection of vertex attributes, an optional collection of indices, and members describing the primitive type and winding order. These types are easier to work with than the lower-level renderer types. A Context.`CreateVertexArray` overload can create a vertex array containing vertex and index buffers, given a mesh, allowing the user to avoid the bookkeeping of dealing with renderer types directly.

allows client code to use the higher-level mesh class to describe geometry and let Context.`CreateVertexArray` do the work of laying out vertex buffers and other bookkeeping. This `CreateVertexArray` overload only uses publicly exposed renderer types, so it does not need to be rewritten for each supported rendering API.

Another benefit of Mesh is that algorithms that compute geometry can create a mesh object. This decouples geometric computation from rendering, which makes sense; for example, an algorithm to compute triangles for an ellipsoid should not depend on our renderer types. In Chapter 4, we cover several algorithms that create a mesh object that approximates the ellipsoidal surface of the globe. In fact, we use the Mesh type throughout the book when building geometry.

A mesh contains a collection of vertex attributes, similar to how a vertex array can have separate vertex buffers for each attribute. The VertexAttributeType enumeration in Listing 3.22 lists the supported data types. There is a concrete class for each type that inherits from VertexAttribute<`T`> and VertexAttribute, as shown in Listing 3.22 and

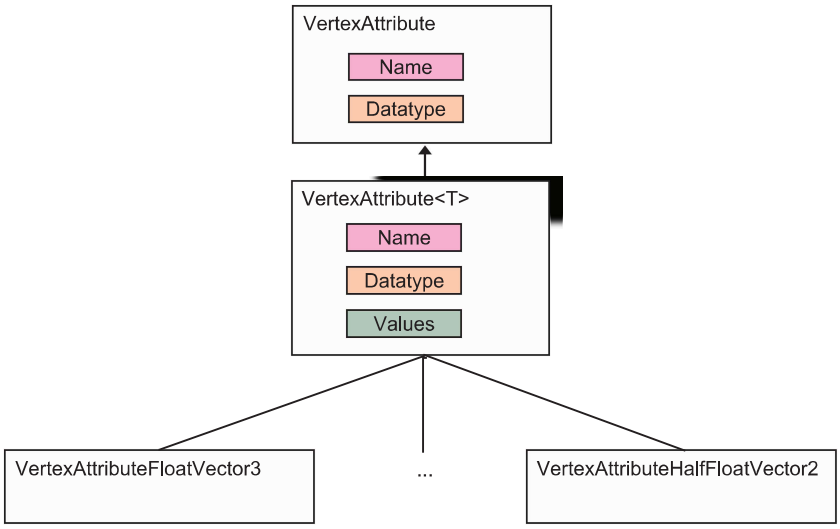


Figure 3.13. A Mesh contains a collection of VertexAttributes indexed by a user-defined name. Actual implementations of VertexAttribute contain a collection of fully typed attributes, as opposed to VertexBuffer, which is untyped and can therefore store multiple attribute types in the same collection.

```
public enum VertexAttributeType
{
  UnsignedByte,
  HalfFloat,
  HalfFloatVector2,
  HalfFloatVector3,
  HalfFloatVector4,
  Float,
  FloatVector2,
  FloatVector3,
  FloatVector4,
  EmulatedDoubleVector3
}

public abstract class VertexAttribute
{
  protected VertexAttribute(string name,
                            VertexAttributeType type);
  public string Name { get; }
  public VertexAttributeType Datatype { get; }
}

public class VertexAttribute<T> : VertexAttribute
{
  public IList<T> Values { get; }
}
```

Listing 3.22. VertexAttribute and VertexAttribute<T> interfaces.

```
public enum IndicesType
{
  UnsignedShort,
  UnsignedInt
}

public abstract class IndicesBase
{
  protected IndicesBase(IndicesType type);
  public IndicesType Datatype { get; }
}

public class IndicesUnsignedShort : IndicesBase
{
  public IndicesUnsignedShort();
  public IndicesUnsignedShort(int capacity);
  public IList<short> Values { get; } // Add one index
  public void AddTriangle(TriangleIndicesUnsignedShort triangle);
      // Add three indices
}
```

Listing 3.23. IndicesBase interface and an example concrete class, IndicesUnsignedShort. IndicesByte and IndicesUnsignedInt are similar.

Figure 3.13. Each attribute has a user-defined name (e.g., "position," "normal") and its data type. A concrete attribute class has a strongly typed `Values` collection that contains the actual attributes. This allows client code to create a strongly typed collection (e.g., VertexAttributeFloatVector3), populate it, and then add it to a mesh. Then, Context.`CreateVertexArray` can inspect the attribute's type using its `Datatype` member and cast to the appropriate concrete class. This is very similar to the class hierarchy design used for uniforms in Section 3.4.4.

A mesh's indices are handled similar to vertex attributes, except a mesh only has a single indices collection, not a collection of them. As shown in Listing 3.23 and Figure 3.14, two different index data types are supported: unsigned shorts and unsigned ints. Again, client code can create the specific type of indices desired (e.g., IndicesUnsignedShort), and Context `.CreateVertexArray` will use the base class's `Datatype` property to cast to the correct type and create the appropriate index buffer.

Listing 3.24 shows example client code using a Mesh and `context.CreateVertexArray` to create a vertex array containing a single triangle. First, a Mesh object is created, and its primitive type and winding order are set to triangles and counterclockwise, respectively. Next, memory is allocated for three single-precision 3D vector vertex attributes. These attributes are added to the mesh's attributes collection. Similarly, memory is allocated for three indices of type `unsigned` short, which are added to the mesh. Then, the actual vertex attributes and indices are assigned. Note the use of a helper class, TriangleIndicesUnsignedShort, to add indices for one triangle in a single line of code, as opposed to calling `indices.Values.Add` three times.

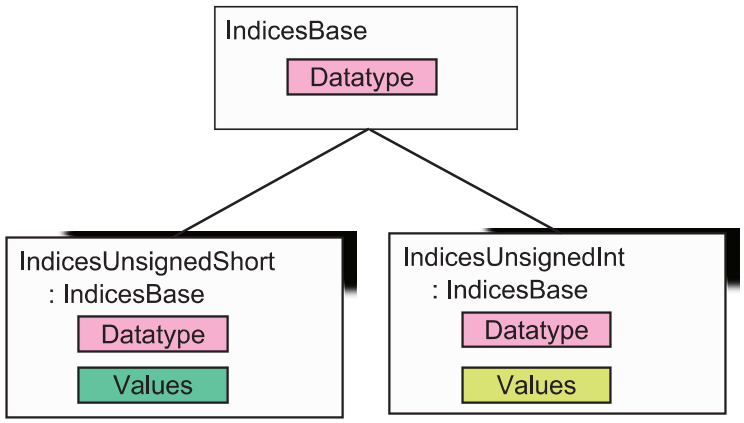


Figure 3.14. A Mesh contains an IndicesBase member. Each concrete class derived from IndicesBase contains a collection of indices of unsigned shorts or unsigned ints.

Finally, a vertex array is created from the mesh. A shader's list of attributes is passed to `CreateVertexArray` along with the mesh to match up the mesh's attribute names with the names of the shader's attributes.

It is helpful to think of Mesh and its related types as geometry in system memory for general application use and to think of VertexArray and related

```
Mesh mesh = new Mesh();
mesh.PrimitiveType = PrimitiveType.Triangles;
mesh.FrontFaceWindingOrder = WindingOrder.Counterclockwise;

VertexAttributeFloatVector3 positionsAttribute =
    new VertexAttributeFloatVector3("position", 3);
mesh.Attributes.Add(positionsAttribute);

IndicesUnsignedShort indices = new IndicesUnsignedShort(3);
mesh.Indices = indices;

IList<Vector3F> positions = positionsAttribute.Values;
positions.Add(new Vector3F(0, 0, 0));
positions.Add(new Vector3F(1, 0, 0));
positions.Add(new Vector3F(0, 0, 1));

indices.AddTriangle(new TriangleIndicesUnsignedShort(0, 1, 2));

// ...

ShaderProgram sp = Device.CreateShaderProgram(vs, fs);
VertexArray va = context.CreateVertexArray(
    mesh, sp.VertexAttributes, BufferHint.StaticDraw);
```

Listing 3.24. Creating a VertexArray for a Mesh containing a single triangle.

types as geometry in driver-controlled memory for rendering use. Strictly speaking, though, a vertex array does not contain geometry; it references and interprets vertex and index buffers.

Although we often favor the ease of use of Mesh and Context.`CreateVertexArray`, there are times when the flexibility of vertex and index buffers makes it worthwhile to create them directly. In particular, if several vertex arrays should share a vertex or index buffer, the renderer types should be used directly.

Try This

Our implementation of Context.`CreateVertexArray` creates a separate vertex buffer for each vertex attribute. Implement two variations: one that stores attributes in a single noninterleaved vertex buffer and another that stores attributes in a single interleaved buffer. What are the performance differences?

3.6 Textures

Textures represent image data in driver-controlled memory. In virtual globes, games, and many graphics applications, textures can account for more memory usage than vertex data. Textures are used to render the high-resolution imagery that has become a commodity in virtual globes. The texels in textures do not have to represent pixels though; as we shall see in Chapter 11, textures are also useful for terrain heights and other data.

Our renderer exposes a handful of classes for using 2D textures, including the texture itself, pixel buffers for transferring data to and from textures, a sampler describing filtering and wrap modes, and texture units for assigning textures and samplers for rendering.

3.6.1 Creating Textures

In order to create a texture, client code must first create a description of the texture, Texture2DDescription, shown in Listing 3.25 and Figure 3.15. A description defines the width and height of the texture, its internal format, and if mipmapping should be used. It also has three derived properties based on the format: `ColorRenderable`, `DepthRenderable`, and `DepthStencilRenderable`. These describe where a texture with the given description can be attached when using framebuffers (see Section 3.7). For

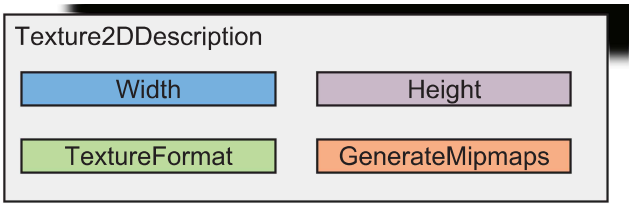


Figure 3.15. A Texture2DDescription describes the immutable state of a texture. A Texture2DDescription is passed to Device.`CreateTexture2D` to describe the resolution, internal format, and mipmaps of the texture to create.

```
public enum TextureFormat
{
  RedGreenBlue8,
  RedGreenBlueAlpha8,
  Red8,
  Red32f,
  Depth24,
  Depth32f,
  Depth24Stencil8,
  // ... See TextureFormat.cs for the full list.
}

public struct Texture2DDescription :
    IEquatable<Texture2DDescription>
{
  public Texture2DDescription(int width, int height,
                              TextureFormat format);
  // ... Constructor overload with generateMipmaps.

  public int Width { get; }
  public int Height { get; }
  public TextureFormat TextureFormat { get; }
  public bool GenerateMipmaps { get; }

  public bool ColorRenderable { get; }
  public bool DepthRenderable { get; }
  public bool DepthStencilRenderable { get; }
}
```

Listing 3.25. Texture2DDescription interface.

example, if `ColorRenderable` is false, the texture cannot be attached to a framebuffer's color attachments.

Once a Texture2DDescription is created, it is passed to Device.`CreateTexture2D`, to create an actual texture of type Texture2D:

```
Texture2DDescription description = new Texture2DDescription(
    256, 256, TextureFormat.RedGreenBlueAlpha8);
Texture2D texture = Device.CreateTexture2D(description);
```

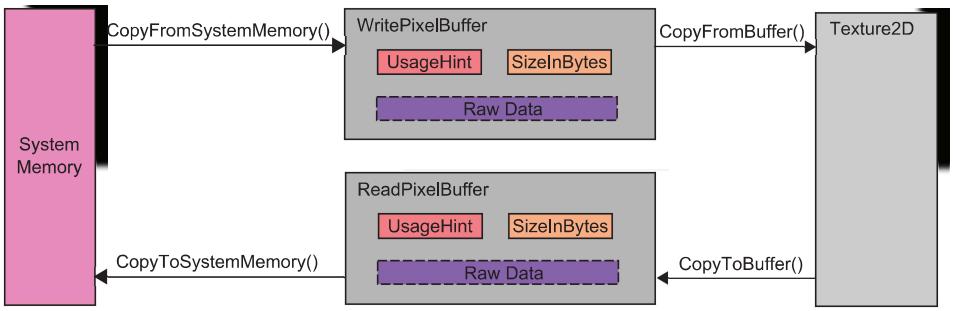


Figure 3.16. Pixel buffers are used to transfer data between system memory and a texture. The method WritePixelBuffer is used to transfer from system memory to a texture, while ReadPixelBuffer goes in the opposite direction. Pixel buffers are untyped, similar to vertex buffers.

The obvious next step is to provide the texture with data. This is done using pixel buffers, as shown in Figure 3.16. There are two types of pixel buffers: WritePixelBuffer, which is used to transfer data from system memory to a texture, and ReadPixelBuffer, for transferring from a texture to system memory. In other words, write pixel buffers write to textures, and read pixel buffers read from textures. Pixel buffers look and feel very much like vertex buffers. They are untyped, meaning they only contain raw bytes, but they provide `CopyFromSystemMemory` and `CopyToSystemMemory` overloads with a generic parameter `T` so client code doesn't have to cast.

Pixel buffers are so similar to vertex buffers that both write and read pixel buffers have almost an identical interface to vertex buffers, shown previously in Listing 3.18. The main difference is that pixel buffers also support methods to copy to and from .NET's Bitmap instead of just arrays. These methods are appropriately named `CopyFromBitmap` and `CopyToBitmap`.

One could argue that vertex buffers and pixel buffers should use the same abstract class, or at least write and read pixel buffers should use the same class. We avoid doing so because we prefer the strong typing provided by separate classes. A method that takes a write pixel buffer, like Texture2D.`CopyFromBuffer`, can only take a write pixel buffer. This is checked at compile time. If both pixel-buffer types used the same class, the check would have to be done at runtime, making the method harder to use and less efficient. Although we're not too concerned about the cost of an extra `if` statement, we are concerned about designing methods that are easy to use correctly and hard to use incorrectly.

Question ○○◯◯

If vertex and pixel buffers need to be treated polymorphically, it makes sense to introduce a Buffer abstract base class, with derived abstract classes VertexBuffer, WritePixelBuffer, and ReadPixelBuffer. Methods that want a specific type use the derived class, and methods that work with any buffer use the base class. One use case is to render to a pixel buffer, then interpret this buffer as vertex data. Can you think of another?

Given that pixel buffers work so similarly to vertex buffers, it is no wonder that client code to copy data to pixel buffers looks similar. The example in Listing 3.26 copies two *red, green, blue, alpha* (RGBA) pixels to a pixel buffer.

Client code can also use `CopyFromBitmap` to copy data from a Bitmap to a pixel buffer:

```
Bitmap bitmap = new Bitmap(filename);
WritePixelBuffer pixelBuffer = Device.CreateWritePixelBuffer(
    PixelBufferHint.Stream,
    BitmapAlgorithms.SizeOfPixelsInBytes(bitmap));
pixelBuffer.CopyFromBitmap(bitmap);
```

Given the interface for Texture2D shown in Listing 3.27, it is easy to see that a `CopyFromBuffer` method should be used to copy data from a pixel buffer to a texture. Two arguments to this method are used to interpret the raw bytes stored in the pixel buffer, similar to how a VertexBufferAttribute is used to interpret the raw bytes in a vertex buffer. This requires specifying the format (e.g., RGBA) and data type (e.g., un-

```
BlittableRGBA[] pixels = new BlittableRGBA[]
{
  new BlittableRGBA(Color.Red),
  new BlittableRGBA(Color.Green)
};

int sizeInBytes = ArraySizeInBytes.Size(pixels);
WritePixelBuffer pixelBuffer = Device.CreateWritePixelBuffer(
    PixelBufferHint.Stream, sizeInBytes);
pixelBuffer.CopyFromSystemMemory(pixels);
```

Listing 3.26. Copying data to a WritePixelBuffer.

signed byte) of the data in the pixel buffer. The necessary conversions will take place to convert from this format and data type to the internal texture format that was specified in the description when the texture was created. To copy the entire pixel buffer created above using an array of BlittableRGBAs, the following could be used:

```
texture.CopyFromBuffer(writePixelBuffer,
    ImageFormat.RedGreenBlueAlpha, ImageDatatype.UnsignedByte);
```

Other overloads to `CopyFromBuffer` allow client code to modify only a subsection of the texture and to specify the row alignment.

```
public enum ImageFormat
{
  DepthComponent,
  Red,
  RedGreenBlue,
  RedGreenBlueAlpha,
  // ... See ImageFormat.cs for the full list.
}

public enum ImageDatatype
{
  UnsignedByte,
  UnsignedInt,
  Float,
  // ... See ImageDatatype.cs for the full list.
}

public abstract class Texture2D : Disposable
{
  public virtual void CopyFromBuffer(
      WritePixelBuffer pixelBuffer,
      ImageFormat format,
      ImageDatatype dataType);
  public abstract void CopyFromBuffer(
      WritePixelBuffer pixelBuffer,
      int xOffset,
      int yOffset,
      int width,
      int height,
      ImageFormat format,
      ImageDatatype dataType,
      int rowAlignment);
  // ... Other CopyFromBuffer overloads

  public virtual ReadPixelBuffer CopyToBuffer(
      ImageFormat format, ImageDatatype dataType)
  // ... Other CopyToBuffer overloads

  public abstract Texture2DDescription Description { get; }
  public virtual void Save(string filename);
}
```

Listing 3.27. Texture2D interface.

Texture2D also has a `Description` property that returns the description used to create the texture. This object is immutable; the resolution, format, and mipmapping behavior of a texture cannot be changed once it is created. Texture2D also contains a method, `Save`, to save the texture to disk, which is useful for debugging.

To simplify texture creation, an overload to Device.`CreateTexture2D` takes a Bitmap. This can be used to create a texture from a file on disk in a single line of code, such as the following:

```
Texture2D texture = Device.CreateTexture2D(new Bitmap(filename),
    TextureFormat.RedGreenBlue8, generateMipmaps);
```

Similar to the `CreateVertexArray` overload that takes a Mesh, this favors ease of use over flexibility, and we use it in many examples throughout this book.

Texture rectangles. In addition to regular 2D textures that are accessed in a shader using normalized texture coordinates (e.g., $(0, width)$ and $(0, height)$ are each mapped to the normalized range $(0, 1)$), our renderer also supports 2D texture rectangles that are accessed using unnormalized texture coordinates in the range $(0, width)$ and $(0, height)$. Addressing a texture in this manner makes some algorithms cleaner, like ray-casting height fields in Section 11.2.3. A texture rectangle still uses Texture2D but is created with Device.`CreateTexture2DRectangle`. Texture rectangles do not support mipmapping and samplers with any kind of repeat wrapping.

3.6.2 Samplers

When a texture is used for rendering, client code must also specify the parameters to use for sampling. This includes the type of filtering that should occur for minification and magnification, how to handle wrapping texture coordinates outside of the $[0, 1]$ range, and the degree of anisotropic filtering. Filtering can affect quality and performance. In particular, anisotropic filtering is useful for improving the visual quality for horizon views of textured terrain in virtual globes. Texture wrapping has several uses, including tiling detail textures for terrain shading, as discussed in Section 11.4.

Like Direct3D and recent versions of OpenGL, our renderer decouples textures and samplers. Textures are represented using Texture2D, and samplers are represented using TextureSampler, shown in Listing 3.28 and Figure 3.17. Client code can explicitly create a sampler using Device.`CreateTexture2DSampler`:

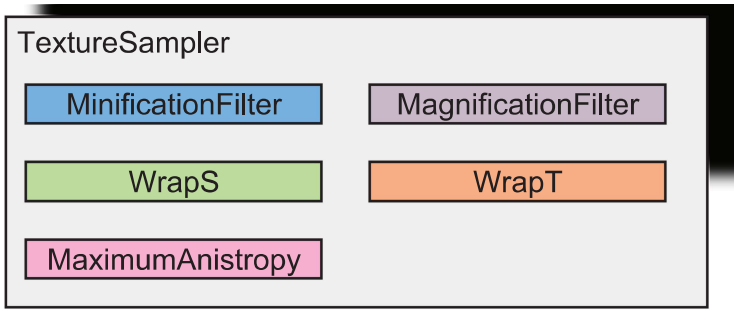


Figure 3.17. A TextureSampler describes sampling state, including filtering and wrap modes.

```
TextureSampler sampler = Device.CreateTexture2DSampler(
    TextureMinificationFilter.Linear,
    TextureMagnificationFilter.Linear,
    TextureWrap.Repeat,
    TextureWrap.Repeat);
```

The Device also contains a collection of common samplers, so the above call to `CreateTexture2DSampler` could be replaced with just the following:

```
TextureSampler sampler = Device.Samplers.LinearRepeat;
```

The latter approach has the benefit of not creating an additional renderer object, and thus, another GL object.

```
public enum TextureMinificationFilter
{
  Nearest,
  Linear,
  // .. Mipmapping filters
}

public enum TextureMagnificationFilter
{
  Nearest,
  Linear
}

public enum TextureWrap
{
  Clamp,
  Repeat,
  MirroredRepeat
}
```

```
public abstract class TextureSampler : Disposable
{
  public TextureMinificationFilter MinificationFilter { get; }
  public TextureMagnificationFilter MagnificationFilter { get; }
  public TextureWrap WrapS { get; }
  public TextureWrap WrapT { get; }
  public float MaximumAnistropy { get; }
}
```

Listing 3.28. TextureSampler interface.

Try This ○○○◯

The property Device.Samplers contains four premade samplers: NearestClamp, LinearClamp, NearestRepeat, and LinearRepeat. Would a sampler cache similar to the shader cache in Section 3.4.6 be more useful? If so, design and implement it. If not, why?

3.6.3 Rendering with Textures

Given a Texture2D and a TextureSampler, it is a simple matter to tell the Context we want to use them for rendering. In fact, we can tell the context we want to use multiple textures, perhaps each with a different sampler, for the same draw call. This ability for shaders to read from multiple textures is called multitexturing. It is widely used in general, and throughout this book; for example, Section 4.2.5 uses multitexturing to shade the side of the globe in sunlight with a day texture and the other side with a night texture, and Section 11.4 uses multitexturing in a variety of terrain-shading techniques. In addition to reading from multiple textures, a shader can also read multiple times from the same texture.

The number of texture units, and thus, the maximum number of unique texture/sampler combinations that can be used at once, is defined by Device.NumberOfTextureUnits. Context contains a collection of TextureUnits, as shown in Figure 3.18. Each texture unit is accessed with an index between 0 and Device.NumberOfTextureUnits − 1. Before calling Context.Draw, client code assigns a texture and sampler to each texture unit it needs. For example, if day and night textures are used, the client code may look like the following:

```
context.TextureUnits[0].Texture = dayTexture;
context.TextureUnits[0].TextureSampler =
    Device.TextureSamplers.LinearClamp;
context.TextureUnits[1].Texture = nightTexture;
```

```
context.TextureUnits[1].TextureSampler =
    Device.TextureSamplers.LinearClamp;
context.Draw(/* ... */);
```

The GLSL shader can define two sampler2D uniforms to access the texture units using the renderer's automatic uniforms:

```
uniform sampler2D og_texture0;  // day - texture unit 0
uniform sampler2D og_texture1;  // night - texture unit 1
```

Alternatively, custom-named uniforms can be used and explicitly set to the appropriate texture unit in client code using Uniform<int>.

Why is the collection of texture units a part of the context and not part of the draw state? What use cases does this make easier? What are the downsides?

○○○○ Question

It is often useful to compute mipmaps offline when more time can be spent on high-quality filtering, instead of at runtime with an API call like `glGenerateMipmap`. Modify Texture2D and related classes to support precomputed mipmaps.

○○○○ Try This

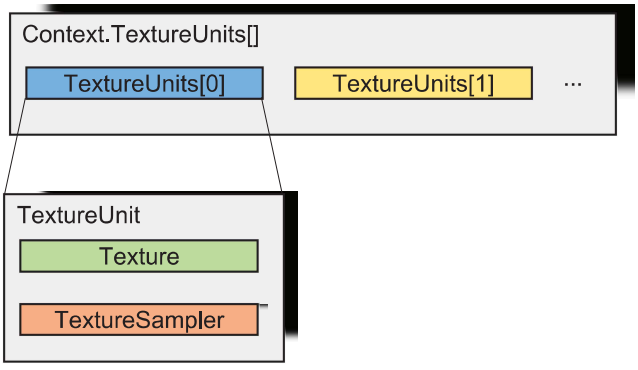


Figure 3.18. Each Context has Device.`NumberOfTextureUnits` texture units. A texture and a sampler are assigned to a texture unit for rendering.

Try This ○○◯◯

Our renderer only supports 2D textures because, believe it or not, 2D textures were the only texture type needed to write the example code for this entire book! There are many other useful texture types: 1D textures, 3D textures, cube maps, compressed textures, and texture arrays. Adding renderer support for these types is a straightforward extension of the existing design. Have at it.

3.6.4 GL Renderer Implementation

The GL implementation for textures is spread across several classes. WritePixelBufferGL3x and ReadPixelBufferGL3x contain the implementations for write and read pixel buffers, respectively. These classes use the same GL buffer object patterns used by vertex buffers, described in Section 3.5.1. Namely, `glBufferData` is used to allocate memory in the constructor, and `CopyFromSystemMemory` is implemented with `glBufferSubData`. The pixel- and vertex-buffer implementations actually share code under the hood. One notable difference is that write pixel buffers use a target of `GL_PIXEL_UNPACK_BUFFER`, and read pixel buffers use a target of `GL_PIXEL_PACK_BUFFER`.

Texture2DGL3x is the the implementation for an actual 2D texture. Its constructor creates the GL texture and allocates memory for it. First, a GL name is created using `glGenTexture`. Next, a call to `glBindBuffer` is made, with a target of `GL_PIXEL_UNPACK_BUFFER` and a buffer of zero. This ensures that no buffer object is bound that could provide the texture with data. Then, `glActiveTexture` is used to activate the last texture unit; for example, if Device.`NumberOfTextureUnits` is 32, it will activate texture unit 31. This somewhat nonintuitive step is required so that we know what texture unit is affected and can account for it before the next draw call if client code wants to render with a different texture on the same texture unit. Once the GL state is configured, a call to `glTexImage2D` with a null data argument finally allocates memory for the texture.

The implementation for `CopyFromBuffer` has some similarities to the constructor. First, a call to `glBindBuffer` is made to bind the write pixel buffer (unpack pixel-buffer object in GL terms), which has the image data we wish to copy to the texture. Next, a call to `glPixelStore` sets the row alignment. Then, `glTexSubImage2D` is used to transfer the data. If the texture is to be mipmapped, a final call to `glGenerateMipmap` generates mipmaps for the texture.

TextureSamplerGL3x is the implementation for texture samplers. Given that this class is immutable, and given the clean GL API for sampler objects, TextureSamplerGL3x has one of the cleanest implementations in our

renderer. A GL name for the sampler is created in the constructor with `glGenSamplers` and deleted later with `glDeleteSamplers` when the object is disposed. The constructor calls `glSamplerParameter` several times to define the sampler's filtering and wrapping parameters.

Finally, texture units are implemented in TextureUnitsGL3x and TextureUnitGL3x. It is the texture unit's responsibility to bind textures and samplers for rendering. A delayed approach is used, similar to the approach used for uniforms and vertex arrays. When client code assigns a texture or sampler to a texture unit, the texture unit is not activated with `glActiveTexture`, nor are the texture or sampler GL objects bound immediately. Instead, the texture unit is marked as dirty. When a call to Context.`Draw` is made, it iterates over the dirty texture units and cleans them. That is, it calls `glActiveTexture` to activate the texture unit and `glBindTexture` and `glBindSampler` to bind the texture and sampler, respectively. The last texture unit, which may have been modified in Texture2D.`CopyFromBuffer`, is treated as a special case and explicitly binds its texture and sampler if they are not null.

3.6.5 Textures in Direct3D

Textures in D3D have some similarities to our renderer interfaces. In D3D, a description, `D3D11_TEXTURE2D_DESC`, is used to create a texture with ID3D11Device::`CreateTexture2D`. A difference is that the usage hint (e.g., static, dynamic, or stream) is specified in the D3D description; in our renderer, it is only specified as part of the write or read pixel buffer. Our texture's `CopyFromBuffer` and `CopyToBuffer` can be implemented with ID3D11DeviceContext::`CopyResource`. It is also possible to implement a write pixel buffer using just an array in system memory and then implement the texture's `CopyFromBuffer` using ID3D11DeviceContext::`UpdateSubresource` or ID3D11DeviceContext::`Map`. Our pixel buffers don't map directly to D3D, so it may be better to do some redesign rather than to complicate the D3D implementation.

Since our renderer's sampler is immutable, D3D samplers map quite nicely to ours. In D3D, sampler parameters are described using a `D3D11_SAMPLER_DESC`, which is passed to ID3D11Device::`CreateSamplerState` to create an immutable sampler. A nice feature of D3D is that if a sampler already exists with the same description that is passed to `CreateSamplerState`, the existing sampler is returned, making a renderer or application-level cache less important.

To access data in a texture for rendering, D3D requires a shader-resource view to be created using ID3D11Device::`CreateShaderResourceView`. The texture's resource view and sampler can be bound to different stages of the pipeline on slots (texture units) using ID3D

`11DeviceContext` methods like `PSSetShaderResources` and `PSSetSamplers`. Similar to D3D shaders themselves, resource views and samplers are bound to individual pipeline stages, whereas in GL and our renderer, they are bound to the entire pipeline.

In D3D, the origin for texture coordinates is the upper left corner, and rows go from the the top of the image to the bottom. In GL, the origin is the lower left corner, and rows go from the bottom to the top. This doesn't cause a problem when reading textures in a shader; the same texture data can be used with the same texture coordinates with both APIs. However, this origin difference does cause a problem when writing to a texture attached to a framebuffer. In this case, rendering needs to be "flipped" by reconfiguring the viewport transform, depth range, culling state, and projection matrix and utilizing ARB_fragment_coord_conventions [84].

3.7 Framebuffers

The last major component in our renderer is the framebuffer. Framebuffers are containers of textures used for render to texture techniques. Some of these techniques are quite simple. For example, the scene might be rendered to a high-resolution texture attached to a framebuffer that is then saved to disk, to implement taking screen captures at a resolution much higher than the monitor's resolution.[22] Other techniques using framebuffers are more involved. For example, deferred shading writes to multiple textures in the first rendering pass, outputting information such as depth, normals, and material properties. In the second pass, a full screen quad is rendered that reads from these textures to shade the scene. Many techniques using framebuffers will write to textures in the first pass, then read from them in a later pass.

```
public abstract class ColorAttachments
{
  public abstract Texture2D this[int index] { get; set; }
  public abstract int Count { get; }
  // ...
}

public abstract class Framebuffer : Disposable
{
  public abstract ColorAttachments ColorAttachments { get; }
  public abstract Texture2D DepthAttachment { get; set; }
  public abstract Texture2D DepthStencilAttachment { get; set; }
}
```

Listing 3.29. ColorAttachments and Framebuffer interfaces.

[22] This is exactly how we created many figures in this book.

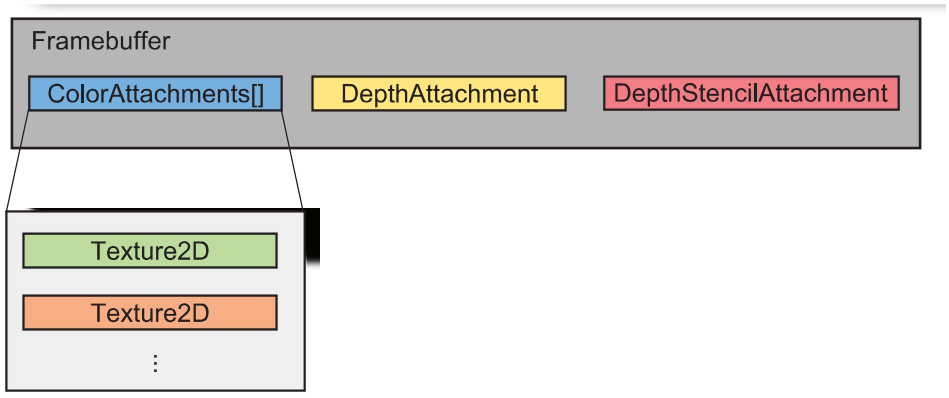


Figure 3.19. A Framebuffer can have up to Device.`MaximumNumberOfColorAttachments` color attachments and a depth or depth/stencil attachment. All attachments are Texture2D objects.

In our renderer, framebuffers are represented by Framebuffer, shown in Listing 3.29 and Figure 3.19. Clients can create framebuffers with Context.`CreateFramebuffer`. Like vertex arrays, framebuffers are lightweight container objects and are not sharable among contexts, which is why `CreateFramebuffer` is a method on Context, not Device. Once a framebuffer is created, we can attach multiple textures as color attachments and another texture as a depth or depth/stencil attachment.

It is important that the texture's format is compatible with the framebuffer attachment type, which is why Texture2DDescription contains `ColorRenderable`, `DepthRenderable`, and `DepthStencilRenderable` properties mentioned in Section 3.6.1. The example on the next page creates a framebuffer with a color and depth attachment.

