LiU-ITN-TEK-A-16/057--SE

Design och Implementering av ett Out-of-Core Globrenderingssystem Baserat på Olika Karttjänster

Kalle Bladin

Erik Broberg

2016-12-02

Department of Science and Technology
Linköping University
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings universitet
601 74 Norrköping

LiU-ITN-TEK-A-16/057--SE

Design och Implementering av ett Out-of-Core Globrenderingssystem Baserat på Olika Karttjänster

Examensarbete utfört i Medieteknik
vid Tekniska högskolan vid
Linköpings universitet

Kalle Bladin
Erik Broberg

Handledare Alexander Bock
Examinator Anders Ynnerman

Norrköping 2016-12-02

Master's thesis

Design and Implementation of an Out-of-Core Globe Rendering System Using Multiple Map Services

Submitted in partial fulfillment of
the requirements for the award of the degree of

Master of Science
in
Media Technology and Engineering

Submitted by

Kalle Bladin & Erik Broberg

Examiner
Anders Ynnerman

Supervisor
Alexander Bock

Department of Science and Technology
Linköping University 2016

Abstract

This thesis focuses on the design and implementation of a software system enabling out-of-core rendering of multiple map datasets mapped on virtual globes around our solar system. Challenges such as precision, accuracy, curvature and massive datasets were considered. The result is a globe visualization software using a chunked level of detail approach for rendering. The software can render texture layers of various sorts to aid in scientific visualization on top of height mapped geometry, yielding accurate visualizations rendered at interactive frame rates.

The project was conducted at the American Museum of Natural History (AMNH), New York and serves the goal of implementing a planetary visualization software to aid in public presentations and bringing space science to the public.

The work is part of the development of the software OpenSpace, which is the result of a collaboration between Linköping University, AMNH and the National Aeronautics and Space Administration (NASA) among others.

Acknowledgments

We would like to give our sincerest thanks to all the people who have been involved in making this thesis work possible. Thanks to Anders Ynnerman for giving us this opportunity. Thanks to Carter Emmart for being an inspiring and driving force for the project and for sharing his passion in bringing knowledge and interest in astronomy to the general public. Thanks to Vivian Trakinski for making us feel needed and useful within the Openspace project and within the museum.

Thanks to Alexander Bock for his dedication in the project and the support he has given as a mentor along with Emil Axelsson during the whole project. Thanks to all the people in the OpenSpace team, including our peers Michael and Sebastian, which have been both inspiring, helpful and enjoyable to work and share the experience with.

We would like to thank all the people we have met during our time at AMNH. Kayla, Eozin, Natalia have not only been our trusted lunch mates but also great friends outside of work. Thanks to Jay for all the hardware support, and also all the rest of the people at the Science Bulletins and Rose Center Engineering for being so welcoming and helpful.

We would also like to thank Lucian Plesea for his expert support in mapping services together with Vikram Singh for setting up the map servers we could use in our software. Also, thanks to Jason, Ally and David for providing us with high resolution Mars imagery data that we could use for rendering.

A big thank you to Masha and the rest of the CCMC team as well as Ryan, who made our visit to NASA Goddard Space Flight Center the best experience possible by inspiring us and giving us insight in parts of NASA's space science.

All our friends and family who travelled from Sweden to visit us in New York, we're happy for sharing a great time with you during our leisure.

Last but not least we are very happy to have made new great friends outside of the thesis work during our stay in the United States. You have made this experience even more enjoyable.

Contents

List of Figures

Chapter 1

Introduction

Scientific visualization of space research, also known as astro-visualization, works as an important tool for scientists to communicate their work in exploring the cosmos. 3D computer graphics has shown to be an efficient tool for bringing insights from geological and astronomical data, as spatial and temporal relations can intuitively be interpreted through 3D visualizations.

Researching and mapping celestial bodies other than the Earth is an important part of expanding the space frontier; rendering these globes using real gathered map and terrain data is a natural part of any scientifically accurate space visualization software.

Important parts of a software for visualizing celestial bodies include the ability to render terrains together with color textures of various sources. The focus of this thesis is put on globe rendering using high fidelity geographical data such as texture maps, maps of scientific parameters, and digital terrain models. The globe rendering feature with the research involved was implemented for the software OpenSpace. The implementation was separated enough from the main program to avoid dependencies and make the thesis independent of specific implementation details.

1.1 Background

1.1.1 OpenSpace

OpenSpace is an open-source, interactive data visualization software with the goal of bringing astro-visualization to the broad public and serve as a platform for scientists to talk about their research. The software supports rendering across multiple screens, allowing immerse visualizations on tiled displays as well as in dome displays using multiple projectors [1].

With a real time rendering software such as OpenSpace, the human curiosity involved in exploration easily becomes obvious when the user is given the ability to freely fly around in space and near the surface of other worlds and discover places they probably never can visit in real life. Even more so is the case of public presentations where researchers such as geologists can go into details about their knowledge and showing it through scientific visualization.

An important part of the software is to avoid the use of procedurally generated data. This is to express where the frontier of science and exploration is currently at and how it progresses through space missions with the goal of mapping the Universe. A general globe browsing feature provides a means of communicating this progress through continuous mapping of planets, moons and dwarf planets within our solar system.

1.1.2 Globe Browsing

The term globe browsing can be described as exploration of geospatial data on a virtual representation of a globe. The word globe is a general term used to describe nearly elliptical celestial objects such as planets, moons, dwarf planets and asteroids.

Globe rendering with the purpose of multi-scale browsing has been used for quite some time in flight simulators, map services and astro-visualization. Prerendered flight paths were visualized as early as the late 1970s by NASA's Jet Propulsion Laboratory [2].

Google Earth [3] enables browsing of the Earth within a web browser using geometries for cities of high detail. The National Oceanic and Atmospheric Administration (NOAA) provides a sophisticated sphere rendering system, "Science On a Sphere", with the ability to visualize a vast amount of geospatial data on spheres with a temporal dimension [4].

There are other commercial softwares that enables larger scale visualization of the Universe with real positional data gathered through research by the National Aeronautics and Space Administration (NASA), the European Space Agency (ESA) and others. Satellite Toolkit (STK) enables this by integrating ephemeris information through the SPICE interface [5] which allows accurate placing of celestial bodies and space crafts within our solar system using real measured data. Uniview from SCISS AB also enables SPICE integration with sophisticated rendering techniques and dome theatre support [6].

There are other significant globe browsing softwares used in dome theatres such as World Wide Telescope (WWT) [7], Evans & Sutherland's Digistar [8] and Sky-Skan's DigitalSky [9].

Table 1.1: Relevant features of different globe browsing softwares

	Observable Universe scale	Focus on Dome configuration support	Ephemeris data integrated	Scientific data only	Free to use	Open source
Google Maps				✔	✔	
STK			✔	✔		
Uniview	✔	✔	✔	✔		
WWT	✔	✔	✔	✔	✔	✔
Digistar	✔	✔	✔	✔		
DigitalSky	✔	✔	✔	✔		
Outerra					✔	
Space Engine	✔		✔		✔	
OpenSpace	Future Plan	✔	✔	✔	✔	✔

Other relevant softwares that currently do not support dome configuration rendering but none the less are very adequate in their techniques of integrating globe browsing and globe rendering include Outerra [10] and Space Engine [11]. Both focusing on merging real data with procedurally generated terrains where real data is not available.

Geographic information systems (GIS) are softwares with the purpose of gathering a wide range of geographic map data and visualizing it in various different ways. Even though most of these softwares use GIS features, many of them are not considered GIS. However, they all have the globe browsing feature in common. Technicalities in how it is implemented varies as their end target users are different.

In table 1.1, features relevant to globe browsing in public presentations are shown and compared between different visualization softwares that integrate globe browsing.

