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

© 2011 by Taylor & Francis Group, LLC
CRC Press is an imprint of Taylor & Francis Group, an Informa business

No claim to original U.S. Government works
Version Date: 2011912

International Standard Book Number-13: 978-1-4398-6558-3 (eBook - PDF)

This book contains information obtained from authentic and highly regarded sources. Reasonable efforts have been made to publish reliable data and information, but the author and publisher cannot assume responsibility for the validity of all materials or the consequences of their use. The authors and publishers have attempted to trace the copyright holders of all material reproduced in this publication and apologize to copyright holders if permission to publish in this form has not been obtained. If any copyright material has not been acknowledged please write and let us know so we may rectify in any future reprint.

Except as permitted under U.S. Copyright Law, no part of this book may be reprinted, reproduced, transmitted, or utilized in any form by any electronic, mechanical, or other means, now known or hereafter invented, including photocopying, microfilming, and recording, or in any information storage or retrieval system, without written permission from the publishers.

For permission to photocopy or use material electronically from this work, please access www.copyright.com (http://www.copyright.com/) or contact the Copyright Clearance Center, Inc. (CCC), 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400. CCC is a not-for-profit organization that provides licenses and registration for a variety of users. For organizations that have been granted a photocopy license by the CCC, a separate system of payment has been arranged.

**Trademark Notice:** Product or corporate names may be trademarks or registered trademarks, and are used only for identification and explanation without intent to infringe.

**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.

Patrick Says

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.

Kevin Says

# Contents


Foreword vii

Preface vii

1 Introduction 1
- 1.1 Rendering Challenges in Virtual Globes . . . . . 1
- 1.2 Contents Overview . . . . . 5
- 1.3 OpenGlobe Architecture . . . . . 8
- 1.4 Conventions . . . . . 10

I Fundamentals 11

2 Math Foundations 13
- 2.1 Virtual Globe Coordinate Systems . . . . . 13
- 2.2 Ellipsoid Basics . . . . . 17
- 2.3 Coordinate Transformations . . . . . 22
- 2.4 Curves on an Ellipsoid . . . . . 34
- 2.5 Resources . . . . . 39

3 Renderer Design 41
- 3.1 The Need for a Renderer . . . . . 42
- 3.2 Bird's-Eye View . . . . . 46
- 3.3 State Management . . . . . 50
- 3.4 Shaders . . . . . 63
- 3.5 Vertex Data . . . . . 84
- 3.6 Textures . . . . . 101
- 3.7 Framebuffers . . . . . 112
- 3.8 Putting It All Together: Rendering a Triangle . . . . . 115
- 3.9 Resources . . . . . 119

4 Globe Rendering 121
4.1 Tessellation . . . 121
4.2 Shading . . . 133
4.3 GPU Ray Casting . . . 149
4.4 Resources . . . 154

II Precision 155

5 Vertex Transform Precision 157
5.1 Jittering Explained . . . 158
5.2 Rendering Relative to Center . . . 164
5.3 Rendering Relative to Eye Using the CPU . . . 169
5.4 Rendering Relative to Eye Using the GPU . . . 171
5.5 Recommendations . . . 177
5.6 Resources . . . 180

6 Depth Buffer Precision 181
6.1 Causes of Depth Buffer Errors . . . 181
6.2 Basic Solutions . . . 188
6.3 Complementary Depth Buffering . . . 189
6.4 Logarithmic Depth Buffer . . . 191
6.5 Rendering with Multiple Frustums . . . 194
6.6 W-Buffer . . . 198
6.7 Algorithms Summary . . . 198
6.8 Resources . . . 199

III Vector Data 201

7 Vector Data and Polylines 203
7.1 Sources of Vector Data . . . 203
7.2 Combating Z-Fighting . . . 204
7.3 Polylines . . . 207
7.4 Resources . . . 219

8 Polygons 221
8.1 Render to Texture . . . 221
8.2 Tessellating Polygons . . . 222
8.3 Polygons on Terrain . . . 241
8.4 Resources . . . 250

9 Billboards 251
9.1 Basic Rendering . . . 252
9.2 Minimizing Texture Switches . . . 258
9.3 Origins and Offsets . . . 267
9.4 Rendering Text . . . 271
9.5 Resources . . . 273