We render to a framebuffer by assigning it to Context.`Framebuffer`. If the depth-testing renderer state is enabled, the framebuffer must have a depth or depth/stencil attachment. Forgetting a depth attachment is a common OpenGL beginner mistake. In this case, our renderer throws an exception in Context.`Draw`. If only OpenGL was so informative!

```
Framebuffer framebuffer = context.CreateFramebuffer();
framebuffer.ColorAttachments[0] = Device.CreateTexture2D(
    new Texture2DDescription(640, 480,
        TextureFormat.RedGreenBlue8, false));
framebuffer.DepthAttachment = Device.CreateTexture2D(
    new Texture2DDescription(640, 480,
        TextureFormat.Depth32f, false));
```

When multiple color attachments are used, it is important to match up the fragment shader's output variable with the framebuffer's color attachment's index, similar to how vertex-shader attribute inputs are matched up with vertex-array attributes. For example, a fragment shader may have two output variables:

```
out vec4 dayColor;
out vec4 nightColor;
```

As described in Section 3.4.3, the color attachment indices for the fragment-shader output variables can be queried with ShaderProgram.`FragmentOutputs`. This index can then be used to assign a texture to the appropriate color attachment, associating the shader's output variable with the texture:

```
ShaderProgram sp = // ...
framebuffer.ColorAttachments[sp.FragmentOutputs("dayColor")] =
    dayTexture;
framebuffer.ColorAttachments[sp.FragmentOutputs("nightColor")] =
    nightTexture;
```

After a texture attached to a framebuffer is written to, it can be assigned to a texture unit and read in a later rendering pass, but it is not possible to both write to and read from the same texture during a single draw call.

We could move the `Framebuffer` property from Context to DrawState to reduce the amount of context-level global state. It is convenient to keep the framebuffer as part of the context because it allows objects to render without having to know which framebuffer they are rendering to. Although this design relies on context-level state, a framebuffer is much easier to manage than the entire renderer state.

3.7.1 GL Renderer Implementation

The GL implementation for framebuffers is in FramebufferGL3x and ColorAttachmentsGL3x. A name for the GL framebuffer object is created and deleted with `glGenFramebuffers` and `glDeleteFramebuffers`. As we've seen throughout this chapter, a delayed technique is used so no other GL calls are made until client code calls Context.`Draw` or Context.`Clear`. Many objects like shaders and textures do not affect clears, so it is important to remember that clears depend on the framebuffer.

As usual, when client code assigns textures to framebuffer attachments, the attachment is marked as dirty. Before drawing or clearing, the frame-

buffer is bound with `glBindFramebuffer`, and the attachments are cleaned by calling `glFramebufferTexture`. If no texture is attached to a dirty attachment (e.g., because it was assigned null), then `glFramebuffer Texture` is called with a texture argument of zero. Likewise, if Context. `Framebuffer` was assigned null, `glBindFramebuffer` is called with a framebuffer argument of zero, which means the default framebuffer is used.

3.7.2 Framebuffers in Direct3D

In D3D, render-target and depth/stencil views can be used to implement our renderer's Framebuffer. A render-target view can be created for a color attachment using ID3D11Device::`CreateRenderTargetView`. Likewise, a depth/stencil view can be created for the depth/stencil attachment using ID3D11Device::`CreateDepthStencilView`. For rendering, these views can be bound to the output-merger pipeline stage using ID3D11Device Context::`OMSetRenderTargets`.

Given that D3D treats color output (render targets) and depth/stencil output separately, implementing our Context.`Clear` requires calling both ID3D11DeviceContext::`ClearRenderTargetView` and ID3D11DeviceContext :: `ClearDepthStencilView` if both color and depth/stencil buffers are to be cleared.

3.8 Putting It All Together: Rendering a Triangle

Chapter03Triangle is an example that uses the renderer to render a single red triangle. It pulls together many of the concepts and code snippets in this chapter into an example you can run, step through in the debugger, and most importantly, build your own code upon. The Chapter03Triangle project references `OpenGlobe.Core` and `OpenGlobe.Renderer`. It has no need to call OpenGL directly; all rendering is done through publicly exposed renderer types.

Chapter03Triangle contains just a single class. Its private members are shown in Listing 3.30. All that is needed is a window to render to and the state types used for issuing a draw call. Of course, the draw state includes

```
private readonly GraphicsWindow _window;
private readonly SceneState _sceneState;
private readonly ClearState _clearState;
private readonly DrawState _drawState;
```

Listing 3.30. Chapter03Triangle private members.

a shader program, vertex array, and render state, but these are not directly held as members in the example.

The bulk of the work is done in the constructor shown in Listing 3.31. It creates a window and hooks up events for window resizing and rendering. A window needs to be created before other renderer types because a window creates the GL context under the hood.

Question ○○◯◯

How would you change the GL renderer implementation to remove this restriction that a window needs to be created before other renderer types? Is it worth the effort?

A default scene state and clear state are created. Recall that the scene state contains state used to set automatic uniforms, and the clear state contains state for clearing the framebuffer. For this example, the defaults are acceptable. In the scene state, the camera's default position is $(0, -1, 0)$, and it is looking at the origin with an up vector of $(0, 0, 1)$, so it is looking at the xz-plane with x to the right and z being up.

A shader program consisting of simple vertex and fragment shaders is created next. The shader sources are hard coded in strings. If they were more complicated, we would store them in separate files as embedded resources, as described in Section 3.4. The vertex shader transforms the input position using the automatic uniform `og_modelViewPerspectiveMatrix`, so the example doesn't have to explicitly set the transformation matrix. The fragment shader outputs a solid color based on the `u_color` uniform, which is set to red.

Next, a vertex array for an isosceles right triangle in the xz-plane, with legs of unit length is created by first creating a Mesh, then calling Context.`CreateVertexArray`, similar to Listing 3.24.

Then, a render state is created with disabled facet culling and depth testing. Even though this is the default state in OpenGL, these members default to true in our renderer since that is the common use case. This example disables them for demonstration purposes; they could also be left alone without affecting the output.

Finally, a draw state is created that contains the shader program, vertex array, and render state previously created. A camera helper method, `ZoomToTarget`, adjusts the camera's position so the triangle is fully visible.

Once all the renderer objects are created, rendering is simple, as shown in Listing 3.32. The framebuffer is cleared, and a draw call draws the triangle, instructing the renderer to interpret the vertices as triangles.

```
public Triangle()
{
  _window = Device.CreateWindow(800, 600,
                                "Chapter 3:  Triangle");
  _window.Resize += OnResize;
  _window.RenderFrame += OnRenderFrame;
  _sceneState = new SceneState();
  _clearState = new ClearState();

  string vs =
      @"in vec4 position;
        uniform mat4 og_modelViewPerspectiveMatrix;
        void main() { gl_Position = og_modelViewPerspectiveMatrix↩
            *
                                    position; }";
  string fs =
      @"out vec3 fragmentColor;
        uniform vec3 u_color;
        void main() { fragmentColor = u_color; }";
  ShaderProgram sp = Device.CreateShaderProgram(vs, fs);
  ((Uniform<Vector3F>)sp.Uniforms["u_color"]).Value =
      new Vector3F(1, 0, 0);

  Mesh mesh = new Mesh();
  VertexAttributeFloatVector3 positionsAttribute =
      new VertexAttributeFloatVector3("position", 3);
  mesh.Attributes.Add(positionsAttribute);
  IndicesUnsignedShort indices = new IndicesUnsignedShort(3);
  mesh.Indices = indices;
  IList<Vector3F> positions = positionsAttribute.Values;
  positions.Add(new Vector3F(0, 0, 0));
  positions.Add(new Vector3F(1, 0, 0));
  positions.Add(new Vector3F(0, 0, 1));
  indices.AddTriangle(new TriangleIndicesUnsignedShort(0, 1, 2));
  VertexArray va = _window.Context.CreateVertexArray(
      mesh, sp.VertexAttributes, BufferHint.StaticDraw);

  RenderState renderState = new RenderState();
  renderState.FacetCulling.Enabled = false;
  renderState.DepthTest.Enabled = false;

  _drawState = new DrawState(renderState, sp, va);

  _sceneState.Camera.ZoomToTarget(1);
}
```

Listing 3.31. Chapter03Triangle's constructor creates renderer objects required to render the triangle.

```
private void OnRenderFrame()
{
  Context context = _window.Context;
  context.Clear(_clearState);
  context.Draw(PrimitiveType.Triangles, _drawState, _sceneState);
}
```

Listing 3.32. Chapter03Triangle's `OnRenderFrame` issues the draw call that renders the triangle.

Try This ○○◯◯

Change the PrimitiveType.`Triangles` argument to PrimitiveType.`LineLoop`. What is the result?

When issuing the draw call, we could also explicitly state the index offset and count, like the following, which uses an offset of zero and a count of three:

```
context.Draw(PrimitiveType.Triangles, 0, 3,
    _drawState, _sceneState);
```

Of course, for this index buffer containing three indices, this has the same result as not providing an offset and count; it draws using the entire index buffer. Although Chapter03Triangle is a simple example, the rendering code for many examples is nearly this easy: issue Context.`Draw` and let the renderer bind the shader program, set automatic uniforms, bind the vertex array, set the render states, and finally draw! The only difference is that more advanced examples may also assign the framebuffer, textures, and samplers and set shader uniforms. Rendering is usually concise; the bulk of the work is in creating and updating renderer objects.

Try This ○○◯◯

Modify Chapter03Triangle to apply a texture to the triangle. Use the Device.`CreateTexture2D` overload that takes a Bitmap to create a texture from an image on disk. Add texture coordinates to the mesh using VertexAttributeHalfFloatVector2. Assign the texture and a sampler to a texture unit, and modify the vertex and fragment shader accordingly.

Try This ○○◯◯

Modify Chapter03Triangle to render to a framebuffer, then render the framebuffer's color attachment to the screen using a textured full screen quad. Create the framebuffer using Context.`CreateFramebuffer`, and assign a texture that is the same size as the window to one of the framebuffer's color attachments. Consider using ViewportQuad in `Open Globe.Scene` to render the quad that reads the texture and renders to the screen.

3.9 Resources

A quick Google search will reveal a plethora of information on OpenGL and Direct3D. The OpenGL SDK[23] and the DirectX Developer Center[24] are among the best references.

It is instructive to look at different renderer designs. Eric Bruneton's Ork[25] (OpenGL rendering kernel) is a C++ API built on top of OpenGL. It is fairly similar to our renderer but also includes a resource framework, scene graph, and task graph, making it closer to a full 3D engine.

Wojciech Sterna's *GPU Pro* article discusses the differences between D3D 9 and OpenGL 2 [157]. The source code for the Blossom Engine, available on the *GPU Pro* website,[26] includes a renderer that supports both D3D 9 and GL 2 using preprocessor defines, avoiding the virtual-call overhead of using abstract classes like our design. Although this couples the implementations, it is instructive to see the differences between D3D and GL implementations right next to each other.

David Eberly's Wild Magic Engine[27] contains an abstract renderer with both D3D 9 and GL implementations. The engine has been used and described in many of his books, including *3D Game Engine Design* [43].

OGRE[28] is a popular open source 3D engine that supports both OpenGL and Direct3D via an abstract RenderSystem class.

The web has good articles on sorting by state before rendering [47, 52].

Two articles in *Game Engine Gems 2* go into detail on the delayed GL techniques used throughout this chapter and the approach for automatic uniforms used in Section 3.4.5 [31, 32].

Later in this book, Section 10.4 describes multithreading with OpenGL and related abstractions in our renderer.

Try This

Perhaps the ultimate exercise for this chapter is to write a D3D implementation of the OpenGlobe renderer. SlimDX, a DirectX .NET wrapper, can be used to make Direct3D calls from C#.[29] A good first step is to make the GL implementation work more like D3D by checking out the following extensions, which are core features since OpenGL 3.2: ARB_vertex_array_bgra [86], ARB_fragment_coord_conventions, and ARB_provoking _vertex [85]. How will you handle shaders?

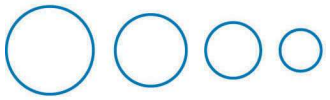
[23] http://www.opengl.org/sdk/
[24] http://msdn.microsoft.com/en-us/directx/
[25] http://ork.gforge.inria.fr/
[26] http://www.akpeters.com/gpupro/
[27] http://www.geometrictools.com/
[28] http://www.ogre3d.org/

Try This ○○◯◯

Write an OpenGL ES 2.0 implementation of the OpenGlobe renderer. How much code from the GL 3.x implementation can be reused? How do you handle lack of geometry shaders?

Patrick Says ○○◯◯

If you embark on either of the above adventures, we would love to hear about it.



4 Globe Rendering

This chapter covers a central topic in virtual globes: rendering a globe, with a focus on ellipsoid tessellation and shading algorithms.

We cover three tessellation algorithms used to produce triangles approximating a globe's surface. First, we discuss a simple subdivision-surfaces algorithm for a unit sphere that is typical of an introductory computer graphics course. Then, we cover ellipsoid tessellations based on a cube map and, finally, the geodetic grid.

Our shading discussion begins by reviewing simple per-fragment lighting and texture mapping with a low-resolution satellite-derived image of Earth. We compare per-fragment procedural generation of normals and texture coordinates to precomputed normals and texture coordinates. Specific virtual globe techniques for rendering a latitude-longitude grid and night lights using fragment shaders finish the shading discussion.

The chapter concludes with a technique for rendering a globe without tessellation by using GPU ray casting.

4.1 Tessellation

GPUs primarily render triangles, so to render an ellipsoid-based globe, triangles approximating the ellipsoid's surface are computed in a process called *tessellation* and then fed to the GPU for rendering. This section covers three popular ellipsoid tessellation algorithms and the trade-offs between them.

4.1.1 Subdivision Surfaces

Our first algorithm tessellates a unit sphere centered at the origin. This easily extends to ellipsoids by scaling each point by the ellipsoid's radii,

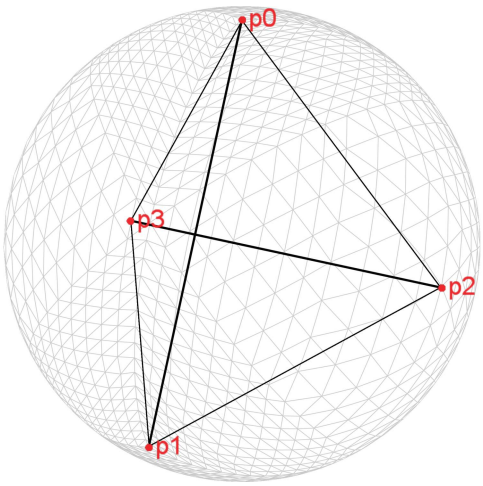


Figure 4.1. A tetrahedron with endpoints on the unit sphere. The faces of the tetrahedron are recursively subdivided and projected onto the unit sphere to approximate the sphere's surface.

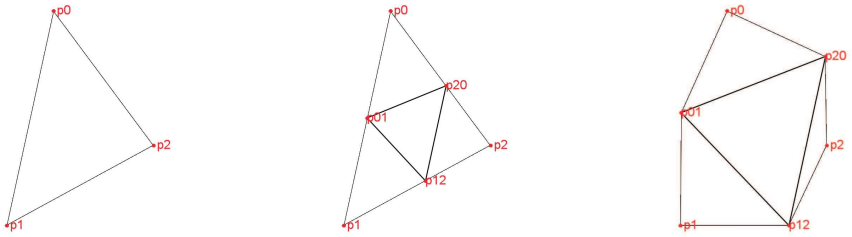
(a, b, c), and then computing geodetic surface normals (see Section 2.2.3) for shading.

The subdivision-surfaces algorithm is concise and easy to implement and doesn't even require any trigonometry. Start with a *regular tetrahedron*, that is, a pyramid composed of four equilateral triangles with endpoints that lie on the unit sphere, as shown in Figure 4.1.

The tetrahedron endpoints, $p0$, $p1$, $p2$, and $p3$, are defined as

$$\begin{aligned} p0 &= (0, 0, 1), \\ p1 &= \frac{(0, 2\sqrt{2}, -1)}{3}, \\ p2 &= \frac{(-\sqrt{6}, -\sqrt{2}, -1)}{3}, \\ p3 &= \frac{(\sqrt{6}, -\sqrt{2}, -1)}{3}. \end{aligned} \tag{4.1}$$

The tetrahedron is a very coarse polygonal approximation to a unit sphere. In order to get a better approximation, each triangle is first subdivided into four new equilateral triangles, as shown in Figures 4.2(a) and 4.2(b). For a given triangle, introducing four new triangles creates three new points, each of which is a midpoint of the original triangle's edge:



(a) (b) (c)

Figure 4.2. Subdividing a triangle and projecting new points onto the unit sphere. This process is repeated recursively until the tessellation sufficiently approximates the sphere's surface. (a) A triangle with its endpoints on the unit sphere. (b) A triangle is subdivided into four new triangles. (c) New points are normalized so they lie on the unit sphere.

$$\begin{aligned} p01 &= \frac{(p0 + p1)}{2}, \\ p12 &= \frac{(p1 + p2)}{2}, \\ p20 &= \frac{(p2 + p0)}{2}. \end{aligned} \tag{4.2}$$

Splitting a triangle into four new triangles alone does not improve the approximation; the midpoints are in the same plane as the original triangle and, therefore, not on the unit sphere. To project the midpoints on the unit sphere and improve our approximation, simply normalize each point:

$$\begin{aligned} p01 &= \frac{p01}{\|\mathbf{p01}\|}, \\ p12 &= \frac{p12}{\|\mathbf{p12}\|}, \\ p20 &= \frac{p20}{\|\mathbf{p20}\|}. \end{aligned} \tag{4.3}$$

The geometric result is shown in Figure 4.2(c).

This choice of subdivision produces nearly equilateral triangles.[1] Other subdivision choices, such as creating three new triangles using the original triangle's centroid, create long, thin triangles that can adversely affect shading due to interpolation over a long distance.

Subdivision continues recursively until a stopping condition is satisfied. A simple stopping condition is the number of subdivisions, n, to perform.

[1] After normalization, the triangles are not all equilateral.

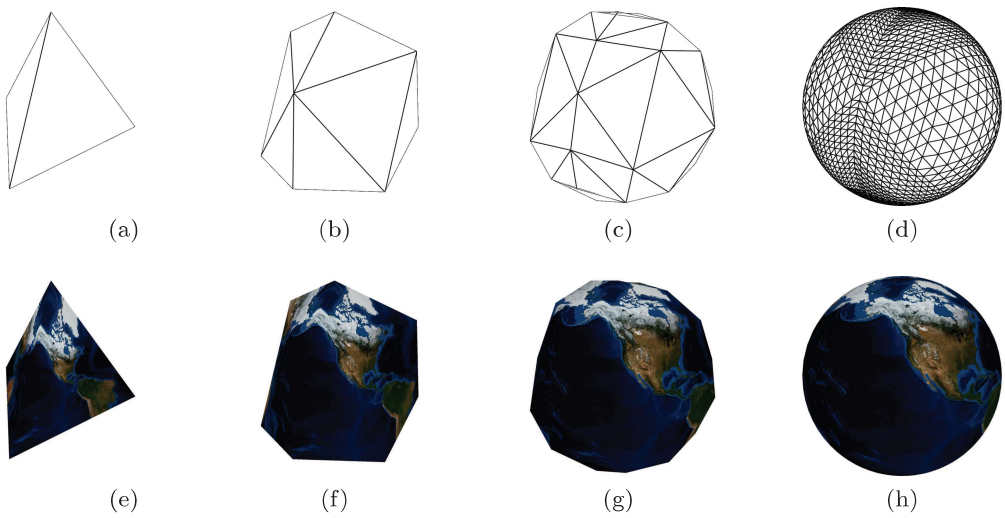


Figure 4.3. (a) and (e) $n = 0$ yields four triangles. (b) and (f) $n = 1$ yields 16 triangles. (c) and (g) $n = 2$ yields 64 triangles. (d) and (h) $n = 5$ yields 4,096 triangles.

For $n = 0$, four (4^1) triangles representing the original tetrahedron are produced; for $n = 1$, 16 (4^2) triangles are produced; for $n = 2$, 64 (4^3) triangles are produced. In general, n subdivisions yield 4^{n+1} triangles. Figure 4.3 shows the wireframe and textured globe for various values of n.

Larger values of n result in a closer approximation to a sphere at the cost of extra computation time and more memory. When n is too small, the visual quality is affected, because the underlying polygonal approximation of the sphere is seen. When n is too large, performance suffers because of extra computation and memory usage. Fortunately, n does not need to be the same for all viewer locations. When the viewer is near to the sphere, a higher value of n can be used. Likewise, when the viewer is far from the sphere, and thus the sphere covers fewer pixels, a lower value of n can be used. This general technique, called *level of detail*, is discussed in Section 12.1.

The tetrahedron is one of five *platonic solids*. Platonic solids are convex objects composed of equivalent faces, where each face is a congruent convex regular polygon. We use a tetrahedron because it is the simplest, having only four faces, but any platonic solid can be subdivided into a sphere using the same algorithm. Each has different trade-offs: platonic solids with more faces require less subdivision. Example code and discussion on subdividing an octahedron (eight faces) and an icosahedron (20 faces) is provided by Whitacre [179] and Laugerotte [95].

4.1.2 Subdivision-Surfaces Implementation

OpenGlobe includes example implementations of the globe-tessellation algorithms described in this chapter. Subdivision-surface tessellation for a unit sphere is implemented in SubdivisionSphereTessellatorSimple. The public interface, shown in Listing 4.1, is a static class with one method named `Compute` that takes the number of subdivisions and returns an indexed triangle list representing the sphere's approximation.

The first order of business for `Compute` is to initialize a Mesh object used to return the triangle list. This mesh, initialized in Listing 4.2, simply contains a collection of position attributes and a collection of indices. The tessellation is performed such that the winding order for front-facing triangles is counterclockwise.

For efficiency, the expected number of vertices and triangles are computed and used to set the capacity for the positions and indices collections, respectively. Doing so avoids unnecessary copying and memory allocations.

Listing 4.3 shows the real work of `Compute`, which is to create the initial tetrahedron using the four endpoints from Equation (4.1). Once these points are added to the position attribute collection, a call is made to `Subdivide` for each of the four triangles that make up the initial tetrahedron. The method `Subdivide` is a private method that recursively subdivides a triangle. The indices passed to each `Subdivide` call define a counterclockwise winding order (see Figure 4.1).

```
public static class SubdivisionSphereTessellatorSimple
{
  public static Mesh Compute(int numberOfSubdivisions)
  { /* ... */ }
}
```

Listing 4.1. SubdivisionSphereTessellatorSimple public interface.

```
Mesh mesh = new Mesh();
mesh.PrimitiveType = PrimitiveType.Triangles;
mesh.FrontFaceWindingOrder = WindingOrder.Counterclockwise;

VertexAttributeDoubleVector3 positionsAttribute =
    new VertexAttributeDoubleVector3(
        "position", SubdivisionUtility.NumberOfVertices(
            numberOfSubdivisions)
mesh.Attributes.Add(positionsAttribute);

IndicesInt indices = new IndicesInt(3 *
    SubdivisionUtility.NumberOfTriangles(numberOfSubdivisions));
mesh.Indices = indices;
```

Listing 4.2. `SubdivisionSphereTessellatorSimple.Compute` mesh initialization.

```
double negativeRootTwoOverThree = -Math.Sqrt(2.0) / 3.0;
const double negativeOneThird = -1.0 / 3.0;
double rootSixOverThree = Math.Sqrt(6.0) / 3.0;

IList<Vector3d> positions = positionsAttribute.Values;
positions.Add(new Vector3d(0, 0, 1));
positions.Add(new Vector3d(
    0, 2.0 * Math.Sqrt(2.0) / 3.0, negativeOneThird));
positions.Add(new Vector3d(
    -rootSixOverThree, negativeRootTwoOverThree, negativeOneThird↩
        ));
positions.Add(new Vector3d(
    rootSixOverThree, negativeRootTwoOverThree, negativeOneThird)↩
        );

Subdivide(positions, indices,
    new TriangleIndices<int>(0, 1, 2),numberOfSubdivisions);
Subdivide(positions, indices,
    new TriangleIndices<int>(0, 2, 3), numberOfSubdivisions);
Subdivide(positions, indices,
    new TriangleIndices<int>(0, 3, 1), numberOfSubdivisions);
Subdivide(positions, indices,
    new TriangleIndices<int>(1, 3, 2), numberOfSubdivisions);

return mesh;
```

Listing 4.3. `SubdivisionSphereTessellatorSimple.Compute` initial tetrahedron.

```
private static void Subdivide(IList<Vector3d> positions,
    IndicesInt indices, TriangleIndices<int> triangle, int level)
{
  if (level > 0)
  {
    positions.Add(Vector3d.Normalize((positions[triangle.I0]
        + positions[triangle.I1]) * 0.5));
    positions.Add(Vector3d.Normalize((positions[triangle.I1]
        + positions[triangle.I2]) * 0.5));
    positions.Add(Vector3d.Normalize((positions[triangle.I2]
        + positions[triangle.I0]) * 0.5));
    int i01 = positions.Count - 3;
    int i12 = positions.Count - 2;
    int i20 = positions.Count - 1;
    --level;
    Subdivide(positions, indices, new TriangleIndices<int>(
        triangle.I0, i01, i20), level);
    Subdivide(positions, indices, new TriangleIndices<int>(
        i01, triangle.I1, i12), level);
    Subdivide(positions, indices, new TriangleIndices<int>(
        i01, i12, i20), level);
    Subdivide(positions, indices, new TriangleIndices<int>(
        i20, i12, triangle.I2), level);
  }
  else
  {
    indices.Values.Add(triangle.I0);
    indices.Values.Add(triangle.I1);
    indices.Values.Add(triangle.I2);
  }
}
```

Listing 4.4. `SubdivisionSphereTessellatorSimple.Subdivide`.

The recursive call to `Subdivide` (see Listing 4.4) is the key step in this algorithm. If no more subdivision is required (i.e., `level` $= 0$), then the method simply adds three indices for the input triangle to the mesh and returns. Therefore, in the simplest case when the user calls `SubdivisionSphereTessellatorSimple.Compute`(0), `Subdivide` is called once per tetrahedron triangle, resulting in the mesh shown in Figure 4.3(a). If more subdivision is required (i.e., `level` > 0), three new points on the unit sphere are computed by normalizing the midpoint of each triangle edge using Equations (4.2) and (4.3). The current level is decreased, and `Subdivide` is called four more times, each with indices for a new triangle. The four new triangles better approximate the sphere than does the input triangle. Indices for the input triangle are never added to the mesh's indices collection until recursion terminates. Doing so would add unnecessary triangles underneath the mesh.

Run Chapter04SubdivisionSphere1 to see SubdivisionSphereTessellatorSimple in action. Experiment with the number of subdivisions performed, and verify that your output matches Figure 4.3.

○○○○ Try This

This implementation introduces duplicate vertices because triangles with shared edges create the same vertex at the edge's midpoint. Improve the implementation to avoid duplicate vertices.

○○○○ Try This

4.1.3 Cube-Map Tessellation

An interesting approach to tessellating an ellipsoid is to project a cube with tessellated planar faces onto the ellipsoid's surface. This is called *cube-map tessellation*. Triangles crossing the IDL can be avoided by using an axis-aligned cube centered around the origin in WGS84 coordinates and rotated $45°$ around z, such that the x and y components for the corners are $(-1, 0)$, $(0, -1)$, $(1, 0)$, and $(0, 1)$.

Cube-map tessellation starts with such a cube. Each face is then tessellated into a regular grid such that the face remains planar but is made up of additional triangles.[2] Cubes with two, four, and eight grid partitions

[2]Tessellations like this were also used to improve the quality of specular highlights and spotlights in fixed-function OpenGL when only per-vertex lighting was possible [88].

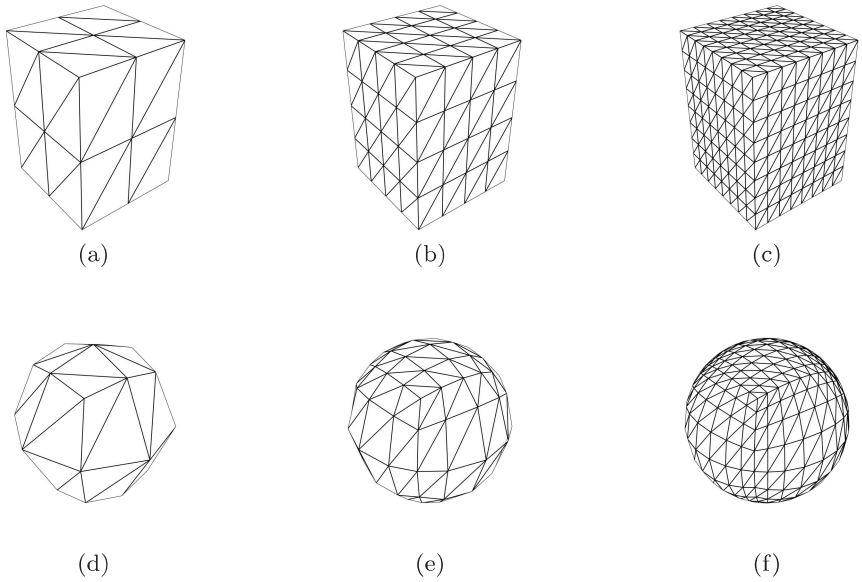


Figure 4.4. Cube-map tessellation tessellates the planar faces of a cube, then projects the points onto an ellipsoid. (a) Two partitions. (b) Four partitions. (c) Eight partitions. (d) Two partitions. (e) Four partitions. (f) Eight partitions.

are shown in Figures 4.4(a), (b), and (c), respectively. More partitions ultimately lead to a better ellipsoid approximation. Each point on the tessellated cube is then normalized to create the unit sphere. An ellipsoid is created by multiplying the normalized point by the ellipsoid's radii. Figures 4.4(d), (e), and (f) show ellipsoids created from two, four, and eight grid partitions.

Normalizing and scaling the original points act as a perspective projection. This results in straight lines on the cube becoming geodesic curves on the ellipsoid, not necessarily lines of constant latitude or longitude. As can be seen in Figure 4.4(f), some distortion occurs, particularly with triangles at a cube face's corner. Grid squares in the middle of the face are not badly distorted though. The distortion is not a showstopper; in fact, Spore used cube-map tessellation for procedurally generated spherical planets, favoring it over the pole distortion of other tessellation algorithms [28].

Like the subdivision-surfaces algorithm, this approach avoids oversampling at the poles but may create triangles that cross over the poles. Conceptually, cube-map tessellation is as simple as other subdivision techniques, and its implementation is only slightly more involved.

An implementation is provided in CubeMapEllipsoidTessellator. Given an ellipsoid shape, the number of grid partitions, and the desired vertex

attributes, CubeMapEllipsoidTessellator.`Compute` returns a mesh representing a polygonal approximation to the ellipsoid. First, it creates the eight corner points for the cube. Then it computes positions along each of the cube's 12 edges at a granularity determined by the number of partitions; for example, two partitions would result in one position added to each edge at the edge's midpoint, as in Figure 4.4(a). Next, each face is tessellated row by row, starting with the bottom row. A final pass is made over each point to project the point onto the ellipsoid and optionally compute a normal and texture coordinates.

Try This

Experiment with cube-map tessellation by replacing the call to SubdivisionSphereTessellatorSimple.`Compute` in Chapter04Subdivision Sphere1 with a call to CubeMapEllipsoidTessellator.`Compute`.

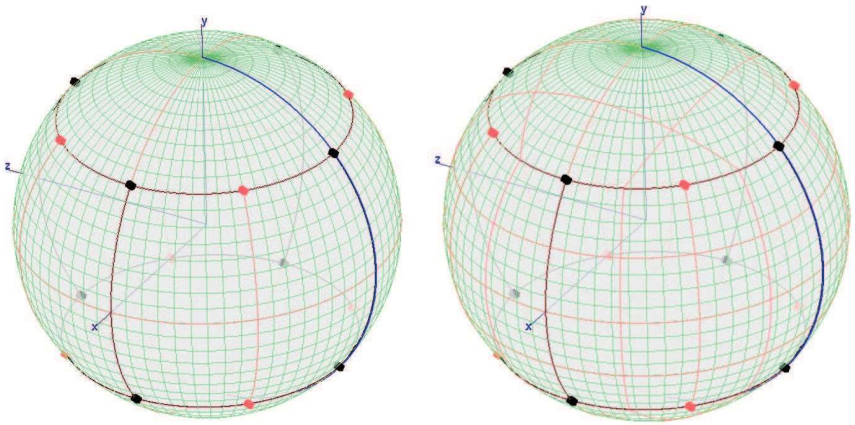
An alternative implementation for tessellating cube faces is to recursively divide each face into four new faces similar to a quadtree. Each face can serve as a separate tree using chunked LOD (see Chapter 14), as done by Wittens [182].

An alternative to tessellating each face separately is to tessellate a single face, then rotate it into place, and project it onto the ellipsoid. This can be done on the CPU or GPU; for the latter, memory is only required for a single cube's face.

Miller and Gaskins describe an experimental globe-tessellation algorithm for NASA World Wind that is similar to cube-map tessellation [116]. The faces of a cube are mapped onto the ellipsoid with one edge of the cube aligned with the IDL. The four nonpolar faces are subdivided into quadrilaterals representing a rectangular grid along lines of constant latitude and longitude. Note that this is different than cube-map tessellation, which creates geodesic curves.

The polar faces are tessellated differently to minimize issues at the poles. First, a polar face is divided into four regions, similar to placing an X through the face (see Figure 4.5(a)). Afterwards, polar regions are subdivided into four regions at their edge's midpoints (see Figure 4.5(b)). Note that edges in the polar regions are not lines of constant latitude or longitude but instead geodesic curves. Special care is taken to avoid cracks when triangulating quadrilaterals that share an edge with a polar and nonpolar region.