1.2 Objectives

The goal of this thesis is not only to provide OpenSpace with a globe browsing feature. There are some specific demands that play a significant role in the focus of this work. These are listed below:

1. Ability to retrieve map data from the most common standardized web map data interfaces: WMS, TMS and WMTS.
2. Ability to render height maps and color textures with up to 25 cm per pixel resolution
3. Ability to layer multiple map datasets on top of each other. This makes it possible to visualize a range of scientific parameters on top of a textured globe

4. Globe browsing should be done at interactive frame rates: at least 60 frames per second on modern consumer gaming graphics cards
5. Correct positional mapping of objects rendered near the globe surface
6. Support animation of map datasets with time resolution
7. An intuitive interaction mode is also required to get the most out of globe browsing. It must support:
   (a) Horizontal following of reference ellipsoid
   (b) Decrease in sensitivity when closer to the surface of the globe
   (c) Terrain following to avoid popping down under the surface

1.3 Delimitations

To focus on the important aspects of globe browsing and its purposes for public presentations, some important delimitations had to be taken into consideration.

We do not consider rendering of globes with distances to the origin greater than the radius of our solar system. Direct imaging of exoplanets is far from usable for mapping and is mainly a method for locating planets [12]. Since we don't yet have map data for exoplanets, and OpenSpace does not currently aim at producing procedurally generated content, visualization of exoplanets is not the focus for this thesis.

One important delimitation for the project is to limit the geometry of a globe to height mapped grids. This will make it possible to perform vertex blending on the GPU as well as simplify the implementation to a uniform method of rendering across the whole globe.

We will not focus on re-projecting maps between different georeferenced coordinate systems. Therefore the implementation must be limited to reading a specific map projection. Reprojecting maps can be considered future work to generalize the ability to read map data as well as optimizing rendering output and performance.

One goal of OpenSpace is to produce awe inspiring visual effects. The current state of the project requires the foundations to be in place and globe browsing is one of the main features that need to be implemented before real sophisticated rendering techniques can be developed. We will not consider rendering of atmospheres or water effects and we will not consider shading techniques that requires changing of the current rendering pipeline of OpenSpace such as deferred shading.

1.4 Challenges

There are multiple technical challenges to tackle when designing a virtual globe renderer. Cozzi and Ring [2] define the main challenges as following:

- **Precision** - In order to render small scale details of a virtual globe and also dolly out to see multiple virtual globes within the solar system, the limited precision of computer arithmetics needs to be considered.
- **Accuracy** - Modeling globes as spheres is usually not a very accurate approach, as many planets and moons that rotate have different polar and equatorial radii.
- **Curvature** - The curved nature of globes implies some extra challenges as opposed to worlds modeled based on flat surfaces. The challenge includes finding a suitable 2D-parameterization for tessellation and mapping.
- **Massive datasets** - It is usual for real world geographical datasets to be too large to fit in GPU memory, RAM and even local drives. Instead, data need to be fetched from remote servers on demand using a so called out-of-core approach to rendering.
- **Multithreading** - The need for multithreading is necessary as the program needs to retrieve geographical data from multiple sources, while at the same time retain a steady frame rate for rendering.

Details in issues and proposed solutions to these challenges will be discussed throughout the thesis.

Chapter 2

Theoretical Background

A sophisticated globe rendering system needs to rely on some theoretical foundations and algorithms developed for globe rendering. These foundations work as a base for the research performed for the thesis and the implementation. The research is based on the proposed challenges.

2.1 Large Scale Map Datasets

Global maps with high level of detail can easily become too large to be stored and read locally on a single machine. A common way of storing large maps is by representing them using several overviews. An overview is a map representing the same geographical area as the original map but down sampled by a factor of two just like a lower level of a mip map texture. Figure 2.1 shows how the size of the maps in raster coordinates decreases with the overview number.

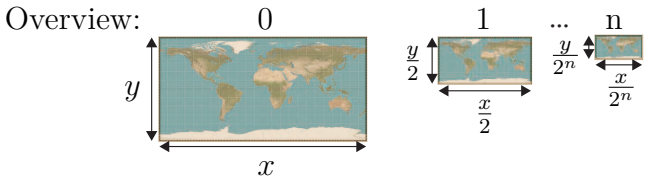


Figure 2.1: The size of the map is decreasing exponentially with the overview. Figure adapted from [13]

The physical disk space of large global map datasets is often measured in terabytes or even petabytes. In order to deal with such large datasets, web based services allow clients to specify parts of the map to download at a time. This is an important aspect in the out-of-core rendering required for globe visualization.

2.1.1 Web Map Services

To standardize web requests for map data, the Open GIS Consortium (OGC) specified a web map service interface [14] and from that, specifications of several other map service interfaces have followed. The most common standards are Web Map Service (WMS), Tile Map Service (TMS) and Web Map Tile Service (WMTS). Some other, more specific, examples of WMS-like services are WorldWind, VirtualEarth and AGS.

WMS

The WMS interface instructs the map server to produce maps as image files with well defined geographic and dimensional parameters. The image files can have different format and compression depending on the provider. A WMS server has the ability to dynamically produce map patches of arbitrary size which puts some load on the server side [14]. The basic elements supported by all WMS providers are the *GetCapabilities* and the *GetMap* operations. *GetCapabilities* gives information about the available maps on the server and their corresponding georeferenced metadata. The *GetMap* operation returns the map or a part of the map as an image file.

WMS requests are done using HTTP GET where the standardized request parameters are provided as query parameters in the URL [14]. For example, setting the query parameter `BBOX=-180,-90,180,90` specifies the size of the map in georeferenced coordinates while the parameters `WIDTH` and `HEIGHT` specify the size of the requested image in raster coordinates. All name and value pairs for the *GetMap* request are defined under the OpenGIS Web Map Server Implementation Specification [14].

TMS

Tile Map Service (TMS) was developed by the Open Source Geospatial Foundation (OSGeo) as a simpler solution to requesting maps from remote servers. The specification uses integer indexing for requesting specific precomputed map tiles instead of letting the server spend time on producing maps of arbitrary dimensions. The TMS interface is similar to WMS but simpler and it does not support all the features of WMS [15].

WMTS

Web Map Tile Service (WMTS) is another standard by OGC that requires tiled requests. It supports many of the features of WMS but, similar to TMS, removes the load of image processing from the server side and instead

forces the client to handle combination and cutouts of patches if required. The standard specifies the *GetCapabilities*, *GetTile* and *GetFeatureInfo* operations. These operations can be requested with different message encodings such as Key-Value Pairs, XML messages or XML messages embedded in SOAP envelopes [16].

Tiled WMS

Before the WMTS standard was developed, some servers had already embarked on the tiled requests by limiting the valid bounding boxes to values that only produce precomputed tiles. These services can be referred to as Tiled WMS and are nothing more than specified WMS services where the server limits the number of valid requests [16].

2.1.2 Georeferenced Image Formats

There are several different standards for handling image data used in GIS softwares. Some common file formats with image data and/or georeferenced information that are used in the below mentioned imagery products are:

- **GeoTIFF** - TIFF image with the inclusion of georeferenced metadata
- **IMG** - Image file format with georeferenced metadata
- **JPEG2000** - Georeferenced image format with lossy or lossless compression
- **CUB** - Georeferenced image file standard created by Integrated Software for Imagers and Spectrometers (ISIS)

2.1.3 Available Imagery Products

There are several organizations working on gathering GIS data that can be visualized as flat 2D maps or projected on globes. Many of them provide their map data through web map services. However, they are often defined in different formats and sometimes available only as downloadable image files.

