TerraVis: A Stereoscopic Viewer for Interactive Seismic Data Visualization

Justin W. Stoecker
University of Miami, justin.w.stoecker@gmail.com

Follow this and additional works at: https://scholarlyrepository.miami.edu/oa_theses

Recommended Citation
https://scholarlyrepository.miami.edu/oa_theses/240
A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science

TerraVis: A Stereoscopic Viewer for Interactive Seismic Data Visualization

Justin W. Stoecker

Approved:

Victor J. Milenkovic, Ph.D.
Professor of Computer Science

Terri A. Scandura, Ph.D.
Dean of the Graduate School

Hüseyin Koçak, Ph.D.
Professor of Computer Science

Ubbo Visser, Ph.D.
Professor of Computer Science
Accurate earthquake prediction is a difficult, unsolved problem that is central to the ambitions of many geoscientists. Understanding why earthquakes occur requires a profound understanding of many interrelated processes; our planet functions as a massive, complex system. Scientific visualization can be applied to such problems to improve understanding and reveal relationships between data. There are several challenges inherent to visualizing seismic data: working with large, high-resolution 3D and 4D data sets in a myriad of formats, integrating and rendering multiple models in the same space, and the need for real-time interactivity and intuitive interfaces. This work describes a product of the collaboration between computer science and geophysics. TerraVis is a real-time system that incorporates advanced visualization techniques for seismic data. The software can process and efficiently render digital elevation models, earthquake catalogs, fault slip distributions, moment tensor solutions, and scalar fields in the same space. In addition, the software takes advantage of stereoscopic viewing and head tracking for immersion and improved depth perception. During reconstruction efforts after the devastating 2010 earthquake in Haiti, TerraVis was demonstrated as a tool for assessing the risk of future earthquakes.
Acknowledgments

This thesis would not be possible without the support of many individuals to whom I am extremely thankful. I would like to acknowledge Professor Victor Milenkovic, my thesis advisor, for guiding my efforts and providing invaluable feedback. I would also like to thank Professor Timothy Dixon for the opportunity to work on such an exciting project, for supporting me financially, and for providing resources that made the work possible. I am grateful to Professor Hüseyin Koçak for his inspiring ability to bring people together to collaborate on larger projects, and also to Professor Burton Rosenberg for providing ideas and comments throughout the duration of this work. I would like to acknowledge Professor Ubbo Visser for joining the defense committee, especially given short notice, and for all members of my thesis committee I am appreciative of the patience in evaluating my efforts and providing me the opportunity to defend. I would also like to thank RSMAS Ph.D. student Scott Baker for his help in understanding many of the data sets and geology concepts. Finally, I would like to extend my appreciation to the Department of Computer Science for its continued support and graduate assistantship.
Contents

List of Figures viii

List of Tables xi

1 Introduction 1
   1.1 Contributions .............................................. 2
   1.2 Organization .............................................. 3

2 Background 4
   2.1 Earthquake Data Visualization ............................... 4
   2.2 NASA World Wind ......................................... 6
   2.3 GEON ..................................................... 8
   2.4 Google Earth ............................................... 9
   2.5 Other Visualizations ..................................... 10
   2.6 Summary ................................................ 11

3 Design 14
   3.1 Architecture ............................................... 15
   3.2 Java and OpenGL ....................................... 17

4 Digital Elevation Models 19
   4.1 Data Loading ............................................. 20
A.2 Vertex Generation ........................................... 119
A.3 Triangle Indexing ........................................... 122

Bibliography .................................................. 129
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>USGS ShakeMap</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>NASA World Wind</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>GEON OEFView</td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>Google Earth</td>
<td>10</td>
</tr>
<tr>
<td>2.5</td>
<td>Nine Point Five</td>
<td>11</td>
</tr>
<tr>
<td>3.1</td>
<td>TerraVis execution flow</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>TerraVis library dependencies</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>Digital elevation model of Hawaii</td>
<td>20</td>
</tr>
<tr>
<td>4.2</td>
<td>Digital elevation model loading/importing pipeline</td>
<td>21</td>
</tr>
<tr>
<td>4.3</td>
<td>A 4 × 6 DEM grid</td>
<td>22</td>
</tr>
<tr>
<td>4.4</td>
<td>Resampling with two DEMs</td>
<td>26</td>
</tr>
<tr>
<td>4.5</td>
<td>Interpolating elevation for a point</td>
<td>27</td>
</tr>
<tr>
<td>4.6</td>
<td>DEM vertex format</td>
<td>29</td>
</tr>
<tr>
<td>4.7</td>
<td>DEM vertex normal sampling</td>
<td>32</td>
</tr>
<tr>
<td>4.8</td>
<td>Haiti InSAR texture overlay</td>
<td>33</td>
</tr>
<tr>
<td>4.9</td>
<td>InSAR texture coordinate mapping</td>
<td>34</td>
</tr>
<tr>
<td>4.10</td>
<td>Vertex usage in a DEM grid without indexing</td>
<td>37</td>
</tr>
<tr>
<td>4.11</td>
<td>DEM mesh triangle indexing</td>
<td>39</td>
</tr>
<tr>
<td>4.12</td>
<td>DEM mesh triangle strip</td>
<td>40</td>
</tr>
</tbody>
</table>
4.13 System and GPU memory usage in rendering pipeline .......................... 42
4.14 Scan-line ordering of triangles ..................................................... 44
4.15 Post-transform vertex caching in a DEM mesh with scan-line ordering 45
4.16 Post-transform vertex caching in a DEM mesh with optimal ordering 46
4.17 Graphics pipeline ................................................................. 47
4.18 DEM relief shading with gradients .............................................. 50
4.19 Comparison of shading DEM shading stages ................................. 53

5.1 Earthquake catalog loading ......................................................... 56
5.2 Visualization of ANSS catalog .................................................... 57
5.3 Earthquake catalog mesh vertex format ....................................... 59
5.4 Earthquake catalog viewing window ......................................... 60
5.5 Modified binary search for earthquake event times ......................... 61
5.6 Focal mechanism diagrams ....................................................... 64
5.7 Focal mechanism shading by seismic wave forces ......................... 65
5.8 Southern California focal mechanism visualization ....................... 66
5.9 Focal mechanism vertex format ................................................. 67
5.10 Fragment shading focal mechanisms using nodal plane normals ...... 68

6.1 Strike-dip-rake block model ....................................................... 70
6.2 Slip distribution rendering in MATLAB and TerraVis .................... 72
6.3 Slip distribution array ............................................................ 73
6.4 Loading a slip distribution ......................................................... 74
6.5 Slip distribution vertex format ................................................. 75
6.6 Vertex coloring of a slip distribution mesh .................................. 76

7.1 Isosurface rendering of a volume of Gutenberg-Richter b-values from Hawaii .......................................................... 79
7.2 Direct volume rendering of b-values .......................................... 81
List of Tables

9.1 Depth to IR lights using FOV calculation . . . . . . . . . . . . . . . . 106
9.2 Comparison of depth calculations by technique . . . . . . . . . . . . . 107
Chapter 1

Introduction

Accurate earthquake prediction has long been central to many geoscientists’ ambitions. Achieving this goal requires a profound understanding of a complex system with many interrelated processes. Looking at the past may be the only way to glean some information on how such a system works and provide an insight into the causes of one the most catastrophic events. Massive volumes of geologic data already exist; however, without advanced methods of processing such data, it is simply too dense to comprehend or use in scientific study.

As a field, visualization is growing rapidly and has the aim of providing researchers across all disciplines with tools and techniques for deeper analysis of their data. The work of this thesis focuses on providing geophysicists visualization and interactivity with geologic data sets. A large amount of geologic data is spatial in nature, making the visual medium effective for exploring the data. The three-dimensional aspect of geologic data also provides an opportunity to apply interesting methods of viewing and data interaction.

One of the primary motivations for this work was the magnitude 7.0 earthquake that struck Haiti in January of 2010. Among engineers and scientists, there was great deal of discussion concerning the risks of rebuilding the capital of Haiti, Port-au-
Prince, at its existing location. This discussion highlighted the urgency and demand for tools to better assess the risks of future earthquakes. An effective visualization can not only be useful in helping scientists understand their data, but also in influencing and communicating ideas to policy-makers.

1.1 Contributions

This thesis describes a visualization program, TerraVis, capable of rendering different types of geologic data integrated into a single interactive scene. The purpose of this research is not to produce a commercial-grade visualization tool that can handle generic data presented in many different formats; TerraVis is a platform for experimenting with visualization techniques. The contributions of this work focus on visualizing specific data models, viewing data interactively, and exploring issues that are useful in assessing the risks of earthquakes:

- A pipeline for processing, constructing, and rendering triangle mesh representations of digital elevation models. Digital elevation models are used to georeference all other data models used in the system.

- Scalable and efficient rendering of massive earthquake data. Users can selectively view earthquake events within a window of time that ranges from seconds to decades. Slip distributions may also be rendered alongside earthquake catalogs.

- Robust visualization of volumetric data using direct volume rendering. An interface is provided for interactively changing volume transfer functions, allowing users to experiment with the optical properties assigned to features in data sets.

- Improved depth perception through stereoscopic 3D viewing and head tracking. Depth perception is especially useful for viewing dense clusters of earthquakes
and volume data. The system is highly accessible and has no dependence on expensive hardware.

1.2 Organization

Chapter 2 explores some of the current techniques in visualizing geologic data models. Some examples of existing applications of visualization to the field of geology are mentioned. Other frameworks and tools for geologic data visualization are discussed.

Chapter 3 covers the architecture and major components of TerraVis. This chapter emphasizes the software engineering aspects of the project, and describes the dependencies and layout of the program.

Chapters 4, 5, 6, and 7 each explore a particular data model that is visualized by TerraVis. For each of these chapters, a brief introduction is provided on the relevance of the data. Following the background, details are provided concerning the complexities in loading the data, processing it as a visible object, and rendering it. Algorithms in the form of pseudocode are provided throughout each chapter as necessary.

Chapters 8 and 9 are focused on the supplementary elements of TerraVis: stereoscopic viewing and head tracking. Chapter 8 provides introductory material on the principles of stereo viewing before describing how it is implemented. Chapter 9 includes an overview of the hardware used to perform head tracking, methods of calculating a viewer’s head position, and how it is applied to rendering.

Finally, Chapter 10 discusses the significance of this work. Ideas and recommendations for extending this project are provided in the final section.
Chapter 2

Background

This chapter will introduce some of the techniques for visualizing different types of seismic data. Not all of these methods are integrated into TerraVis, but they provide context for visualizations described in future chapters. In addition, other existing tools and frameworks for seismic data visualization are described.

2.1 Earthquake Data Visualization

Some of the most basic measurements used to study earthquakes are frequency and magnitude. The magnitude of an earthquake refers to its intensity, and is measured according to the Richter Magnitude Scale\(^1\). The magnitude is related to the amplitude of an earthquake recorded by seismographs. A network of seismograph stations can be used to improve the understanding of a particular event: location, magnitude, time and duration can be determined from several seismographs. With higher magnitude earthquakes, the loss of life and damage to civil infrastructures becomes more pronounced. This makes the use of magnitude, and the frequency of high magnitude earthquakes, especially relevant.

\(^1\)http://earthquake.usgs.gov/learn/topics/richter.php
The U.S. Geological Survey (USGS) and National Earthquake Information Center (NEIC) provide vast amounts of data accessible to researchers. One of the more conventional views of seismograph data is as a two-dimensional waveform plot. These plots can be compared directly and used to evaluate seismic wave properties. Seismic waves can also be numerically simulated, which is useful for experimenting with hypothetical events and preparing hazard rescue plans. Tung-Ju Hsieh provides more details on the visualization of waveform data and data obtained through numerical simulation [9].

Another form of earthquake data is historical catalogs. Massive collections of recorded earthquake events are stored in catalogs. Events in a catalog often include attributes such as time, duration, magnitude, latitude and longitude, depth, and number of recording stations. Using some frame of reference, such as a digital elevation model, these earthquakes can be geo-referenced and rendered. A common way to view earthquake catalogs is using globe or map overlays where circles are drawn at event epicenters. In three dimensions, a point or view-aligned circle at the earthquake hypocenter may be used to indicate the origin of the event.

Surface deformation measurements can be used to visualize the effects of an earthquake. Ground displacement maps texture or color the surface of a digital elevation model according to the amount of movement. Interferometric synthetic aperture radar (InSAR) uses satellite radar imagery to measure changes in surface elevation, and can be used to illustrate elevation changes in 3D [19]. The USGS also offers ShakeMap, a tool that provides maps of “ground motion and shaking intensity following significant earthquakes”\(^2\). These maps are available nearly in real-time, and can be downloaded as images or viewed in software such as Google Earth. This data is accessible not only for areas in the United States, but is also constructed from reports from other countries. An example shake map is shown in Figure 2.1.

\(^2\)http://earthquake.usgs.gov/earthquakes/shakemap/
Figure 2.1: Shake map for Spirit Lake, WA from a 4.3 magnitude earthquake. Only light to moderate shaking was perceived at the surface.

2.2 NASA World Wind

World Wind\(^3\) is a project initiated by the NASA Ames Research Center. World Wind provides an interactive viewer that presents a virtual globe of Earth (or other planets and moons) upon which other data sets can be overlaid. Terrain data includes satellite imagery from several sources including NASA’s Blue Marble project and the USGS; high-resolution imagery becomes visible when the viewer zooms in closer to the Earth.

\(^3\)http://worldwind.arc.nasa.gov/java/
While World Wind was originally written in C# and used Direct3D for rendering, the newest version is open-source, written in Java, and makes use of the Java OpenGL Bindings (JOGL); this change was an effort to increase the platform-independence of the tool. Technically, World Wind is an SDK and not a stand-alone visualization tool. Developers must write modules that build upon World Wind to provide useful data visualization, as the core SDK does not provide an interface for loading and rendering data sets. An extensive collection of demos are provided on NASA’s website for World Wind.

![Analytic Surface Demo](image)

Figure 2.2: NASA World Wind visualization of a scalar field of random altitudes over the Gulf of Mexico and Florida.

One of the greatest strengths of World Wind is its extensive API that can handle geographic data in a robust manner. However, it does not offer any immediately useful applications without developer intervention. Also, the view is constrained to a globe perspective or flat map projection, which makes it better suited to visualizing data sets that span over a wide area. It is possible to zoom into a specific location,
but demos emphasize surface overlays, surface extrusions (such as buildings in an urban area), and aerial data such as weather.

### 2.3 GEON

In contrast to World Wind, the Geosciences Network (GEON) project provides tools that are developed with specific data models in mind. The stated intention of the GEON project is to construct a "cyberinfrastructure in support of an environment for integrative geoscience research"\(^4\). The major focus of GEON is integrating data sets, particularly 3D and 4D, into a single cohesive model.

![Figure 2.3: GEON OEFView displaying earthquakes and tomography.](image)

The GEON project is built around the *OpenEarth Framework (OEF)*, a collection of software and libraries that act as an interface and database to access geologic data

\(^4\)http://www.geongrid.org/index.php/about/
sets. The OEF attempts to address one of the major problems in geosciences (and other fields): the integration of multi-disciplinary data sets. Web services are utilized to filter and collect data from a variety of sources. This framework allows many data sets of varying formats to be read and loaded into a visualization in real-time.

Rendering of data sets is performed interactively, and may be controlled by the user. Currently, there are two visualization applications within the GEON project: \textit{OEFView} and \textit{GEON IDV}. OEFView builds on top of NASA World Wind, and is capable of displaying data as points, lines, surfaces, or volumes (polygonal isosurfaces) inside the same coordinate space. The GEON Integrated Data Viewer (IDV)\footnote{http://geon.unavco.org/unavco/IDV_for_GEON.html} is an alternative system that provides visualization of more data types including digital elevation models, earthquakes and focal mechanisms, tomography, InSAR imagery, and mantle convection models.

\section*{2.4 Google Earth}

Google Earth\footnote{http://www.google.com/earth/index.html} is software similar to NASA World Wind that presents a virtual globe users can view. Most data is viewed as layers overlaid on top of the globe, as in World Wind. Google Earth integrates with Google Maps to provide road names, annotations, picture links, places of interest, street views, and other features useful for more common usage. However, Google Earth also provides the KML file format to allow users to add custom content. The USGS offers a few data types in KML files, including real-time earthquakes (and catalogs), plate boundaries, shake maps, and faults.
2.5 Other Visualizations

Nine Point Five⁷ is an online, interactive browser for viewing earthquakes overlaid on top of a 3D globe. The visualization pulls earthquake data directly from the U.S. Geological Survey (USGS) and uses satellite imagery provided by NASA. Users can view earthquakes in three different ways: particle clouds (see Fig. 2.5), lines, and rings. It also has a filter for earthquake magnitude and a adjustable time range within which earthquakes are visible. The most unique aspect of this visualization is that it is entirely browser-based, and is powered by WebGL, making it highly accessible and easy to share.

⁷http://www.ninepointfive.org/
Another web-based visualization is provided by Lars Grammel, of the University of Victoria in Canada. Grammel’s visualization\(^8\) displays earthquakes as circles drawn on top of 2D terrain provided by Google Maps. The visualization provides more flexible viewing of events: users can sort earthquakes by time (decades), casualties, magnitude, and continent. Several variations of this type of visualization exist online.

\textbf{2.6 Summary}

Older visualization methods in the geosciences, such as waveform plots, are static and often two-dimensional. As graphics hardware has becoming increasingly powerful and accessible in recent years, the current trend for modern tools is to present 3D and 4D data sets. In addition, the integration and rendering of multiple data sets in the same space is critical.

\(^8\)http://web.uvic.ca/~lgrammel/blog/earthquake-visualization/
In some cases, tools attempt to abstract geologic data types according to how each can be visualized. Polygonal meshes or grids tend to be used for digital elevation models. Volumetric data is typically viewed using polygonal isosurfaces. Simple primitives such as points and lines are often used for earthquakes and faults, respectively. Any type of imagery, such as InSAR and ShakeMap, is usually draped over a digital elevation model as a texture.

Interactivity is another important component of modern visualization software: rendering a model in MATLAB is useful, but it’s even more desirable to manipulate the view of the model in real time. Some of the more prominent tools mentioned above, such as NASA World Wind and Google Earth, focus heavily on providing the highest degree of interactivity. By providing a globe view, users can view data from the perspective of a satellite down to street level. This type of visualization is highly effective for viewing large-area data sets; however, it is not well-suited to viewing subterranean geometry.

Another challenge in visualizing geologic data is the myriad of file formats. Nearly every software has its own formats, and entire projects such as the OpenEarth Framework are devoted to streamlining and managing data collections. However, such frameworks are also far from complete, and there is no truly standardized way to access geologic data yet. Part of the problem stems from the multi-disciplinary nature of research in the geosciences. Data formatting is a separate issue from the goal of effective visualization, but it demonstrates the importance of avoiding introducing new formats.

From the above observations, some of the basic requirements for an effective visualization tool begin to emerge. The key components appear to be 3D/4D visualization, interactivity, and integration of multiple data sets. As programs move into the 3D domain, one thing that can be improved upon is the generally poor interface a user experiences when navigating the scene. Not only has graphics performance improved
significantly in recent years, but hardware for producing stereoscopic viewing and even head tracking has become accessible; these are staples of a high-end visualization lab, and can now be incorporated into a tool that any researcher can use.
Chapter 3

Design

TerraVis is designed to be a modular, yet efficient, renderer of multiple data integrated into the same 3D scene. The program performs real-time rendering, meaning that the scene is repeatedly drawn in terms of frames per second; this style of rendering contrasts with offline rendering, where single images or frames are calculated over a long period of time. The execution of the program exhibits a standard rendering loop for real-time graphics applications, as shown in Figure 3.1. There are three primary stages in the lifespan of the program:

- **Startup**: The graphics configuration is set, resources are loaded, and a window is created for the program. The rendering loop and permanent modules like the user interface are initialized. Before the rendering loop starts, the program checks the system to ensure it meets graphics hardware requirements.

- **Render Loop**: The majority of the program’s execution time is spent updating, rendering, and displaying results to the screen. The *update* stage is where user interface components are refreshed and computations are performed. The *render* stage is where components are drawn, and is separated from the updating to make sure a scene can be rendered several times without affecting the world model.
• **Shutdown**: Once the user wishes to quit the program, some cleanup is necessary. Many of the resources used by TerraVis are stored in video memory, and need to be explicitly disposed. Once disposal is finished, the program can safely exit.

![Program execution flow of TerraVis](image)

Figure 3.1: Program execution flow of TerraVis. The program is *done* when the user requests the program exits.

### 3.1 Architecture

TerraVis follows a design pattern similar to model-view-controller. The core of the program is the *viewer* class, which stores the *world model* and *user interface* of the program. All data is stored in the world model and the user interface controls the manipulation of data. However, the view of the entire world model is not centralized into a single renderer. The viewer class delegates the updating and rendering of the world model to the individual elements of the world model. Likewise, the user interface also performs its own rendering.