10 Exploiting Parallelism in Resource Preparation 275
10.1 Parallelism Everywhere . . . 275
10.2 Task-Level Parallelism in Virtual Globes . . . 278
10.3 Architectures for Multithreading . . . 280
10.4 Multithreading with OpenGL . . . 292
10.5 Resources . . . 304

IV Terrain 305

11 Terrain Basics 307
11.1 Terrain Representations . . . 308
11.2 Rendering Height Maps . . . 313
11.3 Computing Normals . . . 335
11.4 Shading . . . 343
11.5 Resources . . . 363

12 Massive-Terrain Rendering 365
12.1 Level of Detail . . . 367
12.2 Preprocessing . . . 376
12.3 Out-of-Core Rendering . . . 381
12.4 Culling . . . 390
12.5 Resources . . . 400

13 Geometry Clipmapping 403
13.1 The Clipmap Pyramid . . . 406
13.2 Vertex Buffers . . . 408
13.3 Vertex and Fragment Shaders . . . 411
13.4 Blending . . . 414
13.5 Clipmap Update . . . 417
13.6 Shading . . . 435
13.7 Geometry Clipmapping on a Globe . . . 436
13.8 Resources . . . 443

14 Chunked LOD 445
14.1 Chunks . . . 447
14.2 Selection . . . 448
14.3 Cracks between Chunks . . . 449
14.4 Switching . . . 450
14.5 Generation . . . 452
14.6 Shading . . . 459
14.7 Out-of-Core Rendering . . . 460
14.8 Chunked LOD on a Globe . . . 462
14.9 Chunked LOD Compared to Geometry Clipmapping . . . 463
14.10 Resources . . . 465

A Implementing a Message Queue 467

Bibliography 477

Index 491

# 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.

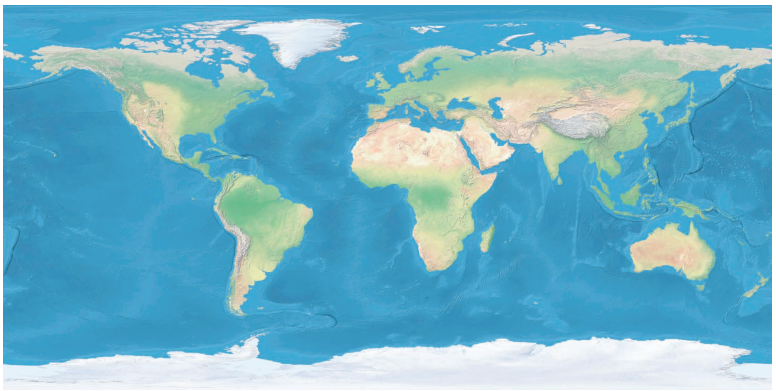


**Figure 1.** Satellite-derived land imagery with shaded relief and water from Natural Earth.

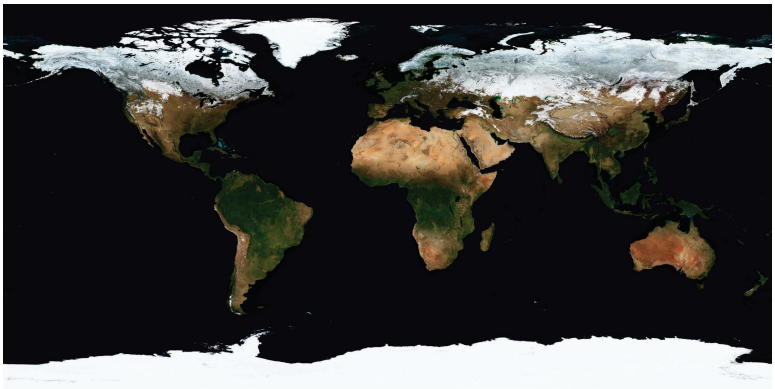
## NASA Visible Earth

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.