Earth

NASA Global Imagery Browse Services (GIBS) provides several global map datasets with information about Earth's changing climate [17]. The two satellites Aqua and Terra orbit the Earth and are continuously measuring

multi band quantities such as corrected reflectance and surface reflectance along with a range of scientific parameters such as surface and water temperatures, ozone, carbon dioxide and more. Many of the GIBS datasets are updated temporally so that changes can be seen over time. Map tiles are requested through a TMS interface and a date and time parameter can be set to specify a map within a certain time range.

Environmental Systems Research Institute (ESRI) is the provider of the software ArcGIS that lets their users create and publish map data through different types of web map services. There are a lot of free maps to use; not only for Earth but other globes are covered too. ESRI supports a web publication interface where web maps can be searched and studied in an online map viewer.

National Oceanic and Atmospheric Administration (NOAA) for the U.S. Department of Commerce gathers and provides weather data of the US that are hosted through different web map services using the ArcGIS online [18] interface by ESRI.

Mars

The first global color images taken of Mars were by the two orbiters of the Viking missions that launched in late 1975. NASA Ames worked on creating the Mars Digital Image Models (MDIM) by blending a mosaic of images taken by the orbiters. United States Geological Survey (USGS) provides downloading of image files in CUB or JPEG2000 format. The maps are still the highest resolution global color maps of the planet [19].

NASA's Mars Reconnaissance Orbiter (MRO) is a satellite that has been orbiting Mars since 2006, gathering map data by taking pictures of the surface. The satellite has three cameras; the Mars Color Imager (MARCI) for mapping out daily weather forecasts, the Context camera (CTX) for imaging terrain and the High Resolution Imaging Science Experiment (HiRISE) camera for mapping out smaller high resolution patches covering limited surface areas of interest. NASA enables downloading of local patches and digital elevation models (DEMs) and grayscale images taken by the CTX [20] and the HiRISE [21] cameras in IMG and GeoTIFF formats.

Moon

The Lunar Mapping Modeling Project (LMMP) is an initiative by NASA to gather and publish map data of the Moon from a vast range of lunar missions. The Lunar Reconnaissance Orbiter (LRO) is a satellite orbiting the moon and gathering map data for future landing missions. These maps

have been put together into global image mosaics as well as DEMs. Most global maps from LMMP can be accessed via the "OnMoon" web interface [22].

2.2 Modeling Globes

We will discuss different proposed methods used for modeling and rendering of globes. The globe can be modeled either as a sphere or an ellipsoid and there are different tessellation schemes for meshing the globe. The tessellation depends on a map projection and out-of-core rendering requires a dynamic level of detail approach for rendering.

2.2.1 Globes as Ellipsoids

Planets, moons and asteroids are generally more accurately modeled as ellipsoids than as spheres. Planets are often stretched out along their equatorial axes due to their rotation which causes the centripetal force to counter some of the gravitational force acting on the mass. This effect was proven in 1687 by Isaac Newton in Principia Mathematica [23]. The rotation causes a self-gravitating fluid body in equilibrium to take the form of an oblate ellipsoid, otherwise known as a biaxial ellipsoid with one semimajor and one semiminor axis. Globes can be modeled as triaxial ellipsoids for more accuracy when it comes to smaller, more irregularly shaped objects. For example Phobos, one of Mars' two moons, is more accurately modeled as a triaxial ellipsoid with radii of $27 \times 22 \times 18$ km [2].

The World Geodetic System 1984 (WGS84) standard defined by National Geospatial-Intelligence Agency (NGA) models the Earth as a biaxial ellipsoid with a semimajor axis of 6,378,137 meters and a semiminor axis of 6,356,752.3142 meters [2]. This is what is known as a reference ellipsoid; a mathematical description that approximates the geoid of the earth as closely as possible. The WGS84 standard is widely used for GIS and plays an important role in accurate placements of objects such as satellites or spacecrafts with position coordinates relatively close to the Earth's surface. In the WGS84 coordinate system, the x-axis points to the prime meridian, the z-axis points to the North pole and the y-axis completes the right handed coordinate system, see figure 2.2.

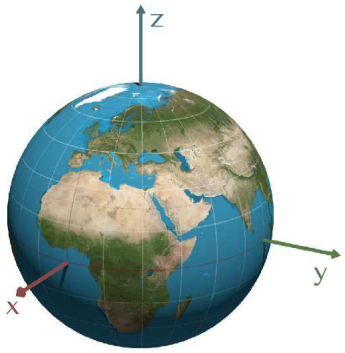


Figure 2.2: The WGS84 coordinate system and globe. Figure adapted from [13]

2.2.2 Tessellating the Ellipsoid

Triangle models are still the most common way of modeling renderable objects in 3D computer graphics softwares, even though other rendering techniques such as volumetric ray casting also can be considered for terrain rendering [2, p. 149].

A triangle mesh, or more generally a polygon mesh, is defined by a limited number of surface elements. This means that ellipsoids need to be approximated by some sort of tessellation or subdivision surface when modeled as a polygon mesh. There are several techniques for tessellating an ellipsoid. Some of them are covered in this section.

Geographic Grid Tessellation

Tessellating the ellipsoid using a geographic grid is a very straightforward approach. Ellipsoid vertex positions can be calculated using a transform from geographic coordinates to Cartesian model space coordinates [2, p. 25]. Figure 2.3 shows three geographic grid tessellations of a sphere with constant number of latitudinal segments of 4, 8 and 16 respectively.

A common issue with geographic grids is something referred to as polar pinching. At both of the poles, segments will be pinched to one point which leads to an increasing amount of segments per area. This in turn results in oversampling in textures as well as possible visual artifacts in shading due to the very thin quads at the poles as well as possible performance penalties for highly tessellated globes.

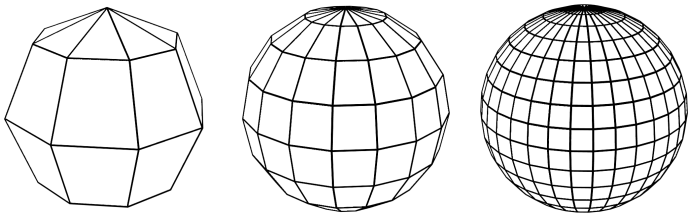


Figure 2.3: Geographic grid tessellation of a sphere with constant number of latitudinal segments of 4, 8 and 16 respectively

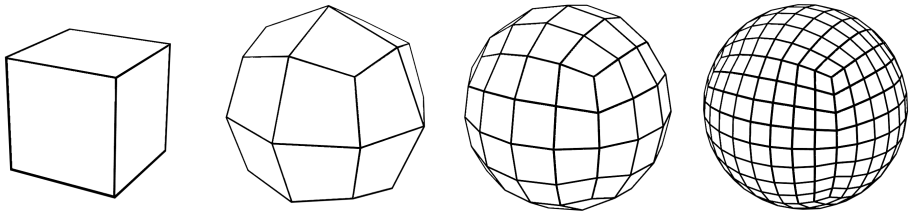


Figure 2.4: Quadrilateralized spherical cube tessellation with 0, 1, 2 and 3 subdivisions respectively

Quadrilateralized Spherical Cube Tessellation

Another common tessellation method for spheres which can be generalized to ellipsoids is the quadrilateralized spherical cube tessellation. The standard approach is to subdivide a cube centered in the origin and then normalize the coordinates of all vertices to map them on a sphere. There are also other more complicated schemes designed to work with specific map projections [24].

To model an ellipsoid from a sphere, the vertices can be linearly transformed with a scaling in the x, y and z directions individually. Figure 2.4 shows a tessellated spherical cube of four different detail levels.