Miller and Gaskins suggest two criteria for selecting a latitude range for polar regions. If the goal is to maximize the number of lines of constant



(a) (b)

Figure 4.5. NASA World Wind experimental globe-tessellation algorithm. (Images courtesy of James Miller, University of Kansas, and Tom Gaskins, NASA [116].)

latitude or longitude, only regions above a high latitude (e.g., 60° or 70°) or below a low latitude should be polar regions. Ultimately, they use a range of above/below +/- 40° for polar regions so all quadrilaterals are approximately the same size.

4.1.4 Geographic-Grid Tessellation

Perhaps the most intuitive ellipsoid-tessellation approach is to use a *geographic grid*. This algorithm has two steps.

First, a set of points distributed over the ellipsoid are computed. This can be done with a nested for loop that iterates over the spherical coordinates $\phi \in [0, \pi]$ and $\theta \in [0, 2\pi]$ at a chosen granularity. Recall that ϕ is similar to latitude and θ is similar to longitude. Most applications use a different granularity for ϕ and θ since their ranges are different.[3] Each grid point is converted from spherical coordinates to Cartesian coordinates using Equation (4.4):

$$\begin{aligned} x &= a\cos\theta\sin\phi, \\ y &= b\sin\theta\sin\phi, \\ z &= c\cos\phi. \end{aligned} \tag{4.4}$$

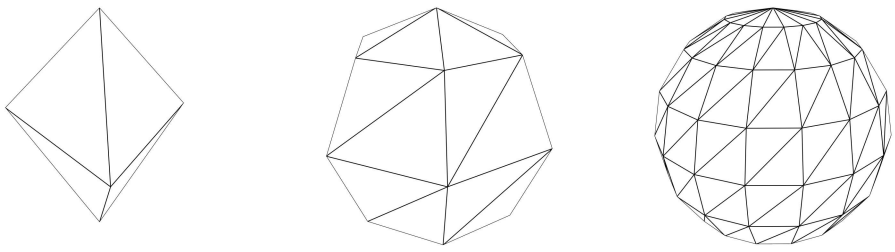
[3]For example, GLUT's gluSphere function takes the number of stacks along the z-axis and number of slices around the z-axis. These are used to determine the granularity for ϕ and θ, respectively.

```
IList<Vector3d> positions = // ...
positions.Add(new Vector3d(0, 0, ellipsoid.Radii.Z));

for (int i = 1; i < numberOfStackPartitions; ++i)
{
  double phi = Math.PI * (((double)i) /
               numberOfStackPartitions);
  double cosPhi = Math.Cos(phi);
  double sinPhi = Math.Sin(phi);

  for (int j = 0; j < numberOfSlicePartitions; ++j)
  {
    double theta = (2.0 * Math.PI) *
                   (((double)j) / numberOfSlicePartitions);
    double cosTheta = Math.Cos(theta);
    double sinTheta = Math.Sin(theta);

    positions.Add(new Vector3d(
        ellipsoid.Radii.X * cosTheta * sinPhi,
        ellipsoid.Radii.Y * sinTheta * sinPhi,
        ellipsoid.Radii.Z * cosPhi));
  }
}
positions.Add(new Vector3d(0, 0, -ellipsoid.Radii.Z));
```

Listing 4.5. Computing points for geographic-grid tessellation.

Here, (a, b, c) is the radii of the ellipsoid, which is $(1, 1, 1)$ in the trivial case of the unit sphere. Since $\sin(0) = \sin\pi = 0$, the north pole ($\phi = 0$) and south pole ($\phi = \pi$) are treated as special cases outside of the for loop. A single point, $(0, 0, \pm c)$, is used for each. This step of the algorithm is shown in Listing 4.5. Note that Math.`Cos(theta)` and Math.`Sin(theta)` could be computed once and stored in a lookup table.

Once points for the tessellation are computed, the second step of the algorithm is to compute triangle indices. Indices for triangle strips are generated in between every row, except the rows adjacent to the poles. To avoid



(a) (b) (c)

Figure 4.6. Geographic-grid tessellation connects a distribution of points on the ellipsoid with triangles. (a) $\Delta\theta = 2\pi/3$, $\Delta\phi = \pi/2$. (b) $\Delta\theta = \pi/3$, $\Delta\phi = \pi/4$. (c) $\Delta\theta = \pi/8$, $\Delta\phi = \pi/8$.

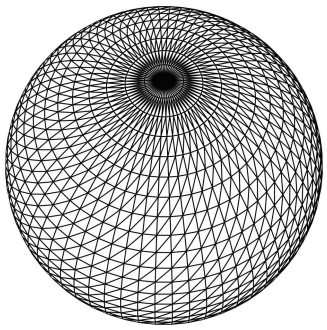


Figure 4.7. Standard geographic-grid tessellation results in too many triangles at the poles.

degenerate triangles, these rows are connected to their closest pole using triangle fans. For a complete implementation, see `GeographicGridEllipsoidTessellator`.

Figure 4.6(a) shows a very coarse approximation to an ellipsoid with just two rows of triangle fans. In Figure 4.6(b), the nonpolar rows are triangle strips. Figure 4.6(c) begins to look like an ellipsoid.

This algorithm is conceptually straightforward and widely used. It was used in the initial release of NASA World Wind [116] and in STK and Insight3D. It avoids triangles that cross the IDL and produces triangles whose edges are lines of constant latitude and longitude, except, of course, the bisecting edges.

The major weakness of this approach is the singularity created at the poles, shown in Figure 4.7. The thin triangles near the poles can lead to lighting and texturing artifacts. Since fragments on shared edges are redundantly shaded, this can even add fragment-shading costs [139]. The overcrowding of triangles at the poles leads to additional performance issues because view frustum culling is now less effective near the poles (e.g., a view at a pole contains many more triangles than a similar view at the equator).

Over-tessellation at the poles can be dealt with by adjusting the tessellation based on latitude. The closer a row is to a pole, the fewer points, and thus fewer triangles, in between rows are needed. Gerstner suggests scaling by $\sin\phi$ to relate the area of a triangle to the area of the spherical section it projects onto [61].

4.1.5 Tessellation-Algorithm Comparisons

It is hard to pick a clear winner between the three ellipsoid-tessellation algorithms discussed. They are all fairly straightforward to implement and used successfully in a wide array of applications. Table 4.1 lists some

Algorithm	Oversampling at Poles	Triangles Avoid Poles	Triangles Avoid IDL	Similar Shape Triangles	Aligned with Constant Latitude/Longitude Lines
Subdivision Surfaces	No	No	No	Yes	No
Cube Map	No	No	Yes	No	No
Geographic Grid	Yes	Yes	Yes	No	Yes

Table 4.1. Properties of ellipsoid-tessellation algorithms.

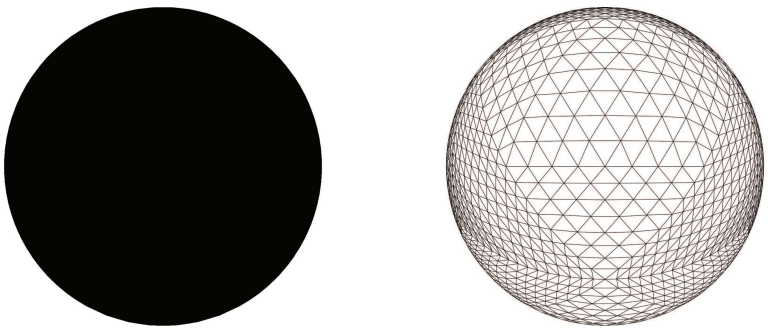
properties to consider when choosing among them. This isn't a hard and fast list; algorithm variations can change their properties. For example, geographic-grid tessellation can minimize oversampling at the poles by adjusting the tessellation based on latitude. Likewise, any tessellation can avoid triangles that cross the IDL by splitting triangles that cross it.

4.2 Shading

Tessellation is the first step in globe rendering. The next step is *shading*, that is, simulating the interaction of light and material to produce a pixel's color. First basic lighting and texturing are reviewed, then specific virtual globe techniques for rendering a latitude-longitude grid and night lights are covered.

4.2.1 Lighting

This section briefly examines lighting computations in a vertex and fragment shader used to shade a globe. Starting with the pass-through shaders of Listing 4.6, diffuse and specular terms are incrementally added.



(a) (b)

Figure 4.8. The pass-through shader produces a solid color without texturing or lighting. (a) Solid globe. (b) Wireframe globe.

```
// Vertex shader
in vec4 position;
uniform mat4 og_modelViewPerspectiveMatrix;

void main()
{
  gl_Position = og_modelViewPerspectiveMatrix * position;
}

// Fragment shader
out vec3 fragmentColor;
void main() { fragmentColor = vec3(0.0, 0.0, 0.0); }
```

Listing 4.6. Pass-through shaders.

The vertex shader transforms the input `position` to clip coordinates by multiplying by the automatic uniform, `og_modelViewPerspective`. Like all OpenGlobe automatic uniforms, simply declaring this uniform in a shader is enough to use it. There is no need to explicitly assign it in C# code; it is automatically set in Context.`Draw` based on the values in SceneState.

As you can imagine, these shaders produce a solid black globe, as shown in Figure 4.8(a).

Try This ○○◯◯

While reading this section, replace the `vs` and `fs` strings in SubdivisionSphere1's constructor with the vertex and fragment shaders in this section, and see if you can reproduce the figures.

Without lighting, the curvature of the globe is not apparent. To light the globe, we will use a single point light source that is located at the eye position, as if the viewer were holding a lantern next to his or her head. To keep the shaders concise, lighting is computed in world coordinates. In many applications, lighting is computed in eye coordinates. Although this makes some lighting computations simpler since the eye is located at $(0, 0, 0)$, using eye coordinates also requires transforming the world position and normal into eye coordinates. For applications with multiple lights and objects in different model coordinates, this is an excellent approach. Since our example only has a single coordinate space, light is computed in world coordinates for conciseness.

For visual quality, lighting is computed per fragment. This prevents the underlying tessellation of the globe from showing through and eliminates artifacts with specular highlights that are common with per-vertex lighting. Per-fragment lighting also integrates well with texture-based effects discussed in Section 4.2.5.

```
in vec4 position;
out vec3 worldPosition;
out vec3 positionToLight;

uniform mat4 og_modelViewPerspectiveMatrix;
uniform vec3 og_cameraLightPosition;

void main()
{
  gl_Position = og_modelViewPerspectiveMatrix * position;
  worldPosition = position.xyz;
  positionToLight = og_cameraLightPosition - worldPosition;
}
```

Listing 4.7. Diffuse-lighting vertex shader.

```
in vec3 worldPosition;
in vec3 positionToLight;
out vec3 fragmentColor;

void main()
{
  vec3 toLight = normalize(positionToLight);
  vec3 normal = normalize(worldPosition);
  float diffuse = max(dot(toLight, normal), 0.0);
  fragmentColor = vec3(diffuse, diffuse, diffuse);
}
```

Listing 4.8. Diffuse-lighting fragment shader.

In order to compute per-fragment *Phong* lighting, a linear combination of *diffuse*, *specular*, and *ambient* terms determines the intensity of the light in the range $[0, 1]$. First, consider just the diffuse term. A characteristic of rough surfaces, diffuse light scatters in all directions and generally defines the shape of an object. Diffuse light is dependent on the light position and surface orientation but not the view parameters. To compute the diffuse term, the pass-through vertex shader needs to pass the world position and position-to-light vector to the fragment shader, as shown in Listing 4.7.

This vertex shader utilizes `og_cameraLightPosition`, an automatic uniform representing the position of the light in world coordinates. The fragment shader in Listing 4.8 uses the diffuse term to compute the fragment's intensity.

The dot product is used to approximate the diffuse term, since diffuse light is most intense when the vector from the fragment to the light is the same as the surface normal at the fragment ($\cos(0) = 1$) and intensity drops off as the angle between the vectors increases (cos approaches zero). The vector `positionToLight` is normalized in the fragment shader since it may not be unit length due to interpolation between the vertex and fragment shader, even if it were normalized in the vertex shader. Since the globe is

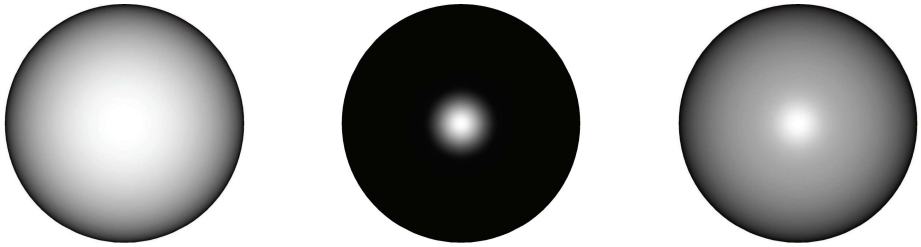


Figure 4.9. Phong lighting with one light located at the camera. (a) Diffuse light scatters in all directions. (b) Specular light highlights the reflection of shiny surfaces. (c) 70% diffuse and 30% specular light is used.

approximated by a sphere in this example, the fragment's surface normal is computed analytically by simply normalizing the world position. The general case of computing the geodetic surface normal for a point on an ellipsoid requires using `GeodeticSurfaceNormal`, introduced in Section 2.2.2. Figure 4.9(a) shows the globe shaded with just diffuse light.

Specular light shows highlights capturing the reflection of shiny surfaces, such as bodies of water. Figure 4.9(b) shows the globe shaded with just specular light, and Figure 4.9(c) shows the globe shaded with both diffuse and specular light. Specular light is dependent on the eye location, so the vertex shader in Listing 4.7 needs to output a vector from the vertex to the eye. Listing 4.9 shows the final vertex shader for Phong lighting. This shader forms the foundation for many shaders throughout this book.

```
in vec4 position;
out vec3 worldPosition;
out vec3 positionToLight;
out vec3 positionToEye;

uniform mat4 og_modelViewPerspectiveMatrix;
uniform vec3 og_cameraEye;
uniform vec3 og_cameraLightPosition;

void main()
{
  gl_Position = og_modelViewPerspectiveMatrix * position;
  worldPosition = position.xyz;
  positionToLight = og_cameraLightPosition - worldPosition;
  positionToEye = og_cameraEye - worldPosition;
}
```

Listing 4.9. Vertex shader for diffuse and specular lighting.

```
in vec3 worldPosition;
in vec3 positionToLight;
in vec3 positionToEye;
out vec3 fragmentColor;

uniform vec4 og_diffuseSpecularAmbientShininess;

float LightIntensity(vec3 normal, vec3 toLight, vec3 toEye,
                     vec4 diffuseSpecularAmbientShininess)
{
  vec3 toReflectedLight = reflect(-toLight, normal);

  float diffuse = max(dot(toLight, normal), 0.0);
  float specular = max(dot(toReflectedLight, toEye), 0.0);
  specular = pow(specular, diffuseSpecularAmbientShininess.w);

  return (diffuseSpecularAmbientShininess.x * diffuse) +
         (diffuseSpecularAmbientShininess.y * specular) +
          diffuseSpecularAmbientShininess.z;
}

void main()
{
  vec3 normal = normalize(worldPosition);
  float intensity = LightIntensity(normal,
      normalize(positionToLight), normalize(positionToEye),
      og_diffuseSpecularAmbientShininess);
  fragmentColor = vec3(intensity, intensity, intensity);
}
```

Listing 4.10. Final Phong-lighting fragment shader.

The specular term is approximated in the fragment shader by reflecting the vector to the light around the surface normal and computing the dot product of this vector with the vector to the eye. The specular term is most intense when the angle between the vector to the eye and the normal is the same as the angle between the vector to the light and the normal. Intensity drops off as the difference in angles increases. Listing 4.10 shows the final Phong-lighting fragment shader.

The bulk of the work is done in a new function `LightIntensity`, which is used throughout the rest of this book. The diffuse term is computed as before, and the specular term is computed as described above. The specular term is then raised to a power to determine its sharpness. Small exponents produce a large, dull specular highlight, and high values create a tight highlight. Finally, the diffuse and specular terms are multiplied by user-defined coefficients, and an ambient term, representing light throughout the entire scene, is added to create a final light intensity.

4.2.2 Texturing

Lighting the globe makes its curvature apparent, but the real fun begins with texture mapping. Texture mapping helps achieve one of the primary

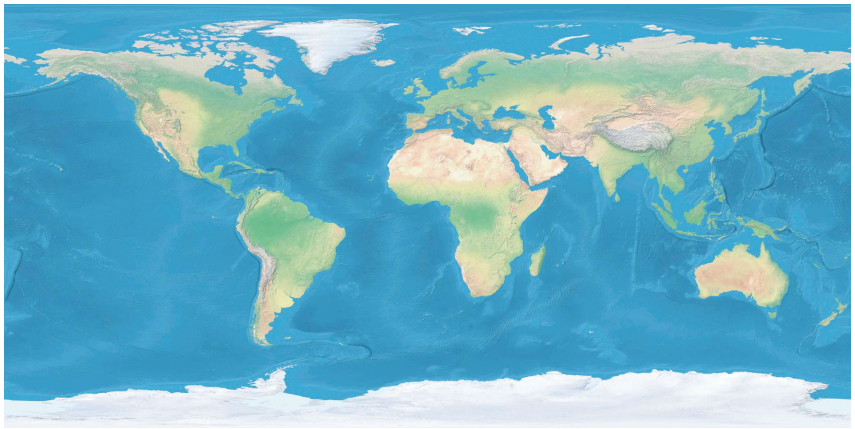


Figure 4.10. Satellite-derived land imagery with shaded relief and water from Natural Earth.

tasks of virtual globes: displaying high-resolution imagery. Our discussion focuses on computing texture coordinates per fragment, similar to how normals were computed per fragment in the previous section. For now, it is assumed the texture fits into memory, and there is enough precision in GLSL's `float` for texture coordinates spanning the globe.

World rasters, like the one in Figure 4.10, typically have a $2:1$ aspect ratio and WGS84 data. To map such a texture onto the globe, texture coordinates need to be computed. Given a fragment on the globe with a geodetic surface normal, n, with components in the range $[-1, 1]$, the goal is to compute texture coordinates, s and t, in the range $[0, 1]$. This mapping is shown in Equation (4.5):

$$\begin{aligned} s &= \frac{\operatorname{atan2}(n_y, n_x)}{2\pi} + 0.5, \\ t &= \frac{\arcsin n_z}{\pi} + 0.5. \end{aligned} \tag{4.5}$$

First, consider t since it is the simpler of the two expressions. Since n_z relates to latitude, it can be used to compute t, the vertical texture coordinate. Intuitively, t should be 1 when n_z is 1 (north pole), 0.5 when n_z is 0 (equator), and 0 when n_z is -1 (south pole). The mapping cannot be linear because of the curvature of the Earth, so arcsin is used to capture the curvature. Given a value, x, in the range $[-1, 1]$, arcsin returns the angle, ϕ, in the range $[-\pi/2, \pi/2]$ that satisfies $x = \sin(\theta)$. Above, $\frac{\arcsin n_z}{\pi}$ is in the range $[-0.5, 0.5]$. Adding 0.5 puts t in the desired $[0, 1]$ range.

```
in vec3 worldPosition;
out vec3 fragmentColor;

uniform sampler2D og_texture0;
uniform vec3 u_globeOneOverRadiiSquared;

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

vec2 ComputeTextureCoordinates(vec3 normal)
{
  return vec2(
      atan(normal.y, normal.x) * og_oneOverTwoPi + 0.5,
      asin(normal.z) * og_oneOverPi + 0.5);
}

void main()
{
  vec3 normal = GeodeticSurfaceNormal(
      worldPosition, u_globeOneOverRadiiSquared);
  vec2 textureCoordinate = ComputeTextureCoordinates(normal);
  fragmentColor = texture(og_texture0, textureCoordinate).rgb;
}
```

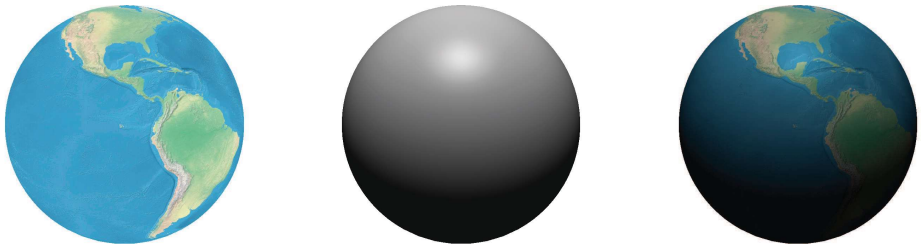
Listing 4.11. Globe-texturing fragment shader.

Likewise, n_x and n_y relate to longitude and can be used to compute s, the horizontal texture coordinate. Starting with $n_x = -1$ and $n_y = 0$ (International Date Line), sweep the vector $[n_x, n_y]$ around the positive z-axis. As θ, the angle between the vector and the negative x-axis in the xy-plane, increases, atan2(n_y, n_x) increases from $-\pi$ to π. Dividing this by 2π puts it in the range $[-0.5, 0.5]$. Finally, adding 0.5 puts s in the desired $[0, 1]$ range.

A fragment shader for texture mapping the globe based on Equation (4.5) is shown in Listing 4.11. The function that computes texture coordinates given the fragment's normal, `ComputeTextureCoordinates`, is used throughout this book. Note that the GLSL function atan is equivalent to atan2.

The result of this fragment shader is shown in Figure 4.11(a). Although textured nicely, the globe lacks curvature because the shader does not include lighting. A simple way to combine lighting and texture mapping is to modulate the intensity of the light with the color of the texture:

```
  fragmentColor = intensity *
                  texture(og_texture0, textureCoordinate).rgb;
```



(a) (b) (c)

Figure 4.11. Combining texturing and lighting. (a) Texture mapping without lighting provides color but not shape. (b) Lighting without texture mapping provides shape but not color. (c) Combining texture mapping and lighting yields both shape and color.

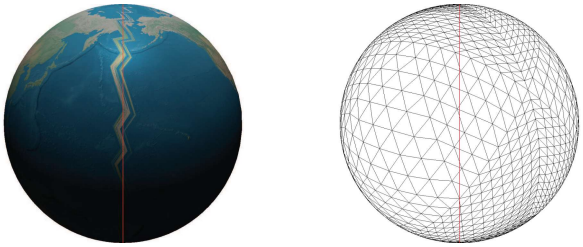
The result is shown in Figure 4.11(c). The final fragment color is the product of the color from Figures 4.11(a) and 4.11(b).

Mapping a rectangular texture to a globe results in problems at the poles because there is a high ratio of texels to pixels. Texture filtering doesn't help and can even make things worse. EVE Online solves this by interpolating between two texture coordinates: a planar projection for the poles and a spherical projection for the "belly" [109]. Another approach to avoiding distortion at the poles is to store the globe texture as a cube map [57]. This has the additional benefit of spreading texel detail more evenly throughout the globe, but the cube map introduces slight distortion at the boundary of each cube face.

4.2.3 CPU/GPU Trade-offs

Now that we have successfully computed geometry for a globe and shaded it, let's think critically about the algorithms with respect to processor and memory usage. Only positions for the ellipsoid are computed on the CPU and stored in vertex attributes. Normals and texture coordinates, required for shading, are computed per fragment on the GPU. Alternatively, normals and texture coordinates could be computed and stored once, per vertex, on the CPU. The per-fragment approach has several advantages:

- Reduced memory usage since per-vertex normals and texture coordinates do not need to be stored. Less vertex data results in less system bus traffic, lower memory bandwidth requirements, and lower vertex assembly costs [123].
- Improved visual quality since the normal is analytically computed per fragment, not interpolated across a triangle. This quality improvement is similar to normal and bump mapping, which modify normals



(a) (b)

Figure 4.12. Per-vertex texture-coordinate interpolation across the IDL causes artifacts unless special care is taken for triangles crossing the IDL. (a) Artifacts along the IDL. (b) Wireframe showing triangles crossing the IDL.

per fragment based on texture lookups, making a flat triangle appear to have more detail due to lighting changes.

- More concise code since the CPU tessellation is simplified without adding significant complexity to the shaders.

○○○○ Try This

To better understand the trade-offs between per-vertex and per-fragment approaches, experiment with the per-vertex example, Chapter04SubdivisionSphere2, and see how it differs from the per-fragment example, Chapter04SubdivisionSphere1. In particular, SubdivisionSphereTessellator, the tessellator that computes positions, normals, and texture coordinates, is nearly twice the length of SubdivisionSphereTessellatorSimple, the tessellator that just computes positions.

The last item is particularly important because, when it comes to graphics code, it is easy to get caught up in performance and not realize how important it is to produce simple, clean, concise code. The per-fragment approach simplifies CPU tessellation because the tessellator does not need to compute and store normals and texture coordinates per vertex. Since per-vertex texture coordinates are interpolated across triangles, special care needs to be taken for triangles that cross the IDL and poles [132]. Simply using Equation (4.5) results in a triangle's vertex on one side of the IDL having an s texture coordinate near one and a vertex on the other side of the IDL having s near zero, so almost the entire texture is interpolated across the triangle. The resulting artifacts are shown in Figure 4.12. Of

course, this can be detected, and the texture coordinate can be adjusted for repeat filtering, but the per-fragment approach eliminates the need for special case code.

There are downsides to the per-fragment approach. On current GPUs, inverse trigonometry functions are not well optimized or highly precise.

4.2.4 Latitude-Longitude Grid

Almost all virtual globes have the ability to show a latitude-longitude grid, also called a lat/lon grid, on the globe, as shown in Figure 4.13. The grid makes it easy for users to quickly identify the approximate geographic position of points on the globe.

The grid is composed of lines of constant latitude, often highlighting the equator, Arctic Circle, Tropic of Cancer, Tropic of Capricorn, and Antarctic Circle, and lines of constant longitude, often highlighting the prime meridian and the IDL. As the viewer zooms in, the resolution of the grid usually increases, similar to how a level-of-detail algorithm refines a mesh as the viewer zooms in (see Figures 4.14(a)–4.14(c)).

Most grid-rendering approaches compute lines of latitude and lines of longitude at a resolution appropriate for the current view and render the grid using line or line-strip primitives. Using indexed lines can have an advantage over line strips since lines of latitude and lines of longitude typically share the same vertices. If the viewer doesn't move drastically, the same grid tessellation can be reused for many frames.

This approach is tried and true, and in fact, we use it in STK, with the exception that the resolution is defined by the user, not the viewer's current zoom level. Even so, we present a shader-based approach here that

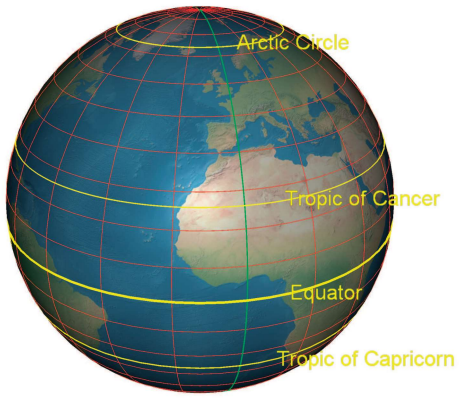


Figure 4.13. Latitude-longitude grid with annotations.

(a) (b) (c)

Figure 4.14. The latitude-longitude grid increases resolution as the viewer zooms in. A view-dependent grid resolution ensures the grid doesn't clutter the globe at distant views and has enough detail at close views.

procedurally generates the grid in the fragment shader during the same pass the globe is rendered. This has several advantages:

- CPU time is not required to compute the grid.
- An extra pass is not needed to render the grid.
- No additional memory or bus traffic is required.
- Z-fighting between the grid and globe does not occur.

There are, of course, disadvantages:

- The time required to render the globe and grid in one pass is higher than the time required to just render the globe.
- The accuracy is limited by the GPU's 32-bit floating-point precision.
- Text annotations still need to be handled separately.

Even with the disadvantages, procedurally generating the grid in the fragment shader is an attractive approach.

Our typical lighting and texturing fragment shader needs the additional uniforms shown in Listing 4.12. The `x` component of each uniform corresponds to lines of longitude and the `y` component corresponds to lines of latitude. The uniform `u_gridLineWidth` defines the line width and `u_gridResolution` defines the grid resolution, that is, the spacing between lines, in the range $[0, 1]$. The number of partitions is determined by $\frac{1}{\text{resolution}}$. For example, a latitude line spacing of 0.05 results in 20 partitions made by lines of constant latitude.

```
uniform vec2 u_gridLineWidth;
uniform vec2 u_gridResolution;
```

Listing 4.12. Uniforms for the latitude-longitude grid fragment shader.

The texture coordinate is used to determine if the fragment is on a grid line. If so, the fragment is shaded as such, otherwise, the fragment is shaded normally using the globe's texture. The s and t texture coordinates span the globe from west to east and south to north, respectively. The mod function is used to determine if the line resolution evenly divides the texture coordinate. If so, the fragment is on the grid line. The simplest approach detects a grid line if (any(equal(mod(`textureCoordinate`, `u_gridResolution`), `vec2`(0.0, 0.0)))). In practice, this results in no fragments on the grid because explicitly comparing floating-point numbers against zero is risky business.

The obvious next attempt is to test for values around zero, perhaps any(lessThan(mod(`textureCoordinate`, `u_gridResolution`), vec2(0.001, 0.001))). This does in fact detect fragments on the grid, but the width of the grid lines, in pixels, varies with the viewer's zoom because of the hard-coded epsilon. When the viewer is far away from the globe, texture coordinates are changing rapidly from fragment to fragment in screen space, so 0.001 creates thin lines. When the viewer is close to the surface, texture coordinates are changing slowly from fragment to fragment, so 0.001 creates thick lines.

To shade lines with a view-independent constant pixel width, the epsilon should be related to the rate of change in texture coordinates from fragment to fragment. The key is to determine how much a texture coordinate is changing between adjacent fragments and use this, or some multiple of it, as the epsilon. Thankfully, GLSL provides functions that return the rate of change (i.e., the derivative) of a value from fragment to fragment. The function dFdx returns the rate of change of an expression in the x screen direction and dFdy returns the rate of change in the y direction. For example, dFdx(texture `Coordinate`) returns a vec2 describing how fast the `s` and `t` texture coordinates are changing in the x screen direction.

The epsilon to determine if a fragment is on a grid line should be a vec2, with its `s` component representing the maximum tolerance for the `s` texture coordinate in either x or y screen space direction. Likewise, its `t` component represents the maximum tolerance for the `t` texture coordinate in either x or y screen space direction. This is shown in Listing 4.13, along with the entire fragment shader `main` function. Run Chapter04LatitudeLongitudeGrid for a complete example.

Fragments on the grid are shaded solid red. A variety of shading options exist. The color could be a uniform or two uniforms, one for lines of latitude

```
void main()
{
  vec3 normal = GeodeticSurfaceNormal(
      worldPosition, u_globeOneOverRadiiSquared);
  vec2 textureCoordinate = ComputeTextureCoordinates(normal);
  vec2 distanceToLine = mod(textureCoordinate, u_gridResolution);
  vec2 dx = abs(dFdx(textureCoordinate));
  vec2 dy = abs(dFdy(textureCoordinate));
  vec2 dF = vec2(max(dx.s, dy.s),
                 max(dx.t, dy.t)) * u_gridLineWidth;

  if (any(lessThan(distanceToLine, dF)))
  {
    fragmentColor = vec3(1.0, 0.0, 0.0);
  }
  else
  {
    float intensity =
        LightIntensity(normal,
                       normalize(positionToLight),
                       normalize(positionToEye),
                       og_diffuseSpecularAmbientShininess);
    fragmentColor = intensity *
                    texture(og_texture0, textureCoordinate).rgb;
  }
}
```

Listing 4.13. Lat/lon fragment grid shader.

and one for lines of longitude. Certain lines, like the equator and prime meridian, can be detected based on the texture coordinate and shaded a different color, as done in Figure 4.13. The line color can be blended with the globe texture and made to fade in/out based on the viewer's zoom. Finally, antialiasing and line patterns can be computed.

Add antialiasing to the grid lines in Chapter04LatitudeLongitudeGrid by utilizing the prefiltering technique in the `OpenGlobe.Scene.WireFrame` fragment shader based on Brentzen [10]. Add a pattern to the grid lines based on the example described by Gateau [56].

○○○○ Try This

The grid resolution can be made view-dependent by modifying `u_gridResolution` on the CPU based on the viewer's height. Any function that maps the viewer's height to a grid resolution can be used. A flexible approach is to define a series of nonoverlapping height intervals (e.g., $[0, 100), [100, 1,000), [1,000, 10,000)$) that each correspond to a grid resolution. When the globe is rendered, or when the viewer moves, the

interval containing the viewer's height is found and the corresponding grid resolution is used.

For a large number of intervals, a tree data structure can be used to quickly find the interval containing the viewer's height, but this is usually unnecessary. A simple sorted list of intervals can be used efficiently by exploiting temporal coherence; instead of testing against the entire list of intervals in order, first test the viewer's height against the interval it was in last time. If it is outside this interval, the adjacent intervals are tested, and so on. In the vast majority of cases, the lookup is constant time.

4.2.5 Night Lights

A common feature in virtual globes and planet-centric games is to show nighttime city lights on the side of the globe not illuminated by the sun. This is a classic use of multitexturing: a daytime texture is used where the sun illuminates the surface and a night-lights texture is used where it does not. The texture in Figure 4.15 shows Earth's city lights at nighttime.

Most of our night-lights implementation is in the fragment shader. Our normal pass-through vertex shader, such as the one from Chapter04SubdivisionSphere1, can be used with the exception that the light position uniform should be the sun position, `og_sunPosition`, instead of the light attached to the camera, `og_cameraLightPosition`.