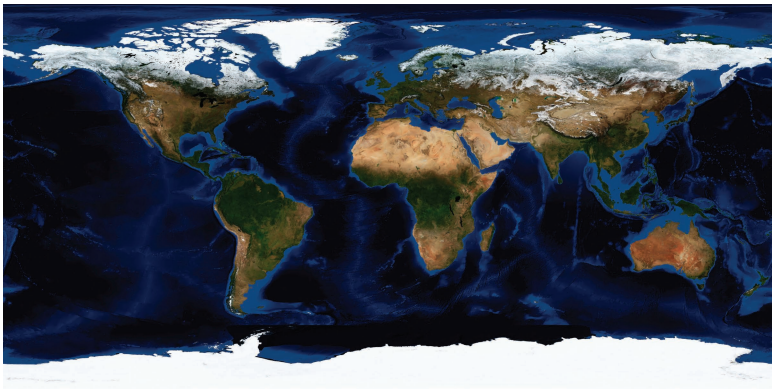
## NASA World Wind

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

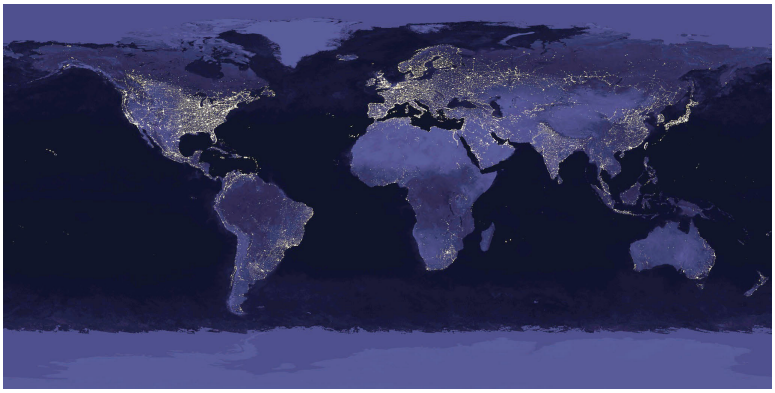
[1] http://www2.jpl.nasa.gov/srtm/



(a)



(b)



(c)

**Figure 2.** Images from NASA Visible Earth.

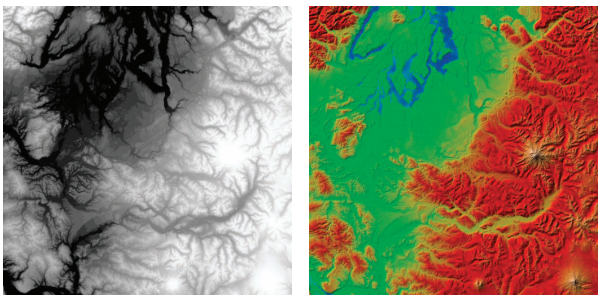
Dataset (NED) from the United States Geological Survey (USGS);[2] and SRTM30_PLUS: SRTM30, coastal and ridge multibeam, estimated topography, from the Institute of Geophysics and Planetary Physics, Scripps Institution of Oceanography, University of California San Diego.[3]

## 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.

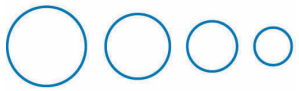
[2] http://ned.usgs.gov/
[3] http://topex.ucsd.edu/WWW_html/srtm30_plus.html
[4] http://rocky.ess.washington.edu/data/raster/tenmeter/onebytwo10/