Hierarchical Triangular Mesh

The hierarchical triangular mesh (HTM) is a method of modeling the sky dome as a sphere proposed by astronomers in the Sloan Digital Sky Survey [25]. Instead of uniformly dividing cube faces, an alternative option is to subdivide a normalized octahedron by, in each subdivision step, split every triangle into four new triangles, see figure 2.5.

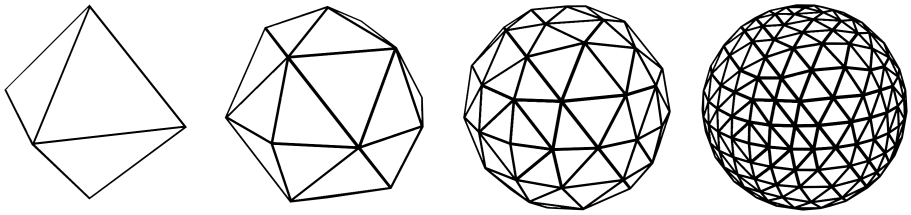


Figure 2.5: Hierarchical triangular mesh tessellation of a sphere with 0, 1, 2 and 3 subdivisions respectively

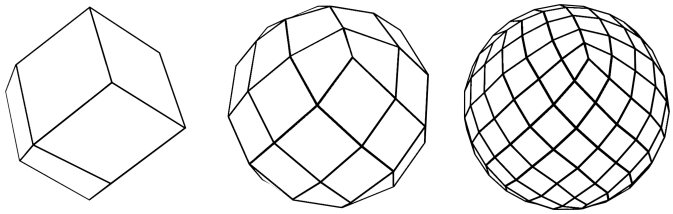


Figure 2.6: HEALPix tessellation of three different levels of detail

Hierarchical Equal Area IsoLatitude Pixelation

Hierarchical Equal Area IsoLatitude Pixelation (HEALPix) is a spherical tessellation scheme with corresponding map projection. The base level of the tessellation is built up of twelve quads, similar to a rhombic dodecahedron, which each can be subdivided further. The tessellation in figure 2.6 shows how the vertices in the HEALPix tessellation leads to curvilinear quads.

Geographic Grid Tessellation With Polar Caps

In their description of the ellipsoidal clipmaps method, Dimitrijević and Rančić introduces polar caps to avoid polar issues related to geographic grids [24]. The polar caps are simply used as a replacement of the problematic, oversampled regions around the poles. The caps can be modeled as grids projected onto the ellipsoid surface in their own georeferenced coordinate systems. One obvious issue with polar caps is the edge problem that occurs due to the fact that the caps are defined as separate meshes with vertices that do not coincide with the geographic vertices of the equatorial region, see figure 2.7. Dimitrijević and Rančić solves the issue by using a type of edge blending between the equatorial and polar segments [24]. Figure 2.7 shows a sphere tessellated with one equatorial region and two polar regions.

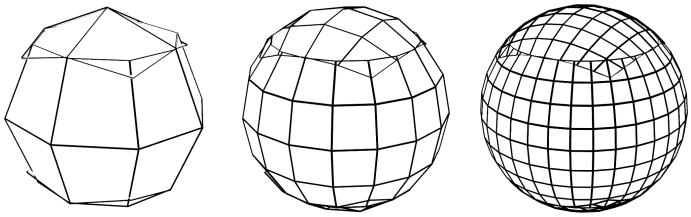


Figure 2.7: Geographic tessellation of a sphere with polar caps

2.2.3 2D Parameterisation for Map Projections

A map projection P defines a transformation from Cartesian model space coordinates to georeferenced (projected) coordinates, as in equation 2.1. The inverse projection P^{-1} is used to find positions on the globe surface in model space given georeferenced coordinates as in equation 2.2.

$$\begin{pmatrix} s \\ t \end{pmatrix}_{georeferenced} = \vec{P}(x, y, z), \tag{2.1}$$

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix}_{modelspace} = \vec{P}^{-1}(s, t), \tag{2.2}$$

where x, y and z are the Cartesian coordinates of a point on the ellipsoid surface. The parameters s and t are georeferenced coordinates defining all positions on the globe. The georeferenced coordinates can have different definition range depending on which projection is used. An example can be letting is $s = \phi \in [-90, 90]$ and $t = \theta \in [-180, 180]$ which are latitude and longitude respectively for geographic projections.

The globally positive Gaussian curvature of any intrinsic ellipsoid surface makes it impossible to unproject it on a flat 2D surface without any distortions. Since it is embedded in a 3D space, some distortions must be introduced when unprojecting the surface. The distortion can differ depending on the projection used. Equal-area projections preserve the size of a projected area as $\partial s \partial t / \partial s_0 \partial t_0 = 1$, while conformal projections preserve the shape of projected objects as $\partial s / \partial t = 1$; s_0 and t_0 are coordinates at the center of the projection with no distortion. No global projection can be both area-preserving and conformal [24].

There are several possibilities for defining a coordinate transform for map projections. A common approach is to project the ellipsoid onto another shape that allows for being flattened out without distortion, such as a cube, a cylinder or a plane. These types of shapes are known as developable shapes and have zero Gaussian curvature.

The choice of map projection is tied together with the choice of ellipsoid tessellation. This is because the map often needs to be tiled up when rendering. Each tile has its local texture coordinate system which need to have a simple transform from the georeferenced coordinate system for texture sampling. If the tiles can be affinely transformed to the georeferenced coordinate system, texture sampling can be done on the fly; otherwise the georeferenced coordinates need to be re-projected which may be computationally heavy or impossible for real time applications.

The European Petroleum Survey Group (EPSG) [26] has defined several standards for map projections of the Earth. Many of these are mentioned when discussing the different projections.

Geographic Projections

Geographic projections are widely used standards for parameterization of ellipsoids. The ellipsoid is projected onto a cylinder which is then unrolled to form the 2D plane of the projected coordinates.

Geographic coordinates are defined with a latitude ϕ and a longitude θ and works together with geographic tessellations of ellipsoids. A common issue with geographic projections is oversampling around the poles, as mentioned in section 2.2.2. At the poles, all longitudes will always map onto one point and the distortion increases with the absolute value of the latitude. Figure 2.8 shows an unprojected geographic map and how it wraps around the globe.

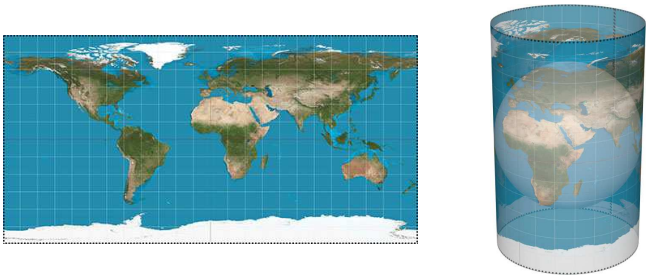


Figure 2.8: Geographic map projection. Figure adapted from [13]

Geocentric projection The simplest geographic parameterization uses geocentric coordinates. Here the latitude and longitude are defined as the angle between a vector from the origin to a point on the ellipsoid surface and the $xy-$ and $xz-$planes respectively.

Geodetic projection Another standard, well used in ellipsoid representations makes use of so called geodetic coordinates. This variety of geographic coordinate systems is defined by the normal of the surface of the ellipsoid where the longitudinal angle is the angle between the normal and the yx-plane.

Figure 2.9 shows the difference between geocentric latitudes along with difference in surface normals.