Each data element of TerraVis is best viewed as its own model-view-controller object: it is compartmentalized, and the loading, construction, and rendering of data models is handled by separate components that produce a generic model called a *world object*. World objects are generally renderable meshes, such as a digital elevation model. To ensure the program remains interactive while data is being read and prepared, the loading of world objects is always done in a separate *loader thread*. 
Each world object has its own loader thread, and once it finishes constructing TerraVis is notified and the world object is added to the world model.

Because rendering is decentralized, the order and types of data loaded into TerraVis is flexible. The only requirement is that a digital elevation model is loaded first: all other data needs to be geo-referenced according to the dimensions of the digital elevation model. A digital elevation model can also have its transparency adjusted, meaning it must be rendered after opaque geometry. To address issues with blending geometry, all world objects have a rendering priority; the lower the value (relative to other world objects), the earlier it is rendered. Once a world object is loaded and added to the world model, all world objects are sorted according to rendering priority.

Figure 3.2: Library dependencies of TerraVis.

A number of libraries are used by TerraVis (Fig. 3.2). Some internal libraries were developed specifically for this project, and others are provided by third parties under open source licenses:

- JOGL, described in the following section, provides a Java binding to the OpenGL graphics API; all components that need hardware-accelerated rendering make use of this library.

- WiiRemoteJ is a pure Java library for accessing sensor data from the Nintendo Wii Console Remote.
• BlueCove is used to enable development of software that uses Bluetooth wireless radio communication. This library is required by WiiRemoteJ.

• NetCDF provides machine-independent data formats that are commonly used in the geological sciences.

• DVR is a direct volume rendering component that can be used to visualize volumetric data. It may function as a stand-alone program, but is used here as a module of TerraVis for rendering volumes inside the same space as other models.

• JSGL is a custom JOGL-based graphics utility library. It provides classes for vector and matrix math, 3D cameras, windowing, and other graphics-related purposes.

• JWiiT allows a program to connect to a Wii Remote for head tracking. The library can be used to query the current head position, if connected, as well as produce viewing matrices to render a virtual window.

### 3.2 Java and OpenGL

TerraVis is written primarily in Java. The use of Java makes TerraVis highly portable, and the system has been run on Microsoft Windows, Linux, and Macintosh OS X operating systems. However, standard Java does not provide a high-performance graphics API necessary for three-dimensional rendering. The most popular graphics APIs are Microsoft’s Direct3D and OpenGL\(^9\), managed by the Khronos Group.

Rendering in TerraVis is done using the OpenGL graphics API. The Java Bindings for OpenGL (JOGL)\(^10\) allows Java programs to access OpenGL for graphics

\(^9\)http://www.opengl.org/
\(^10\)http://jogamp.org/
programming. The graphics code in JOGL programs will look almost identical to that found in C or C++ OpenGL programs, as the API is automatically generated from C header files. This is one of the greatest strengths of JOGL, and it is quite easy to port OpenGL programs written in C or C++ to JOGL.
Chapter 4

Digital Elevation Models

A digital elevation model (DEM) is used to model the elevations of a surface. In the context of this project, elevation models indicate the topography and bathymetry of a geographic region. Elevation models only describe the bare surface of the Earth, and do not include man-made structures or vegetation. The information stored in DEMs is obtained with the use of many technologies such as aerial surveys, LIDAR, GPS, and others. This data is often freely available, and can be downloaded from government organizations such as the U.S. Geological Survey (USGS)\(^1\).

The following sections describe the steps in converting a digital elevation model to a renderable mesh. The process of reading file formats that describe DEMs, a standardized DEM representation, and the pipeline used to import and load DEMs as grids is covered in Section 4.1. Section 4.2.1 covers how the grid of points in a DEM is converted to 3D vertices; these vertices are stitched together as triangles to form a single mesh, as described in Section 4.2.4. Finally, in Section 4.3, the shader programs used to render the mesh are explained.

\(^1\)http://ned.usgs.gov/
4.1 Data Loading

A DEM is the foundation on which other data TerraVis supports is geo-referenced, and must be loaded before other data. TerraVis loads DEMs that are represented as regular grids of three-dimensional points. Usually, points are positioned by latitude, longitude, and elevation. The format for such a DEM may be very simple, as points are spaced at regular intervals; however, several different file formats do exist to store DEM data. These files may be ASCII or binary, and may come with a header file that provides further details about the grid dimensions. The design shown in Fig. 4.2 can be used to extend TerraVis to handle DEMs in any format by writing an additional importer implementation.

Once a standard DEM object is created from an importer, it may be converted into a DEM mesh. The mesh object contains only the data needed to render a DEM. The import pipeline is only intended to be used on new data sets, as raw files can be hundreds of megabytes in size and contain millions of points. Once a .dem file
Figure 4.2: The stages involved in importing or loading a DEM mesh. File formats are in blue; utility classes are in green; data classes are in red. The output of any DEM importer is a standardized DEM object, which can be converted to a mesh.

is created, it can be quickly loaded in a few seconds instead of a few minutes. Each .dem file stores the computation of vertex positions, normals, and texture coordinates; however, triangle indices are needed to render the mesh as a solid surface. These indices are not stored in .dem files: the indices are quick to compute, and their efficient storage depends on the client GPU’s video memory limitations.

TerraVis provides importer implementations for two different file formats which are described below. Before introducing how these grids are imported, the desired format for the standardized DEM object is described.

4.1.1 Standard DEM Objects

The purpose of the DEM class is to standardize the format of a digital elevation model so that a mesh can be easily constructed from it. The vertex positions stored in a DEM mesh are calculated from the (lon, lat, elevation) points of this object. The DEM takes advantage of regularly spaced grids by only storing elevation data to reduce memory usage.

The only information needed to construct the 3D model is coordinates for a grid corner, the latitude and longitude step along the grid, and the number of vertices along side of the grid. A DEM object contains the following fields:
Figure 4.3: Top-down view of a DEM with 24 elevations organized into 4 rows and 6 columns. The numbers indicate the positions of the elevations in the array, which are assigned in row-major order. Lines show the outlines of triangles for the mesh. The lower-left corner is the \((\minLon, \minLat)\) point in the grid; the upper-right is the \((\maxLon, \maxLat)\) point. When mapped to 3D, the origin is at the lower-left.

- **Rows**: the number of rows in the grid, which is the same as the number of changes in latitude.
- **Cols**: the number of columns in the grid, which is the same as the number of changes in longitude.
- **LatLonBox**: each DEM has upper and lower bounds on its longitude, latitude, and elevation values. A \(\text{LatLonBox}\) is used to contain information about these bounds.
- **Elevations**: a 1D array of elevations in meters for every point in the grid. The values are in row-major order, so the elevation \(P_{i,j} = \text{elevations}[i \times \text{cols} + j]\).
- **LatStep**: change in degrees latitude between each pair of adjacent points in the grid. A point \(P_{i,j}\) has latitude \(\text{Lat}_{0,0} + i \times \text{latStep}\). 

\[\text{Lat}_{0,0} = \minLat, \quad \text{Lon}_{0,0} = \minLon, \quad \text{Lon}_{\text{max},0} = \maxLon, \quad \text{Lat}_{0,\text{max}} = \maxLat\]
• **LonStep**: change in degrees longitude between each pair of adjacent points in the grid. A point \( P_{i,j} \) has longitude \( Lon_{0,0} + j \times \text{lonStep} \).

### 4.1.2 XYZ Grids

The National Geophysical Data Center (NGDC) offers online tools, such as the GEO-DAS Grid Translator\(^{12}\), to obtain DEM grids from many regions of the world. One file format this data can be downloaded in is .xyz, which is stored in ASCII and easy to read. Each line in the file corresponds to a point and has longitude (degrees), latitude (degrees), and elevation (meters) separated by whitespace. For example, the first three points in a DEM containing Haiti:

```
Listing 4.1: Sample .xyz file format

1 -75.000000 21.000000 -1555.0
2 -74.983333 21.000000 -1515.0
3 -74.966667 21.000000 -1448.0
4 ...
```

A negative value of longitude indicates degrees west, and a negative value of latitude indicates degrees south. Points are sorted by decreasing latitude first, then increasing longitude second. The first point in the file is always the \((\text{minLon}, \text{maxLat})\) point of the grid. The file must be processed to calculate the number of rows, columns, and the \( \text{LatLonBox} \). There is no header information in the file, so an additional step is required to import XYZ DEMs. Since the values are sorted by decreasing latitude, the number of rows can be found by counting the number of times latitude changes.

The points in the file are already in the correct order for the \textit{DEM object}, so the elevations could be read one-by-one into a variable-size list which is then later copied into an array. However, some points may have an unknown elevation and must be treated differently. Any elevation with an unknown elevation cannot be included in the min and max elevation bounds of the DEM; a safe null value is typically lower

\(^{12}\text{http://www.ngdc.noaa.gov/mgg/gdas/gd_designagrid.html}\)
than the deepest trench or greater than the tallest point on Earth. This must be
manually set by a user, as there is no way of determining this value otherwise.

Algorithm 1 ParseHeaderDimensions(nullValue)

1: rows, cols, points ← 0
2: minLon, minLat, minEle, prevLat ← ∞
3: maxLon, maxLat, maxEle ← −∞
4: while (line ← nextLine()) ≠ EOF do
5:   lon ← split(line)[0]
6:   lat ← split(line)[1]
7:   ele ← split(line)[2]
8:   if lat ≠ prevLat then
9:     rows ← rows + 1
10:   end if
11:   prevLat ← lat
12:   points ← points + 1
13:   UpdateMinMax(minLon, maxLon, lon)
14:   UpdateMinMax(minLat, maxLat, lat)
15:   if ele ≠ nullValue then
16:     UpdateMinMax(minEle, maxEle, ele)
17:   end if
18: end while
19: cols ← points/rows

The split routine is any operation used on strings to tokenize them by a delimi-
ter. In this case, it is intended to tokenize each line by whitespace to separate the
longitude, latitude, and elevation values and store them in an array. The UpdateMin-
Max routine is called to ensure the boundary values are correct when the algorithm
terminates.

Algorithm 2 UpdateMinMax(curMin, curMax, newVal)

1: curMin ← min(curMin, newVal)
2: curMax ← max(curMax, newVal)

After the dimensions of the grid are known, the data can be put into DEM object
format. TerraVis creates a separate header file containing the information from the
ParseHeaderDimensions routine. The program takes up to three steps to import an
XYZ DEM: create the header file (if it doesn’t exist), read the header file, and finally read elevations. When reading elevations into the DEM object, any elevation that equals the null value is assigned the minimum known elevation minus one.

### 4.1.3 NetCDF Grids

A second format supported by TerraVis is the GDAL\(^{13}\) .grd format of NetCDF, “a set of software libraries and machine-independent data formats that support the creation, access, and sharing of array-oriented scientific data”\(^ {14}\). The .grd importer uses the Java NetCDF library to read the binary data. Each DEM file separates data into variables that have attributes and dimensions.

The GDAL Geographics variable contains a few of the attributes that needed to be parsed from XYZ files:

- **Westernmost easting** is the minimum value of degrees longitude.
- **Easternmost easting** is the maximum value of degrees longitude.
- **Southernmost northing** is the minimum value of degrees latitude.
- **Northernmost northing** is the minimum value of degrees latitude.

The Band1 variable contains the elevation values for each point, and has two dimensions: \(x\) corresponding to the number of columns, and \(y\) corresponding to the number of rows. It also contains the fill value attribute which is assigned to points without a known elevation. Unknown elevations are assigned the DEM’s minimum known elevation minus one.

\(^{13}\)http://www.gdal.org/

\(^{14}\)http://www.unidata.ucar.edu/software/netcdf/
4.1.4 Resampling

Occasionally, high-resolution elevation data may be available for topography but not bathymetry; this was a problem when working with DEMs from Haiti. TerraVis requires regular grids, so the DEM cannot contain points with irregular spacing. In this case, it is necessary to resample and construct a new, higher-resolution grid that covers the area shared by both models.

Figure 4.4: A lower resolution grid is shown with red vertices. A new grid is constructed (white vertices) that has about twice the resolution. The new grid is positioned using the high-resolution DEM, which may not exactly fit the dimensions of the low-resolution DEM. For white vertices above sea-level (green area), high-resolution elevations are used directly; otherwise, the elevation is interpolated.

To resample, a new grid is created with latitude and longitude spacing that matches the high-resolution DEM. For each point in this new grid, the elevation is taken from the high-resolution DEM directly if it is available; otherwise, the elevation is interpolated by using four of the points from the low-resolution DEM. For a point $q$ in the new grid, the row of low-resolution points above (in latitude) is the top row and the row below is the bottom row. The four points in the top and bottom rows may not be coplanar, so the quad is split into four triangles (see Fig. 4.5).
Fig. 4.5: Resampling at point $q$ which lies between two rows of low-resolution points, the top and bottom rows ($t$ and $b$).

The vector $r$ is directed from the center of the quad, $c$, to $q$. The triangle that $q$ belongs to is found by checking the vectors from the center of the quad to the four points: $p_i$ is the vector to the right of $r$ (clockwise rotation), and $p_j$ is to the left (counter-clockwise rotation). It may be determined if a vector $v$ is to the right of $r$ by checking $v \times r \geq 0$. Once $p_i$ and $p_j$ are known, the elevation for $q$ is computed as shown in Algorithm 3.

**Algorithm 3** InterpolateElevation($q$, $p_i$, $p_j$, $c$)

1: $a \leftarrow (p_j \times r)/(p_j \times p_i)$
2: $b \leftarrow (p_i \times r)/(p_i \times p_j)$
3: $r2 \leftarrow p_i * a + p_j * b$
4: $elevation \leftarrow c.z + r2.z$
A linear combination of \( p_i \) and \( p_j \) is used to find \( r \) such that it has the same \( x \) and \( y \) values of \( q \), as shown in the following equations:

\[
a * p_i + b * p_j = r
\]

\[
p_j \times (a * p_i + b * p_j) = p_j \times r
\]

\[
a \times (p_j \times p_i) = p_j \times r
\]

\[
a * p_i + b * p_j = r
\]

\[
p_i \times (a * p_i + b * p_j) = p_i \times r
\]

\[
b \times (p_i \times p_j) = p_i \times r
\]

### 4.2 Mesh Construction

The *DEM object* must be converted into a mesh before it can be rendered. Each vertex in the mesh needs 3D coordinates as well as additional information for lighting and texturing. To make the mesh solid, the vertices must be also indexed as triangles. These steps require significant time to complete, as DEMs can contain millions of points.

Once a mesh is constructed, vertex data can be written to file in a binary format (.dem) used by TerraVis. This binary file can be efficiently read later to avoid most of the computation necessary to generate a mesh. The loading process is handled by a *DEM loader*, which is simply a thread that reads the .dem file and notifies any listeners when finished. The aforementioned importers can be run as separate programs to read the raw data, import it to a *DEM object*, create a mesh from the object, and immediately write the mesh to a .dem file.
4.2.1 Vertex Mapping

The vertex data used for a DEM mesh includes position, normal, and texture coordinate information. These values are interleaved in the same buffer. Vertex data is calculated on the CPU and stored in main memory until it can be transferred into a vertex buffer object (VBO). A VBO allows data to be stored in video memory and only needs to be uploaded once with static data. For this mesh, each vertex stores 8 floats of data (see Fig. 4.6), which marks the total size or stride between vertices.

![stride](image)

Figure 4.6: Vertex format of data stored in a vertex buffer object. The vertex, normal, and texture coordinate pointers are shown marked vp, np, and tcp respectively. For example, the start of the position data for a vertex vi is vp + stride * i. The stride is 32 bytes.

A programmer can arrange vertex data in any many ways, so OpenGL must know the locations of position, normal, and texture coordinate data inside the buffer(s). For DEMs, all vertex data is interleaved in a single buffer which provides peak performance. There are three OpenGL pointers that provide enough information to process the custom vertex format:

1. The **vertex pointer** indicates the starting byte of the position of the vertex. It can be set with a call to `glVertexPointer(size, type, stride, ptr)`. Here, each

http://www.opengl.org/registry/specs/ARB/vertex_buffer_object.txt
vertex has 3 position components \((x, y, z)\), so the \textit{size} is set to 3. The \textit{type} is \texttt{GL_FLOAT}, and the \textit{ptr} is set to 0 since the positions data is the first in the buffer.

2. The \textbf{normal pointer} indicates the start of the normal vector components for each vertex. It can be set with a call to \texttt{glNormalPointer(type, stride, ptr)}. Again, the \textit{type} is \texttt{GL_FLOAT}, and the \textit{ptr} is set to 12 since the normal data is located after 3 floats each of size 4 bytes.

3. The \textbf{texture coordinate pointer} indicates the start of the texture mapping coordinates for each vertex. It can be set with a call to \texttt{glTexCoordPointer(size, type, stride, ptr)}. Here, each vertex has two-dimensional texture coordinates, so the \textit{size} is set to 2. The \textit{type} is again \texttt{GL_FLOAT}, and the \textit{ptr} is set to 24 since the texture coordinates are located after 6 floats each of size 4 bytes.

Similar to textures, OpenGL must know the source of vertex data by binding the vertex buffer before the DEM is rendered. During rendering, the buffer pointers are set after the VBO is bound. All of the pointer calls use the same \textit{stride}, which is set at \(8 \times 4 = 32\) bytes.

As mentioned in Section 4.1.1, the input for a mesh is a model that stores points according to latitude, longitude, and elevation in meters. In TerraVis, digital elevation models are mapped into a rectangular region of 3D units where the origin is the \((\text{minLon}, \text{minLat})\) point. The \textit{x} axis is used for longitude; the \textit{z} axis is used for latitude; and the \textit{y} axis is used for elevation. Positions are mapped as follows:

\begin{algorithm}
\textbf{Algorithm 4} CalcCoords(lat,lon,ele)
\begin{algorithmic}[1]
\State \(x \leftarrow -(\text{lon} - \text{minLon}) \times \text{unitsPerDegLon}\)
\State \(y \leftarrow \text{ele} \times \text{unitsPerMeter}\)
\State \(z \leftarrow (\text{lat} - \text{minLat}) \times \text{unitsPerDegLat}\)
\end{algorithmic}
\end{algorithm}
The values *unitsPerDegLon*, *unitsPerMeter*, and *unitsPerDegLat* are constants. The *minLon* and *minLat* values are taken from the *LatLonBox* of the digital elevation model. The assumption is made that a degree latitude or longitude is constant and equal to roughly 111,325 meters; this is certainly not true in some regions, but the skewed perspective is not noticeable on most DEMs. Every DEM will be mapped to a region with a rectangular base, and all position data in TerraVis is converted in the same manner ensuring that objects are placed properly in relation to the DEM.

Algorithm 5 CalcVertexPositions

1: for i := 0 to length(elevations) do
2:   col ← mod(i, numCols)
3:   row ← i/numCols
4:   lat ← maxLat + row * latStep
5:   lon ← minLon + col * lonStep
6:   pos[i] ← CalcCoords(lat, lon, elevations[i])
7: end for

4.2.2 Normal Vectors

Once vertex positions are calculated, the normal vectors used in shading can be generated for the entire mesh. Each vertex has a single normal that is the average of the normals of each adjacent triangle. Unless a vertex is on an edge of the mesh, it has six adjacent triangles. Each vertex in the mesh visited and its normal vector calculated independently. For a vertex $v_i$, the row preceding it is called *north*, the row after is *south*, the column before is *west*, and the column after is *east*. The eight vertices adjacent to $v_i$ are labeled accordingly; given the manner in which triangles are oriented, the *northeast* and *southwest* vertices are never used.

To calculate a triangle normal given three CCW vertices, a standard cross product is used (note the result is not necessarily unit length): $\text{triNormal}(a, b, c) = (c - b) \times (a - b)$. The steps to calculate the normal vector for a vertex $v_i$ at row $y$, column $x$ is shown in Algorithm 6. If a vertex is in the interior of the mesh, all six adjacent
Figure 4.7: Six triangles are used by every vertex that is not on an edge.

triangles with be sampled and averaged; otherwise, the number of adjacent triangles will be 1 at the NW and SE corners, 2 at the NE and SW corners, or 3 on any edge.