### 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.

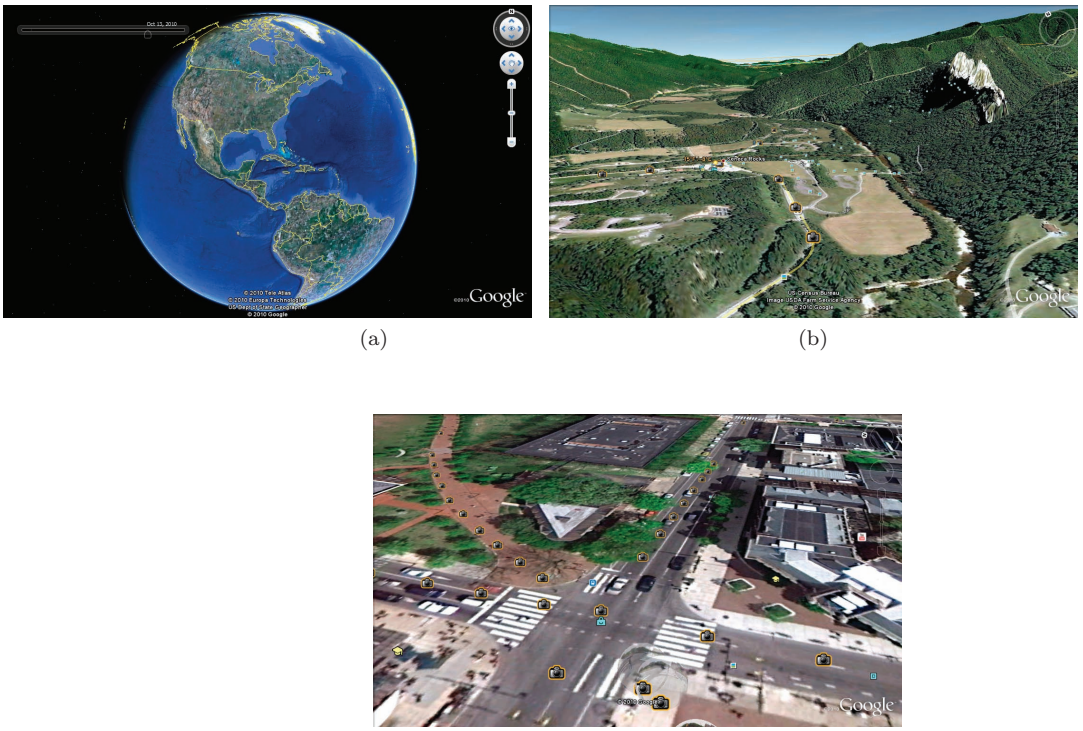
Obviously, implementing a 3D engine for virtual globes requires careful management of these datasets. Storing the entire world in memory and brute force rendering are certainly out of the question. Virtual globes, though, face additional rendering challenges beyond massive data management. This chapter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



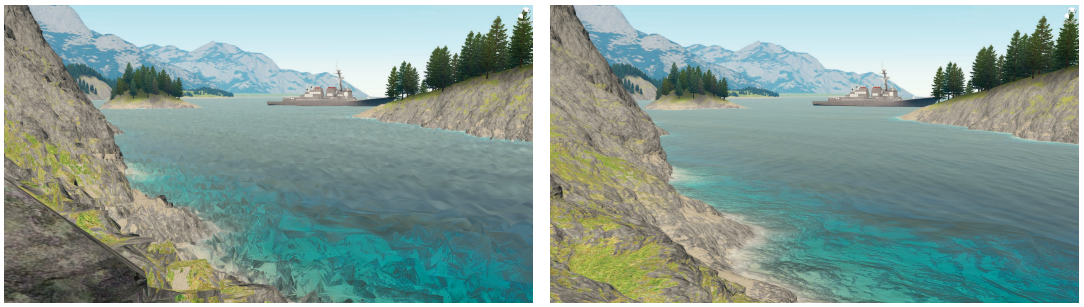

(a) (b)

(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.

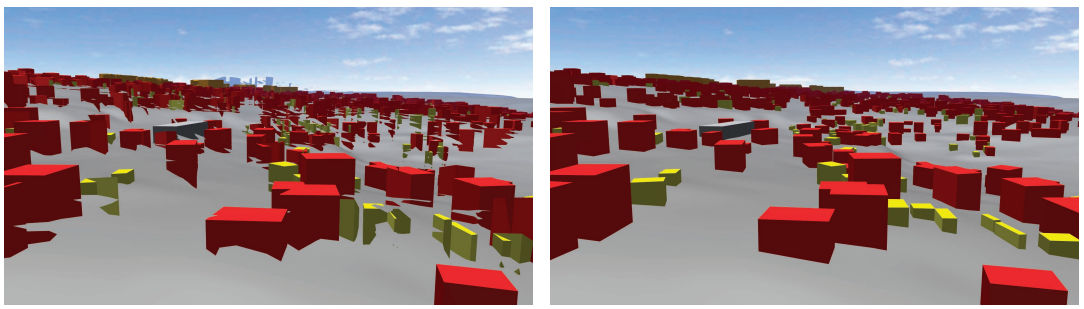
- *Accuracy.* In addition to eliminating rendering artifacts caused by precision errors, virtual globes should also model Earth accurately. Assuming Earth is a perfect sphere allows for many simplifications, but Earth is actually about 21 km longer at the equator than at the poles. Failing to take this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