Our normal fragment shader needs more modifications, including the four new uniforms shown in Listing 4.14. A day texture, `u_dayTexture`, and night texture, `u_nightTexture`, are required. Instead of having a sharp transition between day and night for areas of the globe experiencing

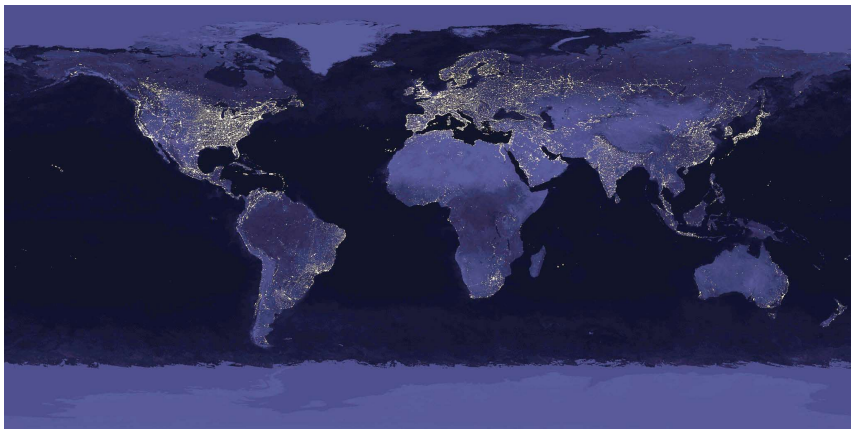


Figure 4.15. This city-lights texture, from NASA Visible Earth, is applied to the surface of the globe not illuminated by the sun.

```
uniform sampler2D u_dayTexture;
uniform sampler2D u_nightTexture;
uniform float u_blendDuration;
uniform float u_blendDurationScale;
```

Listing 4.14. Uniforms for the night-lights shader.

dusk and dawn, the day and night textures are blended based on `u_blendDuration`, the duration of the transition period in the range $[0, 1]$. The function `u_blendDurationScale` is simply precomputed as $\frac{1}{(2*\text{u\_blendDuration})}$ to avoid redundant computation in the fragment shader.

The bulk of the fragment shader is shown in Listing 4.15. The diffuse-lighting component is used to determine if the fragment should be shaded as day, night, or dusk/dawn. If diffuse is greater than the blend duration, then the fragment is shaded using the day texture and Phong lighting. Think of it as if there were no transition period (`u_blendDuration` $= 0$); if the dot product between the fragment's surface normal and the vector to the light is positive, then the fragment is illuminated by the sun. Likewise, if diffuse is less than `-u_blendDuration`, then the fragment should be shaded using just the night texture and no lighting. Dusk/dawn is handled by the final condition when diffuse is in the range [`-u_blendDuration`, `u_blendDuration`] by blending the day color and night color using mix.

```
vec3 NightColor(vec3 normal)
{
  return texture(u_nightTexture,
                 ComputeTextureCoordinates(normal)).rgb;
}

vec3 DayColor(vec3 normal, vec3 toLight, vec3 toEye,
    float diffuseDot, vec4 diffuseSpecularAmbientShininess)
{
  float intensity = LightIntensity(normal, toLight, toEye,
      diffuseDot, diffuseSpecularAmbientShininess);
  return intensity * texture(u_dayTexture,
      ComputeTextureCoordinates(normal)).rgb;
}

void main()
{
  vec3 normal = normalize(worldPosition);
  vec3 toLight = normalize(positionToLight);
  float diffuse = dot(toLight, normal);

  if (diffuse > u_blendDuration)
  {
    fragmentColor = DayColor(normal, toLight,
                             normalize(positionToEye), diffuse,
                             og_diffuseSpecularAmbientShininess);
  }
  else if (diffuse < -u_blendDuration)
  {
```

```
    fragmentColor = NightColor(normal);
  }
  else
  {
    vec3 night = NightColor(normal);
    vec3 day = DayColor(normal, toLight,
                        normalize(positionToEye), diffuse,
                        og_diffuseSpecularAmbientShininess);
    fragmentColor = mix(night, day,
        (diffuse + u_blendDuration) * u_blendDurationScale);
  }
}
```

Listing 4.15. Night-lights fragment shader.

Example images are shown in Figure 4.16, and a complete example is provided in Chapter04NightLights. The shader can be written more concisely if both the day and night color are always computed. For fragments not in dusk/dawn, this results in a wasted texture read and possibly a wasted call to `LightIntensity`.

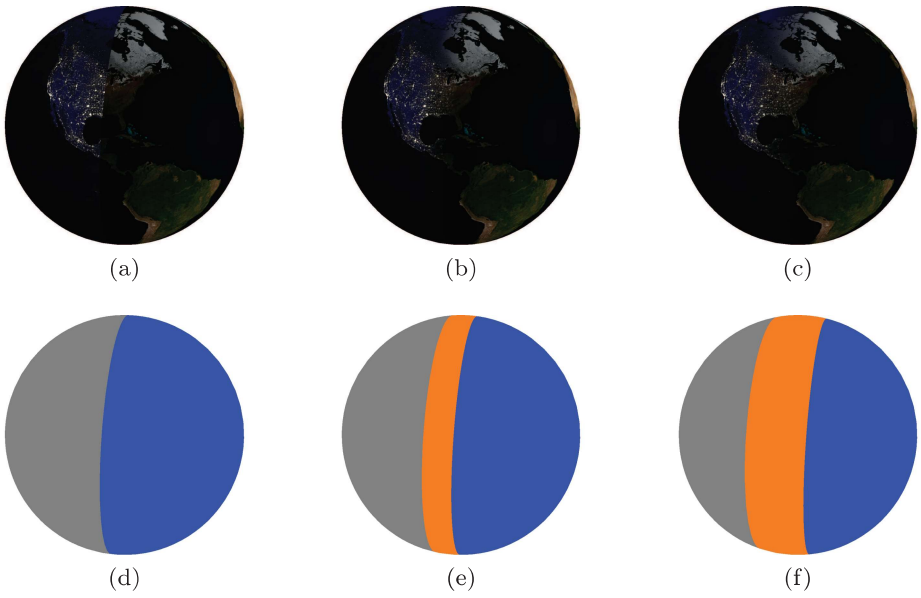


Figure 4.16. Night lights with various blend durations. In the bottom row, night is gray, day is blue, and the transition is orange. The top row shows the shaded result. (a) `u_blendDuration` = 0. (b) `u_blendDuration` = 0.1f. (c) `u_blendDuration` = 0.2f. (d) `u_blendDuration` = 0. (e) `u_blendDuration` = 0.1f. (f) `u_blendDuration` = 0.2f.

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.

Try This

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.

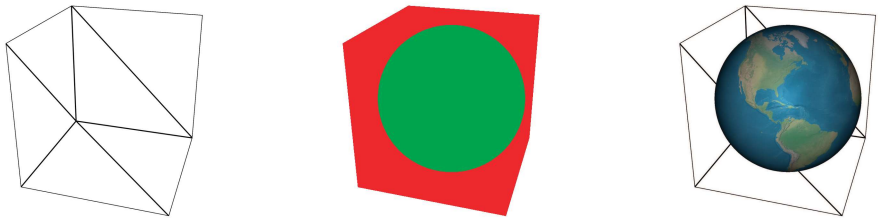
A strength of ray casting is that objects do not need to be tessellated into triangles for rendering. If we can figure out how to intersect a ray with an object, then we can render it. Therefore, no tessellation is required to render a globe represented by an ellipsoid because there is a well-known equation 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(`world Position` − `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].

Part II

Precision

5

Vertex Transform Precision

Consider a simple scene containing a few primitives: a point at the origin and two triangles in the plane $x = 0$ on either side of the point, spaced 1 m apart. Such a scene is shown in Figure 5.1(a). When the viewer zooms in, we expect the objects to appear coplanar, as shown in Figure 5.1(b).

What happens if we change the object positions such that they are drawn in the plane $x = 6{,}378{,}137$, which is Earth's equatorial radius in meters? When viewed from a distance, the objects still appear as expected—ignoring z-fighting artifacts, the subject of the following chapter. When zoomed in, other artifacts become noticeable: the objects appear to jitter, that is, they literally bounce around in a jerky manner as the viewer moves! In our example, this means the point and plane no longer appear

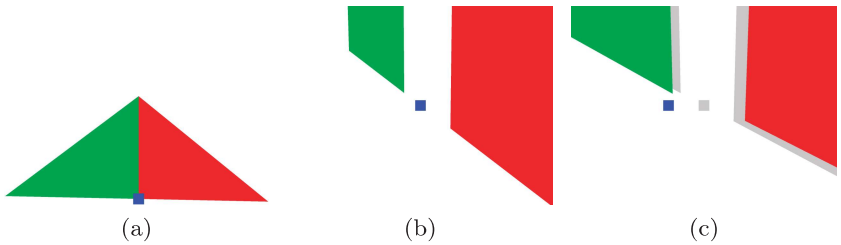


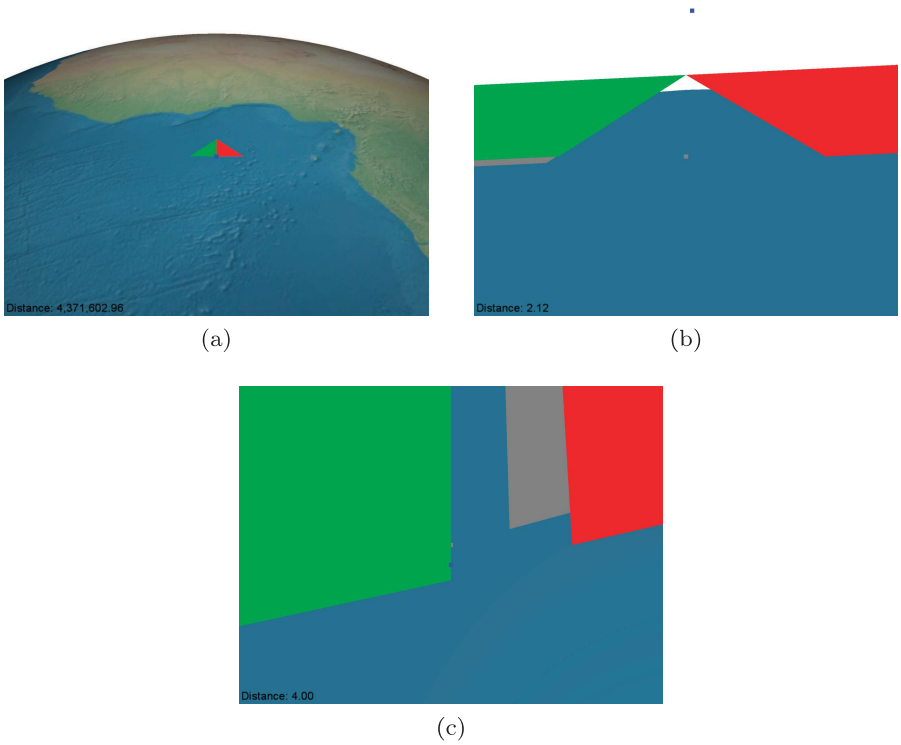
Figure 5.1. (a) A simple scene containing a point and two triangles. (b) Zoomed in on the scene, it is obvious that the point and triangles are in the same plane. (c) Since the scene is rendered with large world coordinates, jittering artifacts occur when zoomed in even closer than in (b). Here, the correct objects are shown in gray and the incorrectly positioned objects are shown in their original colors.

coplanar, as shown in Figure 5.1(c). Instead, they bounce in a view-dependent manner.

Jitter is caused by insufficient precision in a 32-bit floating-point value for a large value like $6,378,137$. Considering that typical positions in WGS84 coordinates are of similar magnitude, jitter is a significant challenge for virtual globes, where users expect to be able to zoom as close as possible to an object, without the object starting to bounce. This chapter explores why jittering occurs and approaches to combat it.

5.1 Jittering Explained

In a scene with large world coordinates, like virtual globes, jittering can occur when zoomed in and the viewer rotates or an object moves. Besides being visually disruptive, jitter makes it hard to determine exactly where



(a) (b) (c)

Figure 5.2. (a) From a distance, jitter is not apparent. (b) and (c) As the viewer zooms in, jittering becomes obvious. The correct point and triangle locations are shown in gray. The distance between the triangles is 1 m, and, in both images, the viewer is within a few meters of the point.

an object is located. Buildings bounce back and forth, vector data snap on and off of the underlying imagery, and terrain tiles overlap, then separate.

Jitter is shown in the series of images in Figure 5.2. The best way to get a feel for jitter artifacts is to run an application that exhibits it (not all virtual globes completely eliminate it) or to watch a video, such as the jittering space shuttle video in Ohlarik's article on the web [126].

Most of today's GPUs support only 32-bit floating-point values, which does not provide enough precision for manipulating large positions in WGS84 coordinates. CPUs use 64-bit double precision, which does provide enough precision, so many approaches to eliminating jitter utilize double-precision computation on the CPU or emulate it on the GPU. Although new Shader Model 5 hardware supports double precision, techniques described in this chapter are useful for supporting older hardware that is still in widespread use, and some techniques, namely RTC (see Section 5.2), are also useful for saving memory over using double precision.

5.1.1 Floating-Point Roundoff Error

Let's dive deeper into the behavior of floating-point numbers to understand why jittering occurs. It is not satisfactory to just say 32 bits is not enough precision—although that does summarize it. At the same time, we do not need to discuss every detail of the IEEE-754 specification [158]. Although GPUs sometimes deviate from the IEEE-754 for binary floating-point arithmetic [122], these deviations are not the culprit for jitter.

A rule of thumb is that 32-bit single-precision numbers have about seven accurate decimal digits, and 64-bit double-precision numbers have about 16. It is not possible to represent every rational number using 32, or even 64, bits. As such, many floating-point numbers are rounded to a nearby representable value. Listing 5.1 shows that the next representable 32-bit floating-point value after `6378137.0f` is `6378137.5f`. For example, `6378137.25f` is not a representable value and is rounded down to `6378137.0f`; likewise, `6378137.26f` is not representable and is rounded up to `6378137.5f`.

To make matters worse, gaps between representable floating-point values are dependent on the values themselves: larger values have larger gaps between them. In fact, if the numbers are large enough, whole numbers are skipped. Listing 5.2 shows that the next representable value after `17000000.0f` is `17000002.0f`. When modeling the world using meters, values in this 2-m gap are rounded up or down. Seventeen million meters may sound large considering that it is over ten million meters above Earth, but it is not impractical for all virtual globe applications. For example, in space applications, this is in the range for medium Earth orbit satellites.

```
float f = 6378137.0f;     // 6378137.0
float f1 = 6378137.1f;    // 6378137.0
float f2 = 6378137.2f;    // 6378137.0
float f25 = 6378137.25f; // 6378137.0
float f26 = 6378137.26f; // 6378137.5
float f3 = 6378137.3f;    // 6378137.5
float f4 = 6378137.4f;    // 6378137.5
float f5 = 6378137.5f;    // 6378137.5
```

Listing 5.1. Roundoff error for 32-bit floating-point values.

```
float f = 17000000.0f;  // 17000000.0
float f1 = 17000001.0f; // 17000000.0
float f2 = 17000002.0f; // 17000002.0
```

Listing 5.2. Roundoff error increases for large values to the point where whole numbers are skipped.

So far, we know there are a finite number of representable floating-point values, the gaps between these values increase as the values themselves increase, and nonrepresentable values are rounded to representable ones. There are other concerns too; in particular, addition and subtraction of values of different magnitudes drop least-significant bits, and thus precision, from the smaller values. The larger the variation between values, the more precision that is lost. To add icing to the cake, rounding errors accumulate from every floating-point operation.

5.1.2 Root Cause of Jittering

With the behavior of floating-point numbers in mind, let's revisit the jittery scene containing one point and two triangles, introduced at the beginning of the chapter. Consider the point located at $(6378137, 0, 0)$. When the viewer zooms in to within about 800 m, the point starts to jitter if the viewer rotates. Jitter becomes significant when zoomed in to just a few meters.

Why does it jitter? And why does jitter increase as the viewer gets closer?

The vertex shader used to transform this jittery point is shown in Listing 5.3. On the surface, everything looks OK: a model-view-perspective matrix transforms the input position from model coordinates (e.g., WGS84) to clip coordinates. This transformation matrix is computed on the CPU by multiplying individual model ($\mathbf{M}$), view ($\mathbf{V}$), and perspective ($\mathbf{P}$)

```
in vec4 position;
uniform mat4 og_modelViewPerspectiveMatrix;

void main()
{
  gl_Position = og_modelViewPerspectiveMatrix * position;
}
```

Listing 5.3. A seemingly innocent vertex shader can exhibit jitter.

matrices using double precision. Example values for these matrices, when the viewer is 800 m from the point, are shown below:

$$\mathbf{M} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

$$\mathbf{V} = \begin{pmatrix} 0.78 & 0.63 & 0.00 & -4,946,218.10 \\ 0.20 & -0.25 & 0.95 & -1,304,368.35 \\ 0.60 & -0.73 & -0.32 & -3,810,548.19 \\ 0.00 & 0.00 & 0.00 & 1.00 \end{pmatrix},$$

$$\mathbf{P} = \begin{pmatrix} 2.80 & 0.00 & 0.00 & 0.00 \\ 0.00 & 3.73 & 0.00 & 0.00 \\ 0.00 & 0.00 & -1.00 & -0.02 \\ 0.00 & 0.00 & -1.00 & 0.00 \end{pmatrix}. \tag{5.1}$$

The fourth column of $\mathbf{V}$, the translation, contains very large values because when the viewer is near the point, the viewer is far from the WGS84 origin. The model-view-perspective matrix, $\mathbf{MVP}$, is computed as $\mathbf{P} * \mathbf{V} * \mathbf{M}$. Values in its fourth column are even larger than in $\mathbf{V}$:

$$\mathbf{MVP} = \begin{pmatrix} 2.17 & 1.77 & 0.00 & -13,844,652.95 \\ 0.76 & -0.94 & 3.53 & -4,867,968.95 \\ -0.60 & 0.73 & 0.32 & 3,810,548.18 \\ -0.60 & 0.73 & 0.32 & 3,810,548.19 \end{pmatrix}. \tag{5.2}$$

When $\mathbf{MVP}$ is assigned to the vertex shader's uniform, it is converted from 64-bit to 32-bit values, which results in some roundoff error in the fourth column:

```
(float)-13,844,652.95 = -13,844,653.0f
(float)-4,867,968.95 = -4,867,969.0f
(float)3,810,548.18 = 3,810,548.25f
(float)3,810,548.19 = 3,810,548.25f
```

The vertex shader then multiplies the point's model position, p_{WGS84}, by **MVP** in 32-bit floating point:

$$
\begin{aligned}
p_{\text{clip}} &= \mathbf{MVP} * p_{\text{WGS84}} \\
&= \begin{pmatrix} 2.17f & 1.77f & 0.00f & -13,844,653.0f \\ 0.76f & -0.94f & 3.53f & -4,867,969.0f \\ -0.60f & 0.73f & 0.32f & 3,810,548.25f \\ -0.60f & 0.73f & 0.32f & 3,810,548.25f \end{pmatrix} \begin{pmatrix} 6,378,137.0f \\ 0.0f \\ 0.0f \\ 1.0f \end{pmatrix} \\
&= \begin{pmatrix} (2.17f)(6,378,137.0f) + -13,844,653.0f \\ (0.76f)(6,378,137.0f) + -4,867,969.0f \\ (-0.60f)(6,378,137.0f) + 3,810,548.25f \\ (-0.60f)(6,378,137.0f) + 3,810,548.25f \end{pmatrix}.
\end{aligned}
$$

This is where jittering artifacts are introduced: the matrix multiply is operating on large values with sparse 32-bit floating-point representations. A small change in the object or viewer position may not result in any change as roundoff errors produce the same results, but then suddenly roundoff error jumps to the next representable value and the object appears to jitter.

Jittering is noticeable when zoomed in (when considering Earth-scale scenes, 800 m is pretty close), but it is not noticeable when zoomed out. Why is there no jitter when the viewer is, say, 100,000 m from the point? Are the numbers smaller? The point didn't move, so p_{WGS84} is still the same, as are **M** and **P**, but **V**, and therefore **MVP**, changed because the viewer changed. For example,

$$
\mathbf{V} = \begin{pmatrix} 0.78 & 0.63 & 0.00 & -4,946,218.10 \\ 0.20 & -0.25 & 0.95 & -1,304,368.35 \\ 0.60 & -0.73 & -0.32 & -3,909,748.19 \\ 0.00 & 0.00 & 0.00 & 1.00 \end{pmatrix},
$$

$$
\mathbf{MVP} = \begin{pmatrix} 2.17 & 1.77 & 0.00 & -13,844,652.95 \\ 0.76 & -0.94 & 3.53 & -4,867,968.95 \\ -0.60 & 0.73 & 0.32 & 3,909,748.18 \\ -0.60 & 0.73 & 0.32 & 3,909,748.19 \end{pmatrix}.
$$

These matrices are different, but they look very similar to **V** and **MVP** when the viewer was 800 m away, in Equations (5.1) and (5.2), respectively. If the values are similar, shouldn't they exhibit similar floating-point precision problems?

Actually, the same behavior does occur—we just don't notice it because the amount of error introduced is less than a pixel. Jitter is view-dependent: when the viewer is very close to an object, a pixel may cover less than a

meter, or even a centimeter, and therefore, jittering artifacts are prominent. When the viewer is zoomed out, a pixel may cover hundreds of meters or more, and precision errors do not introduce enough jitter to shift entire pixels; therefore, jitter is not noticeable. The field of view can also affect jitter since it affects pixel size; in narrower fields of view, each pixel covers less world space and is, therefore, more susceptible to jitter.

In summary, jitter is dependent on the magnitude of the components of an object's position and how a pixel covering the object relates to meters. Positions with larger components (e.g., positions far away from the origin) create more jitter. Pixels covering fewer meters, achieved by zooming close to the object or a narrow field of view or both, makes jitter noticeable. Scenes only need to be pixel perfect; jitter within a pixel is acceptable.

Chapter05Jitter is the example project accompanying this chapter. It renders the simple scene composed of a point and two triangles and allows the user to change the algorithm used to eliminate jitter. In its default configuration, it doesn't eliminate jitter. Run the example, zoom in and out, and rotate to get a feel for when jitter starts to occur for this particular scene. How would changing the position of the object affect jitter?

○○○○ Try This

5.1.3 Scaling and Why It Doesn't Help

Since roundoff errors in large numbers create jitter, wouldn't eliminating large numbers also eliminate jitter? Instead of using meters as the unit, why not use a much larger unit—perhaps multiply each position in meters by $\frac{1}{6{,}378{,}137}$ so positions on Earth's surface have values near 1. There are many more representable 32-bit floating values near 1 than there are near 6,378,137.

Unfortunately, this doesn't actually fix jitter. Although there are many more representable floating-point values around 1 than there are around 6,378,137, there are still not enough because the units have scaled, requiring smaller gaps between representations. For example, there is a gap between 6378137.0`f` and 6378137.5`f`, which is 0.5 m when the world units are meters. If the world units are scaled by $\frac{1}{6{,}378{,}137}$, 0.5 m corresponds to $\frac{0.5}{6{,}378{,}137}$, or 0.000000078, so if there are representable values between 1.0`f` and 1.000000078`f`, then precision is gained. There are no such values though; in fact, there is a gap between 1.0`f` and 1.0000001`f`.

Scaling may make values themselves smaller, but it also requires smaller gaps between representable values, making it ineffective against jitter.

Try This ○○◯◯

Run Chapter05Jitter and enable the option to scale world coordinates. This changes the units from meters to 6,378,137 m. Zoom in and rotate to see that this does not eliminate jitter. The point does not appear to jitter, but the triangles do. Why? What is unique about the point's position? What if we changed the point's position slightly? Does it start to jitter?

Since scaling doesn't eliminate jittering artifacts, let's look at some approaches that do, either by using the CPU's 64-bit double precision or emulating it on the GPU.

5.2 Rendering Relative to Center

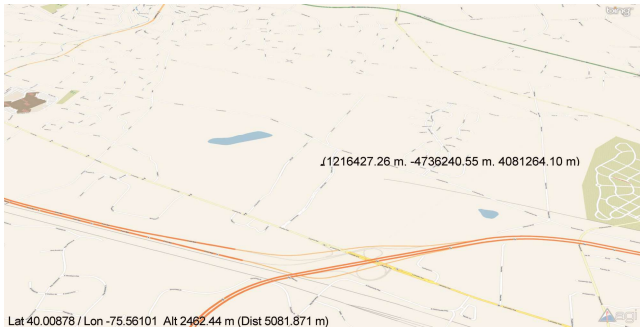
One approach to eliminating jitter is to render an object's position relative to its center (RTC) in eye coordinates. For reasonably sized objects, this results in vertex positions and model-view-projection matrix values small enough for 32-bit floating-point precision, thus, eliminating jitter.

Consider vector data for an area on the globe, perhaps the boundary for a zip code or city. These data are typically provided in geographic coordinates and converted to WGS84 coordinates for rendering (see Section 2.3.1), but using such large positions results in noticeable jitter.

The first step in using RTC to eliminate jitter is to compute the center point of the vertex positions, $\text{center}_{\text{WGS84}}$, by determining the minimum and maximum x-, y-, and z-components of the positions and averaging them. See AxisAlignedBoundingBox and its `Center` property in OpenGlobe.

Next, subtract $\text{center}_{\text{WGS84}}$ from each position to translate the object from WGS84 to its own local coordinates with $\text{center}_{\text{WGS84}}$ as its origin. This subtraction is done using double precision on the CPU. The point $\text{center}_{\text{WGS84}}$ will contain large x-, y-, or z-components, but each position now has much smaller components than before because they are closer to $\text{center}_{\text{WGS84}}$ than the WGS84 origin.

If instead of using vector data, a 3D model, such as a building or bridge, were used, it would already be in this form: a center (or origin) in WGS84 coordinates with large components, and positions relative to the center



(a)

(b)

Figure 5.3. When the viewer zooms from (a) to (b), the WGS84 position of the object does not change, and its components remain too large for single precision, but the distance between the viewer and object becomes manageable for single precision. (Images taken using STK. Imagery © 2010 Microsoft Corporation and © 2010 NAVTEQ.)

manageable with 32-bit floating-point precision. The model would likely also have a different orientation than WGS84, but that's OK.

Having smaller vertex positions is half the solution; the other half is having smaller values in the translation (i.e., the fourth column) of **MVP**. This is achieved by transforming $\text{center}_{\text{WGS84}}$ to eye coordinates, $\text{center}_{\text{eye}}$, by multiplying $\text{center}_{\text{WGS84}}$ by **MV** on the CPU using double precision. As the viewer moves closer to the object's center, $\text{center}_{\text{eye}}$ becomes smaller because the distance between the viewer and the center decreases, as shown in Figure 5.3. Next, replace the x-, y-, and z-components in the fourth column of **MV** with $\text{center}_{\text{eye}}$, forming $\mathbf{MV_{RTC}}$:

$$\mathbf{MV_{RTC}} = \begin{pmatrix} MV_{00} & MV_{01} & MV_{02} & \text{center}_{\text{eye}}x \\ MV_{10} & MV_{11} & MV_{12} & \text{center}_{\text{eye}}y \\ MV_{20} & MV_{21} & MV_{22} & \text{center}_{\text{eye}}z \\ MV_{30} & MV_{31} & MV_{32} & MV_{33} \end{pmatrix}.$$

The final model-view-projection transform, $\mathbf{MVP_{RTC}}$, is formed by $\mathbf{P} * \mathbf{MV_{RTC}}$ on the CPU. This is then converted to 32-bit floating-point values, and used in the shader to transform positions relative to $\text{center}_{\text{eye}}$ to clip coordinates without jitter.

Example. Let's consider the point in our example scene, $p_{\text{WGS84}} = (6378137, 0, 0)$. We'll also pretend it is the only geometry, which makes $\text{center}_{\text{WGS84}} = (6378137, 0, 0)$. The point's position relative to center, p_{center}, is $p_{\text{WGS84}} - \text{center}_{\text{WGS84}}$, or $(6378137, 0, 0) - (6378137, 0, 0)$, which is $(0, 0, 0)$, a very manageable position for a 32-bit floating point number!

Using the same 800-m view distance as before, the model-view matrix, $\mathbf{MV}$, is

$$\mathbf{MV} = \mathbf{V} * \mathbf{M} = \begin{pmatrix} 0.78 & 0.63 & 0.00 & -4,946,218.10 \\ 0.20 & -0.25 & 0.95 & -1,304,368.35 \\ 0.60 & -0.73 & -0.32 & -3,810,548.19 \\ 0.00 & 0.00 & 0.00 & 1.00 \end{pmatrix}.$$

Before rendering the point, $\mathbf{MV}$ is used to compute $\text{center}_{\text{eye}}$:

$$\begin{aligned} \text{center}_{\text{eye}} &= \mathbf{MV} * \text{center}_{\text{WGS84}} \\ &= \begin{pmatrix} 0.78 & 0.63 & 0.00 & -4,946,218.10 \\ 0.20 & -0.25 & 0.95 & -1,304,368.35 \\ 0.60 & -0.73 & -0.32 & -3,810,548.19 \\ 0.00 & 0.00 & 0.00 & 1.00 \end{pmatrix} \begin{pmatrix} 6,378,137.0 \\ 0.0 \\ 0.0 \\ 1.0 \end{pmatrix} \\ &= \begin{pmatrix} 0.0 \\ 0.0 \\ -800.0 \\ 1.0 \end{pmatrix} \end{aligned} \tag{5.3}$$

Since the viewer is looking directly at the point in this case, $\text{center}_{\text{eye}}.xyz$ is $(0, 0, -800)$.[1] We can now create $\mathbf{MV_{RTC}}$ by replacing the large x-, y-, and z-components in the fourth column of $\mathbf{MV}$ with $\text{center}_{\text{eye}}$. The final model-view-perspective matrix is computed as $\mathbf{P} * \mathbf{MV_{RTC}}$. The vertex shader now deals with a manageable vertex position, $(0, 0, 0)$, and a manageable $\mathbf{MVP_{RTC}}$ fourth column, $(0, 0, -800, 1)^T$.

Implementation. The matrix $\mathbf{MVP_{RTC}}$ needs to be computed per object, or more precisely, per center if multiple objects share the same center. If the center changes or the viewer moves, $\mathbf{MVP_{RTC}}$ needs to be recomputed. This only requires a few lines of code, as shown in Listing 5.4. Recall that

[1] The multiplication in Equation (5.3) will not result in exactly this value because only two significant digits are shown here.

```
Matrix4D m = sceneState.ModelViewMatrix;
Vector4D centerEye = m * new Vector4D(_center, 1.0);
Matrix4D mv = new Matrix4D(
    m.Column0Row0, m.Column1Row0, m.Column2Row0, centerEye.X,
    m.Column0Row1, m.Column1Row1, m.Column2Row1, centerEye.Y,
    m.Column0Row2, m.Column1Row2, m.Column2Row2, centerEye.Z,
    m.Column0Row3, m.Column1Row3, m.Column2Row3, m.Column3Row3);

// Set shader uniform
_modelViewPerspectiveMatrixRelativeToCenter.Value =
    (sceneState.PerspectiveMatrix * mv).ToMatrix4F();
// .. draw call
```

Listing 5.4. Constructing the model-view-projection matrix for RTC.

the matrix multiply on the second line is using double precision. A complete RTC code example is in the RelativeToCenter class in Chapter05Jitter.

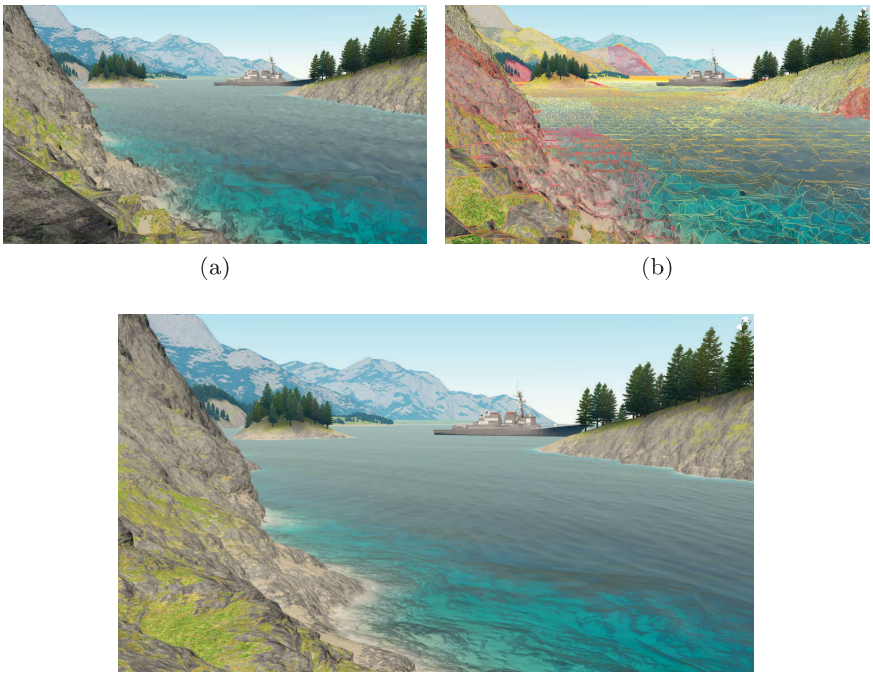
RTC eliminates jitter artifacts at the cost of requiring a custom model-view-perspective matrix per object (or per center). In addition, it requires a center be computed and positions defined relative to that center. This is not a problem for static data, but it can be extra CPU overhead for dynamic data that are changed every frame.