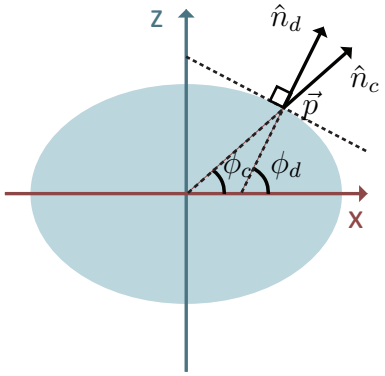


Figure 2.9: Difference between geocentric latitudes, ϕ_c and geodetic latitudes, ϕ_d for a point $\vec{p}$ on the surface of an ellipsoid. The figure also shows the difference between geocentric and geodetic surface normals, $\hat{n_c}$ and $\hat{n_d}$, respectively.

In the case of perfect spheres, geocentric and geodetic projections of any point will yield the same result.

Geodetic coordinates are among the most commonly used geo referenced coordinate systems when mapping ellipsoids to two dimensions.

Cozzi and Ring describes the transform from geodetic coordinates in the ellipsoid class [2, p. 25]. For the Earth, the most commonly used geodetic coordinate space is defined in the EPSG:4326 standard where the WGS84 ellipsoid is used [27].

Mercator Projection

The mercator projection is a cylindrical projection widely used for presenting global maps in unwrapped form. The mercator projection preserves the horizontal to vertical ratio for small objects on the map. Hence, the mercator is a conformal projection in contrast to the geocentric and geodetic projections, which results in a non unit value in the ratio between the longitudinal and latitudinal differentials, see figure 2.10.

The mercator projection compensates for the longitudinal distortion by introducing a latitudinal distortion as well. Due to the polar singularities

(a) The equirectangular projection is not conformal. Figure from [28]

(b) The mercator projection is conformal and preserves aspect ratio. Figure from [29]

Figure 2.10: Unwrapped equirectangular and mercator projections. The mercator projection works when it is unwrapped due to it being conformal (preserving aspect ratio).

which lead to infinite latitudes at the poles when $\phi_d = \pm 90$, the domain of definition for the latitudes need to be constrained in mercator projection, see figure 2.11.

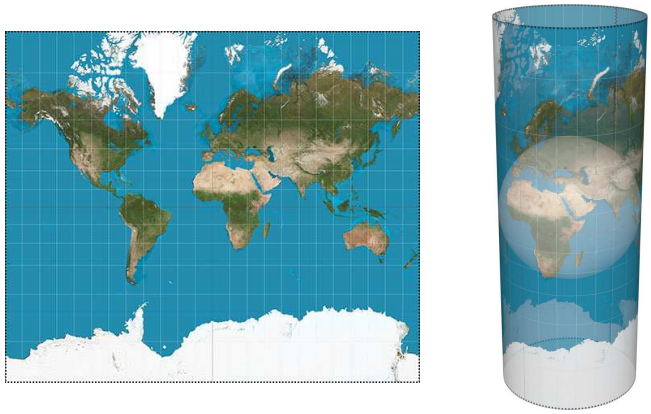


Figure 2.11: Mercator projection. Figure adapted from [13]

The EPSG:3857 standard for mercator projection of the Earth, also known as web mercator, constrains the domain to $\phi \in [-85.06, 85.06]$ [27]. The standard uses a different projection that does not diverge at the polar regions. Web mercator is used by most online web map applications including Google Maps, Bing Maps, OpenStreetMap, Mapquest, ESRI and Mapbox [30].

Cube Map

Cube maps lack the polar singularities apparent in geographic parameterizations. The parameterized coordinates are often discretized to the six sides of the cube, but they can also map directly to a global representation of an unwrapped cube, see figure 2.12.

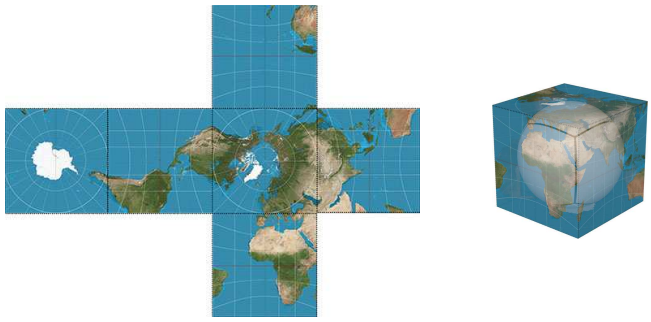


Figure 2.12: Cube map projection. Figure adapted from [13]

Due to the traditions of map projections this is not a common format used for map services so reprojection from a more common format is often required.

There are different cube map projections with different amount of area- and aspect distortions. Dimitrijević and Rančić mention and compares spherical cube, adjusted spherical cube, Outerra spherical cube and quadrilateralized spherical cube [24].

Tessellated Octahedral Adaptive Subdivision Transform

The Tessellated Octahedral Adaptive Subdivision Transform (TOAST) map format used in the globe browsing of Microsoft's World Wide Telescope works together with the HTM tessellation [31]. Each triangular segment of the TOAST map maps to a triangle of a sphere that is tessellated as an octahedron and subdivided to form a sphere, see figure 2.13.

The TOAST format is just as the cube maps not a very well supported format for map providers and harder to use together with some of the most common level of detail approaches due to the fact that most of them are optimized for rectangular and not triangular map tiles.

Hierarchical Equal Area IsoLatitude Pixelation

The map projection that is used for the HEALPix tessellation is equal-area as the name suggests. Figure 2.14 shows how the map wraps onto the sphere.

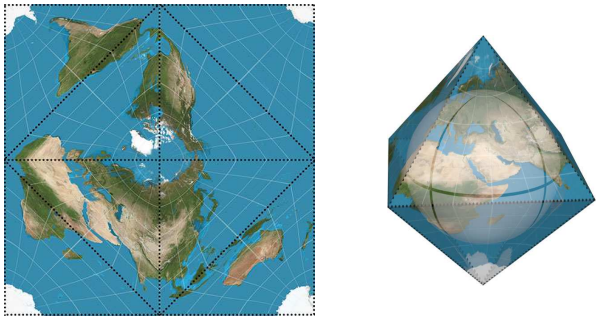


Figure 2.13: TOAST map projection. Figure adapted from [13]

The positive aspect about HEALPix compared to the TOAST format is that the map is tiled into quads and not triangles which means that it is better suitable for the chunked LOD [2] algorithm. The map format is used by NASA for mapping the cosmic microwave background radiation for the Microwave Anisotropy Probe (MAP) [32], but is otherwise an uncommon format when it comes to map services. That means that the maps need to be re-projected from the more common formats for wider support.

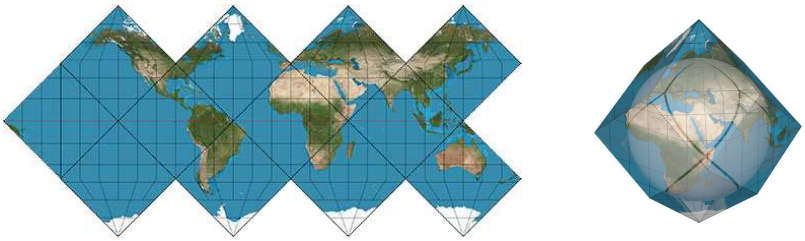


Figure 2.14: HEALPix map projection. Figure adapted from [13]

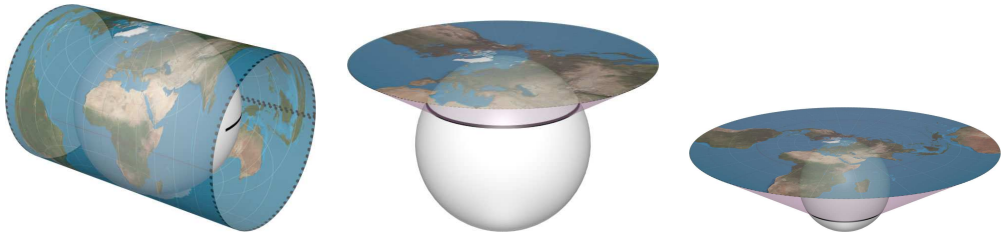
Polar Projections

For parameterization of limited parts of the globe, such as the isolated poles, there are different projections to consider. Most common are different types of azimuthal projections. These projections are defined by projecting all points of the map through a common intersection point and onto a flat surface. The Gnomonic projection maps all great circle segments (geodesics) to straight lines by having the common intersection point in the center of the globe.