\textbf{Algorithm 6} CalculateVertexNormal($x, y$)

1: adjacentTriangles $\leftarrow 0$
2: $n \leftarrow (0, 0, 0)$
3: sampleN $\leftarrow y \neq 0$
4: sampleS $\leftarrow y \neq \text{rows} - 1$
5: sampleW $\leftarrow x \neq 0$
6: sampleE $\leftarrow x \neq \text{cols} - 1$
7: \textbf{if} sampleN \textbf{then}
8: \hspace{1em} \textbf{if} sampleW \textbf{then}
9: \hspace{2em} add the NW triangle normal to $n$
10: \hspace{2em} \textbf{end if}
11: \hspace{1em} \textbf{if} sampleE \textbf{then}
12: \hspace{2em} add the two NE triangle normals to $n$
13: \hspace{1em} \textbf{end if}
14: \textbf{end if}
15: \textbf{if} sampleS \textbf{then}
16: \hspace{1em} \textbf{if} sampleW \textbf{then}
17: \hspace{2em} add the two SW triangle normals to $n$
18: \hspace{2em} \textbf{end if}
19: \hspace{1em} \textbf{if} sampleE \textbf{then}
20: \hspace{2em} add the SE triangle normal to $n$
21: \hspace{1em} \textbf{end if}
22: \textbf{end if}
23: $n \leftarrow n / \text{numAdjacentTris}$
24: Return $n / |n|$
4.2.3 Overlays and Texture Coordinates

Texture coordinates are used for mapping textures to a mesh. These coordinates are sometimes referred to as \textit{UV} coordinates when using 2D textures or, in OpenGL, \textit{STRQ} coordinates for up to four dimensions. For a DEM mesh, these coordinates are used only for 2D texture overlays, such as interferometric synthetic aperture radar (InSAR) data.

![Figure 4.8: InSAR overlay taken from the 2010 Haiti earthquake.](image)

Overlays are provided as two files: an image and a header. The header must contain information units per pixel on both the x and y axes, as well as the longitude and latitude of one corner of the image. TerraVis does not currently have a import interface (such as the one for DEMs) for overlays. Overlays assume the \textit{GDAL raster format}\footnote{http://www.gdal.org/frmt_various.html}, where the header is an \textit{ESRI world file (.pgw)} formatted in plain ASCII as six newline-separated values. A description of the values stored in each line of a .pgw file is shown in Listing 4.2.
Listing 4.2: GDAL .pgw file format

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pixel X size</td>
</tr>
<tr>
<td>2</td>
<td>rotation about the Y axis (usually 0.0)</td>
</tr>
<tr>
<td>3</td>
<td>rotation about the X axis (usually 0.0)</td>
</tr>
<tr>
<td>4</td>
<td>negative pixel Y size</td>
</tr>
<tr>
<td>5</td>
<td>X coordinate of upper left pixel center</td>
</tr>
<tr>
<td>6</td>
<td>Y coordinate of upper left pixel center</td>
</tr>
</tbody>
</table>

Lines 1 and 4 are used for the longitude and latitude step in degrees, respectively. Lines 5 and 6 indicate the \((\text{minLon}, \text{maxLat})\) corner of the overlay. The rotations are ignored, as they should always be 0 degrees. Noting that the bounds are for the pixel centers, not edges, the remaining bounds can be calculated as follows:

\[
\begin{align*}
\text{maxLon} &= (\text{imageWidth} - 1) \times \text{lonStep} \\
\text{minLat} &= (\text{imageHeight} - 1) \times \text{latStep}
\end{align*}
\]

Figure 4.9: Texture coordinates for 4 vertices of a mesh are assigned such that a small portion of the overlay is mapped to triangles.

Ultimately, the desired result is to paint an overlay on the surface of a DEM. The vertex texture coordinates must be assigned such that the pixels of the rendered DEM sample from the appropriate pixels in the overlay (Fig. 4.9). The difficulty in this task is that the overlay is unknown, and may occupy any space on the DEM.
The most common approach to texture mapping a mesh is to fit texture coordinates to a pre-defined texture. This method simply does not work here, as an arbitrary overlay can be used and it is unrealistic to create texture coordinate configurations for all possible overlays. The alternative approach is to create a texture to fit the texture coordinates of the mesh. For example, a quad might have coordinates \((0, 0), (1, 0), (1, 1), (0, 1)\). Any texture applied to this quad will be stretched to fit on that surface. This method is infeasible as well, because it would require an extreme resolution for the overlay to maintain detail.

The solution TerraVis uses is to assign vertex texture coordinates based on the vertex’s position relative to the dimensions of the DEM. The \((\text{minLon}, \text{minLat})\) corner of the DEM receives texture coordinates \((0, 0)\); the \((\text{maxLon}, \text{maxLat})\) corner of the DEM has texture coordinates \((1, 1)\); vertices in the middle are linearly interpolated. The key difference is that the texture coordinates are adjusted inside the fragment shader (see Section 4.3) using offsets determined by the overlay.

\[
\text{Algorithm 7 CalcVertexTexCoords}(row, col)
\]

1: \(tc.x \leftarrow \text{col}/\text{cols}\)
2: \(tc.y \leftarrow 1 - \text{row}/(\text{rows} - 1)\)

The \(uvScale\) and \(uvDelta\) fields are uniform shader variables. They only need to be set when an overlay is assigned, and are read-only memory for each pixel shader. Inside the fragment shader each pixel in the DEM calculates the final texture coordinates it will use to sample the overlay texture. Currently, only a single overlay can be used at a time. However, this approach is scalable to multiple overlays: the \(uvDelta\) and \(uvScale\) variables need only be made into arrays, and multiple texture samplers can be allocated inside the fragment shader.
Algorithm 8 CalcTexCoordsOffsets

1: lonRange ← (dem.maxLon − dem.minLon)
2: latRange ← (dem.maxLat − dem.minLat)
3: oMinX ← (overlay.minLon − dem.minLon)/lonRange
4: oMinY ← (overlay.minLat − dem.minLat)/latRange
5: oMaxX ← (overlay.maxLon − dem.maxLon)/lonRange
6: oMaxY ← (overlay.maxLat − dem.maxLat)/latRange
7: shader.uvScale ← (oMaxX − oMinX, oMaxY − oMinY)
8: shader.uvDelta ← (oMinX, oMinY)

4.2.4 Triangle Indexing

The last step in creating the DEM mesh is to connect all the vertices together as triangles. There are several ways to do this, but it’s crucial to realize the benefits and drawbacks of each approach. The digital elevation model mesh is typically the most expensive geometry to render in TerraVis, so a great amount of attention is given to optimizing the rendering of this mesh.

I’ll start with a comparison of memory requirements for various techniques, all of which utilize vertex buffer objects to store data in video memory. The most straightforward method of rendering the mesh is to instruct OpenGL to treat successive tuples of vertices in the VBO as triangles. This is inefficient, as most vertices will have to be repeated several times: using this technique, consider the number of times a given vertex is used in an \( m \times n \) grid. The frequency a vertex shows up in the VBO is shown in Figure 4.10.

For an \( m \times n \) mesh, one can find the number of interior vertices, \( i \), and edge vertices, \( e \), as follows:

\[
i = (m - 2)(n - 2)
\]

\[
e = 2(m - 2) + 2(n - 2) = 2(m + n - 4)
\]
Figure 4.10: The top-left and bottom-right vertices belong only to 1 triangle each (blue). The bottom-left and top-right vertices are shared by 2 triangles each (green). Vertices on the edges are shared by 3 triangles each (yellow), and every vertex that is in the interior of the grid is used 6 times (red).

There are 4 corner vertices, 2 of which are used twice. Each interior vertex is used 6 times, and each edge vertex is used 3 times (see Figure 4.10). The total number of vertices to be buffered:

\[
\text{numVertices}(m, n) = 6i + 3e + 2 \times 2 + 2
\]
\[
= 6(m - 2)(n - 2) + 6(m + n - 4) + 6
\]
\[
= 6(mn - 2m - 2n + 4 + m + n - 4 + 1)
\]
\[
= 6(mn - m - n + 1)
\]
\[
= 6(m - 1)(n - 1)
\]

If each vertex was only used once, only \(mn\) vertices would need to be stored. The difference, the number of duplicated vertices, is then:

\[
\text{extraVertices}(m, n) = 6mn - 6m - 6n + 6 - mn
\]
\[
= 5mn - 6(m + n - 1)
\]
The space required to store triangles using this approach is tremendous: it takes 5 times additional space for large meshes. To put this in perspective, a Haiti DEM TerraVis uses is a size 1620×2160. Each DEM vertex has eight floats \((x, y, z, n_x, n_y, n_z, tc_x, tc_y)\) for a total size of 32 bytes. If vertices are repeated to create triangles, this would take \(6 \times 1619 \times 2159 \times 32/1024^2 = 640.0\) MB. Without using VBOs, this is an unrealistic amount of data to repeatedly transfer to video memory. Even if VBOs are used, the total capacity of the video memory on an NVIDIA Quadro FX 1800 is 768 MB. That doesn’t leave much room in video memory for other geometry data, textures, screen buffers, and other resources needed by TerraVis or the operating system. The total memory in bytes to store a mesh using this technique:

\[
bytes(m, n) = 32 \times 6(m - 1)(n - 1)
\]

\[
= 192mn - 192(m + n - 1)
\]

A solution to the memory problem from the previous approach is to introduce an additional buffer that contains a series of vertex indices. Instead of reading directly from the vertex buffer, OpenGL will take successive tuples of indices and use the vertices at the indexed locations. A buffer storing indices is called a index buffer, and the data is stored in video memory just as with VBOs.

If an \(m \times n\) grid is split into rectangle cells, the number of cells is simply \((m - 1)(n - 1)\). Each cell is split into two triangles, and each triangle has three indices. The total number of indices needed for all the triangles in the mesh is therefore \(6(m - 1)(n - 1)\). At first glance, this may not appear to help with memory at all: not only do \(mn\) vertices need to be buffered, but also \(6(m - 1)(n - 1)\) indices must now be stored.

It’s important to remember that the size of an index is almost always smaller than a vertex. For smaller meshes, the \textit{unsigned short} data type can be used which is only 2 bytes: this allows up to \(2^{16} = 65,536\) values or a maximum grid size of 256×256.
Figure 4.11: OpenGL processes triangles by using indices from an index buffer that references vertices in a vertex buffer.

Most DEMs are a higher resolution, so unsigned integers, each 4 bytes, are used in TerraVis. Unsigned integers allow $2^{32} = 4,294,967,296$ values or a maximum grid size of $65,536 \times 65,536$. The total memory needed for an index buffered mesh with unsigned integers:

$$\text{bytes}(m, n) = 4 \times 6(m - 1)(n - 1) + 32mn$$

$$= 56mn - 24(m + n - 1)$$

Using the same Haiti DEM of size $1620 \times 2160$, the total memory required is about 186.8 MB. This is a significant improvement for realistic DEM sizes, and cuts the memory to about $56/192 = 29.2\%$ of the VBO-only approach.

### 4.2.5 Triangle Strips and Lists

A second approach to reducing memory usage when rendering a mesh is to utilize a rendering technique called triangle strips. The standard method of rendering triangles is using triangle lists; this is the method described above in which tuples of vertices (or indices) refer to triangles. In a triangle strip, each vertex (after the second vertex)
forms a new triangle with the previous two vertices (see Figure 4.12). Triangle strips can be the most memory efficient way to render triangles, as it has the potential to reduce the number of vertices or indices required to render $N$ triangles from $3N$ to $N + 2$.

![Figure 4.12: An example of a triangle strip. Only 8 vertices are needed to create a strip with 6 triangles. To continue this strip to a second row of triangles, the vertices $v_7$ and $v_1$ would be inserted after $v_7$ is first visited to form two degenerate triangles.](image)

Each row of a regular grid can be easily rendered as a triangle strip. However, it is much more efficient to render a mesh in a single draw command. To wrap a grid into a single triangle strip, *degenerate triangles* can be formed at the end of each row of triangles. A degenerate triangle is recognized as any triangle that has two vertices which are the same, and therefore has no area. Driver implementations of OpenGL can recognize degenerate triangles and cull them before they are rendered, causing little computational overhead.

Each row of triangles, of which there are $(m - 1)$, requires $2n$ vertices. All rows except the last, of which there are $(m - 2)$, also require 2 additional vertices to form degenerate triangles. The memory required to store a VBO with triangle strips is significantly less, as shown in the following equation.
\[ \text{bytes}(m, n) = 32(2n(m - 1) + 2(m - 2)) \]
\[ = 64mn - 64(n - m + 2) \]

For grids, it turns out that the repetition of vertices for each row of triangles causes regular triangle strips to take up more memory than indexed triangle lists; however, it’s still a better choice than non-indexed triangle lists. Using the 1620 × 2160 size mesh, the triangle strip approach would require about 213.6 MB. However, triangle lists can be indexed as well. In this case, the number of indices would be the same as the number of vertices used in non-indexed triangle strips. Additionally, the \( mn \) vertices are factored in, and the total memory requirements shift in favor of strips:

\[ \text{bytes}(m, n) = 4(2n(m - 1) + 2(m - 2)) + 32mn \]
\[ = 40mn - 8(n - m + 2) \]

Given the 1620 × 2160 size mesh, the total required memory using indexed triangle strips is at 133.5 MB. It can be concluded that indexed triangle stripping is the most memory efficient approach to rendering a triangle mesh. With reduced memory requirements, it is quicker to upload and easier to store the data needed to render the mesh.

### 4.2.6 Triangle Ordering Optimization

While indexed triangle strips appear to be a clear choice, memory and bandwidth aren’t the only two factors that should be considered. The order triangles are provided to the graphics hardware can also affect rendering performance. The graphics pipeline has vertex caches that are used to speed up rendering (see Figure 4.13). The first cache
is the *pre-transform vertex cache*, which is a large buffer used by VBOs. Optimizing for this cache is uncommon, and can only be done by sizing VBOs to fit the cache size, which comes with the higher cost of additional draw commands.

The second cache is the *post-transform vertex cache*, which stores vertices that have just been processed by the vertex shader stage in the pipeline. The post-transform cache is a first-in, first-out (FIFO) buffer that varies in size according to hardware. Most current graphics hardware can store 16 to 24 vertices, depending on the number of attributes (position, normal, texture coordinates, etc.) per vertex. Optimizing for this cache involves ordering triangle indices in such a way that new triangles will utilize the vertices that are already stored in this buffer. Only indexed rendering approaches can use this cache, as indices are used to determine if a vertex is in the cache.

![Diagram of GPU pipeline](image)

Figure 4.13: Vertex and index data is stored in video memory until rendering, when portions of the VBO can be copied over to the pre-transform cache. As vertices are processed by the vertex shaders, they are put into a FIFO buffer of limited size, the post-transform cache. As more triangles are rendered, any vertex index that matches a vertex in the cache can cause that vertex to skip re-processing. [4]
Slightly older rendering optimization techniques do take advantage of the vertex cache using indexed triangle strips [8]; the primary goal of such algorithms is reducing memory and bandwidth requirements, making triangle strips the best choice. However, higher data transfer rates and video memory sizes in modern graphics hardware makes these benefits less significant, especially when compared to vertex processing time. Thus, more recent papers focus on minimizing post-transform cache miss rates [5, 14]. While requiring slightly more memory than indexed triangle strips, indexed triangle lists are preferred for reducing cache miss rates.

Ultimately, there is a tradeoff between improved memory and bandwidth efficiency with indexed triangle strips and improved caching with indexed triangle lists. The DEM meshes in TerraVis are static in nature; the vertices are only uploaded once and do not need to change. For this reason, the bandwidth efficiency of triangle strips is meaningless. Furthermore, the difference in memory requirements between the two approaches is not enough to cause concern that mesh data will not fit in video memory. Therefore, DEM meshes in TerraVis use indexed triangle lists to take advantage of vertex caching.

The most straightforward indexed triangle list algorithm is to add vertex indices in scan-line order. Vertices are visited from top (first row) to bottom, left (first column) to right, as shown in Algorithm 9. When triangles are added in this order, only a few of the vertices will be cached.
Figure 4.14: Scan-line ordering of triangles used in Algorithm 9.

Algorithm 9  \textbf{IndexScanline}(buffer, rows, cols)
1: \textbf{for} $y := 0$ to $rows - 1$ \textbf{do}
2: \quad \textbf{for} $x := 0$ to $cols - 1$ \textbf{do}
3: \quad \quad $v_i \leftarrow x + y \times rows$
4: \quad \quad $v_e \leftarrow v_i + 1$
5: \quad \quad $v_s \leftarrow v_i + cols$
6: \quad \quad $v_{se} \leftarrow v_s + 1$
7: \quad \quad \textbf{IndexTriangle}(buffer, $v_i$, $v_s$, $v_e$)
8: \quad \quad \textbf{IndexTriangle}(buffer, $v_s$, $v_{se}$, $v_e$)
9: \quad \textbf{end for}
10: \textbf{end for}

Algorithm 10  \textbf{IndexTriangle}(buffer, $a$, $b$, $c$)
1: \textit{buffer.push}($a$)
2: \textit{buffer.push}($b$)
3: \textit{buffer.push}($c$)

As triangles are added along the first row, new triangles have 2 of their vertices in the post-transform cache, and the third vertex won’t have been processed yet (see Figure 4.15). Triangles in the following rows will reference vertices in the row above that have already been processed; these vertices will need to go through the vertex shader stage again, which can potentially be avoided.
Figure 4.15: Post-transform vertex caching, with cache size of 6, using simple scanline ordering. Green vertices have been cached. Yellow vertices have not been cached, but have also not yet been processed; red vertices are not in the cache and have been previously processed.

There are other orderings that can be used for generic meshes with varying success [1]. However, knowing that DEMs are constructed as a regular grid, there is an optimal approach for this specific geometry. An optimal approach is to pre-cache the top row of vertices and add triangles from top to bottom, but only below the cached vertices. This optimal approach and the following algorithm are described by Ignacio Castano [3]. Knowing the size of the cache, blocks of triangles from the mesh can be added without re-processing any vertex within that block. Each block has width $cachSize - 2$, and triangles are added in scanline order within that block. For the first row of a block, degenerate triangles are added to cache the top row of vertices. Successive triangles below these degenerate triangles will have two of their vertices cached and the third not yet processed. If the grid width is less than or equal to the block width, the entire mesh can be ordered such that no triangle is ever re-processed. More likely, the mesh will need to be split into multiple blocks, as shown in Figure 4.16d; in this case, only the vertices on the block boundaries may be re-processed.
Figure 4.16: Vertex caching, with cache size 6, using optimal grid ordering.

Algorithm 11 IndexOptimized\( (buf, x_0, x_1, y_0, y_1, width, cacheSize) \)

1: if \( x_1 - x_0 + 1 < cacheSize \) then
2: \hspace{1em} if \( 2(x_1 - x_0) + 1 > cacheSize \) then
3: \hspace{2em} for \( x := 0 \) to \( x_1 \) do
4: \hspace{3em} IndexTriangle\( (buf,x,x,x+1) \)
5: \hspace{2em} end for
6: \hspace{1em} end if
7: \hspace{1em} for \( y := 0 \) to \( y_1 \) do
8: \hspace{2em} for \( x := 0 \) to \( x_1 \) do
9: \hspace{3em} \( v_i \leftarrow (width + 1) \ast (y + 0) + (x + 0) \)
10: \hspace{3em} \( v_s \leftarrow (width + 1) \ast (y + 1) + (x + 0) \)
11: \hspace{3em} \( v_e \leftarrow (width + 1) \ast (y + 0) + (x + 1) \)
12: \hspace{3em} \( v_{se} \leftarrow (width + 1) \ast (y + 1) + (x + 1) \)
13: \hspace{3em} IndexTriangle\( (buf,v_i,v_s,v_e) \)
14: \hspace{3em} IndexTriangle\( (buf,v_e,v_s,v_{se}) \)
15: \hspace{2em} end for
16: \hspace{1em} end for
17: end if

### 4.3 Mesh Rendering

For digital elevation models, the desired visualization emphasizes variation in elevation across the surface. Therefore, the approach described below colors the mesh using
relief shading\textsuperscript{17} techniques. In particular, DEMs mesh renderings include elevation-based coloring, basic contour lines, and diffuse shading. The rendering of a DEM mesh is handled by shader programs that run on the graphics card.

The programmable graphics pipeline used in modern graphics hardware allows customized instructions to be written for various rendering stages. A vertex shader program overrides the default vertex transformations done by OpenGL, and is responsible for positioning vertices and any other per-vertex operations. A fragment shader program overrides the default pixel shading, and can be used for per-pixel operations. More recent hardware also includes a programmable geometry shader stage, but it is not utilized for the current DEM mesh. The stages of the graphics pipeline can be seen in Fig. 4.13.

Figure 4.17: Simplified graphics pipeline. Programmable vertex and fragment processing stages are shown in green.

### 4.3.1 Vertex Shader

The vertex shader for the mesh is very simple, as most of the computation is done in the fragment shading stage. The code for this program can be seen in Listing 4.3. As in every vertex shader, the program must output the transformed vertex position. This is done using the \texttt{ftransform()} function in GLSL, which transforms the vertex by the \texttt{modelview} and \texttt{projection} matrices. The vertex texture coordinates also need

\textsuperscript{17}http://www.reliefshading.com/
to be passed to the fragment shader, which is done on line 7. Finally, the fragment shader requires its world space position and normal vector for shading; these values are assigned to vertices, so they need to be interpolated between vertices of a triangle. The *varying* variables declared on line 1 can be accessed in the fragment shader, and are output in the vertex shader on lines 5 and 6.