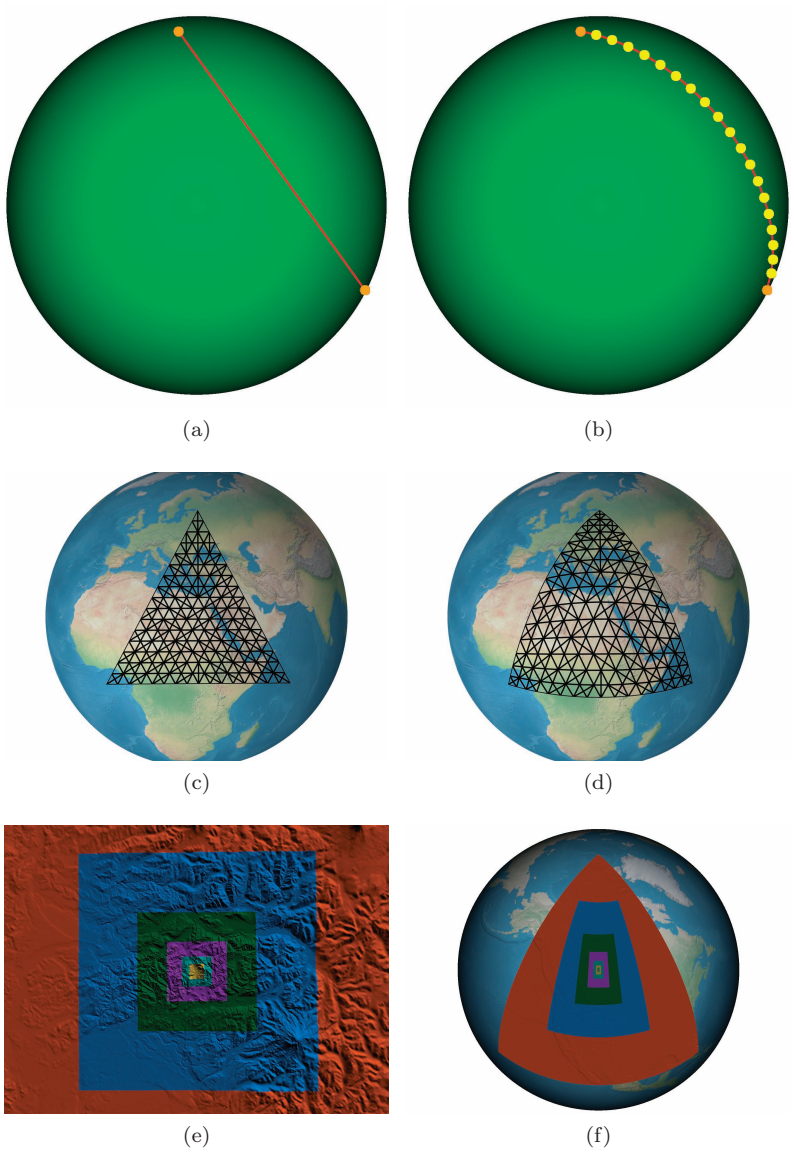



**Figure 1.4.** (a) Lines connecting surface points cut underneath a globe; instead, (b) points should be connected with a curve. Likewise, (c) polygons composed of triangles cut under a globe unless (d) curvature is taken into account. Mapping flat-world algorithms, (e) like geometry clipmapping terrain,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

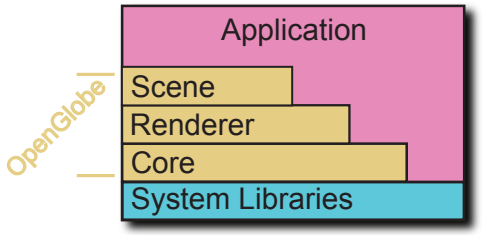



**Figure 1.5.** The stack of OpenGlobe assemblies.

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.

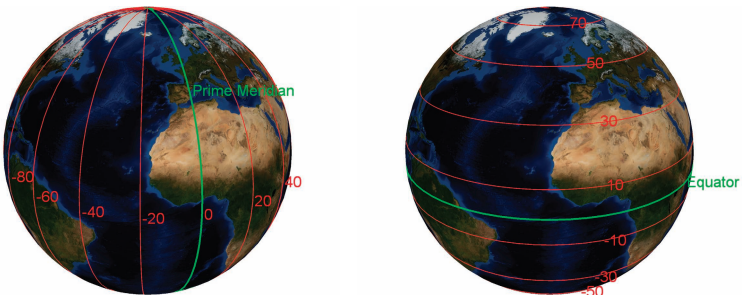
Geographic coordinates are widely used; most vector data are defined in geographic coordinates (see Part III). Even outside virtual globes, geographic coordinates are used for things such as the global positioning systems (GPS).