Stereographic projections are defined when the common intersection point is positioned on the surface of the globe on the opposite side of the pole to project. Polar stereographic projections are used to parameterize the

surface of the poles of the Earth. The standards EPSG:3413 and EPSG:3031 define the stereographic projections for the North Pole and the South Pole respectively [27].

Dimitrijević and Rančić use another polar coordinate system to re-project from geodetic coordinates in runtime. The transformation is a rotation of 90 degrees around the global $x-$axis so that the resulting parametric coordinates of the pole are given in their own geographic space with the meridian as equator. This projection is also known as the Cassini projection and it can be both defined for spheres as well as generalized to ellipsoids. Polar projections are shown in figure 2.15.



(a) Cassini (b) Gnomonic (c) Stereographic

Figure 2.15: Polar map projections. Figure adapted from [13]

2.3 Dynamic Level of Detail

Dynamic level of detail (LOD) is an important part in handling the extensive amount of data used in an out-of-core rendering software. The goal is to maximize the visual information on screen while minimizing the workload. In their book 3D Engine Design for Virtual Globes, Cozzi and Ring describes LOD rendering algorithms by three typical steps: [2, p. 367]

1. **Generation** - Create versions at different level of detail of a model.
2. **Selection** - Choose a version based on some criteria or error metric (e.g. distance to object or the projected area it occupies on the screen).
3. **Switching** - Transition from one version to another in order to avoid noticing of the change in LOD known as popping artifacts.

There are different types of LOD approaches for terrain rendering and a suitable approach should be chosen based on characteristics of the terrain. Terrains can for example be restricted to being represented as height maps -

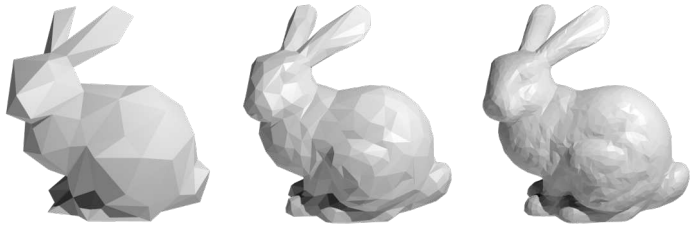


Figure 2.16: A range of predefined meshes with increasing resolution. Dynamic level of detail algorithms are used to choose the most suitable mesh for rendering

a characteristic that can be exploited by the rendering algorithm. Cozzi and Ring describe the following three categories of LOD approaches: Discrete Level of Detail, Continuous Level of Detail and Hierarchical Level of Detail [2, p. 368-371].

2.3.1 Discrete Level of Detail

In the Discrete Level Of Detail (DLOD) approach, multiple different representations of the model are created at different resolutions. DLOD is arguably the most simple LOD algorithm. It works not only for digital terrain models, but for arbitrary meshes. The set of terrain representations can either be predefined or generated using mesh simplification algorithms.

At run time, the main objective is to select one (or generate) a suitable representation. This approach does not provide any means of dealing with large scale datasets which requires multiple levels of detail at the same time. This makes it unsuitable for globe rendering [2].

2.3.2 Continuous Level of Detail

The continuous LOD (CLOD) approach represents a model in a way that allows the resolution to be selected arbitrarily. This is usually implemented by a base mesh combined with a sequence of operations that successively changes the level of detail of the model. Two typical such operations are "edge collapse" (removes two triangles from the mesh) and its inverse, "vertex split" (adds two triangles to the mesh). These operations are illustrated in figure 2.17.

According to Cozzi and Ring [2, p. 368] CLOD has previously been the most popular approach for rendering terrain at interactive rates, with implementations such as Real-time Optimally Adaptive Mesh (ROAM) [33]. The main reason CLOD algorithms are not widely employed these days is

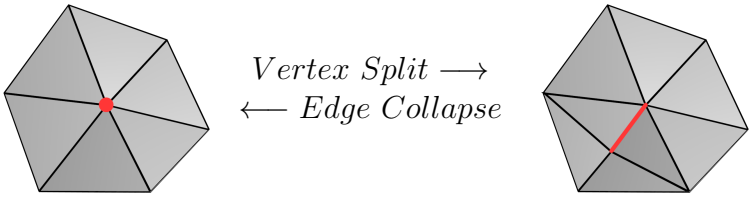


Figure 2.17: Mesh operations in continous LOD

due to the increase in triangle throughput on modern GPUs, causing the CLOD operations done on the CPU in many cases to act as a bottleneck for the rendering.

A special branch of CLOD worth mentioning is the so called infinite LOD. In this approach the terrain is represented by a mathematical function; an implicit surface. These functions can be defined by fractal algorithms and produce complex characteristics or they can define simple geometric shapes such as spheres or ellipsoids. As all points on these types of surfaces are precisely defined, triangle meshes can be generated with no limit on the level of detail. This approach is not suitable for incorporating real world data, but it is used by terrain engines such as Outerra and Terragen to procedurally generate terrain at any desired level of detail [34] [35].

2.3.3 Hierarchical Level of Detail

Hierarchical Level of Detail (HLOD) can be seen as a generalization of DLOD. HLOD algorithms operates on hierarchically arranged, predefined chunks of the full model. Each chunk is processed, stored and rendered separately. By doing this, HLOD approaches tackles the weaknesses of CLOD, essentially by doing the following:

1. Reducing processing time on CPU: The only CPU task that HLOD algorithms has to deal with during runtime is to select a suitable subset of the predefined chunks for rendering. This is a relatively fast procedure in contrast to iteratively applying changes to the raw geometry, as done in CLOD.

2. Reducing data traffic to the GPU: Data is uploaded to the GPU in larger batches but not very often, since the data is static and GPU caching can be done. With CLOD, the geometry data is updated on a per-frame basis and can not be cached on the GPU. Being able to perform GPU caching allows HLOD to better minimize the traffic to the GPU.

HLOD uses spatial hierarchical data structures such as binary trees, quadtrees or octrees for storing the chunk data. The root node of the tree holds a full representation of the model at its lowest level of detail in one single chunk. At successive levels, the model is represented at a higher level of detail but divided up into several chunks. This concept is illustrated with a quad tree holding chunks representing a bunny model in figure 2.18.

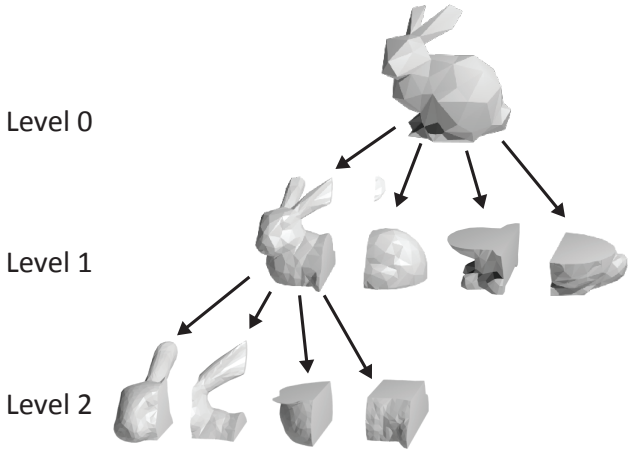


Figure 2.18: Bunny model chunked up using HLOD. Child nodes represent higher resolution representations of parts of the parent models

Generally, selecting all the chunks at a specific level in the tree yields a complete representation of the model. Furthermore, chunks may be selected from different levels for different parts of the model and still yield a full representation of the model. This allows for view dependent rendering of the model. Algorithm 1 describes pseudo code for recursively rendering the full model at view dependent level of detail.

```
RenderLOD (Camera C, ChunkNode N)
    if ErrorMetric(C, N) < threshold then
        Render(N, C)
    else
        for child in children(N) do
            RenderLOD(child, C)
        end
    end
```

Algorithm 1: Selecting chunks to render. The error metric depends on the camera state and the chunk to render. A given chunk always has a smaller error metric than its parent.