**Listing 4.3: (relief.vs) GLSL vertex shader for DEMs**

```glsl
varying vec3 worldPos, normal;

void main()
{
  worldPos = gl_Vertex.xyz;
  normal = normalize(gl_Normal);
  gl_TexCoord[0] = gl_MultiTexCoord0;
  gl_Position = ftransform();
}
```

### 4.3.2 Fragment Shader

The fragment shader is where most of the hard work is done when rendering a DEM mesh. This program operates once for each fragment of the rasterized triangles. The code for this program is split into its functions, Listing 4.4 to Listing 4.8. At the top of the program, several *uniform variables* are declared; these are variables that can be located and used from the main program running on the CPU. For example, the *overlay* variable is a texture sampler that can be bound to a desired texture object; the *useContours* and *useOverlay* variables are simply booleans that can be toggled by the user.

Every fragment shader has a single purpose, which is to output a color for the fragment it is shading. The *main()* function of the fragment shader first calculates a 4-component color *(red, green, blue, alpha)* in the *reliefColor(float)* function. If contours or overlays are to be used, the color is modified by the respective functions *applyContours(vec4)* and *applyOverlay(vec4)*. Finally, the color is weighted by
lighting calculations in \texttt{applyLighting(vec4)} function, and the resulting color is output to the fragment.

Listing 4.4: (relief.fs) Main code of GLSL fragment shader

```glsl
uniform sampler2D overlay;
uniform bool useContours, useOverlay;
uniform vec2 uvScale, uvOffset;
uniform vec3 lightDir, w1, w2, l1, l2, l3;
uniform float ambient, alpha, minElevation, medElevation, maxElevation;

varying vec3 worldPos, normal;

void main()
{
vec4 color = reliefColor(worldPos.y);
if (useContours)
  color = applyContours(color);
if (useOverlay)
  color = applyOverlay(color);
  color = applyLighting(color);
  gl_FragColor = color;
}
```

The first function generates a color for each fragment based on its elevation. Recall that the vertex shader outputs its untransformed position as a varying variable \texttt{worldPos}; each fragment receives this variable as an interpolated position in world space, so the elevation can be retrieved. The \texttt{reliefColor(float)} routine, shown in Listing 4.5, acts as a piecewise function that assigns a color given an elevation \(y\). If an elevation is negative, it is below sea-level and assigned a shade of blue; if positive, it is assigned a color from green to yellow to orange as it approaches the maximum elevation in the DEM. The uniform variables \(w_1, w_2, l_1, l_2, l_3\) are the five base colors assigned to the deepest elevation through to the maximum elevation; the mapping of these colors is shown in Figure 4.18.
If an elevation is between two base colors, it receives a color that is linearly interpolated. There are three such “bins” that a fragment can fall into; one is assigned to fragments below sea-level, and two are assigned to fragments have sea-level. The additional bin for topography fragments is used to provides extra color variation above sea-level, and uses the \textit{medElevation} variable as a point of reference. Within the main application, this uniform variable is set to a third of the \textit{maxElevation}. Any fragments that are below the minimum elevation are made entirely transparent; these fragments correspond to points in a DEM that have unknown elevation. Finally, the alpha component of the color returned from this function is set to the value of the uniform variable \textit{alpha}. This allows the user to adjust the overall transparency of the mesh. The GLSL code for the relief coloring is shown in Listing 4.5.
Listing 4.5: (relief.fs) Relief coloring function

```cpp
vec4 reliefColor(float y)
{
    vec4 color;
    if (y < minElevation) {
        color = vec4(0.0, 0.0, 0.0, 0.0);
    } else if (y < 0.0) {
        color = vec4(mix(w1, w2, 1.0 - y / minElevation), alpha);
    } else if (y < medElevation) {
        color = vec4(mix(l1, l2, y / medElevation), alpha);
    } else {
        color = vec4(mix(l2, l3, (y - medElevation) / (maxElevation - medElevation)), alpha);
    }
    return color;
}
```

An additional feature that helps in capturing elevation in a DEM is contour lines. A contour line is a marker at regular elevation intervals; for example, lines might be marked every 500 meters. By counting the number of rings, one can determine the approximate elevation of all pixels between contour lines. The `applyContours(vec4)` function darkens fragments at elevations corresponding to where a contour line is drawn. This is done by taking the elevation of a fragment modulo a fixed interval. As shown on line 3 of Listing 4.6, elevations are taken modulo 0.02, which corresponds to 200 meters (TerraVis stores a constant of 1 meter = 0.0001 world space units). If the result is within 0.002 (20 meters), the fragment color is darkened by 40%. The effects of apply contour lines can be seen in Figure 4.19.

Listing 4.6: (relief.fs) Contour lines function

```cpp
vec4 applyContours(vec4 color)
{
    if (mod(worldPos.y, 0.02) < 0.002)
        color.rgb *= 0.6;
    return color;
}
```

If an overlay texture is to be applied to a DEM mesh, the `applyOverlay(vec4)` function is run for all fragments. As described in Section 4.2.3, the `uvScale` and `uvOffset` uniform variables are set by the application when an overlay is first loaded.
On line 3 in Listing 4.7, the final texture coordinates are calculated per fragment. The texture coordinates are then used to sample the overlay texture. If the sampled color from the texture is not transparent, its color is weighted by its opacity and mixed with the relief color to blend fragments along the borders.

Listing 4.7: (relief.fs) Overlay texture mapping function

```c
vec4 applyOverlay(vec4 color) {
    vec2 uv = (gl_TexCoord[0].xy - uvOffset) / uvScale;
    vec4 overlayColor = texture2D(overlay, uv);
    if (overlayColor.a > 0.0)
        color.rgb = overlayColor.rgb * overlayColor.a + color.rgb * (1.0 - overlayColor.a);
    return color;
}
```

The final step in shading fragments for a DEM mesh is to apply diffuse lighting. An arbitrary vector is used as the direction of a light source at infinity (a directional light in OpenGL terminology), which is stored in the uniform variable lightDir. The common technique of Phong shading [16] is performed per fragment with an interpolated normal vector. However, before calculating the amount of diffuse lighting, the normal vector is flattened (line 3, Listing 4.8) by scaling its y-component down. This is done to increase the contrast of shading, especially along mountain ridges in the DEM.

Listing 4.8: (relief.fs) Lighting function

```c
vec4 applyLighting(vec4 color) {
    vec3 n = normal * vec3(1.0, 0.25, 1.0);
    float diffuse = max(dot(normalize(n), normalize(-lightDir)), 0.0);
    color.rgb *= min(diffuse + ambient, 1.0);
    return color;
}
```
Figure 4.19: Effects of relief, contour, and lighting functions in the DEM relief fragment shader.
Chapter 5

Earthquake Models

Earthquake events are one of the key data types visualized by TerraVis. In particular, TerraVis is designed to render massive collections of events. The effects of earthquakes are treated as separate data, such as the slip distributions along the fault in Chapter 6 and InSAR imagery in Section 4.2.3. This section discusses modeling the position and intensity of earthquake events. There are two approaches described below: the more basic technique renders catalogs of earthquakes as individual spheres; a second approach renders the focal mechanism (so-called “beachball diagram”) of each earthquake.

5.1 Earthquake Catalogs

Dealing with individual earthquake events can be tedious and cumbersome, as there are many different formats and attributes that may be assigned to a particular event. A collection of earthquake events is oftentimes bound into what is referred to as a catalog. For a given catalog, the values for every event is in a standardized format; each event may be listed as a single line in a plain ASCII file, as all events share the same attributes and measurements (e.g. meters instead of kilometers).
The values for each line in a catalog are usually delimited by spaces or commas, and a catalog file may contain a header that indicates what each column of values represents. The catalog type recognized by TerraVis is provided by the Advanced National Seismic System (ANSS)\textsuperscript{18}. The output that TerraVis can process is described in the “readable catalog format”\textsuperscript{19}, an ASCII format with $n + 2$ lines, where $n$ is the number of events: the first line is a header of column labels, and the second line is a divider. This data can be retrieved for a desired area and data range, making it useful for visualizing different geographic regions.

A second catalog type that TerraVis can load is more geographically specific. The LSH catalog contains 433,166 relocated earthquake events from the southern California region \textsuperscript{18}. The file format is again plain ASCII with space-separated values; a detailed description and download for the LSH catalog can be found online\textsuperscript{20}.

### 5.1.1 Data Loading

For every earthquake catalog, the loading process in TerraVis begins by starting a separate \textit{Earthquake loader} thread that starts reading a header (.\texttt{catalog}) file. The header file is extremely simple and only contains two lines in plain ASCII. The first line indicates the path to the actual data file, and the second line indicates the catalog format. As mentioned before, there are currently only two recognized formats: ANSS (also called Standard) and LSH. Once the header file is read, the loader then delegates the loading to an \texttt{EarthquakeData} object, which wraps a generic catalog.

A catalog contains an array of individual events, each of which contain much more data than needed by TerraVis for visualization. At the very least, the \textit{latitude}, \textit{longitude}, \textit{depth}, and \textit{magnitude} attributes are required to render an event. Additionally, some events may have \textit{date} and \textit{time} attributes that can be used. This information is

\textsuperscript{18}http://www.ncedc.org/anss/
\textsuperscript{19}http://www.ncedc.org/ftp/pub/doc/cat1/catlist.1
\textsuperscript{20}http://www.rsmas.miami.edu/personal/glin/LSH.html
parsed and stored inside an abstract *earthquake event object*; the implementation of an event object varies depending on the catalog format. While much of the catalog information is not used, the use of an abstract event object allows future visualizations to store and make use of the additional attributes as desired.

The parsing of catalogs is trivial, as the data is arranged in a predictable format for both catalog types. The LSH parser only needs to tokenize each line using whitespace as a delimiter. For ANSS catalogs, however, the attributes are not comma or space separated: there may be blank entries in the files for missing data. For this reason, the ANSS catalog parser has hard-coded character positions for the start of each attribute type. As each element is parsed, blank entries can be determined by empty strings. If converting a catalog from another format to fit the ANSS format, care should be taken to ensure values line up to the required positions.

### 5.1.2 Mesh Construction

While an object-oriented design is useful for organizing the events in the catalog, it does not lend itself well to efficient rendering. It is far more efficient to render 1 million triangles in a single batch than 1 million batches of a single triangle; the rendering state changes required to draw objects one at a time quickly becomes prohibitive. Instead, every earthquake event in a catalog is rendered as part of a single mesh.
There may hundreds of thousands of earthquakes in a catalog, so a simple geometric shape is needed to represent each event. A geodesic sphere is a polygonal sphere that can be constructed from an icosahedron. The complexity of the sphere can be varied by changing the “level” of the sphere: at each level, the faces of the icosahedron base are tessellated into many more triangles and pushed outward to form a round surface. This shape is useful for representing earthquake events, as the complexity may be adjusted to be smoother (and more expensive to render) or more polygonal (useful for larger catalogs). An algorithm that describes the creation of a geodesic sphere is described in Appendix A.

![Figure 5.2: View of ANSS earthquake catalog below the surface of Hawaii. A total of 88,571 earthquakes are in this model. Events with larger magnitude appear larger and brighter in color, while less intense ruptures are small and dark purple.](image)

The first step in building the mesh is to generate a geodesic sphere that will serve as the base model for each earthquake. The geodesic sphere has its vertices positioned relative to the origin (0,0,0). For each event, a copy of these vertices are is created, scaled, and translated by a vector pointing toward the epicenter of the corresponding earthquake. Each sphere is illuminated with diffuse reflection, so a normal vector is also stored for each vertex: this is simply the normalized position of the vertex before it is translated. A color is also attributed to earthquakes based on magnitude, where white is assigned to the most intense events and dark purple for minor events.
Algorithm 12 demonstrates the steps involved in constructing and buffering the vertices for the mesh. The initial scale for the sphere, `event.scale`, refers to magnitude of an earthquake on a scale from 1 to 5 from least to greatest intensity. The `sizeConstant` value refers to the minimum radius of an earthquake sphere. The mesh vertices exaggerate the scale to have cubic scaling instead of linear scaling; this is useful for separating higher intensity earthquakes in areas that have hundreds or thousands of small shocks. The color for each sphere is linearly interpolated between violet for small events and white for major events, as shown in lines 4-5.

\begin{algorithm}
\caption{CalcQuakeVerts($buf$)}
\begin{algorithmic}[1]
\STATE $base \leftarrow \text{genGeoSphere}(level)$
\FOR {each earthquake event $e$}
\STATE $scale \leftarrow e.scale^3/5^2 \times sizeConstant$
\STATE $colorMag \leftarrow (e.scale - 1)/4$
\STATE $color \leftarrow \text{lerp}(\text{minMagColor, maxMagColor, colorMag})$
\FOR {$i := 0$ to $base.verts.length$}
\STATE $buf.put(base.verts[i] \times scale + e.position)$
\STATE $buf.put(base.normals[i])$
\STATE $buf.put(color)$
\ENDFOR
\ENDFOR
\end{algorithmic}
\end{algorithm}

As in DEMs, a catalog mesh uses an indexed triangle lists for rendering. Triangles indices are taken directly from the base geodesic sphere and offset by the earthquake index times the number of vertices in each sphere. Indices are stored in a single buffer, just as vertices are, and the triangles are ordered such that the first $3n$ indices belong to the first sphere (which has $n$ triangles); the next next $3n$ indices belong to the second sphere, and so on. Not only is this the most straightforward manner of buffering the indices, it makes it easy to isolate and render a consecutive number of events as listed in the catalog.
Figure 5.3: Each earthquake occupies $m$ vertices, depending on the level of the sphere. All earthquake sphere vertices are stored in a single VBO of length $9mt$ floats, where $t$ is the total number of events.

### 5.1.3 Event Windows and Rendering

For catalogs that do not have dates and times attached to each event, all events are simply rendered as a single large mesh, as shown in Figure 5.2. OpenGL is handed the index and vertex buffers and every earthquake event is rendered as a sphere. The shaders used for the mesh are very simple: the vertex shader transforms the vertex and calculates the lighting, which generates a color to the fragment shader; the fragment simply outputs the color to the fragment.

When catalogs provide dates and times, the mesh can be viewed in a more interesting way. The events are sorted in chronological order inside the catalog, and the resulting indices in the mesh are therefore also organized chronologically. TerraVis takes advantage of this ordering by allowing the user to show events from a starting time and date to an ending time and date. During the construction of the mesh, the event data is handed to an *event window* object. This object allows the user to define a time range, called a *view window*, of visible events; this window may be adjusted as a slider HUD element in the UI, as shown in Figure 5.4.
Figure 5.4: HUD for viewing a window of earthquakes events over time. The gray bar fills represents the window of time in which all earthquakes in the catalog occur; this bar is a visualization of an event window. The blue slider represents the view window, a portion of the event window in which events are visible in TerraVis. A control panel allows the user to scroll the view window and display, for instance, all earthquakes starting on a particular date and ending 15 months later.

A user will typically want to view events inside a fixed interval of time, such as 2 years, rather than a fixed interval of events. However, the time between consecutive events is rarely ever constant. For example, the second earthquake may have occurred a year after the first, and the third only minutes after the second. To display a window of events, it's necessary to first have a function that converts a fixed date and time to an index in the array of events.

An auxiliary array called eventTimes is stored in the event window that has the same size as the number of events and contains a single time value for each corresponding event. The value stored is the total number of milliseconds elapsed
from the start of the epoch (January 1, 1970) to the time of the earthquake event. By using a single value for each event, a binary search may be used to locate an event in the events array.

When searching for a specific time, it is also possible that there are multiple events with the same time, or none at all. When searching for a starting time, the index of the earliest event with a given time should be found; if that time does not exist, the index of the event which occurs soonest after the time should be returned. When searching for an end time, the index of the latest event with the given time should be found; if no event has the specified time, the index should be the event just before that time.

![Examples of selected elements (blue) of the eventTimes array](image)

Figure 5.5: Examples of selected elements (blue) of the eventTimes array given different starting and ending times for the view window. The time values stored in the actual array will be much larger; these values are for illustration purposes only.

The methods used to find the proper indices for the start and end of the view window are shown in Algorithms 14 and 13, respectively. The `FindUpperIndex` function locates the index of the element such that all previous elements are less than or equal to the search time. The index starts in the middle of the eventTimes arrays. If the search time is greater or equal to the last value in the search range, the search is finished and the last element is used. Otherwise, half of the search area is used as the new search area, and the ends of the search area are checked. This process is repeated until the time is found; if the time doesn’t exist, the index will be between the numbers where that time would be inserted.
Algorithm 13 FindUpperIndex\((time, left, right)\)

1: \(nextLargestTime \leftarrow time + 1\)
2: \(i \leftarrow (left + right) / 2\)
3: if \(time \geq eventTimes[right]\) then
4: return \(right\)
5: end if
6: while true do
7: if \(i = left \text{ or } i = right\) then
8: return \(i\)
9: end if
10: if \(nextLargestTime > eventTimes[i]\) then
11: \(left \leftarrow i\)
12: \(i \leftarrow (left + right) / 2\)
13: else
14: \(right \leftarrow i\)
15: \(i \leftarrow (left + right) / 2\)
16: end if
17: end while

The \textbf{FindLowerIndex} function searches for the index of the element such that all previous elements are strictly less than the search time. Lines 1-3 first check if the search time is before the left-most element in the search area; if so, this search can end and the left-most element is used. Otherwise, the function looks for the element with a time just smaller than the desired time using the \textbf{FindUpperIndex} function. If the value after the returned index is the time being searched for, that element is included. Otherwise, the search time lies between two numbers and the next element is not included: for example, looking for 5 in an array \{...,3,7,...\} should return 3.

Algorithm 14 FindLowerIndex\((time, left, right)\)

1: if \(value \leq eventTimes[left]\) then
2: return \(left\)
3: end if
4: \(i \leftarrow \text{FindUpper}(i - 1, left, right)\)
5: \(next \leftarrow \min(i + 1, eventTimes.length - 1)\)
6: if \(eventTimes[next] \geq time\) then
7: return \(next\)
8: end if
9: return \(i\)
When the user changes the starting time of the view window, the start index is found by calling \texttt{FindLowerIndex} \((\text{start}, 0, \text{eventTimes.length}-1)\). Next, the end time is found by calling \texttt{FindUpperIndex} \((\text{end}, \text{startIndex}, \text{eventTimes.length}-1)\). These indices are stored in the event window and can be retrieved by the mesh when it is drawn. Instead of issuing the command to draw all \(n\) event spheres by using every index in the index buffer, the mesh draws \(\text{end} - \text{start} + 1\) spheres. This is done by providing an offset to the start of the index array and limiting the number of indices used to \(3m(\text{end} - \text{start} + 1)\), where \(m\) is the number of triangles per sphere. The \texttt{renderSubMesh} function is called after the index and vertex buffer objects for the catalog mesh are bound, which is shown in Listing 5.1.

Listing 5.1: Rendering sub-mesh using view window offsets

```c
void renderSubMesh(int start, int end) {
    int numEvents = start - end + 1;
    int count = numEvents * indicesPerEvent;
    int offset = start * indicesPerEvent * 4;
    glDrawElements(GL_TRIANGLES, count, GL_UNSIGNED_INT, offset);
}
```

The user may also have TerraVis automatically scroll the event view window at a fixed interval. An event window shifter is used to keep track of a scroll delay (ex. 100ms), a shift amount (ex. 3 days), and a starting time and date. The shift is added to the starting and ending times every scroll delay until the view window reaches the end of the event window. The indices in the \texttt{eventTimes} are recalculated each time the view window changes, and the mesh renders using the most recent indices for the sub-mesh.

### 5.2 Focal Mechanisms

The focal mechanism of an earthquake is a diagram that can be used to show the fault orientation and slip direction along which the earthquake occurred. A focal
mechanism is, in other terms, the visualization of the moment tensor solution of an event, and appears as a “beachball” when drawn graphically. Focal mechanisms are typically viewed as overlays on a 2D map; the USGS, for example, provides data and a simple visualization of recent moment tensor solutions of recent events\(^2\). More precisely, the diagrams are lower-hemisphere stereographic projections: the bottom hemisphere of the beachball is projected onto a plane so it can be viewed in 2D.

![Image of focal mechanisms](image.png)

Figure 5.6: Illustration of the stereographic projection of a focal mechanism for strike-slip and normal faults. The block model shows how the faults slide with direction from white to red. There are actually two possible faults for each focal mechanism: for example, in the strike-slip model the fault may be aligned where the blocks are moving northeast and southwest instead of northwest and southeast.