The real shortcoming of RTC is that it's not effective for large objects. RTC creates vertex positions small enough for 32-bit floating-point precision only when the positions are not too distant from the center. What happens if the object is large, and therefore, many positions are far from the center? Jittering will occur because vertex positions are too large for floating-point precision.

How large can an object be before jittering occurs? For 1 cm accuracy, Ohlarik recommends not exceeding a bounding radius of 131,071 m because 32-bit floating-point numbers generally have seven accurate decimal digits [126]. Many models fit into this size category, such as terrain tiles (as shown in Figure 5.4), buildings, and even cities, but many do not; consider vector data for states and countries, lines for satellite orbits, or in the extreme case, the plane in Figure 5.5.

Run Chapter05Jitter and zoom to 50 m or so, where rotating creates jitter. Switch to RTC and verify that jitter is eliminated. Zoom in to 2 m; the point and triangles start to jitter. Why? The point and triangles are rendered relative to a single center, but they are too large for RTC. The variable that determines the triangles' size, `triangleLength` in Jitter.`CreateAlgorithm`, is 200,000. Change this variable to a smaller value. Does jitter still occur using RTC?

○○○○ Try This



(a) (b)

(c)

Figure 5.4. (a) A scene with jitter. (b) Wireframe overlaid to highlight jitter. (c) The same scene with each terrain tile rendered using RTC. The "center" of a tile is the center of its parent in the terrain's quadtree. The regular structure of a quadtree makes it unnecessary to explicitly compute a center using each position. (Images courtesy of Brano Kemen, Outerra.)

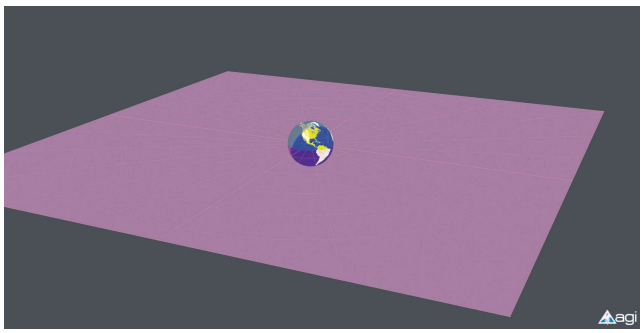


Figure 5.5. RTC does not eliminate jitter for large objects, such as this plane in space. (Image taken using STK. The Blue Marble imagery is from NASA Visible Earth.)

How then do we eliminate jitter for large objects? One approach is to split the object into multiple objects, each with a different center. Doing so hurts batching because each center requires a different $\mathbf{MVP_{RTC}}$ and, therefore, multiple draw calls. For some regular models, like terrain, this is natural, but for arbitrary models, it is usually not worth the trouble when there are other approaches such as rendering relative to eye.

5.3 Rendering Relative to Eye Using the CPU

To overcome the size limitations of RTC, objects can be rendered relative to eye (RTE). Both approaches utilize the CPU's double precision to ensure vertex positions and model-view-perspective matrix values are small enough for the GPU's single precision.

In RTE, each vertex is rendered relative to the eye. On the CPU, the model-view-perspective matrix is modified so the viewer is at the center of the scene, and each vertex position is translated so it is relative to the eye, not its original model coordinates. Similar to $\text{center}_{\text{eye}}$ in RTC, in RTE, vertex position components get smaller as the viewer approaches because the distance between the position and the viewer decreases.

Implementation. To implement RTE, first set the translation in $\mathbf{MV}$ to zero, so the scene is centered at the viewer:

$$\mathbf{MV_{RTE}} = \begin{pmatrix} MV_{00} & MV_{01} & MV_{02} & 0 \\ MV_{10} & MV_{11} & MV_{12} & 0 \\ MV_{20} & MV_{21} & MV_{22} & 0 \\ MV_{30} & MV_{31} & MV_{32} & MV_{33} \end{pmatrix}.$$

Unlike RTC, this matrix can be used for all objects in the same model coordinate system. Each time the viewer moves or a position is updated, position(s) relative to eye are written to a dynamic vertex buffer. A position in model coordinates is made RTE by subtracting off the eye position:

$$p_{\text{RTE}} = p_{\text{WGS84}} - \text{eye}_{\text{WGS84}}$$

This subtraction is done on the 64-bit CPU using double precision, then p_{RTE} is casted to a 32-bit float and written to a dynamic vertex buffer before a draw call is issued. There is a distinction between being relative to eye and being in eye coordinates. The former involves only a translation, no rotation.

Try This ○○◯◯

Run Chapter05Jitter and switch to CPU RTE. Does the scene ever jitter? Switch between CPU RTE and RTC. Given the large default size of the triangles (200,000 m), RTC results in jitter when zoomed very close, but CPU RTE does not.

Listing 5.5 shows the **MVP** and position updates used in CPU RTE. First, the model-view matrix is constructed with a zero translation so it is RTE. Then, the eye position, `eye`, is subtracted from each position and copied into a temporary array, which is then used to update the dynamic-position vertex buffer. The complete RTC code example is in the CPURelativeToEye class in Chapter05Jitter.

If just n positions change, only those n positions need to be written. If the viewer moves, all positions need to be written. Essentially, a previously static object, such as the border for a country, is now dynamic—it doesn't move in model coordinates, but it moves relative to the eye every time the eye moves!

Other vertex attributes for static objects can still be stored in a static vertex buffer; a dynamic vertex buffer is only required for the RTE positions. Both vertex buffers can be referenced from the same vertex-array object (see Section 3.5.3), and, therefore, used in the same draw call.

```
Matrix4D m = sceneState.ModelViewMatrix;
Matrix4D mv = new Matrix4D(
    m.Column0Row0, m.Column1Row0, m.Column2Row0, 0.0,
    m.Column0Row1, m.Column1Row1, m.Column2Row1, 0.0,
    m.Column0Row2, m.Column1Row2, m.Column2Row2, 0.0,
    m.Column0Row3, m.Column1Row3, m.Column2Row3, m.Column3Row3);

// Set shader uniform
_modelViewPerspectiveMatrixRelativeToEye.Value =
    (sceneState.PerspectiveMatrix * mv).ToMatrix4F();

for (int i = 0; i < _positions.Length; ++i)
{
  _positionsRelativeToEye[i] =
      (_positions[i] - eye).ToVector3F();
}

_positionBuffer.CopyFromSystemMemory(_positionsRelativeToEye);
// .. draw call

// ... elsewhere:
private readonly Vector3D[] _positions;
private readonly Vector3F[] _positionsRelativeToEye;
private readonly VertexBuffer _positionBuffer;
```

Listing 5.5. Setup for CPU RTE.

This RTE approach is called CPU RTE because the subtraction that transforms from model coordinates to RTE, $p_{\text{RTE}} = p_{\text{WGS84}} - \text{eye}_{\text{WGS84}}$, happens on the CPU. This eliminates jitter for all practical virtual globe purposes, resulting in excellent fidelity. The downside is performance: CPU RTE introduces CPU overhead, increases system bus traffic, and requires storing the original double-precision positions in system memory. This is especially poor for stationary objects that can no longer use static vertex buffers for positions.

CPU RTE takes advantage of the CPU's double precision at the cost of touching each position every time the viewer moves. Isn't the vertex shader a much better place to compute p_{RTE}? Using a vertex shader will allow us to take advantage of the incredible parallelism of the GPU (see Section 10.1.2), free up the CPU, reduce system bus traffic, and use static vertex buffers for static objects again. How can we utilize the vertex shader on 32-bit GPUs, given RTE relies on the 64-bit subtraction?

5.4 Rendering Relative to Eye Using the GPU

To overcome the performance shortcomings of CPU RTE, a double-precision position can be approximately encoded as two 32-bit floating-point positions and stored in two vertex attributes. This allows emulating the double-precision subtraction, $p_{\text{WGS84}} - \text{eye}_{\text{WGS84}}$, in a vertex shader. This approach is called GPU RTE because the subtraction happens on the GPU.

Implementation. Like CPU RTE, **MV** is first centered around the viewer by setting its translation to $(0, 0, 0)^T$. Two important implementation decisions are how to encode a double using two floats and how to emulate a double-precision subtraction using these values.

Using Ohlarik's implementation, near 1-cm accuracy can be achieved [126]. This is not as accurate as CPU RTE, which uses true double precision, but we have found it sufficient for virtual globes purposes. A double is encoded in a fixed-point representation using the fraction (mantissa) bits of two floats, `low` and `high`, using the function in Listing 5.6.

Ignoring the bit used to capture overflows, there are 23 fraction bits in an IEEE-754 single-precision floating-point value. In Ohlarik's scheme, when a double is encoded in two floats, `low` uses seven fraction bits to store the number's fractional value (i.e., the part after the decimal point). The remaining 16 bits represent an integer in the range $\left[0, 2^{16} - 1\right]$, or 0 to 65,535. All 23 fraction bits of `high` are used to represent numbers in the range $\left[2^{16}, (2^{23} - 1)(2^{16})\right]$, or 65,536 to 549,755,748,352, in increments

```
private static void DoubleToTwoFloats(
    double value, out float high, out float low)
{
  if (value >= 0.0)
  {
    double doubleHigh = Math.Floor(value / 65536.0) * 65536.0;
    high = (float)doubleHigh;
    low = (float)(value - doubleHigh);
  }
  else
  {
    double doubleHigh = Math.Floor(-value / 65536.0) * 65536.0;
    high = (float)-doubleHigh;
    low = (float)(value + doubleHigh);
  }
}
```

Listing 5.6. A method to approximately encode a double in two floats.

of 65,536. The eight exponent bits in each float remain unused to enable the use of floating-point addition and subtraction, as we will see shortly. This encoding is shown in Figure 5.6.

The largest representable whole number is 549,755,748,352 m, well above the Earth's equatorial radius of 6,378,137 m, which is a typical magnitude for positions near the surface. The seven fraction bits used in `low` store the fractional part in 2^{-7}, or 0.0078125, increments. Therefore, in one dimension, the error, due to roundoff, is less than 1 cm; in three dimensions, the maximum error is $\sqrt{0.0078125^2 + 0.0078125^2 + 0.0078125^2}$, or 1.353 cm, which is the accuracy for this GPU RTE approach.

Unlike traditional floating-point representations, where precision depends on the magnitude of the value, this fixed-point encoding has constant accuracy throughout. That is, it is accurate to 1.353 cm for values near zero, as well as for values near 500 billion.

The number of bits used to represent the fraction and integer parts can be adjusted to trade range for accuracy, or vice versa. For many virtual globe uses, such as GIS, 39 integer and seven fraction bits have significantly more range than required, but for other uses, such as space applications, the range is necessary.

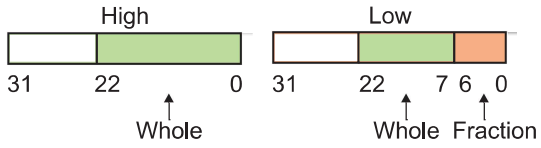


Figure 5.6. Encoding used for GPU RTE.

When a double-precision WGS84 position, p_{WGS84}, is encoded in two single precision values, p^{high} and p^{low}, and the WGS84 eye position, $\text{eye}_{\text{WGS84}}$, is encoded in eye^{high} and eye^{low}, $p_{\text{WGS84}} - \text{eye}_{\text{WGS84}}$ is computed as follows:

$$\begin{aligned} p_{\text{RTE}} &= p_{\text{WGS84}} - \text{eye}_{\text{WGS84}} \\ &\approx (p^{\text{high}} - \text{eye}^{\text{high}}) + (p^{\text{low}} - \text{eye}^{\text{low}}). \end{aligned}$$

When the viewer is distant, $p^{\text{high}} - \text{eye}^{\text{high}}$ dominates p_{RTE}; when the viewer is near, $p^{\text{high}} - \text{eye}^{\text{high}}$ is zero, and p_{RTE} is defined by $p^{\text{low}} - \text{eye}^{\text{low}}$.

On the CPU, the x-, y-, and z-components of each position are encoded in two floating-point values and written to a vertex buffer, storing two attributes per position. Unlike CPU RTE, this vertex buffer does not need to be updated if the viewer moves because $p_{\text{WGS84}} - \text{eye}_{\text{WGS84}}$ occurs in the vertex shader, as shown in Listing 5.7. In addition to the encoded position, the shader also requires a uniform for the encoded eye position. A complete code example is in GPURelativeToEye in Chapter05Jitter.

Although the vertex shader requires only an additional two subtractions and one addition, GPU RTE doubles the amount of vertex buffer memory required for positions. This doesn't necessarily double the memory requirements unless only positions are stored. Unlike CPU RTE, this approach does not require keeping positions in system memory, nor does it require the significant CPU overhead and system bus traffic. With GPU RTE, stationary objects can store positions in static vertex buffers since the CPU is no longer involved in updating each position.

The main drawbacks to GPU RTE are that jittering can still occur for extremely close views, and for some applications, the extra vertex buffer memory may be significant.

```
in vec3 positionHigh;
in vec3 positionLow;
uniform vec3 u_cameraEyeHigh;
uniform vec3 u_cameraEyeLow;
uniform mat4 u_modelViewPerspectiveMatrixRelativeToEye;

void main()
{
  vec3 highDifference = positionHigh - u_cameraEyeHigh;
  vec3 lowDifference = positionLow - u_cameraEyeLow;
  gl_Position = u_modelViewPerspectiveMatrixRelativeToEye *
                vec4(highDifference + lowDifference, 1.0);
}
```

Listing 5.7. The GPU RTE vertex shader.

Try This ○○◯◯

Run Chapter05Jitter and compare CPU RTE to GPU RTE. You will notice jitter with GPU RTE when zoomed in very close given its 1.353-cm accuracy. Recall that the triangles are separated by only 1 m, so when zoomed in to the point where the triangles are outside of the viewport, a pixel can cover less than 1.353 cm, and jitter will occur. How would you change `DoubleToTwoFloats` in Listing 5.6 to decrease jitter? What are the trade-offs?

5.4.1 Improving Precision with DSFUN90

For use cases that require extremely close views, the accuracy of GPU RTE can be improved by using a different encoding scheme. The DSFUN90 Fortran library[2] contains routines for performing double-single arithmetic, which allows approximately 15 decimal digits, based on the work of Knuth [92]. Empirically, we've found that porting this library's double-single subtraction to GLSL results in a GPU RTE implementation that can be used in place of CPU RTE. We call this approach GPU RTE DSFUN90.

A double is encoded in two floats using the method in Listing 5.8. The `high` component is simply the double value cast to a float; `low` is the error introduced by this cast in single precision.

The vertex shader that uses this encoding to perform the double-single subtraction on the GPU is shown in Listing 5.9. With eight subtractions and four additions, this vertex shader is more expensive than the one for stock GPU RTE, which requires only two subtractions and one addition. Given that memory bandwidth and vertex attribute setup have the same cost for both approaches, the extra instructions are not prohibitively expensive. The complete DSFUN90 example is in GPURelativeToEyeDSFUN90 in Chapter05Jitter.

Other double-single arithmetic operations from DSFUN90 have been ported to the GPU using CUDA.[3] It would be straightforward to port these to GLSL.

```
private static void DoubleToTwoFloats(
    double value, out float high, out float low)
{
  high = (float)value;
  low = (float)(value - high);
}
```

Listing 5.8. Encoding a double in two floats for GPU RTE DSFUN90.

[2] http://crd.lbl.gov/~dhbailey/mpdist/
[3] http://sites.virtualglobebook.com/virtual-globe-book/files/dsmath.h

```
in vec3 positionHigh;
in vec3 positionLow;
uniform vec3 u_cameraEyeHigh;
uniform vec3 u_cameraEyeLow;
uniform mat4 u_modelViewPerspectiveMatrixRelativeToEye;

void main()
{
  vec3 t1 = positionLow - u_cameraEyeLow;
  vec3 e = t1 - positionLow;
  vec3 t2 = ((-u_cameraEyeLow - e) + (positionLow - (t1 - e))) +
            positionHigh - u_cameraEyeHigh;
  vec3 highDifference = t1 + t2;
  vec3 lowDifference = t2 - (highDifference - t1);

  gl_Position = u_modelViewPerspectiveMatrixRelativeToEye *
                vec4(highDifference + lowDifference, 1.0);
}
```

Listing 5.9. The GPU RTE DSFUN90 vertex shader.

Run Chapter05Jitter and compare GPU RTE to GPU RTE DSFUN90. Zoom in until GPU RTE starts to jitter, then switch to DSFUN90. When would you use stock GPU RTE over DSFUN90? How does CPU RTE compare to DSFUN90? When would you use CPU RTE over DSFUN90?

Try This

5.4.2 Precision LOD

In addition to increasing precision, GPU RTE DSFUN90 lends itself nicely to precision level of detail (LOD). Take a second look at how a double-precision value is encoded into two single-precision values:

```
double value = // ...
float high = (float)value;
float low = (float)(value - high);
```

The assignment to `high` casts the value to a float, which is the same cast used if the object were naïvely rendered relative to world (RTW) using an unmodified **MVP**, without any concern for jitter. This only requires a single 32-bit vertex attribute but produces jitter when zoomed in close, as explained in Section 5.1.2. A hybrid approach that uses RTW when zoomed out and DSFUN90 when zoomed in can have the best of both algorithms: high precision when zoomed in and low memory usage

when zoomed out—and no jittering anywhere. Similar to how geometric LOD changes geometry based on view parameters, precision LOD changes precision based on the view.

Both RTW and DSFUN90 share the vertex buffer used to store `high`, but only DSFUN90 requires the vertex buffer storing `low`. When the viewer is zoomed out, the high vertex buffer is used with the RTW shader, taking advantage of all the GPU memory savings and performance benefits of RTW. When the viewer zooms in, both the high and low vertex buffers are used with the DSFUN90 shader to render with full precision.

Both vertex buffers can be created at the same time, or the low vertex buffer can be created on demand to reduce the amount of GPU memory used, not just reduce the attribute setup cost and vertex-shader complexity. For most scenes, the viewer is only close to a small percentage of objects, so creating the vertex buffer on demand can be very beneficial. The downside is that the original double-precision positions, or at least the floating-point low part, need to be kept in system memory, or paged back in to create the low vertex buffer. Using separate vertex buffers, instead of interleaving high and low components in the same vertex buffer, can cause a slight performance penalty when both components are called for (see Section 3.5.1).

Precision LOD is a form of discrete LOD with just two discrete levels: low and high. As the viewer zooms in, the LOD level should switch to high before jitter becomes noticeable. This viewer distance should be determined by the position's maximum component, the viewer position's maximum component, and the field of view. This is similar to determining screen-space error.

Saving memory is always a good thing in a world where size is speed, but it is important to put into perspective how much memory can be saved with precision LOD. First, most scenes use significantly more memory for textures than for vertex data. Some virtual globe scenes, however, especially those heavy with vector data, can use a significant amount of memory for vertices. Next, for distant objects, precision LOD cuts the amount of memory required for positions in half, but this does not mean the amount of vertex data is also cut in half. Vertex data may also include attributes such as normals, texture coordinates, etc. If the shading is also LODed, some attributes may not be necessary for simplified shaders (e.g., a simplified lighting model may not require tangent vectors).

As hinted above, precision LOD can be combined with geometric and shader LOD. As the viewer zooms out, the number of vertices rendered, the size of each vertex, and the shading complexity may all be reduced. When shading complexity is reduced, it may also be beneficial to reduce the memory usage of other vertex attributes (e.g., replacing floating-point normals with half-floats).

The complete precision LOD example is in SceneGPURelativeToEyeLOD in Chapter05Jitter.

Run Chapter05Jitter and compare GPU RTE DSFUN90 to precision LOD. At what distant interval does precision LOD jitter? Why? What needs to be adjusted to eliminate jitter in this example?

Try This

5.5 Recommendations

Based on the trade-offs in Table 5.1, our recommendations are as follows:

- Use GPU RTE DSFUN90 for all objects with positions defined in WGS84 coordinates if you can afford the extra vertex memory and vertex-shader instructions. The increase in vertex-buffer memory usage may be insignificant compared to the memory used by textures. Likewise, the increase in vertex-shader complexity may not be significant compared to fragment shading, which is likely to be the case in games with complex fragment shaders. This results in the simplest software design and excellent precision.
- Use RTC for objects defined in their own coordinate system (e.g., a model for a building) with a bounding radius less than 131,071 m.
- For better performance and memory usage, consider a hybrid approach:
  - Use CPU RTE for highly dynamic geometry. If the geometry is changing frequently as the viewer moves, the additional CPU overhead is not significant. Using CPU RTE reduces the vertex memory requirements and reduces system bus traffic in this case.
  - Use RTC for all static geometry with a bounding radius less than 131,071 m. Although this requires a separate model matrix per object, each object only requires 32-bit positions, which do not need to be shadowed in system memory.
  - Use GPU RTE DSFUN90 for all static geometry with a bounding radius greater than 131,071 m. If using RTC results in many small batches due to the need to set the model matrix (or model-view matrix) uniform, use GPU RTE DSFUN90 to increase the batch size.

Algorithm	Strengths	Weaknesses
RTW (relative to world)	Simple, obvious, several objects can be rendered with the same model matrix, and only uses 32-bit positions.	Results in jitter on 32-bit GPUs when zoomed in, and even a few hundred meters away. Scaling the coordinates down can reduce, but not eliminate, jitter.
RTC (relative to center)	Natural for some geometry, like models, results in good precision for reasonably sized geometry, and only uses 32-bit positions.	Large geometry still jitters, but less so than with RTW. Each object requires a different model-view matrix, making it heavy on uniform updates and less large-batch friendly. Slightly unnatural for geometry typically defined in world coordinates like vector data.
CPU RTE (relative to eye)	Excellent precision, several objects can be rendered with the same model-view matrix, only uses 32-bit positions on the GPU, a handful of dynamic vertex buffers can be used instead of many static buffers, appropriate for dynamic geometry.	Significant CPU overhead hurts performance because the CPU needs to touch every position whenever the camera moves. Static geometry essentially becomes dynamic because of camera movement, disabling the use of static vertex buffers for positions. The only approach that requires keeping positions in system memory, along with scratch memory for translating them relative to the eye.
GPU RTE	Very good precision, even for large geometry, until zoomed in to a couple of meters; very little CPU overhead; several objects can be rendered with the same model-view matrix, appropriate for static geometry.	Jittering still occurs when zoomed in very close. A position requires two 32-bit components in a vertex buffer. Requires two subtractions and an addition in the vertex shader.
GPU RTE DSFUN90	Similar to GPU RTE, only with no observed jitter; simpler CPU code than GPU RTE; works well with precision LOD.	Without LOD, a position requires two 32-bit components in a vertex buffer, and the vertex shader requires several instructions: eight subtractions and four additions.

Table 5.1. Summary of trade-offs between approaches to reducing jitter.

- It is reasonable to use GPU RTE instead of GPU RTE DSFUN90 if precision LOD will not be used, and the extra precision of the latter is not required.

Patrick Says

When I joined AGI in 2004, most 3D code in STK used CPU RTE. Notable exceptions included models and terrain chunks, which were small enough to use RTC. Although CPU RTE doesn't sound ideal, a lot of geometry was dynamic, and CPU RTE actually made sense when the code was first written, before vertex processing was done on the GPU!

In 2008, while working on Insight3D, we reevaluated jitter. By this time, GPUs changed dramatically: they included fast programmable vertex shaders, and in many cases, the CPU couldn't feed the GPU quickly enough. We wanted to find a way to take advantage of shaders to stay off the CPU; then my colleague came up with GPU RTE. I couldn't have been happier, but we still thought we could do better.

For moderately-sized geometry, RTC can render with as much precision as GPU RTE, using half the amount of memory for positions. With the well-intentioned goal of saving memory, we developed a hybrid solution (simplified):

- If geometry is dynamic, use GPU RTE.
- If geometry is static with a boundary radius less than 131,071 m, use RTC.
- If geometry is static and greater than 131,071 m, divide the geometry in several meshes and render each using RTC. If some heuristic determines that this results in too many or too few batches, combine batches and render them with GPU RTE. Large triangles and lines were also rendered with GPU RTE, instead of subdividing them and creating multiple RTC batches.

We nicely abstracted this away and hooked up lots of code to use it. It turns out we had a lot of geometry that fit into the last category, and the divide step was problematic and slow. In addition, writing a vertex shader was a pain because the abstraction sometimes used GPU RTE, RTC, or both. Over time, we wound up replacing the abstraction with GPU RTE in many cases; we favored the simplicity over the memory savings of RTC.

Massive-world games also need to eliminate jitter but usually have the luxury of restricting the speed of the viewer, allowing additional techniques. For example, Microsoft Flight Simulator uses a technique similar to CPU

RTE [163]. The origin is not always at the eye; instead, it is only periodically updated to keep it within a few kilometers of the eye. Given the speed of the plane, the origin changes infrequently, not every time the viewer moves. When the origin changes, the updated vertices are double buffered because the update can take several frames. Double buffering has memory costs, so the full vertex memory savings of CPU RTE are not realized.

Flight Simulator's approach is effective because the viewer's speed is bound; in a virtual globe, the viewer could be somewhere completely different several frames later—the viewer can be zoomed out in space looking at the entire Earth one second, then zoomed in to a few meters, looking at high-resolution imagery the next second.

5.6 Resources

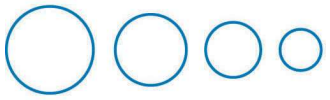
A solid understanding of floating point is useful for more than just transforming vertices. Calculations with large floating-point values are used everywhere in virtual globes. Thankfully, there are many resources on the topic, including Bush's example-filled article [23], Hecker's article in *Game Developer Magazine* [67], Ericson's slides on numerical robustness [46], and a chapter in Van Verth and Bishop's book on mathematics for games [176].

Jittering solutions have received a good bit of attention. Thorne's PhD thesis describes jittering and the continuous floating origin in depth [169]. Ohlarik's article introduces GPU RTE and compares it to other approaches [126].

Thall surveys strategies for double and quad precision on 32-bit GPUs and provides Cg implementations for arithmetic, exponential, and trigonometric functions [167]. The subtraction function can be used for implementing GPU RTE.

Forsyth recommends using RTC with fixed-point positions, instead of double-precision floating point, for better performance, constant precision, and a full 64 bits of precision [51]. The encoding scheme for GPU RTE introduced in Section 5.4 is a fixed-point representation embedded in the fractional bits of two single-precision floats, so a full 64 bits of precision is not achieved.

In addition to virtual globes, games with large worlds also have to combat jitter. Persson describes the technique used in "Just Cause 2" [140].



6 Depth Buffer Precision

At some point in their career, every graphics developer asks, "What should I set the near and far plane distances to?" Ambitious developers may even ask, "Why don't I just set the near plane to a really small value and the far plane to a really huge value?" When they see the dramatic impact of this, the surprised look on their faces is priceless!

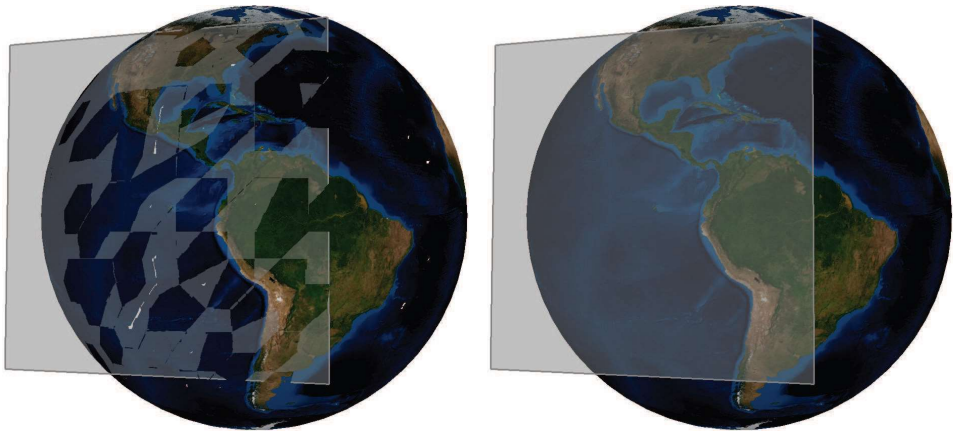
We too were once baffled by visual artifacts, such as those shown in Figure 6.1(a). These artifacts are due to *z-fighting*—the depth buffer cannot tell which object is in front of which. Worst yet, the artifacts tend to flicker as the viewer moves around.

When we first experienced z-fighting, the near and far plane distances and their impact on depth buffer precision was the last place we thought to look.[1] Thankfully, there are good reasons for this behavior and several techniques for improving precision, thus reducing or completely eliminating artifacts. This chapter describes such techniques and their trade-offs, ranging from simple one liners to system-wide changes.

6.1 Causes of Depth Buffer Errors

Virtual globes are unique in that objects very close to the viewer and objects very far from the viewer are present in most scenes. Other 3D applications, such as first-person shooters in indoor environments, have a well-defined upper bound on view distance. In virtual globes and open-world games, the viewer can go anywhere and look in any direction—fun for them, not for us. The viewer may be zoomed in close to a model with

[1] Z-fighting can also occur when rendering coplanar geometry (e.g., two overlapping triangles lying on the same plane). This chapter does not consider this cause of z-fighting; instead, it focuses on z-fighting due to large worlds.



(a) (b)

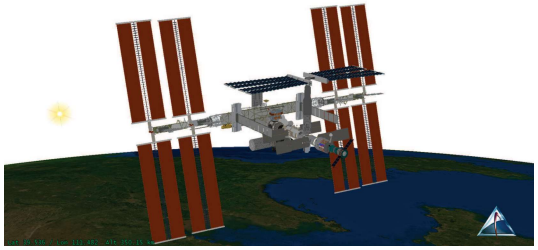
Figure 6.1. (a) A near-plane distance of 1 m creates artifacts due to limited depth precision for distant objects. (b) Moving the near plane out to 35 m eliminates the artifacts, at least for this view. In both cases, the gray plane is 1,000,000 m above Earth at its tangent point, and the far plane is set to a distant 27,000,000 m.

mountains in the distance, as shown in Figure 6.2(a), or the viewer may be zoomed into a satellite, with planets in the distance, as in Figure 6.2(c). Even a scene with just Earth requires a very large view distance, given Earth's semiminor axis of 6,356,752.3142 m (see Section 2.2.1).

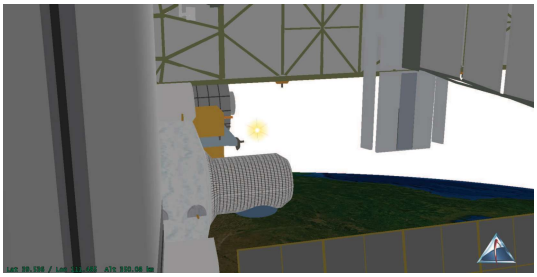
Such views are difficult because they call for a close near plane and a very distant far plane. For orthographic projections, this usually isn't a concern because the relationship between eye space z, z_{eye}, and window space z, z_{window}, is linear. In perspective projections, this relationship is nonlinear [2,11]. Objects near the viewer have small z_{eye} values, with plenty of potential z_{window} values to map to, allowing the depth buffer to resolve visibility correctly. The farther away an object, the larger its z_{eye}. Due to the nonlinear relationship, these z_{eye} values have much less z_{window} values to map to, so the depth buffer has trouble resolving visibility of distant objects close to each other, which creates z-fighting artifacts like those in Figure 6.1(a).

Close objects (small z_{eye}) have lots of depth buffer precision (lots of potential z_{window}). The farther away an object, the less precision it has. This nonlinear relationship between z_{eye} and z_{window} is controlled by the ratio of the far plane, f, to the near plane, n. The greater the $\frac{f}{n}$ ratio, the greater the nonlinearity [2].

(a)



(b)



(c)

Figure 6.2. (a) It is challenging to view both the nearby model and distant terrain in the same scene without depth buffer artifacts. This figure was rendered using a logarithmic depth buffer, discussed in Section 6.4, to eliminate the artifacts. Image courtesy of Brano Kemen, Outerra. (b) The International Space Station (ISS) is shown orbiting Earth, with its solar panels pointed towards the sun. (c) As the viewer zooms towards the ISS, the lack of depth buffer precision in the distance needs to be accounted for. (Images in (b) and (c) were taken using STK, which uses multifrustum rendering to eliminate z-fighting, discussed in Section 6.5.)

The reason for this behavior requires a closer look at the perspective-projection matrix. The OpenGL version of this matrix is

$$\begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

The parameters (l, r, b, t, n, f) define a view frustum.[2] The variables l and r define left and right clipping planes, b and t define bottom and top clipping planes, and n and f define near and far clipping planes (see

[2]These are the same parameters passed to the now deprecated `glFrustum` function.

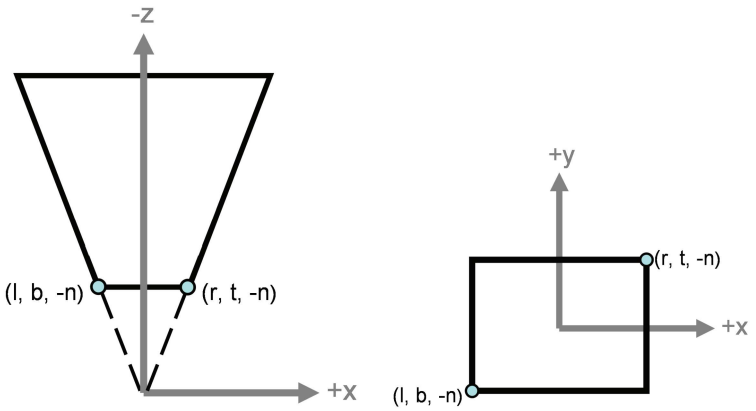


Figure 6.3. A frustum defined by parameters (l, r, b, t, n, f) used for perspective projection. (a) A frustum in the xz-plane. (b) A frustum in the xy-plane.