This example uses a depth first approach for rendering of chunks. Other common schemes for traversing the hierarchy are breadth first and inverse breadth first.

The algorithm traverses the tree and calculates an error metric at each node with respect to the current camera position. If the calculated error is larger than a certain threshold, the algorithm recursively repeats the procedure for all the chunk's children, which have higher level of detail. This general scheme can be used for rendering one-dimensional curves (using a binary tree structure), two-dimensional surfaces (using a quadtree) or volumes (using an octree).

Another key feature of HLOD as opposed to DLOD and CLOD is that it can naturally be integrated with out-of-core rendering, as chunks can be loaded into memory on-demand and deleted when not needed.

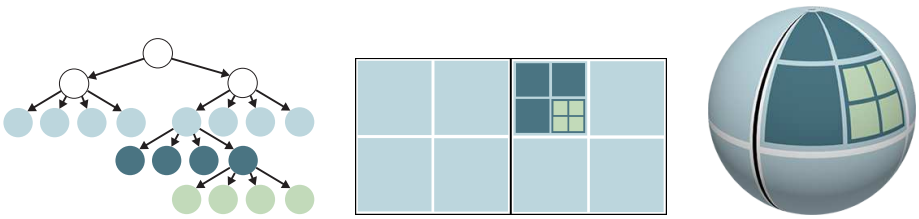
2.4 Level of Detail Algorithms for Globes

A number of different LOD algorithms has been introduced for the purpose of globe rendering. Two common algorithms used are Chunked LOD and Geometry Clipmaps, as pointed out by Cozzi and Ring [2].

2.4.1 Chunked LOD

The Chunked LOD method fits into the HLOD category and works by breaking down the surface of the globe into a quadtree of chunks. There are several different ways of spatially organizing chunks and they depend on the tessellation of the globe.

Using a geographic grid tessellation, the chunks are in geographic space using latitude ϕ and longitude θ coordinates. Figure 2.19 demonstrates the layout of chunks as they are mapped in geographic coordinates onto an ellipsoid representation of a globe.



(a) Chunk tree (b) Geodetic chunks (c) Globe

Figure 2.19: Chunked LOD for a globe

Chunks

Chunks should store the following data:

1. A mesh defining the terrain geometry (positions, normals, texture coordinates).

2. A monotonic geometric error based on the vertices distances to the fully detailed mesh. The children of a chunk always have smaller geometric error as they can better fit the highest level of detail model.

3. A known bounding volume encapsulating the mesh and all the chunk's children. This is used along with the geometric error metric when selecting suitable chunks for rendering.

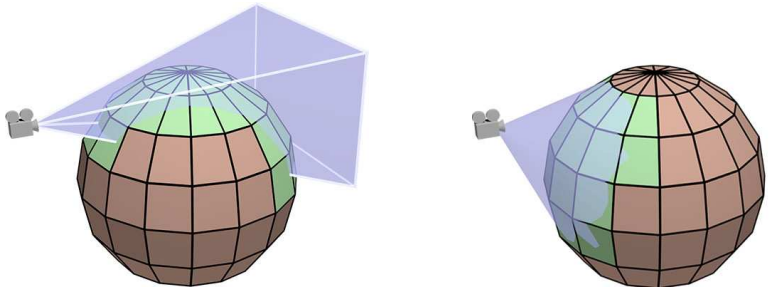
Depending on the implementation of the chunked LOD algorithm, these properties can be either calculated on the fly or preprocessed as suggested by Cozzi and Ring [2, p. 447]. Furthermore, the chunk mesh must have defined edges along its sides such that when two adjacent chunks are rendered next to each other, there is no gap in between them.

Culling

An important thing to consider when dealing with chunks of a virtual globe is the fact that the chunks that are selected for rendering might not actually be visible on the screen. Needless to say, this is a waste of computational power and by eliminating these unnecessary draw calls, the performance of the globe renderer can be increased.

Camera frustum culling This is done by testing a bounding box of the chunk for intersection with the camera frustum. If the chunk is completely outside the frustum, the chunk can safely be culled as it will not be visible in the rendered image. See figure 2.20a.

Horizon culling Even after camera frustum culling there are chunks that still do not contribute to the rendered image because they are positioned behind the horizon. Figure 2.20b illustrates that most of a globe is actually invisible to any observer. This can be used as a basis for culling some of the remaining chunks.



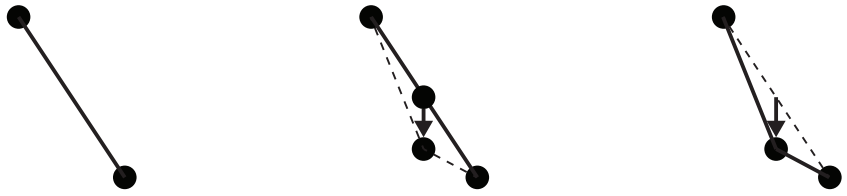
(a) Frustum culling (b) Horizon culling

Figure 2.20: Culling for chunked LOD. Red chunks can be culled due to them being invisible to the camera

Switching

Even when chunks can be selected in a way that guarantees a maximum pixel error per vertex, the fact that full areas of multiple triangles are replaced all at once causes a drastic change in the rendered view. Even when the updates are small per vertex, the update of whole chunk areas may be easily noticed. This is what is referred to as popping.

Minimizing popping artifacts is typically done by smoothly transitioning between levels over time. Cozzi and Ring suggest an approach, where along with each vertex, a delta offset is also stored [2, p. 451]. This delta offset stores the difference between the chunk itself and the same region within the parent chunk. Using this difference, new vertices can be placed on already defined edges and then interpolated into their actual positions. Figure 2.21 illustrates the idea.



(a) An edge at a given chunk level (b) The same edge at the next level (c) The new vertex is interpolated into its true place

Figure 2.21: Vertex positions when switching between levels

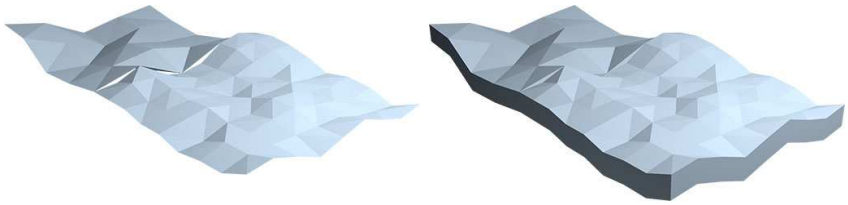
The interpolation parameter can be based on the distance to the camera or changed over time for each chunk.

Cracks and skirts

As chunks are tiled and rendered next to each other, it is desirable to make the borders between chunks as unnoticeable as possible. Even though the chunk meshes are generated according to the requirements mentioned in the Chunk subsection above, it is not possible to guarantee a watertight edge between two adjacent chunks of different LOD. Where adjacent chunks have different detail level, so called T-junctions till emerge. These T-junctions cause cracks between the chunks as unwanted visual artifacts.