Source: modified, with permission, from Chris Rowan
http://all-geo.org/highlyallochthonous/2009/12/5-focal-mechanisms/

The focal sphere is split into two hemispheres, and each hemisphere is split in half and colored according to zones of least compressional (red) or max compressional (white) motion in the seismic waves coming from the event. The two orthogonal planes that cut the zones are called nodal planes, and the actual fault is aligned with one of the nodal planes. Multiple focal mechanisms in the same area can be used to determine the fault orientation.

In TerraVis, focal mechanisms are rendered in 3D with a complete focal sphere. Both the top and bottom hemispheres of the focal mechanisms displayed. As with the ANSS earthquake catalogs, several focal mechanisms are listed in a single file ASCII.

\(^2\)http://earthquake.usgs.gov/earthquakes/eqarchives/fm/
Figure 5.7: The red quadrants of the focal mechanism correspond to areas with
tensional seismic waves originating at the focus; white corresponds to zones with
compressional. This focal mechanism corresponds to a strike-slip fault that may have
slip horizontally (left) or vertically (right).

Loading and storing the mesh data for a collection of moment tensor solutions is
almost the same as regular events, but additional data is provided to construct the
focal mechanisms.

5.2.1 Data Loading and Mesh Construction

TerraVis constructs a set of focal mechanisms by reading a single data file in a separate
thread, called a solution loader, that outputs a solution data object. Unlike most other
models used by TerraVis, the data and mesh are a shared object. This is a design
that should be changed in a future revision, as this was one of the last types of data
to be modeled in TerraVis.

The format TerraVis uses to moment tensor solutions is an ASCII comma-separated
value file. This file is organized in a very similar manner as the ANSS catalog out-
put, and includes attributes such as date, time, positions, magnitude, and so forth.
Additionally, each line contains the strike, dip, rake, and slip attributes that can be
used to define the two nodal planes. The algorithm to create each nodal plane from
these values is described in 6.1, which discusses the slip distribution along a plane.
Figure 5.8: Focal mechanisms shown in 3D underneath a DEM from southern California.

As with catalog earthquakes, each focal mechanism makes use of a geodesic sphere. The vertices for all spheres are stored in a single VBO, and the indices for all triangles in each sphere are stored in a single index buffer. The layout of these buffers is the same as a catalog mesh described in the previous section. However, extra information is packed into the vertices that is needed to render it as a “beachball” diagram.

If vertices are colored according to zone during mesh construction, it would take a significant amount of vertices to create a nicely shaded beachball; instead, the colors on the sphere are colored inside a fragment shader program described in the next section. The vertex format instead stores the information needed to determine which zone a fragment belongs to. Each vertex stores its position, the normal of the first nodal plane, the normal of the second nodal plane, and the position of the vertex in the base geodesic sphere (before scaling and translation). The untransformed vertex position is used so it can be determined where the vertex lies with respect to the center of the sphere inside the shader program.
Figure 5.9: Vertex format for focal mechanisms. The second tuple of float belongs
to the normal of the first nodal plane; the third tuple belongs to the second nodal
plane. The last tuple of floats stores the position of the vertex before it is scaled
and translated from the origin. The normal, texture coordinate, and color pointers
(np, tcp, cp respectively) are used to mark the locations of these values; this does not
mean the values are used for lighting, texturing, and coloring, as the default rendering
is overridden by a shader program.

5.2.2 Mesh Rendering

The vertex shader used while rendering focal mechanisms is relatively simple, and
can be seen in Listing 5.2. The position of each vertex with respect to the center
of the sphere is stored in \textit{spherePos}, and the nodal plane normals are stored in \textit{np1}
and \textit{np2}. Recall that these values are stored in the normal and texture coordinate
locations, but are not used for lighting or texturing.

Listing 5.2: (beachball.vs) Vertex shader for focal mechanism rendering

```
varying vec3 spherePos, np1, np2;

void main()
{
  np1 = gl_Normal;
  np2 = vec3(gl_MultiTexCoord0);
  spherePos = gl_Color.xyz;
  gl_Position = ftransform();
}
```
The fragment shader is where each fragment of the sphere is assigned color. The four zones of the sphere should be colored with alternating red and white. If the position of a fragment is in front of a nodal plane, its dot product with that plane’s normal vector will be positive; otherwise, if it is behind the plane, the dot product will be a negative value. If the fragment is in front of both nodal planes or behind both nodal planes, it is in a tensional quadrant. Otherwise, the fragment is in a compressional quadrant. This can be seen in Figure 5.10.

Listing 5.3: (beachball.fs) Fragment shader for focal mechanism rendering

```c
varying vec3 spherePos, np1, np2;

void main()
{
  vec3 color = vec3(1.0);
  bool a = dot(worldPos, np1) > 0.0;
  bool b = dot(worldPos, np2) > 0.0;
  if ((a && b) || (!a && !b))
    color = vec3(1.0, 0.0, 0.0);
  gl_FragColor = vec4(color, 1.0);
}
```

Figure 5.10: Illustration of how fragment positions are colored depending on the dot product of the position with the normal vectors of the two nodal planes.
Chapter 6

Fault Plane Models

A fault is a fracture in Earth’s crust where movement occurs between two rock formations on either side. There are several names for the types of faults that can occur, such as normal faults, reverse faults, strike slip faults, and thrust faults\(^{22}\). This section describes two simple models used to analyze the movement along a fault. The first model defines a surface plane between two blocks representing the formations on the sides of the fracture. The second model, a slip distribution, illustrates the amount of movement at particular points along the fault.

6.1 Strike-Dip-Rake Model

TerraVis uses strike, dip and rake measurements to orient a fault plane. These measurements are all in degrees (or radians). The strike and dip are used to determine the incline of the fault plane, while rake shows the direction the hanging wall, the block that lies above the fault plane, slides during a rupture. Collectively, these measurements provide a picture of the overall movement along a fault.

\(^{22}\)http://www.iris.edu/gifs/animations/faults.html
The following vectors can be used to describe a strike-dip-rake model:

- **Strike direction**: found by rotating strike degrees clockwise from North with respect to the “up” direction, a vector perpendicular to both the strike direction and North.

- **Dip direction**: a vector perpendicular to the strike direction; it is the normal vector clockwise from the strike direction.

- **Incline direction**: faces down the surface of the fault plane. This vector may be found by rotating the dip direction by dip degrees around the strike direction.

- **Plunge direction**: this vector is normal to the fault plane facing the hanging wall.

- **Rake direction**: found by rotating the strike direction by rake degrees around the plunge direction.

The aforementioned vectors can be calculated as in Algorithm 15. Within TerraVis, North is facing the positive z-axis, and up is the positive y-axis. The following
algorithm uses a \texttt{rotate}(v, radians, axis) function, which simply rotates the vector \( v \) by \textit{radians} around the provided \textit{axis}. The \texttt{normalize}(v) function ensures the vector \( v \) is unit length.

\begin{algorithm}
\textbf{Algorithm 15} CalcSDRVectors\((\text{strike, dip, rake, north, up})\)
\begin{algorithmic}
\STATE \texttt{strikeDir} \leftarrow \texttt{rotate}(\texttt{north, -strike, up}).
\STATE \texttt{dipDir} \leftarrow \texttt{normalize} (\texttt{strikeDir} \times \texttt{up})
\STATE \texttt{inclineDir} \leftarrow \texttt{rotate} (\texttt{dipDir, dip, strikeDir})
\STATE \texttt{plungeDir} \leftarrow \texttt{normalize} (\texttt{inclineDir} \times \texttt{strikeDir})
\STATE \texttt{rakeDir} \leftarrow \texttt{rotate} (\texttt{strikeDir, rake, plungeDir})
\end{algorithmic}
\end{algorithm}

### 6.2 Slip Distribution Model

The strike-dip-rake model is used to describe the orientation of a fault plane; however, visualizing the static plane (or blocks) does not provide some of the more detailed information from an event. The type of fault data that TerraVis supports is called a slip distribution, which builds on top of the strike-dip-rake model described in the previous section. A slip distribution indicates the \textit{slip}, movement along a fault plane, for several points on the plane. Slip is a displacement vector, and the typical visualization is to draw a grid where each cell contains the displacement vector and a color for its magnitude. Currently, TerraVis only has access to the magnitude of slip and the positions of the points along the plane. This type of model can be seen in Figure 6.2.
Figure 6.2: Slip distribution for the 2010 Haiti earthquake. In (a), rendered in MATLAB; in (b), rendered in TerraVis. The colors represent slip intensity in centimeters along the fault plane.

A slip distribution model is provided in ASCII format (a .dat file) with each line corresponding to a cell in a 2D grid. There is a single line of header information that describes the values in the lines below. The first few lines of a sample slip distribution file are shown in Listing 6.1. Each line contains a position in longitude and latitude, the strike, dip, and rake values for the fault plane, a slip amount, and the size of the cell in kilometers.

Listing 6.1: Sample slip distribution .dat file (digits truncated for brevity)

<table>
<thead>
<tr>
<th>Lon</th>
<th>Lat</th>
<th>Strike</th>
<th>Dip</th>
<th>Rake</th>
<th>SLIP</th>
<th>Length</th>
<th>top</th>
<th>bottom</th>
</tr>
</thead>
<tbody>
<tr>
<td>-72.398</td>
<td>18.436</td>
<td>262.000</td>
<td>70.000</td>
<td>30.000</td>
<td>0.064</td>
<td>2.000</td>
<td>0.00</td>
<td>2.000</td>
</tr>
<tr>
<td>-72.398</td>
<td>18.436</td>
<td>262.000</td>
<td>70.000</td>
<td>30.000</td>
<td>0.129</td>
<td>2.000</td>
<td>2.00</td>
<td>4.000</td>
</tr>
<tr>
<td>-72.398</td>
<td>18.436</td>
<td>262.000</td>
<td>70.000</td>
<td>30.000</td>
<td>0.302</td>
<td>2.000</td>
<td>4.00</td>
<td>6.000</td>
</tr>
<tr>
<td>-72.398</td>
<td>18.436</td>
<td>262.000</td>
<td>70.000</td>
<td>30.000</td>
<td>0.256</td>
<td>2.000</td>
<td>6.00</td>
<td>8.000</td>
</tr>
</tbody>
</table>

Figure 6.3 shows how the values in the file correspond to each cell in the grid. The cells are listed in column-major order in the data file, and the grid is rectangular. The longitude and latitude for each line is the same for every cell in a column. This is misleading, and the position cannot be used directly to position each cell: this would indicate every cell belongs to a fault plane with a 90° dip. Whenever the latitude or
longitude values change, it marks the position of the top of a new column in the grid. The strike and dip values must be used to correctly calculate the positions of each grid cell beyond the first in each column.

![Figure 6.3: Values of a slip distribution grid cell.](image)

The top and bottom values mark the (negative) elevation of the top and bottom of the cell in kilometers. The length is the size of the sides of the cell in kilometers, and is the same as the bottom minus the top. The number of columns of the grid can be determined by how many times the latitude and longitude values change; the number of rows is then the number of lines in the file divided by the columns.

### 6.3 Data Loading

A slip distribution .dat file is loaded inside a slip loader, a separate thread that reads the file, parses the values, and constructs the slip mesh. The slip data is an object that describes a slip distribution grid in a standardized format that can be consumed by the mesh.

Each line of the .dat file is read sequentially and the values are parsed into a slip object. After all slips are parsed and the dimensions of the grid are known, the vectors that define the displacement from one slip to another are calculated. First,
the strike, dip, and rake values are converted into vectors as in Algorithm 15. Next, the \texttt{CalcSlipPlaneVectors} function (Algorithm 16) is called to store the \textit{down} and \textit{up} displacement vectors. All the vectors are multiplied by 1000 as they need to be converted from kilometers to meters.

\begin{algorithm}
\caption{CalcSlipPlaneVectors($slips$, \texttt{latLonBox})}
\begin{algorithmic}[1]
\State $\text{origin} \leftarrow \text{latLonBox.calcPos3D}(slips[0].lon, slips[0].lat, \text{slips}[0].\text{top} \times 1000)$
\State $\text{right} \leftarrow \text{strikeDir} \times \text{unitsPerMeter} \times \text{slips}[0].\text{length} \times 1000$
\State $\text{down} \leftarrow \text{inclineDir} \times \text{unitsPerMeter} \times \text{slips}[0].\text{length} \times 1000$
\end{algorithmic}
\end{algorithm}

The \textit{origin}, the starting corner of the slip distribution, is calculated using the \texttt{LatLonBox} from the digital elevation model currently used by TerraVis (see Algorithm 4). Next, the \textit{right} vector is calculated, which is a vector in the direction of the strike and length equal to the distance between cells, in 3D units; the same is done for the \textit{down} vector, which uses the incline direction of the plane.

\section{Mesh Construction and Rendering}

Each slip distribution is rendered as a triangle mesh that is arranged in the same manner as DEM meshes. Vertex and index buffers are used, just as in the digital elevation models. However, the complexity of a slip distribution is far less than a DEM, so the overall construction is far more straightforward. The slip distribution is
rendered as a colored plane without lighting or texturing; each vertex is only assigned a position and color, as shown in Figure 6.5.

**Figure 6.5: Vertex format for slip distribution mesh.**

Slip distributions have significantly lower resolutions than DEMs, so the memory and triangle ordering optimizations used by DEM meshes are not beneficial. For example, the slip distribution we were provided with for the 2010 Haiti earthquake has a size $13 \times 30$. Because of the small size grids, slip meshes only need to use `unsigned shorts` for the triangle indices.

Two triangles are assigned to each cell in the slip distribution. The vertex positions are input by iterating through each slip object and calculating its position using the `down` and `right` vectors from Algorithm 16. After each position, a color is assigned to each vertex corresponding to the slip intensity at that position. There is an additional row and column of vertices than slips; these are needed to create the triangles. Each vertex is assigned a color corresponding to the slip value as its index. Vertices on the last row receive the same color as the vertices on the preceding row; vertices on the last column receive the same color as the vertices on the preceding column.

The function for calculating the color for a particular slip value consists of a selection of linear interpolations. The minimum and maximum slip values for a model
Figure 6.6: Mapping of slip distribution cell to mesh vertices. Given a \(2 \times 4\) slip distribution, there will be \(3 \times 5\) vertices. Colors for pixels in the triangles of the mesh are calculated with bilinear interpolation during rendering.

are stored while the file is read. Five colors are assigned to values that lie between the max and min: blue (min), cyan (medMin), light green (med), yellow (medMax), and red (max). The values \(\text{medMin}\) and \(\text{medMax}\) lie halfway between the medium and min and max, respectively. The algorithm for selecting the color is shown in Algorithm 17. The \(\text{lerp}(a,b,c)\) function returns a color linearly interpolated between \(a\) (at \(c = 0\)) and \(b\) (at \(c = 1\)).

**Algorithm 17** CalcSlipColor\((slip)\)

1: \(\text{med} \leftarrow (\text{max} - \text{min})/2 + \text{min}\)
2: \(\text{medMax} \leftarrow (\text{max} - \text{med})/2 + \text{med}\)
3: \(\text{medMin} \leftarrow (\text{med} - \text{min})/2 + \text{min}\)
4: if \(\text{slip} < \text{medMin}\) then
5: \(\text{color} \leftarrow \text{lerp}(\text{minColor}, \text{medMinColor}, (\text{slip} - \text{min})/(\text{medMin} - \text{min}))\)
6: else if \(\text{slip} < \text{med}\) then
7: \(\text{color} \leftarrow \text{lerp}(\text{medMinColor}, \text{medColor}, (\text{slip} - \text{medMin})/(\text{med} - \text{medMin}))\)
8: else if \(\text{slip} < \text{medMax}\) then
9: \(\text{color} \leftarrow \text{lerp}(\text{medColor}, \text{medMaxColor}, (\text{slip} - \text{med})/(\text{medMax} - \text{med}))\)
10: else
11: \(\text{color} \leftarrow \text{lerp}(\text{medMaxColor}, \text{maxColor}, (\text{slip} - \text{medMax})/(\text{max} - \text{medMax}))\)
12: end if

Triangles in the mesh are rendered using an indexed triangle list. Vertices are added in column-major order, matching the ordering of the slips in the data file, so the loop to buffer indices is only slightly different from the scan-line ordering mentioned in the DEM mesh. The values \(tl, bl, tr,\) and \(br\) refer to the top-left,
bottom-left, top-right, and bottom-right vertices of the current cell in the mesh; \( tl \) is always the current vertex in the loop.

```
Algorithm 18 CalcTriangleIndices(buf)
1: for \( x := 0 \) to \( cols - 1 \) do
2:   for \( y := 0 \) to \( rows - 1 \) do
3:     \( tr \leftarrow y + x \times rows + rows \)
4:     \( tl \leftarrow tr - rows \)
5:     \( bl \leftarrow tl + 1 \)
6:     \( br \leftarrow tr + 1 \)
7:     IndexTriangle(buf, tl, bl, tr)
8:     IndexTriangle(buf, tr, bl, br)
9:   end for
10: end for
```

The slip mesh does use a shader program to control the rendering, however it is trivial. The vertex shader simply transforms the vertex position and passes the color to the fragment shader. The fragment shader then uses the interpolated fragment color and outputs it.
Chapter 7

Volume Models

A special category of data that is common in scientific and engineering fields is volume data. The form of volume data is typically a scalar field: a discrete set of samples in a three-dimensional area. Each element of a volume is called a voxel, and can be thought of as block inside the volume bounds. For example, pressure may be recorded at points in an area near a fault; each voxel is assigned a pressure value and can be positioned by its index in the volume data structure, such as a three-dimensional array. With a high enough sampling rate, volumes can approximate the continuous nature of volumetric phenomena.

There are essentially two approaches to visualizing volumetric data: surface-fitting techniques extract a polygonal mesh for a specified value in the data; direct volume rendering attempts to approximate light transport in a gaseous volume to provide physically accurate visualizations. Polygonal meshes are suitable for objects with well-defined boundaries, but are inadequate for nebulous data. As shown in Figure 7.1, an isosurface can be used to render a mesh around a particular scalar value in a volume; however, much of the information contained in the data set is lost using this technique. TerraVis makes use of direct volume rendering to provide the maximum quality of rendering and interactivity to the scientist.
Figure 7.1: Isosurface rendering of a volume of Gutenberg-Richter b-values obtained in Hawaii. This figure illustrates a surface-fitting approach to visualizing volumes, and uses a technique based on marching cubes \[15\]. A mesh is extracted that surrounds all scalar values less than or equal to a desired b-value. This is not a computationally cheap task and only visualizes a small portion of the entire data. Colors represent normal vectors.

Rendering volumes in TerraVis is not specialized to handle a specific data types as previous chapters describing DEMs, earthquake catalogs, and slip distribution modeling. TerraVis can load and render any 3D single-value scalar field that can be positioned and sized using latitude and longitude and is organized in a regular 3D grid. However, this section explores the application of direct volume rendering within the context of Gutenberg-Richter b-values. Volume rendering is also applicable to many other types of geologic data, such as tomography; however, we have not had the opportunity to experiment with these data types.

This chapter focuses on direct volume rendering and its implementation in TerraVis. However, some light background is provided on the data we have used to demonstrate the volume rendering facilities in TerraVis. The Gutenberg-Richter law
[6] is used in seismology to show a relationship between an earthquake magnitude and amount of earthquakes in a region and time that match or exceed the given magnitude. The amount of earthquakes with a magnitude $M$ is proportional to $10^{-bM}$. Typically, the value of $b$ is around 1.0 in most parts of the world. The data we have worked with is a volume of $b$-values from Hawai‘i, and our hope was that rendering the $b$-values alongside earthquake events could provide some insight that has not yet been achieved.

### 7.1 Direct Volume Rendering

Rather than extracting a polygonal mesh, direct volume rendering makes use of an *optical model* that approximates how light interacts with particles in the volume. Such a model is responsible for converting scalar values in the volume to properties that can be used in rendering: color and opacity. For naturally occurring volumes, such as smoke, a specific result is desired and the optical model is designed to convincingly capture the interaction of light with smoke particles. Even if a volume consists of what would normally be opaque materials, each voxel can be viewed as a container as many particles.

For scientific data, the optical properties of the particles must be defined by the scientist. An optical model can be designed as a function that maps voxel values to colors directly, as the volume data is loaded; such a mapping is called a *transfer function*. It is not easy to come up with an effective transfer function unless the data values are known to correspond to particular materials. Some volumes may contain multiple materials, which is common in medical data. Other volumes may be completely mysterious without a precursory visualization, as is the case with the $b$-values we have modeled.
Figure 7.2: Direct volume rendering of the same volume of Gutenberg-Richter b-values in Figure 7.1. An ANSS earthquake catalog is also rendered alongside the volume, demonstrating the benefit of having translucent, non-polygonal mesh. Colors represent b-values, and are assigned by a transfer function described in [13].