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.

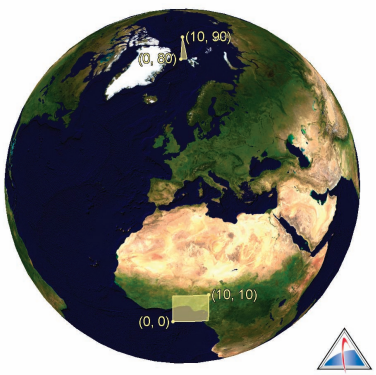



**Figure 2.2.** Extents with the same square number of degrees do not necessarily have the same surface area. (Image taken using STK. The Blue Marble imagery is from NASA Visible Earth.)

$(0°, 80°)$ 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 $2\pi$ 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.

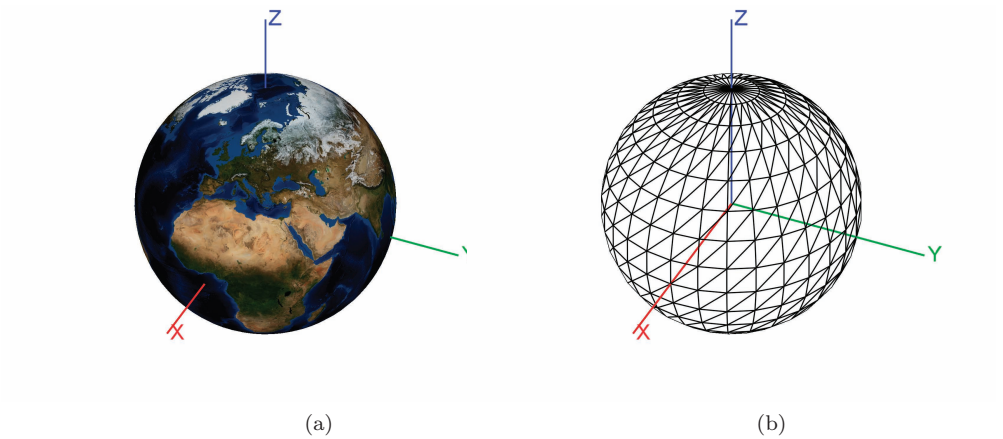
### 2.1.2 WGS84 Coordinate System

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.

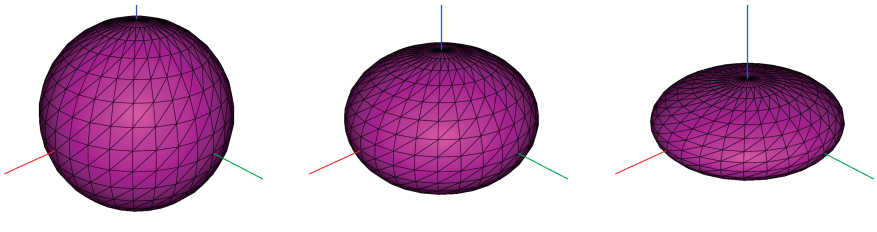



**Figure 2.4.** Oblate spheroids with different semiminor axes. All of these oblate spheroids have a semimajor axis = 1 and a semiminor axis along the $z$-direction (blue). (a) Semiminor axis = 1. The oblate spheroid is a sphere. (b) Semiminor axis = 0.7. (c) Semiminor axis = 0.4.

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.

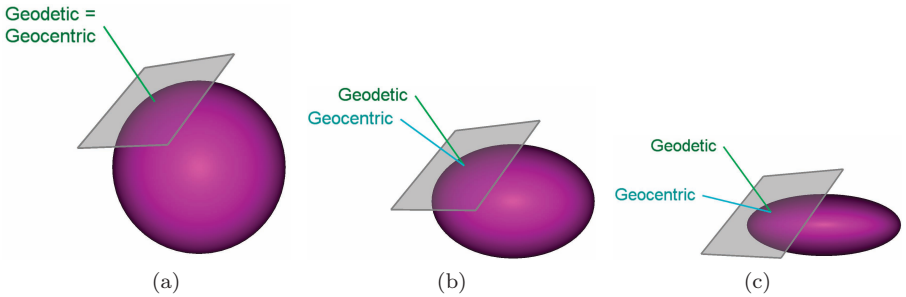



**Figure 2.5.** Geodetic versus geocentric surface normals. The geocentric normal diverges from the geodetic normal as the ellipsoid becomes more oblate. All figures have a semimajor axis = 1. (a) Semiminor axis = 1. (b) Semiminor axis = 0.7. (c) Semiminor axis = 0.4.