Figure 6.3). With the eye located at $(0, 0, 0)$, the corner points $(l, b, -n)$ and $(r, t, -n)$ are mapped to the lower left and upper right corners of the window.

Let's see how the perspective projection and subsequent transformations determine the final depth value, z_{window}, for a point in eye coordinates, $p_{\text{eye}} = (x_{\text{eye}}, y_{\text{eye}}, z_{\text{eye}})$. The perspective-projection matrix transforms p_{eye} to clip coordinates:

$$p_{\text{clip}} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_{\text{eye}} \\ y_{\text{eye}} \\ z_{\text{eye}} \\ 1 \end{pmatrix} = \begin{pmatrix} x_{\text{clip}} \\ y_{\text{clip}} \\ -\left(\frac{f+n}{f-n} z_{\text{eye}} + \frac{2fn}{f-n}\right) \\ -z_{\text{eye}} \end{pmatrix}.$$

The next transform, perspective division, divides each p_{clip} component by w_{clip} yielding normalized device coordinates:

$$p_{\text{ndc}} = \begin{pmatrix} \frac{x_{\text{clip}}}{-z_{\text{eye}}} \\ \frac{y_{\text{clip}}}{-z_{\text{eye}}} \\ \frac{-\left(\frac{f+n}{f-n} z_{\text{eye}} + \frac{2fn}{f-n}\right)}{-z_{\text{eye}}} \\ \frac{-z_{\text{eye}}}{-z_{\text{eye}}} \end{pmatrix} = \begin{pmatrix} x_{\text{ndc}} \\ y_{\text{ndc}} \\ \frac{\left(\frac{f+n}{f-n} z_{\text{eye}} + \frac{2fn}{f-n}\right)}{z_{\text{eye}}} \\ 1 \end{pmatrix}. \tag{6.1}$$

Hence,

$$z_{\text{ndc}} = \frac{\left(\frac{f+n}{f-n} z_{\text{eye}} + \frac{2fn}{f-n}\right)}{z_{\text{eye}}},$$
$$= \frac{f+n}{f-n} + \frac{2fn}{z_{\text{eye}}(f-n)}.$$

In normalized device coordinates, the scene is in the axis-aligned cube with corners $(-1,-1,-1)$ and $(1,1,1)$; therefore, z_{ndc} is in the range $[-1,1]$. The final transformation, the viewport transformation, maps z_{ndc} to z_{window}, where the range of z_{window} is defined by the application using `glDepthRange`(n_{range}, f_{range}):

$$z_{\text{window}} = z_{\text{ndc}} \left(\frac{f_{\text{range}} - n_{\text{range}}}{2}\right) + \frac{n_{\text{range}} + f_{\text{range}}}{2}$$
$$= \left(\frac{f+n}{f-n} + \frac{2fn}{z_{\text{eye}}(f-n)}\right)\left(\frac{f_{\text{range}} - n_{\text{range}}}{2}\right) + \frac{n_{\text{range}} + f_{\text{range}}}{2}.$$

In the common case of $n_{\text{range}} = 0$ and $f_{\text{range}} = 1$, z_{window} simplifies to

$$\begin{aligned} z_{\text{window}} &= \left(\frac{f+n}{f-n} + \frac{2fn}{z_{\text{eye}}(f-n)}\right)\left(\frac{1}{2}\right) + \frac{1}{2} \\ &= \frac{\left(\frac{f+n}{f-n} + \frac{2fn}{z_{\text{eye}}(f-n)}\right) + 1}{2}. \end{aligned} \tag{6.2}$$

For a given perspective projection, everything on the right-hand side of Equation (6.2) is constant except for z_{eye}; therefore,

$$z_{\text{window}} \propto \frac{1}{z_{\text{eye}}}.$$

This relationship is the result of the perspective divide from Equation (6.1). This division causes perspective foreshortening, making objects in the distance appear smaller and parallel lines converge in the distance. Dividing by z_{eye} makes sense for x_{clip} and y_{clip}; when z_{eye} is big, objects in the distance are scaled down. At the same time, z_{clip} is divided, making it also inversely proportional to z_{eye}.

Consider Figure 6.4: when z_{eye} is small, $\frac{1}{z_{\text{eye}}}$ is a quickly moving function; hence, a small change in z_{eye} will yield a large enough change in z_{window} that the depth buffer can resolve depth correctly. As z_{eye} gets large, $\frac{1}{z_{\text{eye}}}$ becomes a slow-moving function; therefore, a small change in z_{eye} may yield the same z_{window}, creating rendering artifacts.

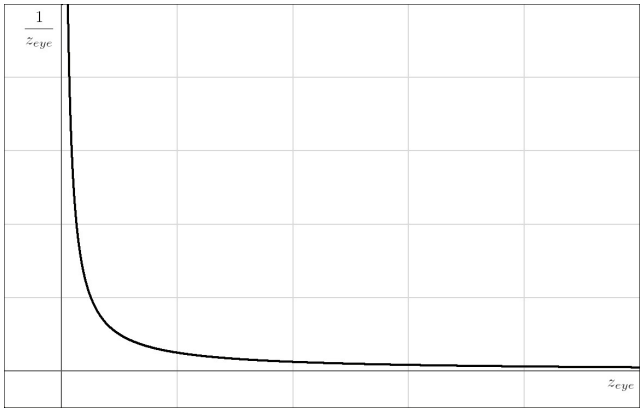


Figure 6.4. Given $z_{\text{window}} \propto \frac{1}{z_{\text{eye}}}$, a small change in z_{eye} when z_{eye} is small results in a large change to z_{window}. When z_{eye} is large, a small change in z_{eye} only very slightly changes $\frac{1}{z_{\text{eye}}}$, creating z-fighting artifacts.

The relationship between z_{eye} and z_{window} is also dependent on f and n. Figure 6.5 shows Equation (6.2) graphed for various values of n. Small values of n cram all the depth buffer precision at the near plane, larger values of n spread precision more evenly throughout. This is shown in the shape of each curve.

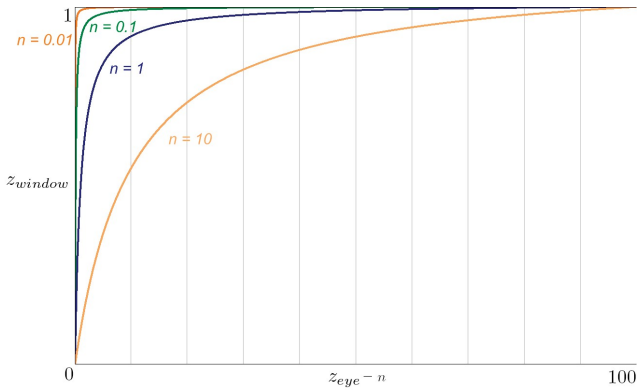


Figure 6.5. The relationship between z_{eye} and z_{window} for various values of near-plane distance, n, and a fixed-view distance, $f - n$, of 100. The x-axis shows the distance from the near plane. As n gets large, precision is spread more evenly throughout.

To see the effects of moving the near plane, run Chapter06DepthBufferPrecision. Move the near plane out until z-fighting between the gray plane and Earth is eliminated. Then move the gray plane closer to Earth. The artifacts occur again because the world-space separation between the objects is not large enough to create unique window space z-values. Push the near plane out even farther to eliminate the artifacts.

Try This

6.1.1 Minimum Triangle Separation

The minimum triangle separation, S_{min}, for a distance from the eye, z_{eye}, is the minimum world-space separation between two triangles required for correct depth occlusion. This is also referred to as the resolution of the depth buffer at the given distance. Based on our discussion thus far, it is easy to see that farther objects require a larger S_{min}. For an x-bit fixed-point depth buffer where the distance to the near plane, n, is much smaller than the distance to the far plane, f, Baker [11] provides an approximation to S_{min}:

$$S_{\text{min}} \approx \frac{z_{\text{eye}}^2}{2^x n - z_{\text{eye}}}.$$

Figure 6.6 shows S_{min} graphed for various values of n. As expected, larger values of n result in a flatter curve. These lower values of S_{min} allow objects to be closer together without creating rendering artifacts.

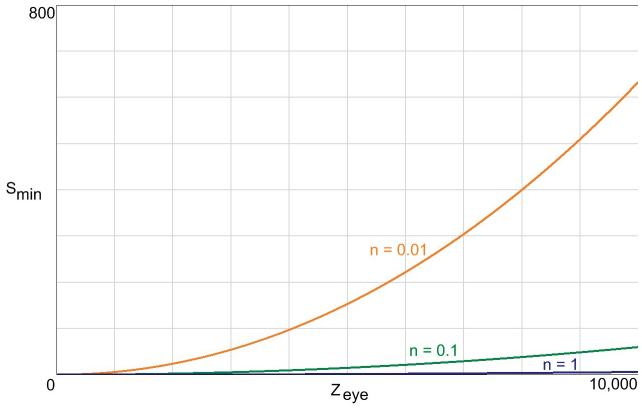


Figure 6.6. Approximate minimum triangle separation as a function of distance for various near-plane values. Larger values of n result in a flatter curve and allow proper depth occlusion for objects close together.

Although this nonlinear behavior is the most commonly cited reason for depth buffer errors, Akeley and Su [1] show that window coordinate precision, field of view, and error accumulated by single-precision projection, viewport, and rasterization arithmetic significantly contribute to effective depth buffer resolution.

Window coordinates, x_{window} and y_{window}, are commonly stored on the GPU in a fixed-point representation with 4 to 12 fractional bits. Prior to rasterization, a vertex is shifted into this representation, changing its position and thus its rasterized depth value. For a given z_{eye}, S_{min} depends on x_{eye} and y_{eye}; S_{min} increases as x_{eye} and y_{eye} move away from the origin. The wider the field of view, the greater the increase in S_{min}. The result is that depth buffer errors, and thus z-fighting, are more likely to occur along the window edges and in scenes with a wider field of view.

Now that we have an understanding of the root cause of z-fighting, it is time to turn our attention to the solutions. We start with some simple rules of thumb, then move on to more involved techniques.

6.2 Basic Solutions

Fortunately, some of the simplest approaches to eliminating artifacts due to depth buffer precision are also the most effective. Pushing out the near plane as far as possible ensures the best precision for distant objects. Also, the near plane doesn't need to be the same from frame to frame. For example, if the viewer zooms out to view Earth from space, the near plane may be pushed out dynamically. Of course, pushing out the near plane can create new rendering artifacts; objects are either entirely clipped by the near plane or the viewer can see "inside" partially clipped objects. This can be mitigated by using blending to fade out objects as they approach the near plane.

Likewise, pulling in the far plane as close as possible can help precision. Given that the far to near ratio, $\frac{f}{n}$, affects precision, pound for pound, moving the far plane in is less effective than moving the near plane out. For example, given an initial $n = 1$ and $f = 1{,}000$, the far plane would have to be moved in 500 m to get the same ratio as moving the near plane out 1 m. Nonetheless, minimizing the far-plane distance is still recommended. Objects can be gracefully faded out using blending or fog as they approach the far plane.

In addition to improving precision, setting the near plane and far plane as close as possible to each other can improve performance. More objects are likely to get culled on the CPU or at least clipped during the rendering pipeline, eliminating their rasterization and shading costs. A low $\frac{f}{n}$ ratio also helps GPU z-cull (also called hierarchical Z) optimizations. Z-cull tiles

store a low-resolution depth value, so the lower the $\frac{f}{n}$, the more likely a tile's depth value will have enough precision to be safely culled when the depth difference is small [136].

Another effective technique commonly used in virtual globes is to aggressively remove or fade out objects in the distance. For example, in a scene with Earth and vector data for major highways, when the viewer reaches a high altitude, the highways will fade out. Removing distant objects can also declutter the scene, especially if the scene contains lots of text and billboards (see Chapter 9), whose pixel size does not decrease with distance. This is similar to LOD, which usually reduces the geometric or shader complexity of an object based on its pixel size to improve performance.

6.3 Complementary Depth Buffering

OpenGL supports three depth buffer formats: 16-bit fixed point, 24-bit fixed point, and 32-bit floating point. Most of today's applications use a 24-bit fixed-point depth buffer because it is supported by a wide array of video cards. A property of fixed-point representation is that values are uniformly distributed, unlike floating-point representation, where values are nonuniformly distributed. Recall that representable floating-point values are close together near zero and spaced farther and farther apart as values move away from zero.

A technique called complementary depth buffering [66,94,137] takes advantage of a 32-bit floating-point depth buffer to compensate for nonlinear depth mapping. For normal depth buffering with $n_{\text{range}} = 0$ and $f_{\text{range}} = 1$, the clear depth is typically one, the depth comparison function is "less," and the distance to the near plane is less than the distance to the far plane, $n < f$. For complementary depth buffering with the same n_{range} and f_{range}, clear depth to zero, use a depth compare of "greater," and swap n and f when constructing the perspective-projection matrix (compare Listings 6.1 and 6.2).

```
ClearState clearState = new ClearState();
clearState.Depth = 1;
context.Clear(clearState);
sceneState.Camera.PerspectiveNearPlaneDistance = nearDistance;
sceneState.Camera.PerspectiveFarPlaneDistance = farDistance;
renderState.DepthTest.Function = DepthTestFunction.Less;
```

Listing 6.1. Normal depth buffering. The depth buffer is cleared to one, the near and far planes are set as usual, and the "less" depth comparison function is used.

```
ClearState clearState = new ClearState();
clearState.Depth = 0;
context.Clear(clearState);
sceneState.Camera.PerspectiveNearPlaneDistance = farDistance;
sceneState.Camera.PerspectiveFarPlaneDistance = nearDistance;
renderState.DepthTest.Function = DepthTestFunction.Greater;
```

Listing 6.2. Complementary depth buffering improves precision for distant objects. The depth buffer is cleared to zero, the near and far planes are swapped, and the "greater" depth comparison function is used.

How does this improve depth buffer precision for distant objects? Floating-point representations are nonlinear, just like depth values. As floating-point values move far away from zero, there are less discrete rep-

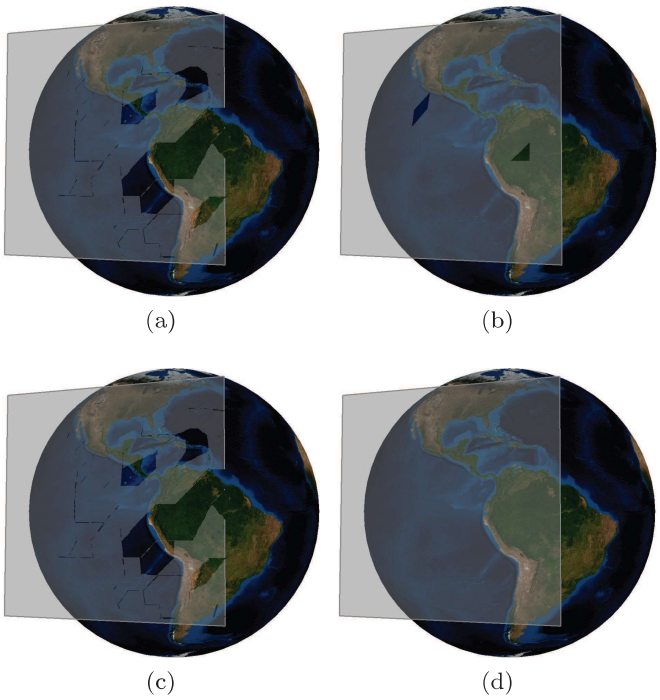


Figure 6.7. Complementary depth buffering with different depth buffer formats. The near plane is at 14 m. Similar to Figure 6.1, the gray plane is 1,000,000 m above Earth at its tangent point, and the far plane is set to a distant 27,000,000 m. In this scene, the difference between a (a) 24-bit fixed-point and (c) 32-bit floating-point buffer is unnoticeable. (a) and (b) Complementary depth buffering provides decent improvements when used with a 24-bit fixed-point buffer. (c) and (d) Complementary depth buffering is most effective when used with a 32-bit floating-point buffer.

resentations. By reversing the depth mapping, the highest floating-point precision is used at the far plane, where the lowest depth precision is found. Likewise, the lowest floating-point precision is used at the near plane, where the highest precision is found. These opposites balance each other out quite a bit. Persson recommends this technique even if a fixed-point buffer is used since it improves the precision during transformations [137]. Figure 6.7 shows complementary depth buffering with both fixed- and floating-point buffers.

Try This

To see this technique in action, run Chapter06DepthBufferPrecision. Swap between different depth buffer formats and toggle complementary depth buffering. The effectiveness of complementary depth buffering with a floating-point buffer can also be seen by running NVIDIA's simple depth float sample (see http://developer.download.nvidia.com/SDK/10.5/opengl/samples.html).

6.4 Logarithmic Depth Buffer

A logarithmic depth buffer improves depth buffer precision for distant objects by using a logarithmic distribution for z_{screen} [82]. It trades precision for close objects for precision for distant objects. It has a straightforward implementation, only requiring modification of z_{clip} in a shader using the following equation:[3]

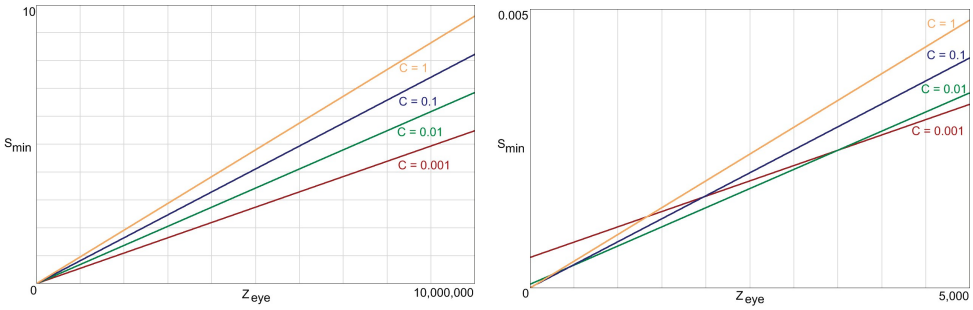
$$z_{\text{clip}} = \frac{2\ln(Cz_{\text{clip}} + 1)}{\ln(Cf + 1)} - 1. \tag{6.3}$$

As in Section 6.1, z_{clip} is the z-coordinate in clip space, the space in the pipeline immediately after the perspective-projection transformation but before perspective divide. The variable f is the distance to the far plane, and C is a constant that determines the resolution near the viewer.

Decreasing C increases precision in the distance but decreases precision near the viewer. Given C and an x-bit fixed-point depth buffer, the approximate minimum triangle separation, S_{min}, at distance z_{eye} is

$$S_{\text{min}} = \frac{\ln(Cf + 1)}{(2^x - 1)\frac{C}{Cz_{\text{eye}}+1}}.$$

[3] This equation uses OpenGL's normalized device coordinates, where $z_{\text{ndc}} \in [-1, 1]$. For Direct3D, where $z_{\text{ndc}} \in [0, 1]$, the equation is $\frac{\ln(Cz_{\text{clip}}+1)}{\ln(Cf+1)}$.

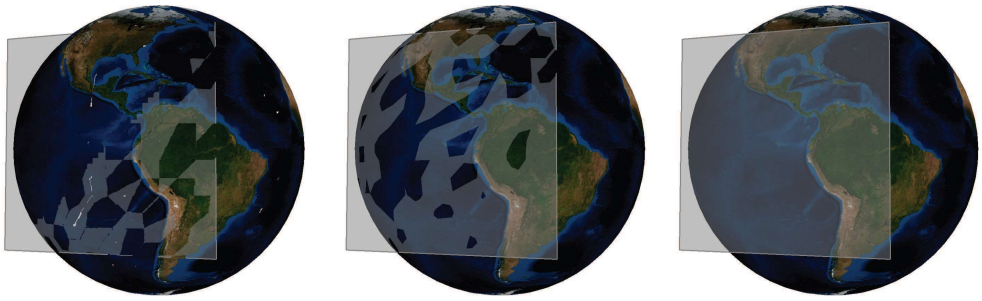


(a) (b)

Figure 6.8. Approximate minimum triangle separation as a function of distance for various C values using a logarithmic depth buffer. Lower C values result in (a) more precision (lower S_{min}) in the distance but (b) less precision up close. Compare these graphs, including scale, to normal depth buffering shown in Figure 6.6.

Figure 6.8 graphs various C values for a 24-bit fixed-point depth buffer and a distant far plane of 10,000,000 m. Figure 6.8(a) shows that lower values of C provide increased precision in the distance. Figure 6.8(b) shows the adverse effect low values of C have on precision near the viewer. This precision is tolerable for many scenes, considering that $C = 0.001$ yields $S_{\text{min}} = 5.49$ at the far plane while still yielding $S_{\text{min}} = 0.0005$ at 1 m.

A logarithmic depth buffer can produce excellent results, as shown in Figure 6.9.



(a) (b) (c)

Figure 6.9. A comparison of (a) normal depth buffering, (b) complementary depth buffering, and (c) a logarithmic depth buffer. A logarithmic depth buffer produces excellent results for distant objects. In this scene, the near plane is at 1 m and the far plane is at 27,000,000 m. The gray plane is 20,000 m above Earth at its tangent point.

```
in vec4 position;

uniform mat4 og_modelViewPerspectiveMatrix;
uniform float og_perspectiveFarPlaneDistance;
uniform bool u_logarithmicDepth;
uniform float u_logarithmicDepthConstant;

vec4 ModelToClipCoordinates(
    vec4 position,
    mat4 modelViewPerspectiveMatrix,
    bool logarithmicDepth,
    float logarithmicDepthConstant,
    float perspectiveFarPlaneDistance)
{
  vec4 clip = modelViewPerspectiveMatrix * position;

  if (logarithmicDepth)
  {
    clip.z =
        ((2.0 * log(logarithmicDepthConstant * clip.z + 1.0) /
          log(logarithmicDepthConstant *
              perspectiveFarPlaneDistance + 1.0)) - 1.0) *
        clip.w;
  }

  return clip;
}

void main()
{
  gl_Position = ModelToClipCoordinates(position,
      og_modelViewPerspectiveMatrix,
      u_logarithmicDepth, u_logarithmicDepthConstant,
      og_perspectiveFarPlaneDistance);
}
```

Listing 6.3. Vertex shader for a logarithmic depth buffer.

Run Chapter06DepthBufferPrecision and enable the logarithmic depth buffer. Move the gray plane in the scene closer to Earth. Compare the results to that of complementary depth buffering or simple modification of the near plane.

Try This

In a vertex-shader implementation, like the one shown in Listing 6.3, Equation (6.3) should be multiplied by w_{clip} to undo the perspective division that will follow. Using a vertex shader leads to an efficient implementation, especially considering that $\frac{2}{\ln(Cf+1)}$ is constant across all vertices and can, therefore, be made part of the projection matrix. But a vertex-shader implementation leads to visual artifacts for visible triangles with a

vertex behind the viewer. The artifacts are caused because the fixed function computes perspective-correct depth by linearly interpolating $\frac{z}{w}$ and $\frac{1}{w}$. Multiplying z_{clip} by w_{clip} in the vertex shader means that $\frac{z}{w}$ used for interpolation is really just z. To eliminate this problem, Brebion writes depth in a fragment shader using Equation (6.3) and z_{clip} passed from the vertex shader [21].

6.5 Rendering with Multiple Frustums

Multifrustum rendering, also called depth partitioning, fixes the depth buffer precision problem by capitalizing on the fact that each object does not need to be rendered with the same near and far planes. Given that a reasonable $\frac{f}{n}$ ratio for a 24-bit fixed-point depth buffer is 1,000 [2], only a handful of unique frustums are required to cover a large distance with adequate depth precision.

For example, near- and far-plane distances for a virtual globe application might be 1 m and 100,000,000 m. Given Earth's semimajor axis of 6,378,137 m, these distances allow the viewer to zoom out quite a bit to view the entire planet and perhaps satellites orbiting it. Using multifrustum rendering with an $\frac{f}{n}$ ratio of 1,000, the frustum closest to the viewer would have a near plane of 1 m and a far plane of 1,000 m. The next closest frustum would have a near plane of 1,000 m and a far plane of 1,000,000 m. The farthest frustum would have a near plane of 1,000,000 m

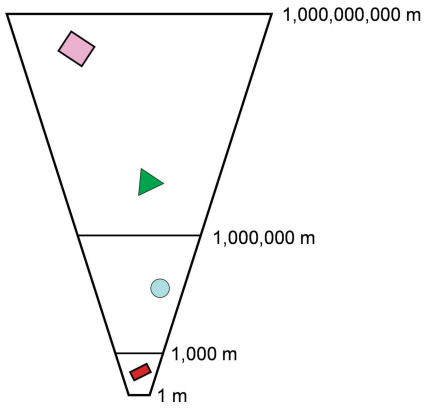


Figure 6.10. Multifrustum rendering with a near plane of 1 m, a far plane of 100,000,000 m, and an $\frac{f}{n}$ ratio of 1,000 (not drawn to scale). When the scene is rendered, first the purple and green objects are drawn, then the cyan object, and finally the red object. To avoid artifacts, the frustums should overlap slightly.

and a far plane of 100,000,000 m (or even 1,000,000,000 m). This is shown in Figure 6.10. As explained in Section 6.2, once the near plane is pushed out far enough, a single frustum covers a large distance with adequate depth precision.

The scene is rendered starting with objects overlapping the frustum farthest from the viewer, then continuing with the next farthest frustum, and so on, until the closest frustum is rendered. Before rendering objects overlapping a frustum, the perspective-projection matrix is computed using the frustum's near and far planes, and the depth buffer is cleared. Clearing the depth buffer ensures that objects do not lose the depth test against objects in a farther frustum; objects in a closer frustum should always overwrite objects in a further frustum because the frustums are sorted far to near.

Rendering a scene like this requires determining which objects are in which frustums. When the viewer moves, the object/frustum relationship will change, even if the objects do not move. One approach to efficiently handling this is to first perform hierarchical-view frustum culling against the union of individual frustums (i.e., a frustum with the near plane of the closest frustum and a far plane of the farthest frustum). Next, sort the objects by distance to the viewer. Finally, use this sorted list to quickly group objects into the correct frustum(s) and render them in front-to-back order within each frustum to aid hardware z-buffer optimizations.

To avoid tearing artifacts, adjacent frustums should have slight overlap. In our example, the middle frustum has the same near plane, 1,000 m, as the first frustum's far plane. To avoid gaps in the rendered image, the middle frustum's near plane should overlap slightly (e.g., be set to 990 m).

Patrick Says

STK uses multifrustum rendering. It defaults to a near plane of 1 m, a far plane of 100,000,000 m, and an $\frac{f}{n}$ ratio of 1,000. This allows users to zoom in very close on a satellite and still accurately see planets in the distance. STK tries to minimize the total number of frustums by pushing the near plane out as far as possible based on the viewed object. There is an advanced panel that allows users to tweak the near plane, far plane, and $\frac{f}{n}$ ratio, but thankfully, this is almost never required.

Multifrustum rendering allows virtually infinite depth precision and even runs on old video cards. It allows a large enough view distance that fog and translucency tricks from Section 6.2 are usually unnecessary.

Multifrustum rendering is not without its drawbacks, however. It makes reconstructing position from depth, which is useful in deferred shading, difficult. It can also have a significant impact on performance if not used carefully.

6.5.1 Performance Implications

Care needs to be taken to avoid dramatic impacts to performance. For many scenes in STK and Insight3D, three or four frustums are required. This makes hierarchical-view frustum culling key to avoid wasting time for objects that are culled by the near or far plane in the majority of the frustums. Simply iterating over every object during each frustum leads to a large amount of wasted CPU time on frustum checks.

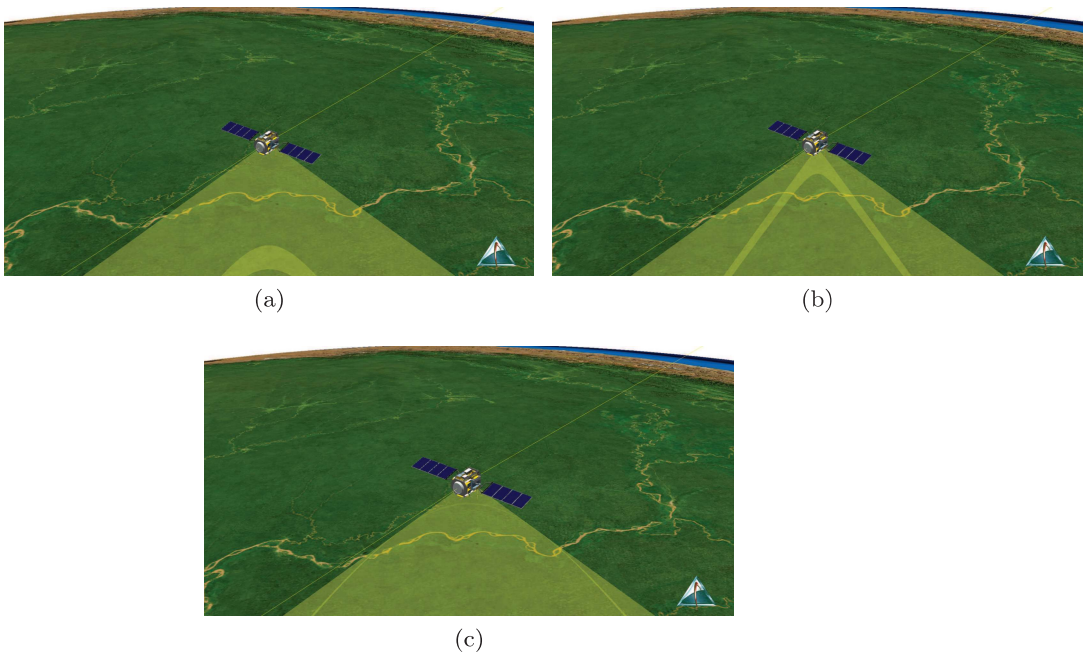
Other optimizations can be used in addition to, or instead of, hierarchical culling. In a first pass, objects can be culled by the left, bottom, right, and top planes. When each frustum is rendered, only the near and far planes need to be checked, improving the efficiency of the frustum check.

Another optimization exploits objects completely in a frustum. A data structure that allows efficient removal, such as a doubly linked list, is used to store objects to be rendered. The list is iterated over during each frustum. When an object is completely in a frustum, it is removed from the list since it will not be rendered in any other frustum. As each frustum is rendered, fewer and fewer objects are touched. Of course, the removed objects are added back before the next frame. This is particularly useful for dynamic objects, where maintaining a spatial data structure for hierarchical-view frustum culling can introduce more overhead than it saves.

The farther the near plane is from the viewer, the larger a frustum can be. Therefore, it is beneficial to move the near plane for the closest frustum as far out as possible to reduce the total number of frustums required.

Regardless of efficient culling, multifrustum rendering creates a performance and visual quality problem for objects straddling two or more frustums. In this case, the object must be rendered twice. This leads to a little overdraw at the fragment level, but it does cost extra CPU time, bus traffic, and vertex processing compared to objects only in a single frustum.

Multifrustum rendering also increases the tension between culling and batching. Batching many objects into a single draw call is important for performance [123,183]. Doing so typically leads to larger bounding volumes around objects. The larger a bounding volume, the more likely it is to overlap more than one frustum and, therefore, lead to objects rendered multiple times. This makes efficient use of batching harder to achieve. Hardware vendors started releasing extensions to reduce CPU overdraw for draw commands [124], which may help reduce the need for batching, but it obviously will not avoid application-specific per-object overhead.



(a) (b) (c)

Figure 6.11. Although multifrustum rendering eliminates occlusion artifacts, it can introduce artifacts for translucent objects straddling two or more overlapping frustums. In the above image sequence the double translucency appears to slide across the yellow satellite sensor as the viewer zooms.

Besides causing performance issues, rendering objects multiple times can lead to artifacts shown in Figure 6.11. Given the slight overlap of adjacent frustums, redundantly rendered translucent objects appear darker or lighter in the overlapped region since the objects are blended on top of each other. As the viewer zooms in and out, this band can appear to slide along the object.

Multifrustum rendering also impacts performance by reducing the effectiveness of early-z (see Section 12.4.5), which benefits from rendering objects in near-to-far order from the viewer. In multifrustum rendering, frustums are rendered in the opposite order—far to near—although objects in each individual frustum can still be rendered near to far.

The final performance consideration for multifrustum rendering is to avoid redundant computations. Since an object may be rendered multiple times, it is important to avoid expensive computations in an object's render method, and instead, move such operations to a method that is called less frequently, like an update method.

Patrick Says ○○◯◯

Although it solves an important problem, multifrustum rendering is a bit of a pain. Besides requiring careful attention to culling, it complicates the render loop and makes debugging harder since an object may be rendered more than once per frame. It even complicates less obvious code paths like picking.

6.6 W-Buffer

The w-buffer is an alternative to the z-buffer that can improve precision when resolving visibility for distant objects. The w-buffer uses w_{clip}, the original z_{eye}, to resolve visibility. This is linear in eye space and uniformly distributed between the near and far planes, within the constraints of a floating-point representation. Thus, compared to the z-buffer, the w-buffer has less precision close to the near plane but more precision moving towards the far plane. Table 6.1 compares the w-buffer to the z-buffer.

The problem is w is nonlinear in window space. This makes interpolating w for rasterization more expensive than interpolating z, which is linear in window space. Perspective-correct rasterization only has to calculate $\frac{1}{z}$ at each vertex and linearly interpolate across a triangle. Furthermore, z's linearity in window space improves hardware optimizations, including coarse-grained z-cull and z-compression, because the depth-value gradients are constant across a given triangle [137]. Today's hardware no longer implements w-buffers.