The design of a transfer function can boil down to a game of trial and error on the part of the scientist visualizing the data. Assigning a random optical model to a volume and iteratively refining it can be time-consuming, especially if it is done in a processing step before the volume is rendered. For volumes that contain data from a well-known medium, such as an MRI scan, materials can sometimes be automatically segmented using general-purpose or specialized algorithms for such. In other cases, it is desirable to make the process of exploring a volume interactive by letting the user change the transfer function during rendering. With such an approach, scientists can experiment with their data as they view it and search for patterns, especially when the volume is rendered alongside other data. This is the approach used by TerraVis, and is demonstrated in this chapter with multiple renderings of the b-values.
Volume rendering is a deep topic in itself, and a thorough dissection of the topics of light transport and volume rendering is outside the scope of this work. An overview of volume rendering is provided by Kaufman and Mueller[12], and more information on the use of volume rendering in real-time graphics is covered by Hadwiger et. al in the 2006 book *Real-Time Volume Graphics*[7]. The remainder of this chapter will focus on the basic implementation details of the volume renderer in TerraVis.

### 7.1.1 3D Texture-Based Rendering

Originally, direct volume rendering was a technique used only in off-line software rendering, which produces high-quality images at non-interactive rates. As graphics hardware became increasingly powerful, various techniques have been used to produce similar results using graphics APIs such as OpenGL. Since TerraVis is a real-time visualization application, it makes of an approach calling 3D texture-based volume rendering. In OpenGL, a 3D texture is simply an extension of a 2D texture along the z-axis; instead of having $width \times height$ pixels, a 3D texture has $width \times height \times depth$ voxels. This type of texture is perfectly suited to storing volume data, and the individual elements can be accessed using 3D texture coordinates.

OpenGL is still a primitive-based rendering API designed work as efficiently as possible with graphics hardware. Graphics pipelines expect to receive vertices and indices that compose primitives such as points, lines, and triangles. As such, achieving volume rendering is not as simple as merely storing a volume in a 3D texture and calling some built-in routines. Ultimately, we want to sample the optical properties of the volume along rays to the camera position. To approximate this, a series of view-aligned polygons are created along the viewing direction. Each polygon is the interior of a plane that intersects with the edges of the volume’s bounding box; the volume is effectively sliced into polygons that are always oriented to face the view position.
Figure 7.3: Slicing a volume into several polygons, sampling volume texture along the polygons, and compositing the result. With a greater number of slices (sampling rate), a higher-quality rendering is produced.


### 7.2 Data Loading

TerraVis loads volumes stored in ASCII files. First, the dimensions and axes of the grid are provided along. The first three lines (not counting commented lines), correspond to $gx$, $gy$, and $gz$: the grid dimensions in $x$, $y$, and $z$ respectively. Each of these lines contains a space-separated list of values, where the number of values indicates the size in that dimension and the value indicates the location. For the $gx$, values are stored as longitude; the $y$-axis stores values in latitude; and the $z$-axis contains depth values in kilometers. Following these three lines is a single line repeating the size of each dimension. Finally, there are $n$ lines that contain the scalar value for each voxel, where $n$ is $gx \times gy \times gz$.

The file format is typical for 3D matrices from MATLAB, and the ASCII output is suitable for volumes with smaller resolutions. The $b$-value data we have used from Hawaii has a resolution of $41 \times 27 \times 75$. Currently, the volume rendering in TerraVis does not make use of optimization techniques for meshes with large resolutions.

The data is, as with other data in TerraVis, loaded inside a separate thread. A $BVData$ object parses the b-values and calculates the $LatLonBox$ for the volume.
This object is then processed and stored a standardized format, called a *volume data* object. Finally, most of the work is done in a *volume* object that converts the 3D array of scalar values into a 3D texture and calculates the proxy geometry (polygon slices) necessary to render the volume. Each volume also stores a transfer function, which is a texture that is sampled by each voxel during rendering.

### 7.3 Proxy Geometry

The polygon slices used to render the volume are collectively referred to as the *proxy geometry*. Each slice is view-aligned, meaning the normal vector for the surface is facing towards the viewer. The vertices for each slice lie on the edges of the bounding box of the volume and store 3D texture coordinates that are used to sample the 3D volume texture.

The first step in calculating the proxy geometry is the generate planes that are spaced at a regular interval from each other. These planes will be used to slice the volume; the greater the number of slices, the higher the resulting quality of the rendering. The volume maintains a *slice stack*, which must be updated whenever the view direction or number of desired slices changes. Algorithm 19 outlines the steps involved in this process.
The distance between each slice is calculated as \( \text{samplingLength}/(\text{numSlices} + 1) \), cutting the sample length into equal parts. The sample length is found by intersecting the viewing ray with the bounding box and using the separation between the nearest and farthest points of the intersection. The \( \text{sliceStep} \) is the vector used to place the center of each plane, and slices are placed starting with the nearest point on the bounding box that is nearest to the camera.

**Algorithm 19** UpdateSliceStack(numSlices)

1: \( \text{sliceDist} \leftarrow |\text{samplingLength}|/(\text{numSlices} + 1) \)
2: \( \text{sliceNorm} \leftarrow -\text{camera.forward} \)
3: \( \text{sliceStep} \leftarrow \text{sliceDist} \times \text{sliceNorm} \)
4: for each \( \text{slice} \) in \( \text{numSlices} \) do
5: AddSlice(\( \text{nearPt} - \text{sliceStep} \), \( \text{sliceNorm} \), \( \text{volumeBounds} \))
6: end for

Figure 7.5: The bounding box on the left is sliced into four equally spaced polygons that are orthogonal to the viewing direction. The right subfigure illustrates how the vertices for a slice would be sorted using smallest angle from the right vector.

When a slice is added, a plane is created with a point and normal vector and intersected with the bounding box. The ordering of the points of intersection can change depending on the viewing direction and plane/box intersection algorithm. Before a polygon can be created, the vertices for the slice are sorted using smallest angles with respect to the right vector of the polygon (the same as the right vector.
for the camera, since polygons are always facing the viewer). The sorting for pairs of vertices is done using Algorithm 20, which returns a negative number if \( v_a \) has a smaller angle than \( v_b \), a positive value if \( v_b \) has a smaller angle than \( v_a \), or 0 if they are equal.

Algorithm 20 SortProxyVerts\((v_a, v_b)\)

1: \( a \leftarrow \text{CalcSortValue}(v_a) \)
2: \( b \leftarrow \text{CalcSortValue}(v_b) \)
3: return \( a - b \)

Algorithm 21 CalcSortValue\((p)\)

1: \( v \leftarrow p - \text{polygonCenter} \)
2: \( v \leftarrow v/|v| \)
3: \( \text{val} \leftarrow (up \times v) \cdot n \)
4: if \((v \times right) \cdot n >= 0\) then
5: \( \text{val} \leftarrow 2 - \text{val} \)
6: end if
7: return \( \text{val} \)

After the vertices for each slice are calculated, texture coordinates are assigned so that each vertex can sample from the volume texture at the correct location. Knowing that each vertex is always on an edge of the bounding box, it’s trivial to calculate these coordinates: the position of the vertex can be used directly as the texture coordinates when each component is divided by the length of the respective side of the bounding box. Finally, the geometry is uploaded to a vertex buffer. Unlike previous data, this buffer is assigned the \textit{dynamic} flag instead of \textit{static}; this allows the graphics hardware to best determine the optimal memory location for the data given that it is expected to change frequently.
7.4 Textures and Gradient Vectors

Two textures are used when rendering a volume: a 3D texture that stores the volume data itself, and a 2D texture that stores the transfer function used to color the volume. Textures may be configured to store texels in many different formats and sizes. One of the most commonly used texture formats is \texttt{GL RGBA}, which assigns each texel red, green, blue, and alpha components of 8-bits each. Different formats are useful for different data types, and in general it is more efficient and faster to render using formats with smaller memory requirements.

Each voxel in volumes TerraVis can load stores a value in single-precision floating-point from 0 to 1. When generating the 3D texture, these values are mapped to 0 to 255. This would permit the use of a single-channel texture using a format such as \texttt{GL LUMINANCE}, or using extra bytes for higher precision values; however, the extra 3 bytes of the \texttt{GL RGBA} format can also be used for storing gradient vectors that are used in shading the volume. A gradient vector is the vector computed from using central differences (Equation 7.1 at each voxel, which indicates an approximation to the direction and amount of change in value at the location.

\begin{align}
G_x(\text{voxels}_{i,j,k}) & \leftarrow (\text{voxels}_{i+1,j,k} - \text{voxels}_{i-1,j,k})/2d \quad (7.1) \\
G_y(\text{voxels}_{i,j,k}) & \leftarrow (\text{voxels}_{i,j+1,k} - \text{voxels}_{i,j-1,k})/2d \quad (7.2) \\
G_z(\text{voxels}_{i,j,k}) & \leftarrow (\text{voxels}_{i,j,k+1} - \text{voxels}_{i,j,k-1})/2d \quad (7.3)
\end{align}

In Equation 7.1, the vector \(d\) is the physical separation between voxels along each axis. The result of calculating gradients at each voxel is similar to an edge detection filter in image processing: voxels in homogeneous regions have very small magnitude, while voxels that lie on the boundaries of surfaces have higher magnitude.

After the gradient vector is calculated for each voxel, each component must be mapped into a 0 to 255 range value so it can be stored in the 3D texture. This is done
by keeping track of the minimum and maximum values of each gradient: the mapping for the x-component of a gradient vector, for example, is shown in Equation 7.4. For the texels in a 3D volume texture, the first three components (RGB) are used to store the x, y, and z components of the gradient vector. The fourth component, the alpha, stores the voxel value.

\[
(g_x - \text{min}G_x)/(\text{max}G_x - \text{min}G_x) \times 255
\]  

(7.4)

The transfer function is stored in a separate texture that is used to assign colors to each voxel. A 1D transfer function texture simply uses the voxel value, which is already 0 to 1, as the texture coordinate used to sample the texture. TerraVis uses a 2D transfer function, and the voxel value and gradient vector magnitude are used as the x and y components of the 2D texture coordinates. The use of a 2D transfer function allows the user to more effectively separate material boundaries in volumes and assign optical properties accordingly [13].

7.5 Rendering

While slices are created in a front-to-back order, they are rendered in back-to-front order. The over operator is used for compositing the pixel colors of each slice, shown in Equation 7.5. In the equation, \(C_i\) and \(A_i\) are the color and alpha (opacity), which is accumulated as each slice is rendered. Volumes are rendered last to ensure they blend properly with opaque geometry in the scene, such as earthquakes and a DEM. Unfortunately, when a DEM is made transparent the volume is not visible if it is behind the DEM from the viewer’s perspective. The blending equations is shown in Equation 7.5.
\[
C_i = C_i + (1 - A_i) \cdot C_{i+1} \\
A_i = A_i + (1 - A_i) \cdot A_{i+1}
\] (7.5) (7.6)

The proxy geometry is rendered using a shader program. The vertex shader is trivial, and simply passes on the transformed vertex positions and texture coordinates. The fragment shader is more complex, and takes into the transfer function and local illumination. The condensed GLSL code for the fragment shader is shown in Listing 7.1.

Listing 7.1: (volume.fs) Main function of volume rendering fragment shader

```glsl
void main()
{
    vec4 data = texture3D(texVolume, texcoords);
    vec3 gradient = data.rgb * gRange + gMin;
    float gMag = length(gradient);
    float value = data.a;
    vec2 texCoords = vec2(value, (gMag - gMagMin) / gMagRange);
    vec4 colorTF = texture2D(texTF, texCoords);
    float opacity = colorTF.a;
    opacity = 1.0 - pow(1.0 - colorTF.a, opacityCorrection);
    vec3 outColor = applyLighting(colorTF);
    gl_FragColor = vec4(outColor * opacity, opacity);
}
```

The first four lines of the `main` function extract the gradient vector and voxel value for the current fragment. Notice that the gradient vector must be converted back to its original values after it was mapped into byte value range; the `gMin` and `gRange` values are uniform variables that represent the minimum gradient values and range of values (`gMax - gMin`), respectively.

The following four lines sample the transfer function texture. First, the texture coordinates are calculated using the voxel value and gradient magnitude mapped to a 0 to 1 range. The opacity from the transfer function cannot be used directly, the
overall opacity of the volume would change with different sampling rates (more or less slice polygons). The \textit{opacityCorrection} value is a uniform variable that ensures the overall transparency of the volume is unaffected as the viewer adjusts the sampling rate. The value of the opacity correction is set as \textit{referenceSampleRate/numSlices}; the user can adjust the reference sampling rate to scale the total opacity of the volume.

The color extracted from the transfer function has local illumination applied to shade the volume. Listing 7.2 shows the code that accomplishes this, which uses the gradient vectors as approximations for normal vectors. The color returned from shading is weighted by the opacity and output to the fragment.

\begin{verbatim}
Listing 7.2: (volume.fs) Local illumination using gradient vectors
vec3 applyLighting(vec3 colorTF, vec3 gradient)
{
    vec3 diffuseColor = colorTF.rgb * lightColor;
    vec3 outColor = ambient * diffuseColor;

    vec3 n = gradient / gMag;
    float diffuse = saturate(dot(lightDir, n));

    if (diffuse > 0.0) {
        vec3 r = normalize(reflect(lightDir, n));
        vec3 eye = normalize(viewPos - pos);
        float specular = pow(saturate(dot(r, eye)), specExponent);
        outColor += diffuse * diffuseColor + specular * vec3(1.0);
    }

    return color;
}
\end{verbatim}
Chapter 8

Stereoscopic Vision

Several technologies can be used to produce stereoscopic vision, a process that provides enhanced depth perception. These technologies take advantage of the differences in images viewed by the left and right eyes. Depth perception is useful in differentiating objects that are similar in shape and color but at different depths, making it valuable for three-dimensional data sets [10].

8.1 Depth Perception

There are number of visual cues that indicate the depth of objects in a scene [17, 2]. Many of these cues can be found in two-dimensional representations of three-dimensional images, while others are a result of having two eyes. In general, depth perception is greater with a higher number of cues. Some of the most crucial depth cues are monocular, meaning they can only require a single eye to have effect. In non-stereo three-dimensional graphics, these cues are essential to differentiating objects at different depths.

- **Perspective** was first understood by artists during the Renaissance, and is a simple observation: objects farther from a viewer appear smaller.
• **Knowledge** of object dimensions relative to other shapes can indicate if an object is close and far from the viewer. This cue only applies when objects are familiar and there are multiple objects for reference.

• **Detail** indicates an object is closer to the viewer; when the object is farther away finer details such as texture and patterns are blurred.

• **Occlusion** of an object by another object is a clear indicator of the ordering of the two objects relative to the viewer.

• **Shading** provides hints about the surface of a shape. The reflection, absorption, and refraction of light describes the geometry and material of a surface. Objects at greater distances also appear more dim.

• Objects in **motion** will seem to move slower the farther they are from a viewer.

While the above monocular cues can be incorporated into any program with real-time 3D graphics, special hardware is needed to capture **binocular cues**. These cues are only present as a result of our visual system and cannot be reproduced in a single 2D image. There are three important binocular cues that affect our depth perception:

• **Binocular disparity** : The horizontal separation between human eyes provides two slightly different views of the world. The disparity between the two images is the main cue addressed by stereographic systems. A single view is not sufficient to reproduce the scene with the depth perception most humans possess.

• **Convergence** : To maintain focus on an object, our eyes rotate to converge on that object’s position. The effort of convergence can be measured as an angle where the two viewing directions intersect.

• **Accommodation** : Changing the muscle tension to focus at different depths is described by accommodation.
Figure 8.1: Eyes rotate to converge at a specific point. Focusing an object at a specific depth is achieved by accommodation, a change in muscle tension to adjust focal length.

8.2 Parallax

For humans, the two perspectives of the left and right eyes are different and do not create a cohesive image on their own. There is a horizontal disparity between the left and right images our eyes receive, providing additional information that allows our brain to construct a more accurate representation of the world. The difference in the perceived location of an object viewed from two different viewpoints is called parallax. Since human eyes are separated horizontally, we are primarily concerned with horizontal parallax. The human brain attempts to fuse together two disparate images, and enhanced depth perception is provided as a result.

Figure 8.2: A sphere is placed in front of three boxes, and the left and right eyes converge to focus on that sphere. When viewed by the left eye, the right-most box is occluded; when viewed by the right eye, the left-most box is occluded. When the two images are fused, our brain would understand that the sphere is positioned somewhere in front of the boxes and horizontally centered between the left and right eye images.
Eyes may be adjusted to focus on a point at a particular depth; this creates a plane where any point on that plane will be in focus. When viewing graphics rendered to a display, the focal plane of our eyes is always at the distance from the eyes to the display. This plane is called the projection plane. When generating stereoscopic 3D images, the effects of horizontal parallax can be used to produce a binocular depth cue.

Figure 8.3: *Negative horizontal parallax*: when viewed by the left eye, \( p \) appears on the right; when viewed by the right eye, \( p \) appears on the left.

Figure 8.4: *Zero horizontal parallax*: the left and right eye views of the projections of \( p \) overlap.

When the eyes converge at a point in front of the projection plane, this type of parallax indicates that an object is coming out of the display. The parallax is described as negative, and when viewed by the left eye, the projected point appears on the right; when viewed by the right eye, the projected point appears on the left. As the point moves closer to the projection plane, the two projections get closer together. Eventually, the eyes converge at a point that lies on the projection plane producing...
zero horizontal parallax. When this occurs, there is no disparity between the left and right images.

If an object is behind the projection plane, the parallax is positive and the projections will be on the same sides as their respective eyes. Positive parallax will continue to decrease angle of convergence as the point goes to infinity. At infinite depth, the left and right eyes will be looking in parallel directions, and this is the maximum amount of positive horizontal parallax that should occur; in other words, the horizontal parallax is equal to the depth between the two eyes (interocular distance).

![Figure 8.5: Positive horizontal parallax: when viewed by the left eye, p appears on the left; when viewed by the right eye, p appears on the right.](image)

**8.3 Stereographic Systems**

With an understanding of the effects of parallax, the strategy of all systems producing stereoscopic vision is to generate two images: one for the left eye, and one for the right eye. The method in how the images are isolated depends on the technology employed. The common technique in visualization is to display both images on the same display and utilize glasses to separate the images. There are two approaches to this: active and passive.

Passive systems use polarized filters in both the projectors and glasses so that the image intended for the left eye is filtered out in the right eye, and the image intended for the right eye is filtered out in the left eye. An older technology using this same
principle is anaglyphs, which separate images into red and cyan components. This is the technique used in most theatres, as the glasses are very cheap and do not have any active components.

What we’re using here is called active, because the glasses are liquid crystal that alternates opaque and transparent in synchronization with an infrared timing signal. This means the glasses are much more expensive, but the display technology is cheaper. We only need a 120hz display that can switch between left and right eye images so each eye gets 60hz.

8.4 Stereo Rendering in OpenGL

The images produced by OpenGL are a result of projecting three-dimensional geometry onto a plane. A viewing frustum represents the bounds of what a virtual camera will see. The usual approach to configuring a camera is to define the viewing frustum in terms of field of view, near plane distance, and far plane distance. For stereo rendering, it’s necessary to have one viewing frustum for each eye.

To achieve stereographic rendering in OpenGL, it is necessary to render an image for the left and right eyes of the viewer. These images should be different from one another by containing horizontal parallax. Therefore, two viewing frustums are used: one for the left eye, and the other for the right eye. TerraVis uses asymmetric viewing frustums for stereo frustums [2]. The frustums are described as asymmetric because the distance between the forward direction of the camera and the left and right planes of the frustum is not equal. The reason for using asymmetric viewing frustums is to ensure that the projection planes of the left and right eye viewing frustums do not produce any vertical parallax.

For perspective projection, it is easiest to setup a viewing frustum using a function such as `gluPerspective`: this function defines the frustum in terms of a field of view,
Figure 8.6: A perspective projection viewing frustum can be viewed as a pyramid with the top chopped off. Geometry inside the pyramid is projected onto the near clipping plane.

aspect ratio, and near and far clipping distances. An asymmetric frustum in OpenGL can be defined using the glFrustum function, which allows the left, right, top, bottom, near and far parameters to be set. These values set the distances from the center of the near clipping plane to the edges of the frustum bounds (see Figure 8.7). While these parameters are less natural to work with, it allows more control and flexibility over the shape of the frustum.

With a virtual stereo camera, there are additional parameters that must be taken into account. The eye separation (interocular separation) is set to the distance between the centers of the left and right eyes. The focal distance determines the depth at which the viewer has their eyes focused. The aspect is the width to height ratio of the viewing frustums, and finally the aperture refers to the horizontal field of view of the stereo frustums.
Figure 8.7: The \textit{left}, \textit{right}, \textit{top}, and \textit{bottom} parameters of \texttt{glFrustum}. The \textit{near} and \textit{far} parameters determine the distances from the eye to the near and far clipping planes, respectively.