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

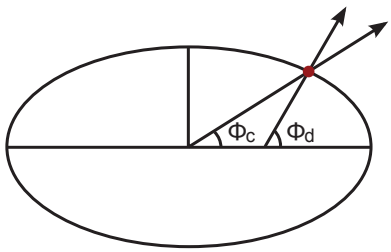



**Figure 2.6.** Comparison of geodetic latitude, $\phi_d$, and geocentric latitude, $\phi_c$.

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 $\lambda$, geodetic latitude by $\phi$, and height by $h$, so a $(longitude, latitude, height)$-tuple is denoted by $(\lambda, \phi, h)$. 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 $(\lambda, \phi, h)$ 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.

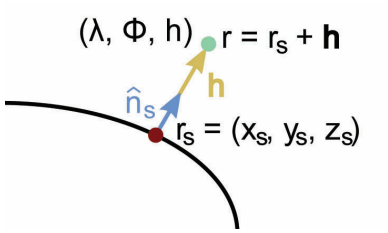



**Figure 2.7.** The geographic point $(\lambda, \phi, h)$ is converted to WGS84 coordinates by using the surface normal, $\hat{\mathbf{n}}_s$, to compute the surface point, $r_s$, which is offset by the height vector, $\mathbf{h}$, to produce the final point, $r$.

Given the surface position $(\lambda, \phi)$, 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 $\gamma$, $x_s$, $y_s$, 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 $\gamma$; if we compute $\gamma$, 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 $\gamma$ by substituting Equation (2.8) into this equation, then isolating $\gamma$:

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

Since $\gamma$ 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 $\gamma$ 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 $(\lambda, \phi)$ 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

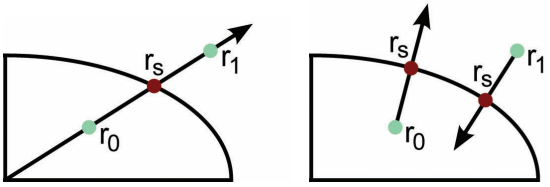



**Figure 2.8.** Scaling two points, $r_0$ and $r_1$, to the surface. (a) When scaling along the geocentric normal, a vector from the center of the ellipsoid is intersected with the ellipsoid to determine the surface point. (b) When scaling along the geodetic normal, an iterative process is used.

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 $\beta$ 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

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

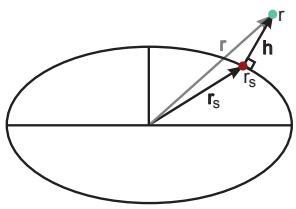



**Figure 2.9.** Computing $r_s$ given $r$ using the geodetic normal.

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, $\alpha$. In order to determine $\alpha$, 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):

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

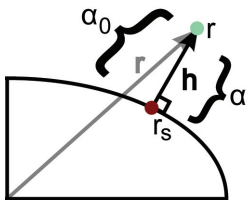



**Figure 2.10.** The Newton-Raphson method is used to find $\alpha$ from the initial guess, $\alpha_0$.

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

We solve for $\alpha$ 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, $\alpha_0$, for $\alpha$, 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 $\beta$ 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

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

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

Given our guess for $r_s$ and $\hat{\mathbf{n}}_s$, we can now determine $\alpha_0$. The unknown $\alpha$ scales $\hat{\mathbf{n}}_s$ to produce the height vector, $\mathbf{h}$. Our initial guess, $\alpha_0$, 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 $\alpha$:

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

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

We iterate to find $\alpha$ 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 $\alpha$ is found. Otherwise, a new $\alpha$ is computed:

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

Iteration continues until $S$ is sufficiently close to zero. Given $\alpha$, $r_s$ 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;
```