Buffer	Eye Space	Window Space
Z	Nonlinear	Linear
W	Linear	Nonlinear

Table 6.1. Relationship of the stored z- and w-buffer values to eye- and window-space values. The linear relationship of the w-buffer in eye space uniformly distributes precision through eye space at the cost of performance.

6.7 Algorithms Summary

Table 6.2 summarizes the major approaches for reducing or completely eliminating z-fighting caused by depth buffer precision errors. When de-

Algorithm	Summary
Adjust near/far plane	Easy to implement. Pushing out the near plane is more effective than pulling in the far plane. Use fog or blending to fade out objects before they are clipped.
Remove distant objects	Distant objects require a larger $S_{\min}$ to resolve visibility so removing distant objects that are near other objects can eliminate artifacts. This can also serve to declutter some scenes.
Complementary depth buffering	Easy to implement. Requires a floating-point depth buffer, which is available on most recent video cards, for maximum effectiveness. Doesn't account for truly massive view distances.
Logarithmic depth buffer	Highly effective for long view distances. Only requires a small vertex or fragment shader change, but all shaders need to be changed. Artifacts from using a vertex shader can be eliminated by using a fragment shader at the cost of losing hardware z-buffer optimizations.
Multifrustum rendering	Allows for arbitrarily long view distances and works on all video cards. Some objects may have to be rendered multiple times, which creates artifacts on translucent objects. A careful implementation is required for best performance.

Table 6.2. Summary of algorithms used to fix depth buffer precision errors.

ciding which approach to use, consider that many approaches can be combined. For example, complementary or logarithmic depth buffering can be used with multifrustum rendering to reduce the number of frustums required. Also, pushing out the near plane and pulling in the far plane will help in all cases.

Many of the approaches described in this chapter are implemented in Chapter06DepthBufferPrecision. Use this application to assist in selecting the approach that is best for you.

6.8 Resources

Solid coverage of transformations, including perspective projections, is found in "The Red Book" [155]. The "OpenGL Depth Buffer FAQ" contains excellent information on precision [110]. In addition, there are many worthwhile depth buffer precision articles on the web [11, 81, 82, 111, 137].

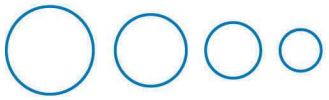
Another technique to overcome depth buffer precision errors is to use imposters, each rendered with their own projection matrix [130].

Reconstructing world- or view-space positions from depth is useful for deferred shading. The precision of the reconstructed position depends on

the type of depth stored and the format it was stored in. A comparison of these, with sample code, is provided by Pettineo [141].

Part III

Vector Data



7 Vector Data and Polylines

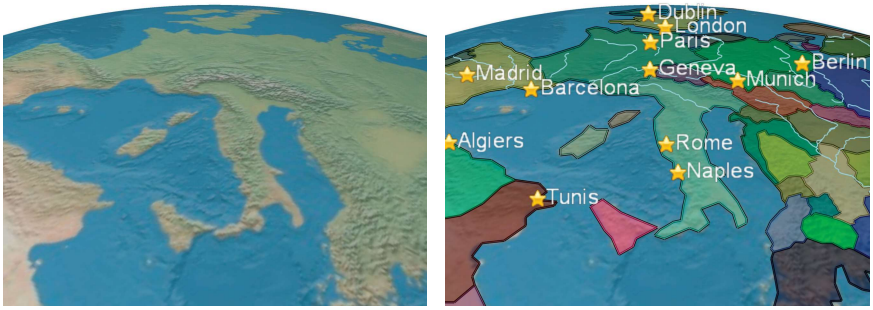
Virtual globes use two main types of data: raster and vector. Terrain and high-resolution imagery, which come in the form of raster data, create the breathtaking landscapes that attract so many of us to virtual globes. Vector data, on the other hand, may not have the same visual appeal but make virtual globes useful for "real work"—anything from urban planning to command and control applications.

Vector data come in the form of *polylines*, *polygons*, and *points*. Figure 7.1(b) shows examples of each; rivers are polylines, countries are polygons, major cities are points. When we compare Figure 7.1(a) to 7.1(b), it becomes clear how much richness vector data add to virtual globes.

This is the first of three chapters that discuss modern techniques for rendering polylines, polygons, and points on an ellipsoid globe. This discussion includes both rendering and preprocessing. For example, polygons need to be converted to triangles in a process called *triangulation*, and images for points should be combined into a single texture, called a *texture atlas*, in a process called *packing*. We suspect that you will be pleasantly surprised with all the interesting computer graphics techniques used to render vector data. On the surface, rendering polylines, polygons, and points sounds quite easy, but in reality, there are many things to consider.

7.1 Sources of Vector Data

Although our focus is on rendering algorithms, we should have an appreciation for the file formats used to store vector data. Perhaps the two most popular formats are Esri's shapefile [49] and the Open Geospatial Consortium (OGC) standard, KML [131].



(a) (b)

Figure 7.1. (a) A globe with just raster data. (b) The same globe also with vector data for rivers, countries, and major cities.

Shapefiles store nontopological geometry and attribute information for geospatial vector data, including line, area, and point features. Using our terminology, a shapefile's line feature is a polyline and an area feature is a polygon. The geometry for a feature is defined using a set of vector coordinates. A feature's attribute information is metadata, such as the city names shown in Figure 7.1(b).

A shapefile is actually composed of multiple files: the main `.shp` file containing the feature geometry, a `.shx` index file for efficient seeking, and a `.dbf` dBase IV file containing feature attributes. All features in the same shapefile must be of the same type (e.g., all polylines). The format is fully described in the *Esri Shapefile Technical Description* [49]. See `OpenGlobe.Core.`Shapefile for a class that can read point, polyline, and polygon shapefiles.

Another format, KML, formerly the Keyhole Markup Language, has gained widespread use in recent years due to the popularity of Google Earth. KML is an XML-based language for geographic visualization. It goes well beyond the geographic locations of polylines, polygons, and points to include such things as their graphical style, user navigation, and the ability to link to other KML files. The KML language is described in the KML 2.2 specification [131]. Google's open source library, *libkml*,[1] can be used to read and write KML files using C++, Java, or Python.

7.2 Combating Z-Fighting

Before getting to the fun stuff of actually rendering vector data, we need to consider the potential z-fighting artifacts discussed in Chapter 6. When

[1] http://code.google.com/p/libkml/

(a) (b)

Figure 7.2. (a) Artifacts due to z-fighting and vector data failing the depth test. (b) Rendering the ellipsoid using the average depth value of its front and back faces eliminates the artifacts.

rendering vector data on a globe, care needs to be taken to avoid z-fighting between the vector data and the globe. The problem is somewhat different than the usual coplanar triangles fighting to win the depth test. Consider a tessellated ellipsoid and a polyline on the ellipsoid. Since both only approximate the actual curvature of the ellipsoid, in some cases, a line segment may be coplanar with one of the ellipsoid triangles; in other cases, the line segment may actually cut underneath triangles. In the former case, z-fighting occurs; in the latter case, either z-fighting occurs or the polyline loses the depth test. Even if the ellipsoid is ray casted, a la Section 4.3, artifacts still occur (see Figure 7.2(a)).

A simple technique can eliminate the artifacts. When rendering the ellipsoid, instead of writing the depth values of its front-facing triangles, use

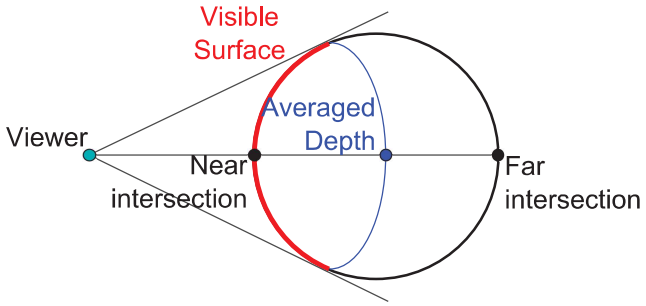


Figure 7.3. The averaged depth values of the front and back faces of the ellipsoid are shown in blue. Using these depth values, instead of the depth values of the front face, shown in red, can avoid z-fighting with vector data and still occlude objects on the other side of the ellipsoid.

the average of the depth values of its front-facing and back-facing triangles, as shown in Figure 7.3. This is easy to do when ray casting an ellipsoid in the fragment shader. The ray-ellipsoid intersection test needs to return both intersections:[2]

```
struct Intersection
{
  bool  Intersects;
  float NearTime; // Along ray
  float FarTime;  // Along ray
};
```

The second and final step is to change the way gl_FragDepth is assigned. When writing depth using the front face, the code looks like this:

```
vec3 position = og_cameraEye + (i.NearTime * rayDirection);
gl_FragDepth = ComputeWorldPositionDepth(position,
    og_modelZToClipCoordinates);
```

It can simply be replaced by averaging `i.NearTime` and `i.FarTime` to get the distance along the ray:

```
vec3 averagePosition = og_cameraEye +
    mix(i.NearTime, i.FarTime, 0.5) * rayDirection;
gl_FragDepth = ComputeWorldPositionDepth(averagePosition,
    og_modelZToClipCoordinates);
```

The lighting computations should still use the original `position`, since that is the actual position being lit. See the fragment shader in `OpenGlobe.Scene`.RayCastedGlobe for a full implementation.

When averaged depth is used, objects on the back face of the ellipsoid still fail the depth test, but objects slightly under the front face pass. This is great for polylines whose line segments cut slightly underneath the ellipsoid, as can be seen in the southern boundaries of Georgia and Alabama in Figures 7.2(a) and 7.2(b). For other objects such as models, this sleight of hand may not be desirable. This approach can be extended to use two depth buffers: one with averaged depth and one with the the ellipsoid's front-facing triangle's depth values. Objects that may cause artifacts with the ellipsoid can be rendered with the averaged depth buffer, and other objects can be rendered with the regular depth values.

[2]When the ray intersects the ellipsoid at its tangent, both `NearTime` and `FarTime` will be the same value.

We use a variation of this approach, called the *depth cone*, in STK. First, the ellipsoid is rendered with its regular depth values, followed by any objects that should be clipped by the front faces of the ellipsoid. Then, a depth cone is computed on the CPU. The cone's base is formed from tangent points on the ellipsoid, and its apex is the center of the opposite side of the ellipsoid from the viewer's perspective. The cone is then rendered with only depth writes enabled, no depth test or color writes. Finally, objects that would have previously created artifacts are rendered "on the ellipsoid."

Patrick Says

7.3 Polylines

It is amazing how much mileage virtual globes get out of the simple polyline. Polylines are used for boundaries around closed features, such as zip codes, states, countries, parks, school districts, and lakes. The list goes on and on. Polylines are also used to visualize open features, such as rivers, highways, and train tracks. Polylines are not limited to geographic information either; they can also represent the path for driving directions, the stages in the Tour de France, or even airplane flight paths and satellite orbits. Since not all polylines are on the ground, this section covers general polyline rendering. Section 7.3.5 discusses the special, but common, case of rendering polylines directly on the ground.

Most polyline datasets used by virtual globes come in the form of *line strips*, that is, a series of points where a line segment connects each point, except the first, to the previous point, as shown in Figure 7.4. To represent closed features, the first point can be duplicated and stored as the last

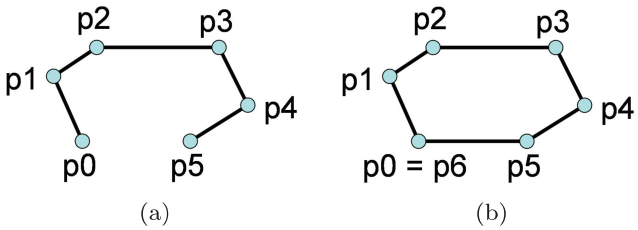


Figure 7.4. (a) An open line strip. A line segment implicitly connects each point, except the first, to the previous point. (b) A closed line strip is created by duplicating the first point at the end.

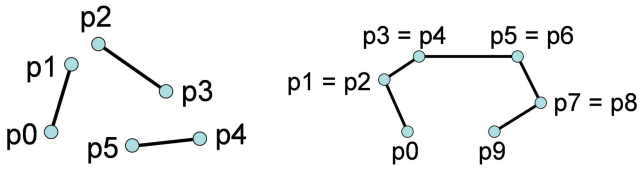
point. Another representation is a *line loop*, which is a line strip that always connects the last point to the first, avoiding the need to store an extra point for closed polylines.

7.3.1 Batching

It is easy enough to render a line strip using PrimitiveType.`LineStrip`—simply create a vertex buffer with a vertex per point in the polyline and issue a draw call using the line strip primitive. But how would you render multiple line strips? For a few dozen, or perhaps a few hundred line strips, it may be reasonable to create a vertex buffer per polyline and issue a draw call per polyline. The problem with this approach is that creating lots of small vertex buffers and issuing several draw calls can introduce significant application and driver CPU overhead. The solution is, of course, to combine line strips into fewer vertex buffers and render them with fewer draw calls. This is known as batching [123, 183].

There are three approaches to batching line strips:

1. *Lines.* The simplest approach is to use PrimitiveType.`Lines`, which treats every two consecutive points as an individual line segment, as in Figure 7.5(a). Since line segments do not need to be connected, this allows storing multiple line strips in one vertex buffer by duplicating each point in a line strip except the first and last points, as in Figure 7.5(b). This converts each line strip to a series of line segments. The line segments for all line strips can be stored in a single vertex buffer and rendered with a single draw call. The downside is that a line strip with n vertices, where $n \geq 2$, now requires $2 + 2\,(n-2) = 2n - 2$ vertices—practically double the amount of memory! Nonetheless, in many cases, using a single draw call to render a vertex buffer of lines will outperform per-line strip vertex buffers and draw calls.



(a) (b)

Figure 7.5. (a) Using the lines primitive, every two points define an individual line segment. (b) The lines primitive can be used to represent a line strip by duplicating interior points.

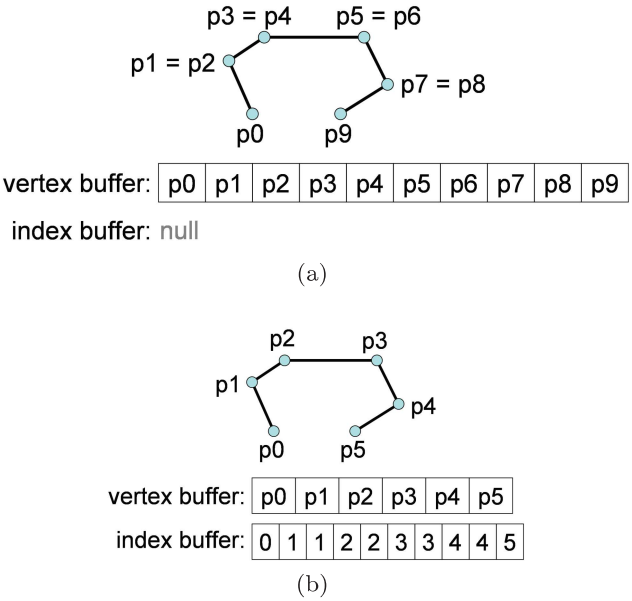


Figure 7.6. (a) Nonindexed lines duplicate points to store line strips. (b) Indexed lines only duplicate indices.

2. *Indexed lines.* The amount of memory used by PrimitiveType.`Lines` can be reduced by adding an index buffer (see Figure 7.6). In much the same way that indexed triangle lists use less memory than individual triangles for most meshes, indexed lines use less memory than nonindexed lines when most line segments are connected. In the case of batching line strips, all line segments, except the ones between strips, are connected. For $n \geq 2$, a line strip stored using indexed lines requires n vertices and $2n - 2$ indices. Since an index almost always requires less memory than a vertex, indexed lines generally consume less memory than nonindexed lines. For example, assuming 32-byte vertices and 2-byte indices, a line strip with n vertices requires the following:

   - $32n$ bytes using a line strip,
   - $32\,(2n - 2) = 64n - 64$ bytes using nonindexed lines,
   - $32n + 2\,(2n - 2) = 40n - 4$ bytes using indexed lines.

   In this example, for $n \geq 3$, indexed lines always use less memory than nonindexed lines and use only slightly more memory than line strips since vertex data dominate the memory usage. For example, a line strip with 1,000 points requires 32,000 bytes when stored as a line

```
RenderState renderState = new RenderState();
renderState.PrimitiveRestart.Enabled = true;
renderState.PrimitiveRestart.Index = 0xFFFF;
```

Listing 7.1. Enabling primitive restart.

strip, 63,936 bytes as nonindexed lines, and 39,996 bytes as indexed lines. Of course, the gap between memory usage for line strips and indexed lines will vary based on the size of vertices and indices.

Indexed lines favor large vertices and small indices. When batching a large number of line strips, it is not uncommon for 4-byte indices to be required. If this becomes a concern, you can limit the number of vertices per batch such that vertices can be indexed using 2-byte indices.

Similar to indexed triangle lists, indexed lines can also take advantage of a GPU's vertex caches to avoid redundantly processing the same vertex.

3. *Indexed line strips and primitive restart.* Another memory-efficient way to batch together line strips is to use indexed line strips. Each line strip requires n vertices and only $n + 1$ indices, except the last line strip, which requires only n indices (see Figure 7.7). Interior line strips store an extra index at their end to signal a *primitive restart* [34], a special token that is interpreted as the end of a line strip and the start of a new one.

   Primitive restart enables drawing multiple indexed line strips in a single draw call. It is part of the render state, which contains a flag to enable or disable it, as well as the index that signifies a restart, as in Listing 7.1.

   In the previous example where indexed lines required 39,996 bytes, indexed line strips require 34,002 bytes, only 2,002 bytes more than

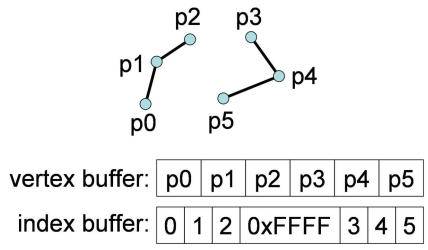


Figure 7.7. Multiple indexed line strips can be rendered in a single draw call by using primitive restart. In this figure, the restart index is $0xFFFF$.

line strips. When considering a single batch with at least one line strip with $n \geq 3$, indexed line strips always use less memory than indexed lines.

7.3.2 Static Buffers

In addition to batching, use static vertex and index buffers where appropriate to achieve the best polyline-rendering performance [120]. How often do country borders, park boundaries, or even highways change? Many polyline datasets are static. As such, they should be uploaded to the GPU once and stored in GPU memory. This makes draw calls more efficient by avoiding the traffic from system memory to video memory, resulting in dramatic performance improvements. A static buffer is requested by passing BufferHint.`StaticDraw` to Device.`CreateVertexBuffer`, Device.`CreateIndexBuffer`, or Context.`CreateVertexArray`.

It is even sometimes appropriate to use a static index buffer with a dynamic or stream vertex buffer. For example, consider a line strip for the path of an airplane currently in flight. When the application receives a position update, the new position is appended to the vertex buffer (which may need to grow every so often). If either indexed lines or indexed line strips are used, the new index or indices are known in advance. Therefore, a static index buffer can contain indices for a large number of vertices, and only the `count` argument passed to Context.`Draw` needs to change for a position update. The index buffer only needs to be reallocated and rewritten when it is no longer large enough.

7.3.3 Wide Lines

Let's turn our attention from performance to appearance. Simple techniques such as line width, solid color, and outline color go a long way in shading polylines. These simple visual cues can make a big difference. For example, a GIS user may want to make country borders wider than state borders. For another example, in a virtual globe used for battlespace visualization, threat zones may be outlined in red and safe zones outlined in green. If one thing is for sure, users want to be able to customize everything!

For line width, we'd like to be able to render lines with a width defined in pixels. In older versions of OpenGL, this was accomplished using the `glLineWidth` function. Since support for wide lines was deprecated in OpenGL 3, we turn to the geometry shader to render wide lines.[3]

[3]Interestingly enough, `glLineWidth` remains but width values greater than one generate an error.

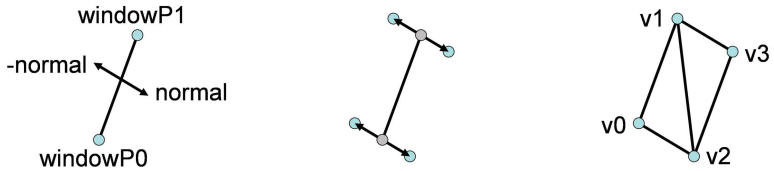


Figure 7.8. (a) A line segment in window coordinates showing normals pointing in opposite directions. (b) Each endpoint is offset along both normals to create four new points. (c) The new points are connected with a triangle strip to form a wide line.

The idea is simple: a geometry shader takes a line segment as input, transforms it into window coordinates, expands it into a two-triangle triangle strip of a given pixel width, then outputs the triangle strip (see Figure 7.8).

The vertex shader simply passes the vertex's model space position through, as in Listing 7.2.

The first order of business for the geometry shader is to project the line's endpoints into window coordinates. Using window coordinates makes it easy to create a triangle strip sized in pixels. In order to avoid artifacts, the line segment needs to be clipped to the near plane, just as the fixed function does against every frustum and clip plane after the geometry shader. Otherwise, one or both endpoints would be transformed to the wrong window coordinates.

Clipping is done by transforming the endpoints into clip coordinates and checking if they are on opposite sides of the near plane. If so, the line segment intersects the near plane and needs to be clipped by replacing the point behind the near plane with the point at the near plane. If both points are behind the near plane, the line segment should be culled. The function in Listing 7.3 transforms the endpoints from model to clip coordinates, clipped to the near plane.

```
in vec4 position;

void main()
{
  gl_Position = position;
}
```

Listing 7.2. Wide lines vertex shader.

```
void ClipLineSegmentToNearPlane(float nearPlaneDistance,
                                mat4 modelViewPerspectiveMatrix,
                                vec4 modelP0, vec4 modelP1,
                                out vec4 clipP0, out vec4 clipP1,
                                out bool culledByNearPlane)
{
  clipP0 = modelViewPerspectiveMatrix * modelP0;
  clipP1 = modelViewPerspectiveMatrix * modelP1;
  culledByNearPlane = false;

  float distanceToP0 = clipP0.z + nearPlaneDistance;
  float distanceToP1 = clipP1.z + nearPlaneDistance;

  if ((distanceToP0 * distanceToP1) < 0.0)
  {
    // Line segment intersects near plane
    float t = distanceToP0 / (distanceToP0 - distanceToP1);
    vec3 modelV = vec3(modelP0) +
                  t * (vec3(modelP1) - vec3(modelP0));
    vec4 clipV = modelViewPerspectiveMatrix * vec4(modelV, 1);

    // Replace the end point closest to the viewer
    // with the intersection point
    if (distanceToP0 < 0.0)
    {
      clipP0 = clipV;
    }
    else
    {
      clipP1 = clipV;
    }
  }
  else if (distanceToP0 < 0.0)
  {
    // Line segment is in front of near plane
    culledByNearPlane = true;
  }
}
```

Listing 7.3. Clipping a line segment to the near plane.

```
vec4 ClipToWindowCoordinates(vec4 v,
                             mat4 viewportTransformationMatrix)
{
  v.xyz /= v.w;
  v.xyz = (viewportTransformationMatrix * vec4(v.xyz, 1.0)).xyz;
  return v;
}
```

Listing 7.4. Transforming from clip to window coordinates.

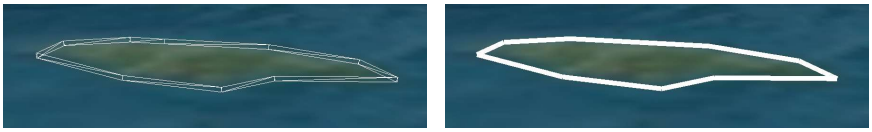
If the line segment is not culled, the endpoints need to be transformed from clip to window coordinates. Perspective divide transforms clip into normalized device coordinates, then the viewport transformation is used to produce window coordinates. These transforms are done as part of the

fixed-function stages after the geometry shader, but we also need to do them in the geometry shader to allow us to work in window coordinates. These transforms are encapsulated in a function, shown in Listing 7.4, that takes a point from clip to window coordinates.

Given the two endpoints in window coordinates, `windowP0` and `windowP1`, expanding the line into a triangle strip for a wide line is easy. Start by computing the line's normal. As shown in Figure 7.8(a), a 2D line has two normals: one pointing to the "left" and one to the "right." The normals are used to introduce two new points per endpoint; each new point is offset from the endpoint along each normal direction, as in Figure 7.8(b).

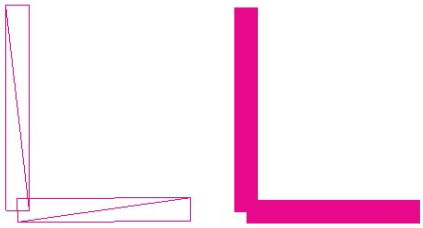
```
vec4 clipP0;
vec4 clipP1;
bool culledByNearPlane;
ClipLineSegmentToNearPlane(og_perspectiveNearPlaneDistance,
                           og_modelViewPerspectiveMatrix,
                           gl_in[0].gl_Position,
                           gl_in[1].gl_Position,
                           clipP0, clipP1, culledByNearPlane);
if (culledByNearPlane)
{
  return;
}

vec4 windowP0 = ClipToWindowCoordinates(clipP0,
    og_viewportTransformationMatrix);
vec4 windowP1 = ClipToWindowCoordinates(clipP1,
    og_viewportTransformationMatrix);

vec2 direction = windowP1.xy - windowP0.xy;
vec2 normal = normalize(vec2(direction.y, -direction.x));

vec4 v0 = vec4(windowP0.xy - (normal * u_fillDistance),
               -windowP0.z, 1.0);
vec4 v1 = vec4(windowP1.xy - (normal * u_fillDistance),
               -windowP1.z, 1.0);
vec4 v2 = vec4(windowP0.xy + (normal * u_fillDistance),
               -windowP0.z, 1.0);
vec4 v3 = vec4(windowP1.xy + (normal * u_fillDistance),
               -windowP1.z, 1.0);

gl_Position = og_viewportOrthographicMatrix * v0;
EmitVertex();

gl_Position = og_viewportOrthographicMatrix * v1;
EmitVertex();

gl_Position = og_viewportOrthographicMatrix * v2;
EmitVertex();

gl_Position = og_viewportOrthographicMatrix * v3;
EmitVertex();
```

Listing 7.5. Geometry shader `main()` function for wide lines.



(a) (b)

Figure 7.9. A ten-pixel-wide line rendered using a geometry shader. (a) Wireframe. (b) Solid.



(a) (b)

Figure 7.10. Wide line strips with sharp turns can create noticeable gaps.

The new points are offset using the line's pixel width. For an x-pixel-wide line, each point is offset by the normal scaled in both directions by $\frac{x}{2}$, which is `u_fillDistance` in Listing 7.5. The four new points are connected using a triangle strip and finally output from the geometry shader. The window-coordinate points are transformed by the orthographic projection matrix, just as we would when rendering anything in window coordinates, such as a heads-up display. Code for the geometry shader's `main`() function is shown in Listing 7.5.

The complete geometry shader for wide lines is in `OpenGlobe.Scene`. Polyline. Figure 7.9 shows both a wireframe and solid line rendered using this algorithm. Most polyline datasets used by virtual globes do not have sharp turns. But for those that do, this algorithm creates noticeable gaps between line segments if the lines are wide enough, as in Figure 7.10. A simple solution to round the turns is to render points the same size as the line width at each endpoint. Other solutions are described by McReynolds and Blythe [113].

7.3.4 Outlined Lines

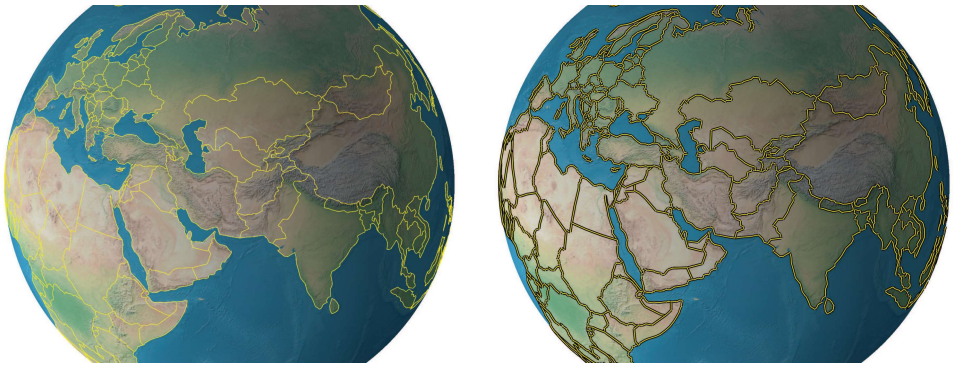
In addition to line width, it is also useful to set a polyline's color. Of course, making each line strip, or even line segment, a different solid color is easy. What is more interesting is using two colors: one for the line's

interior and one for its outline. Outlining makes a polyline stand out, as shown in Figure 7.11.

There are a variety of ways to render outlined lines. Perhaps the most straightforward is to render the line twice: first in the outline color with exterior width, then in the interior color with the interior width. Even if the second pass is rendered with a less-than-or-equal depth-comparison function, this naïve approach results in z-fighting as the outline and interior fight to win the depth test. Z-fighting can be avoided by using the stencil buffer to eliminate the depth test on the second pass, identical to decaling [113]. This is a good solution for supporting older GPUs and is the implementation we use in Insight3D. For hardware with programmable shaders, outlining can be done in a single pass.

One approach is to create extra triangles for the outline in the geometry shader. Instead of creating two triangles, six triangles are created: two for the interior and two on each side of the line for the outline (see Figure 7.12(a) and the implementation in `OpenGlobe.Scene`.OutlinedPolylineGeometryShader). The downside to this approach is that it is subject to aliasing, unless the entire scene is rendered with antialiasing, as shown in Figure 7.12(c). The jagged edges are on both sides of the outline: when the outline meets the interior and when it meets the rest of the scene. We'll call these *interior* and *exterior aliasing*, respectively. Exterior aliasing is present in our line-rendering algorithm even when outlining is not used.

Aliasing can be avoided by using a 1D texture or a one-pixel high 2D texture, instead of creating extra triangles for the outline. The idea is to lay the texture across the line width-wise. A texture read in the fragment shader is then used to determine if the fragment belongs to the line's interior or outline. With linear texture filtering, the interior and outline colors are



(a) (b)

Figure 7.11. (a) No outline. (b) An outline makes polylines stand out.

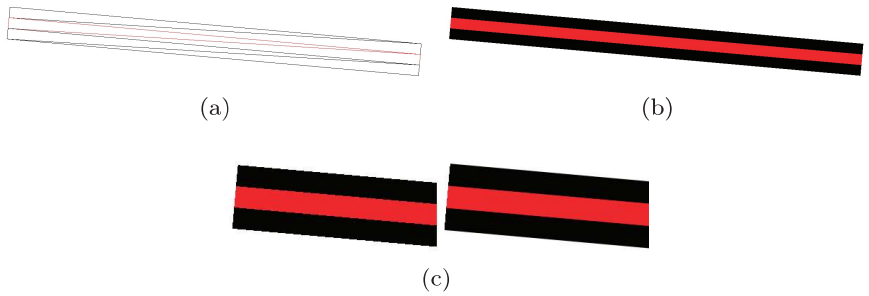


Figure 7.12. (a) Wireframe showing extra triangles for the outline using the geometry shader method. (b) With solid shading. (c) The left side shows aliasing using the geometry shader method; compare this to the right side, which uses the texture method.

smoothly blended without interior aliasing. Exterior aliasing is avoided by storing extra texels representing an alpha of zero on each end of the texture. With blending enabled, this provides exterior antialiasing.

The width of the texture is $interiorWidth + outlineWidth + outlineWidth + 2$. For example, a line with an interior width of three and an outline width of one would use a texture seven pixels wide (see Figure 7.13). The texture is created with two channels: the red channel, which is one for interior and zero for outline, and the green channel, which is zero on the left-most and right-most texels and one everywhere else. The texture only stores 0s and 1s, not actual line colors, so colors can change without rewriting the texture. Likewise, the same texture can be used for different line strips with different colors as long as they have the same interior and outline widths.

The geometry shader assigns texture coordinates when creating the triangle strip. Simply, points on the left side of the original line have a texture coordinate of zero and points on the right side have a texture

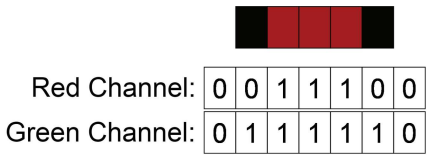


Figure 7.13. An example texture used for outlining. Here, the interior width is three pixels, and the outline width is one pixel. The red channel is one where the interior line is, and the green channel is one where the interior or outline is. Above, an example slice of a line width-wise is shown lined up with the texture.

```
flat in vec4 fsColor;
flat in vec4 fsOutlineColor;
in float fsTextureCoordinate;

out vec4 fragmentColor;

uniform sampler2D og_texture0;

void main()
{
  vec2 texel =
      texture(og_texture0, vec2(fsTextureCoordinate, 0.5)).rg;
  float interior = texel.r;
  float alpha = texel.g;

  vec4 color = mix(fsOutlineColor, fsColor, interior);
  fragmentColor = vec4(color.rgb, color.a * alpha);
}
```

Listing 7.6. Using a texture for outlining.

coordinate of one. In order to capture the left-most and right-most texels used for exterior antialiasing without shrinking the line width, the triangle strip is extruded by an extra half-pixel in each direction.