The calculations to define the left and right viewing frustums are shown in Algorithms 22 and 23, respectively. These are illustrated in more detail by Paul Bourke’s notes on stereographics\textsuperscript{23}. The $wd2$ value is half the width of a symmetric viewing frustum with a horizontal field of view equal to \textit{aperture} degrees; it can be calculated as \texttt{near} $\ast \tan(\texttt{apertureRadians}/2)$. The \texttt{ndfl} value is simply the near clipping plane distance divided by the focal length, and is used to scale the horizontal displacement of the \textit{left} and \textit{right} parameters as the \textit{near} and \textit{focalLength} values increase in disparity.

\begin{algorithm}
\caption{LeftEyeFrustum}
\begin{algorithmic}[1]
\State $left \leftarrow -\texttt{aspect} \ast wd2 + 0.5 \ast \texttt{eyeSeparation} \ast \texttt{ndfl}$
\State $right \leftarrow \texttt{aspect} \ast wd2 + 0.5 \ast \texttt{eyeSeparation} \ast \texttt{ndfl}$
\State $top \leftarrow wd2$
\State $bottom \leftarrow -wd2$
\end{algorithmic}
\end{algorithm}

\textsuperscript{23}\url{http://paulbourke.net/miscellaneous/stereographics/stereorender/}
Algorithm 23 RightEyeFrustum

1: \( \text{left} \leftarrow \text{-aspect} \times \text{wd} - 0.5 \times \text{eyeSeparation} \times \text{ndfl} \)
2: \( \text{right} \leftarrow \text{aspect} \times \text{wd} - 0.5 \times \text{eyeSeparation} \times \text{ndfl} \)
3: \( \text{top} \leftarrow \text{wd} \)
4: \( \text{bottom} \leftarrow -\text{wd} \)

After calculating the viewing frustums, the scene must be rendered once using the left frustum and again using the right frustum. This is done by calling \text{glDrawBuffer(GL\_BACK\_LEFT)} and \text{glDrawBuffer(GL\_BACK\_RIGHT)}. With stereo rendering, four buffers are used (quad-buffered stereo) to take advantage of double-buffering for both the left and right eye images, hence the “back left” and “back right” buffers.

Figure 8.8: Left (blue) and right (red) viewing frustums shown with a focal distance that exceeds the near clipping plane distance. Objects in front of the focal plane will have negative horizontal parallax, and objects behind will have positive horizontal parallax. The images for the left and right eyes is projected onto the near clipping plane.
Chapter 9

Head Tracking

An optional feature of TerraVis is tracking a user’s head to provide even greater immersion and interactivity. By knowing where a user’s head is located relative to the display, the viewing frustum used to render the scene can be adjusted to create a virtual window into the three-dimensional scene. There are several systems and techniques for head tracking; many times, these systems can be expensive and difficult to work with. This chapter describes head tracking using widely available and inexpensive hardware: the Nintendo Wii Remote\textsuperscript{24} (Wiimote). The use of the Wiimote as a head tracking device is inspired by Johnny Chung Lee’s project, \textit{Head Tracking for Desktop VR Displays using the Wii remote}\textsuperscript{25}.

9.1 Hardware

The Wiimote is an ideal choice for head tracking in a lab environment with less natural lighting. A Wiimote contains an infrared camera and built-in image processing that provides camera-space coordinates of up to four infrared light sources. In addition, a Wiimote can communicate with other devices using Bluetooth wireless radio. A

\begin{footnotesize}
\begin{itemize}
\item \textsuperscript{24}http://www.nintendo.com/wii/console/controllers
\item \textsuperscript{25}http://johnnylee.net/projects/wii/
\end{itemize}
\end{footnotesize}
number of libraries exist that provide programmers a convenient API to connect and retrieve information from a Wiimote; TerraVis uses WiiRemoteJ, which is a pure Java library that works on Linux, Windows, and Macintosh OS X operating systems.

The hardware setup for head tracking with a Wiimote requires only a few components: at least one Wii remote, a display, a Bluetooth dongle (or built-in Bluetooth support) attached to a computer, and two infrared light sources affixed to some head gear or glasses with a battery power supply. Such a system is shown in Figure 9.1. For this type of setup, it is required that the infrared lights are separated by a constant distance. So long as the IR lights remain in the Wiimote camera’s field of view, it is possible to calculate the distance to the lights.

![Diagram of Wiimote head tracking setup](diagram.png)

Figure 9.1: Wiimote head tracking setup using a large display, a machine connected to the Wiimote using a Bluetooth dongle, and a custom-built infrared light attachment.

[^26]: http://wiibrew.org/wiki/Wiimote/Library
One of the most important components in the system is the infrared light attachment. The infrared light must be intense enough to be visible at a distance from the Wiimote camera. Infrared LEDs also typically emit light in a cone, meaning they will not appear if the angle between the LED and the camera is too great. We designed two custom-built infrared light attachments to address the needs of intensity and increased emission angle, shown in Figure 9.2. Both designs could be affixed to the top of a pair of NVIDIA 3D Vision glasses, or simply taped to a cap. Both IR LED attachments are powered by a lithium ion rechargeable battery.

![Infrared Light Attachments](image)

Figure 9.2: Two infrared light devices designed and built by Scott Baker. Both models use the same infrared LEDs that emit light in a cone. The first model in (a) has spherical plastic casing intended to scatter the light in multiple directions. In (b), three LEDs are affixed to each side, emitting light in a greater area but from the same general area; the Wiimote sees the three LEDs as a single IR light source. The second device exhibited better performance than the first.

### 9.2 Calculating Depth

The infrared camera of the Wiimote has built-in image processing algorithms that calculate the centers of infrared light sources in *camera space*. Up for four light sources are tracked by the Wiimote, and the coordinates are represented as \((x, y)\)
pairs in $[0,1]$. The coordinate $(0,0)$ represents the bottom-left corner of the camera’s viewing area, and $(1,1)$ is the top-right. The Wiimote is also capable of detecting the relative intensity of infrared light sources, shown in Figure 9.3 as circles with larger radii.

![Figure 9.3: Graphical representation of Wiimote IR camera space. Three IR lights are seen by the camera, and their positions are calculated in the $[0,1]$ range for x and y. The size of the lights indicates their intensity.](image)

The 2D coordinates of an infrared light correspond to a ray of light in 3D. With only a single infrared light, the 2D coordinates alone do not provide enough information to determine the depth of that light. Two techniques are described below for calculating depth to a viewer using two infrared lights. Another approach might be to calibrate the tracking according to light intensity at various depths; this was not explored, as it was felt the intensity varied too much to be used as a reliable and accurate measurement.

### 9.2.1 Depth by Field of View

The first method of calculating the distance from the camera to the viewer is to assume the IR camera has a viewing area that is similar to a viewing frustum. Such a viewing area is shown in Figure 9.1. The exact details of the Wiimote’s IR camera
were not available, so experimentation was necessary to determine the field of view (horizontal and vertical) and effective range of the camera.

If the viewing volume is known, and two infrared lights have a known and fixed horizontal separation, then it is possible to calculate the distance to the center of the two lights. We assume the physical separation between the two lights is a constant $s$, and the distance to the center of the lights is $d$. As shown in Figure 9.4, the horizontal separation in camera space, $\Delta x$, will be 1 when the two lights are on the edges of the viewing volume.

![Figure 9.4: The value $k$ measures distance to the center of two IR lights on the edges of the left and right viewing frustum planes. If two lights have $\Delta x = 1$ in camera space, they have $s$ physical separation, where $s$ is the actual fixed distance between the two lights.](image)

Initially, $s$ is the only known constant as it is measured and depends on the IR light device being used. The value $k$ can also be found by positioning the two IR lights in front of the Wiimote, moving them closer until $\Delta x = 1$, then measuring the distance. Once both $s$ and $k$ are known, the field of view can be calculated. The following equations are equivalent and can be used to find the value of one unknown constant so long as the other two are known.
\[
\frac{s}{2k} = \tan(\text{fov}/2) \tag{9.1}
\]
\[
k = \frac{s}{2 \times \tan(\text{fov}/2)} \tag{9.2}
\]
\[
\text{fov} = 2 \times \arctan(s/(2k)) \tag{9.3}
\]

Figure 9.5: When the two IR lights are further away from the camera, they have a smaller separation in camera space.

When \(\Delta x = 1\), the lights are at a distance \(d = k\). As the IR lights move further back from the camera, \(\Delta x\) will decrease and \(d\) will increase. In other words, \(d\) is inversely proportional to \(\Delta x\), or \(d = k/\Delta x\). Therefore, the final equation to calculate the distance \(d\) to two IR lights given \(\Delta x\), the field of view, and the physical separation between the two lights \(s\):

\[
d = \frac{s}{2\Delta x \times \tan(\text{fov}/2)} \tag{9.4}
\]

One problem we encountered is that measuring \(s\) and \(k\) to calculate the field of view did not produce distances that were as accurate as hoped. One assumption in the above model is that the focal point is exactly at the tip of the Wiimote. Instead of measuring \(k\) to calculate the field of view, we found the best results when adjusting the field of view to produce a \(k\).
Calculated distance using $k/\Delta x$

<table>
<thead>
<tr>
<th>Actual Distance (ft)</th>
<th>Distance in feet with FOV 42.7°</th>
<th>Distance in feet with FOV 45.0°</th>
<th>Distance in feet with FOV 48.3°</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.00</td>
<td>0.95</td>
<td>0.88</td>
</tr>
<tr>
<td>2</td>
<td>1.99</td>
<td>1.88</td>
<td>1.74</td>
</tr>
<tr>
<td>3</td>
<td>2.97</td>
<td>2.80</td>
<td>2.60</td>
</tr>
<tr>
<td>4</td>
<td>3.92</td>
<td>3.75</td>
<td>3.44</td>
</tr>
<tr>
<td>5</td>
<td>4.96</td>
<td>4.68</td>
<td>4.32</td>
</tr>
<tr>
<td>6</td>
<td>5.94</td>
<td>5.66</td>
<td>5.23</td>
</tr>
<tr>
<td>7</td>
<td>6.97</td>
<td>6.58</td>
<td>6.01</td>
</tr>
<tr>
<td>8</td>
<td>7.89</td>
<td>7.45</td>
<td>6.79</td>
</tr>
<tr>
<td>9</td>
<td>8.95</td>
<td>8.45</td>
<td>7.81</td>
</tr>
<tr>
<td>10</td>
<td>9.83</td>
<td>9.43</td>
<td>8.58</td>
</tr>
<tr>
<td>11</td>
<td>10.90</td>
<td>10.43</td>
<td>9.69</td>
</tr>
<tr>
<td>12</td>
<td>12.00</td>
<td>11.32</td>
<td>10.68</td>
</tr>
</tbody>
</table>

Table 9.1: Depth to IR lights calculated using $k/\Delta x$ with $s = 7.25$ inches and different values for fov.

To find the best set of values, we placed the IR light attachment at 12 feet from the Wiimote camera and adjusted the field of view until the calculation $k/\Delta x$ also gave 12 feet. Then, we measured the actual distance against the computed distance for different FOV values; some of the results are shown in Table ???. This experiment showed that a horizontal field of view of 42.7° produced results with maximum error of about 2 inches at 10 feet from the camera.

### 9.2.2 Depth by Calibration

After experimenting with field of view, the second approach we tried to calculate depth was calibration. From the previous experiments, it was noted that $\Delta x$ decreases as $d$ increases, which gave the equation $d = k/\Delta x$. Instead of calculating $k$ using field of view and $s$, variables $A$ and $B$ are used to make this a linear equation:

$$d = A(1/\Delta x) + B$$

(9.5)
Now, instead of determining a value of $k$, it is necessary to find the values for $A$ and $B$. We recorded $\Delta x$ in camera space by moving the lights away from the Wiimmote in 12 inch increments. These measurements are saved as $(x, y)$ pairs, where $x = 1/\Delta x$ and $y$ is distance in inches. After plotting these values, a linear model can be fitted to the data, as shown in Figure 9.6, which gives the values for $A$ and $B$. A comparison of using calibration and using the field of view calculations is shown in Table 9.2.

![Head Tracking Calibration Linear Fit](image)

Figure 9.6: Linear fit for recorded values of $1/\Delta x$ against actual distance. The resulting fit is the line $D = \frac{7.0814}{\Delta x} + 0.1783$.

<table>
<thead>
<tr>
<th>Actual Distance (in)</th>
<th>$\Delta x$</th>
<th>$k/\Delta x$</th>
<th>$A(1/\Delta x) + B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>0.5863</td>
<td>12.00</td>
<td>12.27</td>
</tr>
<tr>
<td>24</td>
<td>0.2946</td>
<td>23.88</td>
<td>24.23</td>
</tr>
<tr>
<td>36</td>
<td>0.1974</td>
<td>35.64</td>
<td>36.06</td>
</tr>
<tr>
<td>48</td>
<td>0.1496</td>
<td>47.04</td>
<td>47.53</td>
</tr>
<tr>
<td>60</td>
<td>0.1182</td>
<td>59.52</td>
<td>60.09</td>
</tr>
<tr>
<td>72</td>
<td>0.0987</td>
<td>71.28</td>
<td>71.92</td>
</tr>
<tr>
<td>84</td>
<td>0.0841</td>
<td>83.64</td>
<td>84.36</td>
</tr>
<tr>
<td>96</td>
<td>0.0743</td>
<td>94.68</td>
<td>95.47</td>
</tr>
<tr>
<td>108</td>
<td>0.0655</td>
<td>107.40</td>
<td>108.27</td>
</tr>
<tr>
<td>120</td>
<td>0.0596</td>
<td>117.96</td>
<td>118.89</td>
</tr>
<tr>
<td>132</td>
<td>0.0538</td>
<td>130.80</td>
<td>131.81</td>
</tr>
<tr>
<td>144</td>
<td>0.0489</td>
<td>144.00</td>
<td>145.10</td>
</tr>
</tbody>
</table>

Table 9.2: Comparison of depth calculations using field of view and calibration.
9.3 Calculating Head Position

To make use of the head tracking information, it’s necessary to have the 3D coordinates of the center of the two IR lights (roughly where the head is). The previous section shows two ways of how to compute the depth, but the viewer may move horizontally and vertically, as well. As mentioned earlier, the 2D camera space coordinates of an IR light represents a ray along which an infinite number of points exist, as shown in Figure 9.7.

Figure 9.7: Two IR lights at $d$ units from the camera (the depth plane) projected to 2D camera space coordinates.

To calculate the 3D positions of the two IR lights, the depth is first calculated using one of the aforementioned techniques. In TerraVis, the calibration method is used. The two IR lights must be somewhere on the depth plane, a plane that is perpendicular to the Wiimote camera’s viewing direction at $d$ units away. The two rays representing the IR lights are calculated using the camera space coordinates,
which is shown in Algorithm 24. The rays are positioned at the camera origin, and directed toward the points on the projection plane. The horizontal and vertical field of view must be known for this calculation. TerraVis uses $fovV = 32.7^\circ$, and $fovH = 42.7^\circ$.

Algorithm 24 CalcIRLightRay(x, y)

1: $v_x \leftarrow \tan\left(\frac{fovH}{2}\right) \ast (x - 0.5) \ast 2$
2: $v_y \leftarrow \tan\left(\frac{fovV}{2}\right) \ast (y - 0.5) \ast 2$
3: $v_z \leftarrow -1$
4: return $v$

After the two IR light rays are created they are intersected with the depth plane, giving two 3D points. These points are averaged, producing a central point that lies between the two IR lights in 3D space. This final point represents the head position relative to the camera. However, what we ultimately want is the head position relative to the center of the display. A configuration file stores a the displacement from the center of the display to the tip of the Wiimote, which is then subtracted from the head position.

One issue with head tracking using this IR lights and the Wiimote is that noise in the data can cause the head position to jitter without the viewer moving. The quality of the Bluetooth connection also plays an important role in the overall quality of the tracking: if the frequency data is received from the Wiimote is higher, the tracking will be smoother. To compensate for these issues, TerraVis stores the last $n$ head positions in a history, and when the current head position is queried it returns the average of all $n$ positions. The larger the value of $n$, the smoother the tracking becomes; however, it also introduces a delay. A small value, such as $n = 4$, provided acceptable results.
9.4 Virtual Window

Once the head position is known, a viewing frustum can be constructed that turns the
scene into a virtual window. For monoscopic viewing, a single asymmetric viewing
frustum (see 8.4) is created such that the display becomes a “virtual window” into
the 3D scene. For stereo viewing, two asymmetric frustums are created: one for each
eye. To process to produce the frustums is nearly the same as in normal stereoscopic
viewing; however, the dimensions of the display must be known.

The steps to generate the perspective projection matrix for head tracking is shown
in Algorithm 25. The function takes the dimensions of the display and an offset
vector as parameters. The offset is used for stereo viewing, and should be set to a
vector pointing to the left and right eyes from the center of the viewing position.
For example, assuming eye separation of 2.8 inches and the viewer at (0,0,0) facing
(0,0,-1), the offset should be set to (0,0,1.4) for the right eye. The values can be used
in a call to glFrustum.

Algorithm 25 VRPerspective(displayW, displayH, offset)

1: head ← headPosition + offset
2: hw ← displayW / 2
3: hh ← displayH / 2
4: left ← -(hw + head.x) / head.z * near
5: right ← (hw - head.x) / head.z * near
6: bottom ← -(hh + head.y) / head.z * near
7: top ← (hh - head.y) / head.z * near

Finally, a translation matrix is used to move the camera in 3D the same as the
viewer moves their head in the physical world. The translation matrix simply uses
the negative of the head position as the translation. If stereo is enabled, the current
eye offset vector is added to the translation. The final view matrix is set to the 3D
camera view matrix multiplied with the head tracking translation matrix.
Chapter 10

Review

Visualization of geologic data is a difficult problem: scientists need to view their data at the highest resolution with the most relevant information as possible. To enhance the understanding of such a complex system as the Earth, it’s insufficient to visualize individual models and expect a geoscientist to put the pieces together. It is necessary to generate an effective and communicative view for each type of data, and ensure these views can be pulled together into a cohesive scene. Beyond this challenge, massive file sizes require attention to carefully optimizing the processing and rendering of data.

Providing a complete system that addresses all of the challenges of geoscience visualization is simply too large of a problem for an individual. The interdisciplinary collaboration of many researchers and developers is required to realize a framework that solves many of the existing problems. Some groups are attempting to achieve this vision, such as GEON, where the OpenEarth Framework serves as an interface to access and visualize data. The work of this thesis has focused primarily on exploring ways to provide highly efficient and effective visualizations.

Within this thesis, the requirements of a modern visualization tool specifically designed for seismic data have been described: interactivity, multiple-data integra-
tion, and 3D/4D viewing. Visualization techniques and implementation details are described for some of the major forms of data including digital elevation models, earthquake catalogs, moment tensor solutions, fault slip distributions, InSAR imagery, and scalar fields. In addition, the application of advanced visualization through stereoscopic rendering and head-tracking is explained. TerraVis is an implementation of the aforementioned techniques and technologies and provides a proof of the ideas and algorithms in the preceding chapters.

10.1 Goals Achieved

The contributions of this thesis have been demonstrated through the use of TerraVis:

• A highly efficient and robust pipeline for loading, constructing, and rendering digital elevation models has been realized. DEMs from any region of the Earth can be loaded into TerraVis and rendered in high resolution with little effort.

• Massive catalogs of earthquakes can be rendered. A user can interactively view a window of earthquakes from seconds to several years without reloading, sorting, or otherwise manipulating the data. Earthquakes can be displayed over time as they occur with minimal computational cost.

• Volumetric data can be visualized using the newest and most advanced form of volume rendering available. These renderings can be viewed alongside any other data types that TerraVis can process.

• Stereoscopic viewing and head tracking has been successfully integrated for enhanced viewing and depth perception. This is one of the few geology visualization tools that offers effective stereo viewing with cost-effective technologies.
• Despite large data sets, rendering is highly optimized. High frame rates can be achieved with even modest graphics hardware\textsuperscript{27}, making visualizations accessible to users without high-end lab equipment.

The software has been used in several presentations to successfully demonstrate visualizations of geologic data. In particular, we have focused on showing seismic data sets from the 2010 earthquake in Haiti. Other regions of interest included Hawaii and southern California. An earlier version of TerraVis was used at the workshop Rebuilding for Resilience: How Science and Engineering can Inform Haiti’s Reconstruction, held in March 2010 at the University of Miami.

10.2 Future Work

There are several ways in which TerraVis could be improved. Additionally, the current state of the software is advanced enough to permit some experimentation with ideas that we have not yet had the opportunity to try. TerraVis is capable of multiple-data integration, but the user must make connections between data sets visually. One area that needs to be looked into is ways to produce and show relationships between data, not just display them side-by-side.