The fragment shader reads the texture using the interpolated texture coordinate from the geometry shader. The red channel is then used to blend between the interior and outline color for interior antialiasing. The green channel is interpreted as an alpha value, which is multiplied by the color's alpha for exterior antialiasing. The complete fragment shader is shown in Listing 7.6.

The full implementation is in `OpenGlobe.Scene`.OutlinedPolyline Texture. One downside to this approach is that overlapping lines can create artifacts since blending is used for exterior antialiasing. Of course, this can be eliminated by rendering in back-to-front order at a potential performance cost.

7.3.5 Sampling

Our polyline-rendering algorithm connects points using straight line segments in Cartesian space. Imagine what happens if two points are on the ellipsoid but very far away; for example, what would a line segment between Los Angles and New York look like? Of course, it would cut underneath the ellipsoid. In fact, if averaged depth wasn't used, most of the line segment would fail the depth test. The solution is to create a curve on the ellipsoid that subsamples each line segment in a polyline using the algorithm presented in Section 2.4. Doing so makes the polyline better approximate the curvature of the ellipsoid.

Some polylines will have a large number of points; for example, consider polylines bounding large countries such as Russia or China. After subsampling, these polylines will have even more points. Discrete LODs can be computed using an algorithm such as the popular Douglas-Peucker reduction [40, 73]. The LODs can then be rendered using traditional selection techniques (e.g., screen-space error).

LOD for individual polylines is not always useful. In virtual globes, polylines are typically rendered as a part of a layer that may only be visible for certain viewer heights or view distances. This can make LODs for individual polylines less important since a polyline's layer may be turned off before the polyline's lower LODs would even be rendered. Perhaps the best balance is a combination of layers and a handful of layer-aware discrete LODs.

Run Chapter07VectorData to see polylines, polygons, and billboards rendered from shapefile data.

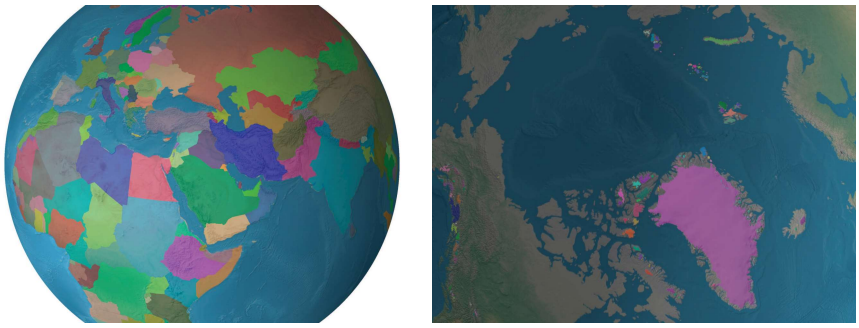
Try This

7.4 Resources

To better understand vector data file formats, it is worth reading the *Esri Shapefile Technical Description* [49], OGC's KML 2.2 specification [131], and Google's "KML Reference" [64].

8 Polygons

Once we can render polylines, the next step is to render polygons on the globe or at a constant height above the globe. Examples of polygons in virtual globes include zip codes, states, countries, and glaciated areas, as shown in Figure 8.1. We define the interior of a polygon by a *line loop*, a polyline where the last point is assumed to connect to the first point.



(a) (b)

Figure 8.1. Randomly colored polygons for (a) countries and (b) glaciated areas.

8.1 Render to Texture

One approach to rendering polygons on the globe is to burn the polygons to a texture, we'll call this a *polygon map*, and render the globe with multi-texturing. The polygon map can be created using a rasterization technique such as flood fill or scan-line fill. In the simplest case, a single one-channel texture can be used, where a value of zero indicates the texel is outside of

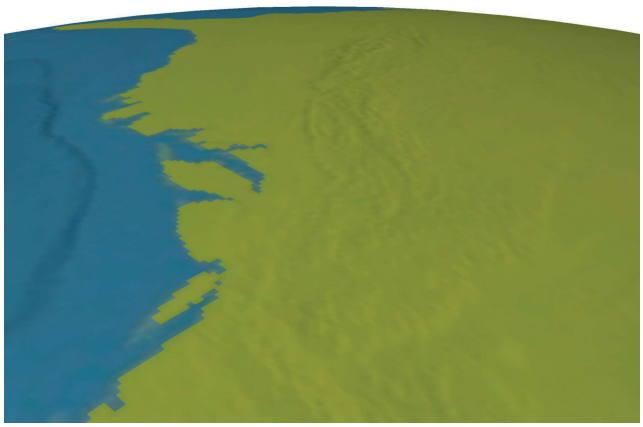


Figure 8.2. A polygon of North America rendered using a polygon map. The low resolution of the map leads to aliasing in the foreground.

a polygon and a value greater than one represents the interior of a polygon and its opacity. When the globe is rendered, the color of its base texture is blended with the polygon's color based on the polygon map. If multiple polygon colors are required, the polygon map can be expanded to four channels or multiple textures can be used. Multiple textures are also useful for supporting overlapping polygons.

A major benefit of this approach is that polygons automatically conform to the globe, even if the globe is rendered with terrain. Also, performance is independent of the number of polygons or number of points defining each polygon. Instead, performance is affected by the size and resolution of the polygon map—which leads to its weakness. Figure 8.2 shows a polygon of North America rendered with this approach. Since the resolution of the polygon map is not high enough, aliasing occurs in the foreground, leading to blocky artifacts. Using a higher-resolution map will, of course, fix the problem at the cost of using a significant amount of memory.

Ultimately, the polygon map can be treated like any other high-resolution texture and rendered with LOD techniques discussed in Part IV. This approach is used in Google Earth and appears to be used in ArcGIS Explorer, among others.

8.2 Tessellating Polygons

Instead of using a polygon map, we will detail an implementation based on tessellating a polygon into a triangle mesh, which is then rendered separately from the globe. This geometry-based approach does not suffer from

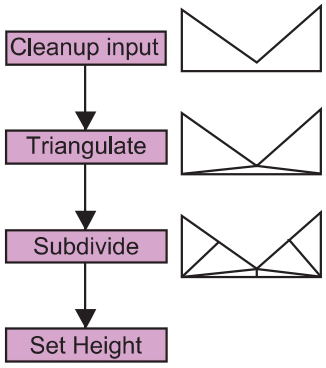


Figure 8.3. The pipeline used to convert a polygon's line loop into a triangle mesh that conforms to the ellipsoid surface of a globe.

large texture memory requirements or aliasing artifacts around its edges. It can also be implemented nearly independently of the globe rendering algorithm, leading to less coupling and clean engine design.

Figure 8.3 shows the pipeline to convert a line loop representing the boundary of a polygon on a globe to a triangle mesh that can be rendered with a simple shader. First, the line loop needs to be cleaned up so the rest of the stages can process it. Next, the polygon is divided into a set of triangles, a process called *triangulation*. In order to make the triangle mesh conform to the ellipsoid surface, the triangles are subdivided until a tolerance is satisfied. Finally, all points are raised to the ellipsoid's surface or a constant height.

8.2.1 Cleaning Up the Input

One thing that makes virtual globes so interesting is that a wide array of geospatial data is available for viewing. The flip side to this is that a plethora of file formats exist, each with their own conventions. For example, KML and shapefiles are two very popular sources of polygon data. Ignoring interior holes, in KML, the `<Polygon>` element's winding order is counterclockwise [64], whereas the winding order for polygons in shapefiles is clockwise [49]. To add to the fun, some files do not always follow the format's specifications.

To make our code as robust as possible, it is important to consider the following cleanup before moving on to the triangulation stage:

- *Eliminate duplicate points.* The algorithms later in our pipeline assume the polygon's boundary is a line loop; therefore, the last point should not be a duplicate of the first point. Consecutive duplicate

```
public static IList<T> Cleanup<T>(IEnumerable<T> positions)
{
  IList<T> positionsList =
      CollectionAlgorithms.EnumerableToList(positions);
  List<T> cleanedPositions =
      new List<T>(positionsList.Count);

  for (int i0 = positionsList.Count - 1, i1 = 0;
       i1 < positionsList.Count; i0 = i1++)
  {
    T v0 = positionsList[i0];
    T v1 = positionsList[i1];

    if (!v0.Equals(v1))
    {
      cleanedPositions.Add(v1);
    }
  }

  cleanedPositions.TrimExcess();
  return cleanedPositions;
}
```

Listing 8.1. Removing duplicate points from a list of points.

points anywhere along the boundary are allowed by some formats, such as shapefiles, but these should be removed as they can lead to degenerate triangles during triangulation. One method for removing duplicate points is to iterate through the points, copying a point to a new list only if it isn't equal to the previous point, as done in Listing 8.1.

This method illustrates a useful technique for iterating over polygon edges [13]. At each iteration, we want to know the current point, `v1`, and the previous point, `v0`. Since the last point implicitly connects to the first point, `i0` is initialized to the index for the last point, and `i1` is initialized to zero. After each iteration, `i0` is set to `i1` and `i1` is incremented. This allows us to iterate over all the edges without treating the last point as a special case or using modulo arithmetic.

Try This ○○◯◯

Make `Cleanup<T>` more efficient: instead of creating a new list, modify the existing list and adjust its size. You will need to modify the method's signature.

- *Detect self-intersections.* Unless the triangulation step in the pipeline can handle self-intersections, it is important to verify that the polygon doesn't cross over itself, like the one in Figure 8.4(c) does.

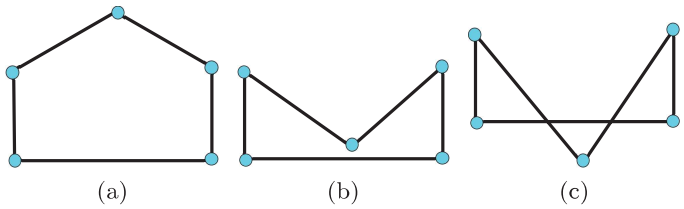


Figure 8.4. (a) Simple convex polygon. (b) Simple concave polygon. (c) Nonsimple polygon.

Even though some format specifications will "guarantee" no self-intersections, it can still be worth checking. One technique to check for self-intersections is to project the points on a plane tangent to the globe at the polygon's centroid. This allows us to treat the 3D polygon as a 2D polygon. Now it is a simple matter of verifying no edge intersects any other edge. The biggest pitfall with this technique is that large polygons that wrap around the globe may be distorted when projected onto the tangent plane. One approach to combating this is to only project the polygon if its bounding sphere's radius is below a threshold. See `OpenGlobe.Core`.EllipsoidTangentPlane for code that computes a tangent plane and projects points onto it.

- *Determine winding order.* Some formats store a polygon's points in clockwise order, others use counterclockwise order. Our algorithms can either handle both winding orders or only handle one and can reverse the order of the points if the winding orders do not match. Reversing can be efficiently implemented in $\mathcal{O}(n)$ by swapping the first and last points, then repeatedly swapping interior points until the middle of the list is reached. Most platforms provide a method for this, such as .NET's List<`T`>.`Reverse`.

  Algorithms later in our pipeline assume counterclockwise order, so if points are provided clockwise, they need to be reversed. One technique for determining winding order is to project the polygon onto a tangent plane as mentioned above, then compute the area of the 2D polygon. The sign of the area determines the winding order: nonnegative is counterclockwise and negative is clockwise (see Listing 8.2).

Eliminating duplicate points, checking for self-intersections, and handling winding order can chew up some CPU time. A naïve implementation that checks for self-intersection is $\mathcal{O}(n^2)$, and the other two steps are $\mathcal{O}(n)$. Although the vast majority of time will be spent in later stages of our pipeline, it would be nice to be able to skip these steps, especially considering that using a tangent plane can sometimes cause problems. However,

```
public static PolygonWindingOrder ComputeWindingOrder(
    IEnumerable<Vector2D> positions)
{
  return (ComputeArea(positions) >= 0.0)
         ? PolygonWindingOrder.Counterclockwise
         : PolygonWindingOrder.Clockwise;
}

public static double ComputeArea(IEnumerable<Vector2D> positions)
{
  IList<Vector2D> positionsList =
      CollectionAlgorithms.EnumerableToList(positions);

  double area = 0.0;
  for (int i0 = positionsList.Count - 1, i1 = 0;
       i1 < positionsList.Count; i0 = i1++)
  {
    Vector2D v0 = positionsList[i0];
    Vector2D v1 = positionsList[i1];

    area += (v0.X * v1.Y) - (v1.X * v0.Y);
  }
  return area * 0.5;
}
```

Listing 8.2. Using a polygon's area to compute its winding order.

we advise against skipping these cleaning steps unless you are using your own file format that only your application writes. This is one of the many challenges virtual globes face: handling a wide array of formats.

8.2.2 Triangulation

GPUs don't rasterize polygons—they rasterize triangles. Triangles make hardware algorithms simpler and faster because they are guaranteed to be convex and planar. Even if GPUs could directly rasterize polygons, the GPU is unaware of our need to render polygons on a globe. So once we've cleaned up a polygon, the next stage in the pipeline is triangulation: decomposing the polygon into a set of triangles without introducing new points. We will limit ourselves to *simple polygons*, which are polygons that do not contain consecutive duplicate points or self-intersections and each point shares exactly two edges.

A *convex polygon* is a polygon where each interior angle is less than 180°, as in Figure 8.4(a). Just because a polygon is simple does not mean it needs to be convex. Although some useful shapes like circles and rectangles are convex, most real-world geospatial polygons will have one or more interior angles greater than 180°, making them concave (see Figure 8.4(b)).

A nice property of simple polygons is that every simple polygon can be triangulated. A simple polygon with n points will always result in $n - 2$

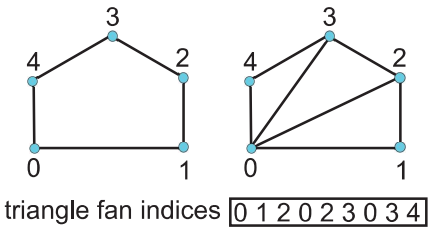


Figure 8.5. Decomposing a convex polygon into a triangle fan.

triangles. For convex polygons, triangulation is easy and fast at $\mathcal{O}(n)$. As shown in Figure 8.5, a convex polygon can simply be decomposed into a triangle fan pivoting around one point on the boundary.

A fan can result in many long and thin triangles (e.g., consider a fan for a circle). These types of triangles have an adverse effect on rasterization because fragments along shared triangle edges are redundantly shaded. Persson suggests an alternative greedy triangulation algorithm based on creating triangles with the largest possible area [139]. If convex polygons are common in your application, consider using a separate code path for them because triangulating concave polygons is considerably more expensive.

There are several algorithms for triangulating concave polygons, with varying runtimes, from a naïve $\mathcal{O}(n^3)$ algorithm to a complex $\mathcal{O}(n)$ one [25]. We're going to focus on a simple and robust algorithm called *ear clipping* that we've found works well for a wide range of geospatial polygons: zip codes, states, country borders, etc. In order to understand the fundamentals, we're going to describe the $\mathcal{O}(n^3)$ implementation first, then briefly look at optimizations to make the algorithm very fast in practice.

Since the type of polygons we are interested in triangulating are defined on the face of an ellipsoid, they are rarely planar in three dimensions (i.e., they form a curved surface in the WGS84 coordinate system). Fortunately, ear clipping can be extended to handle this. We'll start by explaining ear clipping for planar 2D polygons, then we'll extend ear clipping to 3D polygons on the ellipsoid.

Ear clipping in two dimensions. An *ear* of a polygon is a triangle formed by three consecutive points, p_{i-1}, p_i, p_{i+1}, that does not contain any other points in the polygon, and the *tip* of the ear, p_i, is convex. A point is convex if the interior angle between the shared edges, $p_{i-1} - p_i$ and $p_{i+1} - p_i$, is not greater than 180°.

The main operation in ear clipping is walking over a polygon's points and testing if three consecutive points form an ear. If so, the tip of the ear

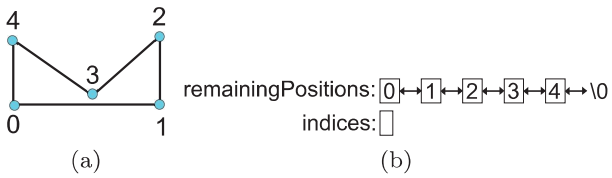


Figure 8.6. (a) A simple, planar polygon before ear clipping. (b) Initial data structures before ear clipping. Only indices are shown in the `remainingPositions` list.

```
internal struct IndexedVector<T> : IEquatable<IndexedVector<T>>
{
  public IndexedVector(T vector, int index) { /* ... */ }
  public T Vector  { get; }
  public int Index { get; }

  // ...
}
```

Listing 8.3. A type used to hold a point and an index.

is removed and the ear's indices are recorded. Ear clipping then continues with the remaining polygon until a single triangle, forming the final ear, remains. As ears are removed and the polygon is reduced, new ears will appear until the algorithm finally converges. The union of the ears forms the triangulation of the polygon.

An implementation can simply take the polygon's boundary points as input and return a list of indices, where every three indices is a triangle in the polygon's triangulation. A full implementation is provided in `OpenGlobe.Core.EarClipping`. We will outline the major steps here by going through an example of triangulating the polygon in Figure 8.6(a).

Since we're not actually going to remove points from the input polygon, the first step is to copy the points to a new data structure. A doubly linked list is a good choice since it supports constant time removal. Given that we want to return a list of indices to the input positions, each node in the linked lists should contain the point and its index[1] using a type such as our IndexedVector<T> in Listing 8.3.

Using IndexedVector<T>, the linked list can be created with the following code:

[1]If the points are already stored in an indexable collection, just the index could be stored in each linked-list node. In our implementation, we follow the .NET convention of passing collections using IEnumerable, which does not support indexing.

```
public static class EarClipping
{
  public static IndicesUnsignedInt Triangulate(
      IEnumerable<Vector2D> positions)
  {
    LinkedList<IndexedVector<Vector2D>> remainingPositions =
        new LinkedList<IndexedVector<Vector2D>>();

    int index = 0;
    foreach (Vector2D position in positions)
    {
      remainingPositions.AddLast(
          new IndexedVector<Vector2D>(position, index++));
    }

    IndicesUnsignedInt indices = new IndicesUnsignedInt(
        3 * (remainingPositions.Count - 2));

    // ...
  }
}
```

The above code also allocates memory for the output indices using the fact that triangulation of a simple polygon of n points creates $n - 2$ triangles. After this initialization, the contents of `remainingPoints` and `indices` are shown in Figure 8.6(b). The rest of the algorithm operates on these two data structures by removing points from `remainingPoints` and adding triangles to `indices`.

First, the triangle formed by points (p_0, p_1, p_2), shown in Figure 8.7, is tested to see if it is an ear. The first part of the test is to make sure the tip is convex. Since we are assured the input points are ordered counterclockwise due to the cleanup stage of our pipeline, we can test if a tip is convex by checking if the sign of the cross product between two vectors along the outside of the triangle is nonnegative, as shown in Listing 8.4.

If the tip is convex, the triangle containment test follows, which checks if every point in the polygon, except the points forming the triangle, is inside the triangle. If all points are outside of the triangle, the triangle is an ear, and its indices are added to `indices` and its tip is removed from

```
private static bool IsTipConvex(Vector2D p0, Vector2D p1,
                                Vector2D p2)
{
  Vector2D u = p1 - p0;
  Vector2D v = p2 - p1;
  return ((u.X * v.Y) - (u.Y * v.X)) >= 0.0;
}
```

Listing 8.4. Testing if a potential ear's tip is convex.

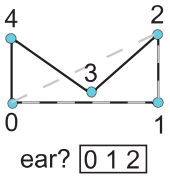


Figure 8.7. Ear test for triangle (p_0, p_1, p_2).

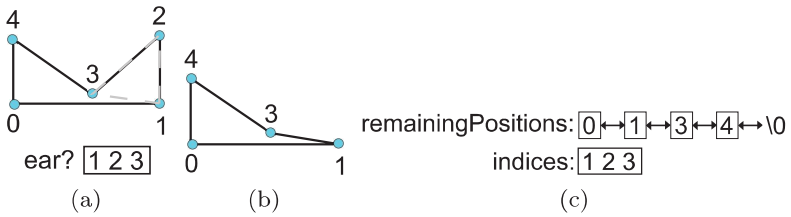


Figure 8.8. (a) Ear test for triangle (p_1, p_2, p_3). (b) Polygon after removing p_2. (c) Data structures after removing p_2.

`remainingPositions`. In our example, although the tip of triangle (p_0, p_1, p_2) is convex, p_3 is inside the triangle, making it fail the triangle containment test.

So ear clipping moves on to test triangle (p_1, p_2, p_3), shown in Figure 8.8(a). This triangle passes both the convex tip and triangle containment test, making it an ear that can be clipped, so p_2 is removed from `remainingPositions` and indices $(1, 2, 3)$ are added to `indices`. The updated polygon and data structures are shown in Figures 8.8(b) and 8.8(c).

The next triangle in question is triangle (p_1, p_3, p_4), shown in Figure 8.9, which fails the convex tip test. So ear clipping moves on to triangle (p_3, p_4, p_0), which is an ear, resulting in the changes shown in Figure 8.10.

After removing p_4, only one triangle remains, shown in Figure 8.10(b). The indices for this triangle are added to `indices` and ear clipping is complete. The final triangulation is shown in Figure 8.11.

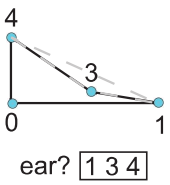


Figure 8.9. Ear test for triangle (p_1, p_3, p_4).

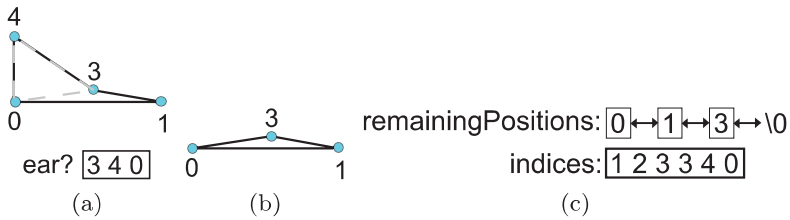


Figure 8.10. (a) Ear test for triangle (p_3, p_4, p_0). (b) Polygon after removing p_4. (c) Data structures after removing p_4.

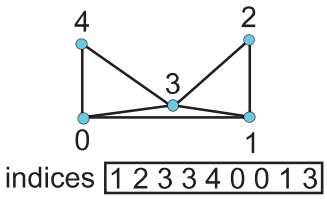


Figure 8.11. Final triangulation after ear clipping.

Figure 8.12 shows the triangulation of a more complex, real-world polygon. Creating a triangle fan for a convex polygon is basically ear clipping, except each triangle is guaranteed to be an ear, making the algorithm much faster since no ear tests are required.

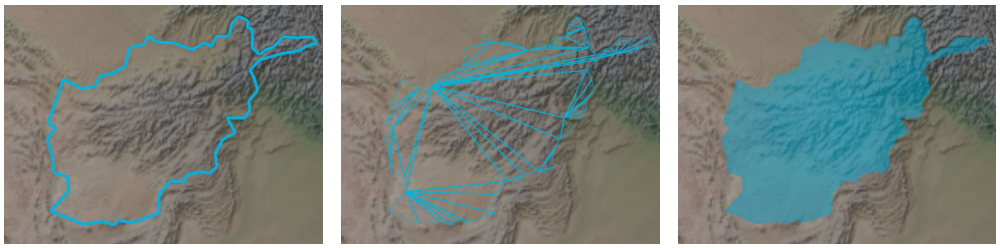


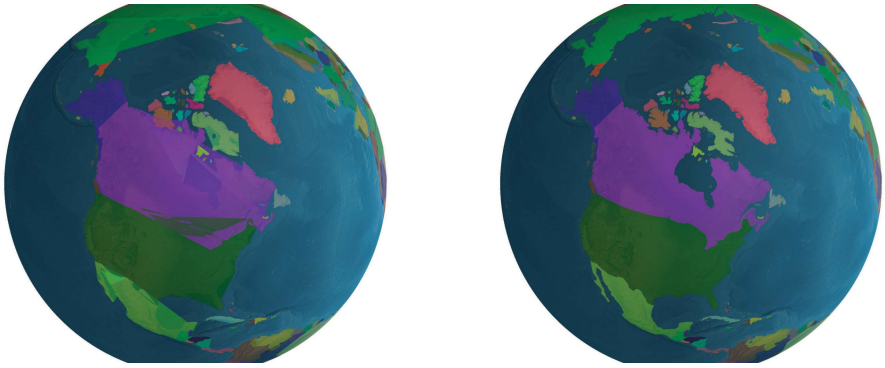
Figure 8.12. (a) The line loop of a polygon for Afghanistan. (b) The wireframe triangulation of the polygon. (c) The triangulation rendered with translucency.

Ear clipping on the ellipsoid. Our approach to ear clipping thus far works well for planar 2D polygons, but it doesn't work for polygons defined on a globe by a set of geographic (longitude, latitude) pairs. We may be tempted to treat longitude as x and latitude as y, then simply use our 2D ear-clipping implementation. In addition to wraparound issues at the poles and the IDL, this approach has a more fundamental problem: our

implementations for `IsTipConvex` and `PointInsideTriangle` are designed for Cartesian, not geographic, coordinates, which makes the primary operation in ear clipping, ear testing, incorrect. For example, consider three points at the same latitude with longitudes of 10°, 20°, and 30°. If the geodetic coordinates are treated as Cartesian coordinates, this is a straight line segment. In reality, in Cartesian WGS84 coordinates, it is two line segments with an angle where they meet. The result is the ear test can have false positives or negatives creating incorrect triangulations, like those shown in Figure 8.13(a).

There are two ways to achieve a correct triangulation, like that of Figure 8.13(b). One way is to project the boundary points onto a tangent plane, then use 2D ear clipping to triangulate the points on the plane. Imagine yourself at the center of the ellipsoid looking out towards the tangent plane. Consider two consecutive points on the polygon. Two rays cast from the center of the globe through each point lie in a plane whose intersection with the ellipsoid forms a curve (see Section 2.4) between the points. This creates the property that curves are projected to line segments on the plane, allowing our ear test to return correct results. Triangulating 3D polygons by projecting them onto a plane is a common approach and is the technique used in fast industrial-strength triangulation (FIST) [70].

An alternative approach is to modify ear clipping to handle points on an ellipsoid. The main observation is that the 2D point in the triangle test can be replaced with a 3D point in the infinite-pyramid test. An infinite



(a) (b)

Figure 8.13. (a) Triangulating a polygon on the ellipsoid by naïvely treating longitude as x and latitude as y results in an incorrect triangulation. (b) A correct triangulation is achieved by projecting the polygon onto a tangent plane or replacing the 2D point in the triangle test with a 3D point in the infinite-pyramid test.

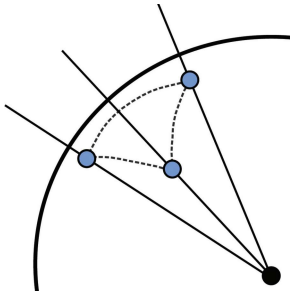


Figure 8.14. An infinite pyramid formed by the center of the ellipsoid and three points on its surface. When ear clipping polygons on an ellipsoid, the point-in-triangle test should be replaced with a point-in-infinite-pyramid test.

pyramid is formed with the ellipsoid center at its apex, extending through a triangular base formed by three points on the ellipsoid's surface, as shown in Figure 8.14. The inside of the pyramid is the space inside the intersection of three planes. Each plane contains the ellipsoid's center and two points in the triangle. The intersection of a plane and the ellipsoid forms a curve between the two points on the surface, shown as dashed lines in Figure 8.14.

The point-in-infinite-pyramid test is performed by first computing outward-facing normals for each plane. A point is in the pyramid if it is behind each plane. Since the pyramid's apex lies in all three planes, the sign of the dot product of the vector from the apex to the point in question with a plane's normal determines the side of the plane the point is on. If all three dot products are negative, the point is inside the pyramid (see Listing 8.5).

```
public static bool PointInsideThreeSidedInfinitePyramid(
    Vector3D point,
    Vector3D pyramidApex,
    Vector3D pyramidBase0,
    Vector3D pyramidBase1,
    Vector3D pyramidBase2)
{
  Vector3D v0 = pyramidBase0 - pyramidApex;
  Vector3D v1 = pyramidBase1 - pyramidApex;
  Vector3D v2 = pyramidBase2 - pyramidApex;
  Vector3D n0 = v1.Cross(v0);
  Vector3D n1 = v2.Cross(v1);
  Vector3D n2 = v0.Cross(v2);
  Vector3D planeToPoint = point - pyramidApex;

  return (planeToPoint.Dot(n0) < 0) &&
         (planeToPoint.Dot(n1) < 0) &&
         (planeToPoint.Dot(n2) < 0);
}
```

Listing 8.5. Testing if a point is inside an infinite pyramid.

```
private static bool IsTipConvex(Vector3D p0, Vector3D p1,
                                Vector3D p2)
{
  Vector3D u = p1 - p0;
  Vector3D v = p2 - p1;

  return u.Cross(v).Dot(p1) >= 0.0;
}
```

Listing 8.6. Testing if the tip of a potential ear on the ellipsoid is convex.

Replacing the point-in-triangle test with a point-in-infinite-pyramid test gets us halfway to ear clipping a polygon on an ellipsoid. The other half is determining if a potential ear's tip is convex. Given three points forming a triangle on the ellipsoid, (p_0, p_1, p_2), form two vectors along the outside of the triangle relative to the polygon: $\mathbf{u} = p_1 - p_0$ and $\mathbf{v} = p_2 - p_1$. If the tip, p_1, is convex, the cross product of $\mathbf{u}$ and $\mathbf{v}$ should yield an upward-facing vector relative to the surface. This can be verified by checking the sign of the dot product with $p_1 - 0$, as shown in Listing 8.6.

Ear clipping a polygon on an ellipsoid is now a simple matter of changing our 2D ear-clipping implementation to use Vector3D instead of Vector2D and using our new `PointInsideThreeSidedInfinitePyramid` and `IsTipConvex` methods. The input is a polygon's line loop in WGS84 coordinates. A complete implementation is provided in `OpenGlobe.Core.EarClippingOnEllipsoid`. Compare this to the 2D implementation in `OpenGlobe.Core.EarClipping`.

Besides creating a triangle mesh for rendering, triangulation is also useful for point in polygon testing. A point on an ellipsoid's surface is also in a polygon on the surface if it is in any of the infinite pyramids defined by each triangle in the polygon's triangulation. This can be made quite efficient by precomputing the normals for each pyramid's plane, reducing each point in pyramid test to three dot products. Geospatial analyses, such as point in polygon tests, are the bread and butter of GIS applications.

8.2.3 Ear-Clipping Optimizations

Given that virtual globe users commonly want to visualize a large number of geospatial polygons, and each polygon can have a large number of points, it is important to optimize ear clipping because a naïve implementation doesn't scale well at $\mathcal{O}(n^3)$. Thankfully, this is a worst-case runtime, and in practice, our naïve implementation will run much closer to $\mathcal{O}(n^2)$. But there is still plenty of room for optimization.

First, let's understand why the runtime is $\mathcal{O}(n^3)$; the triangle containment test is $\mathcal{O}(n)$, finding the next ear is $\mathcal{O}(n)$, and $\mathcal{O}(n)$ ears are needed. Although the triangle containment test is $\mathcal{O}(n)$, n decreases as ear tips are

removed. In practice, the next ear is found much quicker than $\mathcal{O}(n)$, which is why the algorithm rarely hits its $\mathcal{O}(n^3)$ upper bound using real polygons.

At least $\mathcal{O}(n)$ ears are always required, but the other two steps can be improved. In fact, the $\mathcal{O}(n)$ search for the next ear can be eliminated, making the worst-case runtime $\mathcal{O}(n^2)$. The key observation is that removing an ear tip only affects the ear-ness of the triangle to its left and its right, so an $\mathcal{O}(n)$ search for the next ear is not required after ears in the initial polygon are found. This does require some extra bookkeeping.

Eberly describes an implementation that maintains four doubly linked lists simultaneously: lists for all polygon points, reflex points only, concave points only, and ear tips [44]. *Reflex points* are those points whose incident edges form an interior angle larger than 180°. The reason for keeping a list of these points is because only reflex points need to be considered during the triangle containment test, bringing the runtime down to $\mathcal{O}(nr)$, where r is the number of reflex points. For geospatial polygons including zip codes, states, and countries, Ohlarik found 40% to 50% of a polygon's initial points were reflex [127].

The final bottleneck to consider is the linear search, now $\mathcal{O}(r)$, used in the triangle containment test. This can be replaced by storing reflex points in leaf nodes of a spatial binary tree [127]. The linear search is then replaced with only tests for reflex points in the leaves whose AABB's overlap the triangle in question. Empirically, this optimization was found to scale very well on geospatial polygons, despite the need to create the tree.

Similarly, Held presents a detailed description and analysis of using geometric hashing, both bounding volume trees and regular grids, to optimize ear clipping [69]. Although geometric hashing does not improve the worst case runtime, in practice, it can obtain near $\mathcal{O}(n)$ runtimes.

Try This

Add any of the above optimizations to `OpenGlobe.Core.EarClipping`. Record timings before and after. Are the results always what you expect?

8.2.4 Subdivision

If the globe were flat, a polygon would be ready for rendering after triangulation, but consider the wireframe rendering of the large triangle in Figure 8.15(a). From this view, the triangle appears to be on the ellipsoid's surface. By viewing the triangle edge, as in Figures 8.15(b) and 8.15(c), it becomes clear that only the triangle's endpoints are on the ellipsoid; the triangle itself is well under the surface. What we really want is the triangle