For digital elevation models, high resolution grids can be rendered for smaller regions of interest; for larger regions, lower resolution grids must be used. It would be useful to incorporate some level-of-detail algorithms to move between high and low resolution elevation data according to the area the user is viewing. A space partitioning data structure, such as a quadtree, should be used to optimize the rendering even further. Furthermore, data must currently be downloaded and imported from disk. Ideally, the user would be presented with a 2D map where they could select a region,

\textsuperscript{27}TerraVis can render a $1620 \times 2160$ resolution DEM with roughly 90k earthquakes (total of approx. 8.79 million triangles) at over 50 FPS on an AMD Radeon HD 6750M, a mid-range mobile GPU.
have the latitude and longitude bounds calculated, and download the elevation data directly.

The current method of visualizing earthquake catalogs is to render each earthquake as opaque polygonal spheres with color and scaling parameters. This is especially effective when using stereo viewing, as the enhanced depth perception is useful for viewing clusters of earthquakes. However, we would also like to experiment with viewing each event as a translucent object, where clusters become the focus of attention rather than individual events. We have already taken advantage of the 4D nature of catalogs with scrolling time windows, but further investigation in the temporal patterns exhibited by earthquake catalogs is needed.

Slip distributions are, at the moment, displayed as planar meshes. In reality, the slippage is not constrained to a plane and displacement vectors are associated with cells in the distribution. A more interesting visualization would perhaps be the use of volume rendering to show the actual displacement along the fault over time.

The head tracking with a Wii remote is limited by the narrow field of view (about $43^\circ$ horizontally) of the infrared camera. A user cannot move side to side too much if they are closer to the display, and the further away from the display they move the less intense the IR lights become, making the head tracking less smooth.

Finally, the volume rendering component can be greatly improved. The current renderer implementation is not as efficient or high-quality as it could be.
Appendix A

Geodesic Sphere Algorithm

For modeling some point-based data, geodesic spheres are used to provide a variable level-of-detail shape. To construct the sphere, an icosahedron is used as the base, which can be referred to as level-0. To get a higher level sphere, the triangle faces from the icosahedron are tessellated (cut up into smaller triangles) and the vertices are pushed away from the center until they are at radius distance from the center of the icosahedron.

Figure A.1: Three geodesic spheres of varying levels, starting with an icosahedron.
A.1 Tessellation

An icosahedron is used as the basis for any geodesic sphere. The faces of the icosahedron, which are all triangles, can be split into several rows of smaller triangles. The level of a sphere determines the degree of tessellation, and refers to the number of additional triangle rows each icosahedron face is split into. In other terms, the level indicates the number of new vertices spaced evenly along each edge of the original icosahedron. For example, a level 1 sphere has a single vertex added half-way between the two vertices of each edge of the icosahedron.

Figure A.2: The original icosahedron faces tessellated at different levels.

By adding \( n \) new vertices along the edges, each of the 20 faces of the icosahedron is composed of \( n + 1 \) rows of triangles. Each row has two more triangles than the row above, with the first row always having a single triangle (see figure A.2). The number of triangles in a sphere of level \( n \) can then be calculated as follows:
\[ \text{sphereTriangles}(n) = 20 \sum_{i=0}^{n}(2i + 1) \]
\[ = 20(2 \sum_{i=0}^{n} i + \sum_{i=0}^{n} 1) \]
\[ = 20\left(\frac{2n(n + 1)}{2} + (n + 1)\right) \]
\[ = 20(n^2 + 2n + 1) \]
\[ = 20(n + 1)^2 \]

Adding a row of triangle also adds more vertices (figure A.3). It is straightforward to calculate the total number of vertices that each icosahedron face uses after tessellation:

\[ \text{faceVerts}(n) = \sum_{i=0}^{n+2} i \]
\[ = \sum_{i=0}^{n} (i) + (n + 1) + (n + 2) \]
\[ = \frac{n(n+1)}{2} + (n + 1) + (n + 2) \]
\[ = \frac{n^2 + 5n + 6}{2} \]
\[ = \frac{(n + 3)(n + 2)}{2} \]

Calculating the total number of vertices in a sphere is not as simple as multiplying \( \text{faceVerts}(n) \) by 20, the number of icosahedron faces. The vertices on the icosahedron edges are shared between at least two faces. When calculating the mesh for the sphere,
(a) A level-3 sphere has \(1 + 2 + 3 + 4 + 5 = 15\) total vertices per icosahedron face; some of these are shared between faces.

(b) Out of the 15 face vertices: 9 edge vertices (red), 3 icosahedron vertices (black), and 3 inner vertices (green)

Figure A.3: A level-3 sphere has \(1 + 2 + 3 + 4 + 5 = 15\) vertices per icosahedron face

It is preferable to avoid repeating shared vertices for performance and memory. To avoid redundant vertices, each vertex is labeled as one of three types:

1. **Icosahedron vertices** are part of the original icosahedron; there are always 12 total, regardless of the sphere’s level. Each icosahedron face will use 3 of these vertices.

   \[\text{faceIcoVerts}(n) = 3\]

2. **Edge vertices** are any vertices on an icosahedron edge excluding the endpoints (icosahedron vertices). There are 30 edges in an icosahedron, and each face has 3 of these edges. The level \(n\) is the additional number of vertices placed on these edges.

   \[\text{faceEdgeVerts}(n) = 3n\]
3. **Inner vertices** are vertices that do not lie on any icosahedron edge. There are no inner vertices for level-0 or level-1 spheres.

\[
\text{faceInnerVerts}(n) = \text{faceVerts}(n) - \text{faceEdgeVerts}(n) - \text{faceIcoVerts}(n)
\]
\[
= \frac{(n+3)(n+2)}{2} - 3n - 3
\]

To calculate the total number of vertices in a level-\(n\) sphere, the number of edge vertices, icosahedron vertices, and inner vertices per face is summed:

\[
\text{sphereVerts}(n) = 20 * \text{faceInnerVerts}(n) + 30n + 12
\]
\[
= 20 \times \left( \frac{(n+3)(n+2)}{2} - 3n - 3 \right) + 30n + 12
\]
\[
= 10n^2 + 20n + 12
\]

### A.2 Vertex Generation

The first step in creating a geodesic sphere is calculating the vertex positions. The algorithm is split into three steps: one for each of the three types of vertices. The algorithm takes an icosahedron as a base; given the simplicity of the icosahedron, its vertices, edges, and faces can be hard-coded for efficiency. The vertex positions are all created such that the origin is the center of the sphere. Each subroutine adds the vertices to a single array, `sphereVerts`; the index \(i\) is used as a pointer to the next free space in `sphereVerts`. 
Algorithm 26 CalculateSphereVertices(n,r)

1: Allocate sphereVerts[10n^2 + 20n + 12]
2: i ← 0
3: i ← i + CalculateIcoVerts(n,r,i)
4: i ← i + CalculateEdgeVerts(n,r,i)
5: i ← i + CalculateInnerVerts(n,r,i)

The sphere’s icosahedron vertices are already provided by the original icosahedron; however, these vertices need must be at r (radius) units from the center of the sphere (origin). The routine CalculateIcoVerts simply normalizes the positions of each vertex v to be 1 unit from the origin and then scales the distance to r.

Algorithm 27 CalculateIcoVerts(n,r,i)

1: for each vertex v ∈ ico.verts do
2:    sphereVerts[i] ← v/|v| * r
3:    i ← i + 1
4: end for
5: Return i

For the edge vertices, each edge of the icosahedron is visited and its endpoints a and b are stored. A step vector is calculated to position new vertices along the edge. Similar to the icosahedron vertices, each position is pushed to be r units from the origin.

Algorithm 28 CalculateEdgeVerts(n,r,i)

1: for each edge e ∈ ico.edgeVerts do
2:    a ← ico.verts[e[0]]
3:    b ← ico.verts[e[1]]
4:    step ← (b - a)/(n + 1)
5: for j := 1 to n do
6:    v ← a + step * j
7:    sphereVerts[i] ← v/|v| * r
8:    i ← i + 1
9: end for
10: end for
11: Return i
(a) Edge vertices along edge \((a, b)\); the edge vertices do not include \(a\) or \(b\).

(b) Inner vertices for a face. The vertices are labeled \((row, col)\) and added in the order \((2, 1), (3, 1), (3, 2)\).

Figure A.4: Calculating edge and inner vertices for \(n=3\).

Finally, the inner vertices are calculated in a similar manner as the edge vertices. Each of the icosahedron’s 20 triangle faces is visited and its three vertices stored. The \(down\) and \(right\) vectors are calculated to position the inner vertices. For any size \(n\), the first inner vertex will always occur at \(a + 2 \times down + right\); this number of shifts down is called the \(row\), and the number of shifts right is the \(col\). The number of vertices on a row is always 1 more than the previous row. Once again, these positions are pushed to \(r\) distance from the origin.
Algorithm 29 CalculateInnerVerts(n,r,i)

1: for each triangle $t \in ico.triVerts$ do
2:     $a \leftarrow ico.verts[t[0]]$
3:     $b \leftarrow ico.verts[t[1]]$
4:     $c \leftarrow ico.verts[t[2]]$
5:     down $\leftarrow (b - a)/(n + 1)$
6:     right $\leftarrow (c - b)/(n + 1)$
7: for row := 2 to $n$ do
8:     for col := 0 to row - 2 do
9:         $v \leftarrow a + down \times row + right \times (col + 1)$
10:        sphereVerts[$i$] $\leftarrow v/|v| \times r$
11:        $i \leftarrow i + 1$
12:    end for
13: end for
14: Return $i$

Figure A.5: The storage of vertices in an array after running CalculateSphereVertices.

A.3 Triangle Indexing

Calculating the triangle indices for the tessellated icosahedron is a bit trickier than generating the vertices. The vertices were added to sphereVerts in a manner that ensures there are no redundant vertices; however, this necessitates more work when indexing the vertices for each triangle. The icosahedron data structure contains three index arrays:

1. triVerts: contains 3-element arrays for each triangle to index vertices. A triangle $t_i$ indexes three vertices in CCW order.
2. **edgeVerts**: contains 2-element arrays for each edge to index vertices. The start vertex for and edge \( e_i \) is \( edgeVerts[i][0] \) and the end vertex is \( edgeVerts[i][1] \).

3. **triEdges**: contains 3-element arrays for each triangle to index edges. A triangle \( t_i \) indexes three edges in CCW order.

The tessellated triangle indices are calculated by visiting each icosahedron triangle. It is important to be able to reference vertices in a consistent order for each icosahedron triangle: an auxiliary array is created that contains indices to \( sphereVerts \), and the elements of this array are ordered from top-to-bottom, left-to-right as in figure A.6a.

![Diagram](image)

(a) The indices in \( aux \) are ordered from top-to-bottom, left-to-right. Three pointers, \( lp, rp, \) and \( bp \) are shown at their starting positions for the left, right, and bottom edges.

(b) Triangle \( t_1 \) stores vertex indices \((v_0, v_3, v_4)\) and edges indices \((e_3, e_4, e_5)\). Triangle \( t_2 \) stores vertex indices \((v_0, v_1, v_3)\) and edges indices \((e_1, e_2, e_3)\). The arrows on the edges indicate direction, such that \( e_3 = (v_3, v_0) \) where \( v_3 \) is the start. For \( t_2 \), all of the edges list the vertices CCW; for \( t_1 \), however, \( e_3 \) is flipped.

![Figure A.6: Detecting flipped edges.](image)

While creating the auxiliary array, the corner vertices of an icosahedron face \( t_i \) are easy to locate in \( sphereVerts \), as they are always at the same positions as the indices in \( triVerts[i] \). As mentioned previously, these indices are always ordered CCW.
The inner vertices are also not difficult to locate: a pointer to the first inner vertex of a icosahedron face is at \(12 + 30n + i \times \text{faceInnerVerts}(n)\). Inner vertices are always added top-to-bottom, left-to-right.

The edge vertices are more problematic, since their ordering may not be CCW for the triangle face (see figure A.6b). For this algorithm, it is necessary to have pointers to the first edge vertex on the left, right, and bottom edges. The left and right pointers should be at the top and move down; the bottom pointer should be at the left and move right. For some icosahedron triangles, the edge vertices will not all be added in a CCW order as the edges are shared. If an edge is indexed in reverse from what a face expects, it is considered flipped for that face. When an edge is flipped, the edge pointer starts at the opposite end of the edge and moves in the opposite direction.

The algorithm for calculating the sphere triangles iterates through each triangle face in the icosahedron and performs three steps: checking for flipped edges, creating the auxiliary index array, and finally adding the triangle indices. These substeps are performed in the same code block, but split up here for clarity.

\begin{algorithm}
\caption{CalculateSphereTriangles}
\begin{algorithmic}[1]
\For{each triangle \(t \in \text{ico.triVerts}\)}
\State CheckFlippedEdges
\State CalculateAuxIndexArray
\State CalculateTriIndices
\EndFor
\end{algorithmic}
\end{algorithm}

When checking for flipped edges, the vertex indices in \textit{triVerts} are compared with the vertex indices in the triangle's left, right, and bottom edges. If the first vertex index does not match, then the order must be flipped. For the right edge, it is considered flipped if it actually is CCW; this is to make sure the right edge pointer starts at the top of the right edge instead of the bottom.
Algorithm 31 CheckFlippedEdges

1: leftEdge ← ico.edgeVerts[t.triEdges[0]]
2: rightEdge ← ico.edgeVerts[t.triEdges[1]]
3: bottomEdge ← ico.edgeVerts[t.triEdges[2]]
4: flipL ← leftEdge[0] ≠ t.triVert[0]
5: flipB ← bottomEdge[0] ≠ t.triVert[1]
6: flipR ← rightEdge[0] ≠ t.triVert[0]

The auxiliary array is calculated by visiting each row of vertices on the face. The first row is the top of the triangle, which has only a single icosahedron vertex. The last row contains the second and third icosahedron vertices for that face with all of the bottom edge vertices between. Rows between the top and bottom have a single left edge vertex, followed by some inner vertices, and ended with a single right edge vertex.

Algorithm 32 AddEdgeVertex(p,flip)

1: if flip then
2: aux[added] ← p
3: p ← p + 1
4: else
5: aux[added] ← p
6: p ← p - 1
7: end if
8: added ← added + 1
Algorithm 33 CalculateAuxIndexArray

1: $lp \leftarrow \text{flipL}(\text{triEdges}[0] + 1) \times n + 11 : \text{triEdges}[0] \times n + 12$
2: $bp \leftarrow \text{flipB}(\text{triEdges}[1] + 1) \times n + 11 : \text{triEdges}[1] \times n + 12$
3: $rp \leftarrow \text{flipR}(\text{triEdges}[2] + 1) \times n + 11 : \text{triEdges}[2] \times n + 12$
4: Allocate $\text{aux}[(n + 3)(n + 2)/2]$
5: $\text{added} \leftarrow 0$
6: for $\text{row} := 0$ to $n + 1$ do
7:   if $\text{row}$ equals 0 then
8:     $\text{aux}[\text{added}] \leftarrow \text{triV}erts[0]$
9:     $\text{added} \leftarrow \text{added} + 1$
10: else if $\text{row}$ equals $n + 1$ then
11:    $\text{aux}[\text{added}] \leftarrow \text{triV}erts[1]$
12:    $\text{added} \leftarrow \text{added} + 1$
13:    for $j := 0$ to $n - 1$ do
14:       AddEdgeVertex($bp, \text{flipB}$)
15:    end for
16:    $\text{aux}[\text{added}] \leftarrow \text{triV}erts[2]$
17:    $\text{added} \leftarrow \text{added} + 1$
18: else
19:    AddEdgeVertex($lp, \text{flipL}$)
20:    for $j := 0$ to $n - 1$ do
21:       $\text{aux}[\text{added}] \leftarrow ip$
22:       $ip \leftarrow ip + 1$
23:       $\text{added} \leftarrow \text{added} + 1$
24:    end for
25:    AddEdgeVertex($rp, \text{flipR}$)
26: end if
27: end for

Once the auxiliary array has been created, adding the triangles is straight-forward. Each row of triangles is visited in the icosahedron face. Triangles may have their top top vertex facing up or down).
Algorithm 34 CalculateTriIndices

1: \( i \leftarrow 0 \)
2: \( \text{vertsOnRow} \leftarrow 1 \)
3: \( \text{trisOnRow} \leftarrow 1 \)
4: \( \forall \text{row} := 0 \text{ to } n \) do
5: \( a \leftarrow \{i, i + \text{vertsOnRow}, i + \text{vertsOnRow} + 1\} \)
6: \( b \leftarrow \{a[0] + 1, a[0], a[2]\} \)
7: \( \forall j := 0 \text{ to } \text{trisOnRow} \) do
8: \( k \leftarrow \text{tri}/2 \)
9: \( \text{if } j \text{ is even then} \)
10: \( \text{triangles}[t] \leftarrow \{\text{aux}[a[0] + o], \text{aux}[a[1] + o], \text{aux}[a[2] + o]\} \)
11: \( \text{else} \)
12: \( \text{triangles}[t] \leftarrow \{\text{aux}[b[0] + o], \text{aux}[b[1] + o], \text{aux}[b[2] + o]\} \)
13: \( \text{end if} \)
14: \( t \leftarrow t + 1 \)
15: \( \text{end for} \)
16: \( \text{end for} \)
17: \( i \leftarrow \text{vertsOnRow} \)
18: \( \text{vertsOnRow} \leftarrow \text{vertsOnRow} + 1 \)
19: \( \text{trisOnRow} \leftarrow \text{trisOnRow} + 2 \)
public class Icosahedron {

// indices for vertices of each triangle face (20 faces total)
private static final int[][] TRI_VERTS = {
  { 0, 1, 2 }, { 0, 2, 3 },
  { 0, 3, 4 }, { 0, 4, 5 }, { 0, 5, 1 }, { 1, 9, 2 }, { 2, 9, 10 },
  { 2, 10, 3 }, { 3, 10, 6 }, { 3, 6, 4 }, { 4, 6, 7 }, { 4, 7, 5 },
  { 5, 7, 8 }, { 5, 8, 1 }, { 1, 8, 9 }, { 11, 6, 10 },
  { 11, 10, 9 }, { 11, 9, 8 }, { 11, 8, 7 }, { 11, 7, 6 }
};

// indices for edges of each triangle face (20 faces total)
private static final int[][] TRI_EDGES = {
  { 0, 1, 2 }, { 2, 3, 4 },
  { 4, 5, 6 }, { 6, 7, 8 }, { 8, 9, 0 }, { 10, 11, 1 },
  { 11, 12, 13 }, { 13, 14, 3 }, { 14, 15, 16 }, { 16, 17, 5 },
  { 17, 18, 19 }, { 19, 20, 7 }, { 20, 21, 22 }, { 22, 23, 9 },
  { 23, 24, 10 }, { 25, 15, 26 }, { 26, 12, 27 }, { 27, 24, 28 },
  { 28, 21, 29 }, { 29, 18, 25 }
};

// indices for vertices of each edge (30 edges total)
private static final int[][] EDGE_VERTS = {
  { 0, 1 }, { 1, 2 }, { 2, 0 },
  { 2, 3 }, { 3, 4 }, { 4, 0 }, { 4, 5 }, { 5, 0 },
  { 5, 1 }, { 1, 9 }, { 9, 2 }, { 9, 10 }, { 10, 2 }, { 10, 3 },
  { 10, 6 }, { 6, 3 }, { 6, 4 }, { 6, 7 }, { 7, 4 }, { 7, 5 },
  { 7, 8 }, { 8, 5 }, { 8, 1 }, { 8, 9 }, { 11, 6 }, { 11, 10 },
  { 9, 11 }, { 8, 11 }, { 7, 11 }
};

private float[][] vertices;

public Icosahedron(float side) {
  float hs = side / 2;
  float piOver5 = (float) (Math.PI / 5.0);
  float t2 = piOver5 / 2;
  float t4 = piOver5;
  float R = (float) ((side / 2) / Math.sin(t4));
  float H = (float) (Math.cos(t4) * R);
  float Cx = (float) (R * Math.cos(t2));
  float Cy = (float) (R * Math.sin(t2));
  float H1 = (float) (Math.sqrt(side * side - R * R));
  float H2 = (float) (Math.sqrt((H + R) * (H + R) - H * H));
  float Z2 = (H2 - H1) / 2;
  float Z1 = Z2 + H1;

  vertices = new float[][] {
    { hs, Z2, -H }, { -hs, Z2, -H }, { -Cx, Z2, Cy },
    { 0, -Z2, -R }, { -Cx, -Z2, -Cy }, { -hs, -Z2, H },
    { hs, -Z2, H }, { Cx, -Z2, -Cy }, { 0, -Z1, 0 }
  };
}

\caption{Java code for an icosahedron centered at origin.}
Bibliography