The easiest way to tackle this issue is not to try to remove the cracks, but instead try to hide them. The most common approach hides the cracks by simply adding an extra layer of vertices to the sides of sides of the mesh. This extra layer of vertices, which is also known as a skirt, is offset down vertically, as illustrated in figure 2.22.

By adding skirts to the chunk meshes, the model will not be rendered with visible holes in it. Instead, the holes will be filled up with textured triangles.



(a) No skirts (b) Skirts

Figure 2.22: Chunks with skirts hide the undesired cracks between them

2.4.2 Geometry Clipmaps

A clipmap texture is a dynamic mip map where each image is clipped down to a constant size. This reduces the amount of memory of the whole texture to increase linearly instead of exponentially with the number of overlays for LOD textures [36]. Figure 2.23 a shows the difference between the amount of texture data stored in a regular mip map compared to a clip map.

The idea of clipmaps can be applied not only to textures but also to geometries [36]. By representing a terrain by a stack of clipmap geometries of different sizes, the resolution increases closer to the virtual camera, as illustrated in figure 2.24. As the view point moves around, the grids updates their vertex positions accordingly to keep the grid centered in the geometry clipmap stack. The position of each of the levels of the clip map geometries

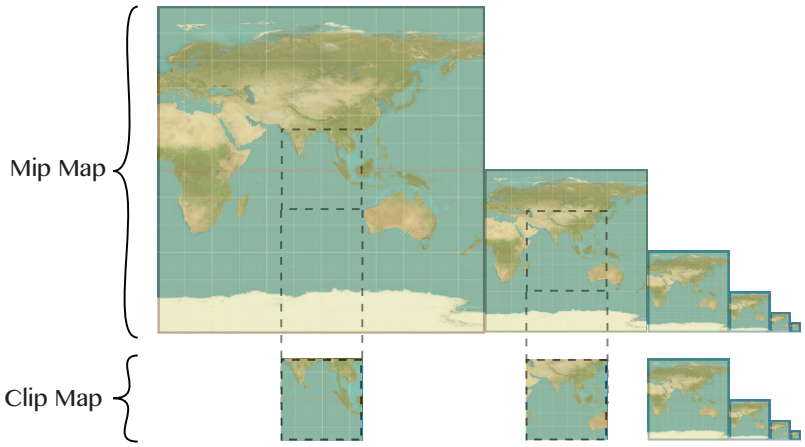


Figure 2.23: Clip maps are smaller than mip maps as only parts of the complete map need to be stored. Figure adapted from [13]

snaps to a discrete coordinate grid with cell sizes equal to the distance between two adjacent vertices. Due to the different grid resolution of different levels, the relative position of each sub grid must change so that they can snap on a grid with higher level. This is illustrated with the dynamic interior part of the grid in figure 2.24.

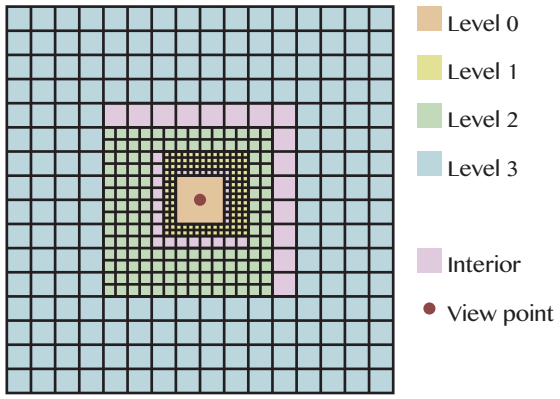


Figure 2.24: The Geometry Clipmaps follow the view point. Higher levels have coarser grids but covers smaller areas. The interior part of the grid can collapse so that higher level geometries can snap to their grid

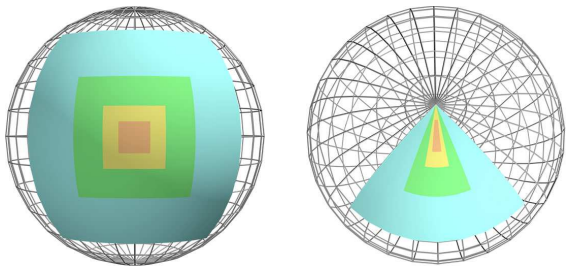
Geometry clipmaps limits the terrain representation to be in the form of height maps. This is because the clipmap geometry moves around when the focus point changes and since the underlying terrain should not follow

the camera position, the clipmap geometries require vertex shader texture fetching. The texture coordinates are offset as the geometry clipmap moves to follow the focus point.

One of the main selling points for geometry clipmaps is the decrease in CPU workload and the increase in GPU triangle throughput [2]. There is no need to traverse a hierarchical structure such a quadtree. The number of draw calls will remain equal the number of clip maps instead of the number of chunks which is often larger. The frame rate will also be relatively consistent if the number of layers in the clipmap stack remains constant [2].

2D Grid Geometry Clipmaps

Using geometry clipmaps to achieve dynamic level of detail for a height mapped grid was proposed by Losasso and Hoppe [37]. The method is limited to rendering of equirectangular grids. When considering the ellipsoidal shape of a globe, the clipmap grid can be represented in geographic coordinates mapped on an ellipsoid where the map texture coordinates in the longitudinal direction wraps around the anti meridian. Rendering the clipmap close to the poles however will lead to polar pinching which breaks the globe as illustrated in figure 2.25.



(a) Near the equator (b) Near a pole

Figure 2.25: Geometry Clipmaps on a geographic grid cause pinching around the poles, which needs to be handled explicitly

Clipmap grids can also be used to model a spherical cube representation of a globe to avoid polar issues. This requires six clipmap partitions; one for each side of the cube.

Spherical Clipmaps

Spherical clipmaps takes advantage of the fact that no observer will ever see more than half a globe at any time. The vertices of the clipmap are described in polar coordinates with the center of the grid always following

Table 2.1: Comparison between geometry clipmaps and chunked LOD

	Geometry Clipmaps	**Chunked LOD**
Preprocessing	Minimal	Extensive
Mesh Flexibility	None	Good
Triangle Count	High	Lower
Ellipsoid Mapping	Challenging	Straightforward
Error Control	Poor	Excellent
Frame-Rate Consistency	Excellent	Poor
Mesh Continuity	Excellent	Poor
Terrain Data Size	Small	Large
Legacy Hardware Support	Poor	Good

the camera position [38]. The coordinates of the vertices will therefore not have any correspondence to texture coordinates which is why the algorithm is not widely adopted [24].

Ellipsoidal Clipmaps

Dimitrijević and Rančić introduces ellipsoidal clipmaps as a level of detail rendering method for globes [24]. It uses a geographic grid for the polar regions of the globe and solves the polar issues with the use of polar caps. The advantages compared to spherical cube clipmaps is that it uses the well known geographic map projection and reduces the number of geometry clipmap partitions from six to two. The globe is divided up into one equatorial region and two polar regions. If the distance is close enough to the globe (3,000 km above the surface for the Earth), a maximum of two partitions is needed at any time [24].

The area distortion and the aspect distortion of the map projection of the ellipsoidal clipmap method is low compared to cube map projections but requires extra care to hide the discontinuities that appear at the edges between the equatorial and polar partitions [24].

The most generally used level of detail algorithms of today are chunked LOD and varieties of geometry clipmaps. Cozzi and Ring compare the algorithms and gives the advantages and disadvantages depending on the needs of the globe browsing software. The main differences are summarized and presented in Table 2.1 [2, p. 464].

2.5 Precision Issues

Modern graphics cards work with floating point variables and single precision is the common standard [39]. With OpenGL 4.0 came the introduction of double precision floating point numbers within the GLSL shader programming language [39]. The double precision floating point operations are significantly slower than their single precision counterparts [24]; we choose to avoid double precision floating point operations on the GPU for this reason.

In most computer graphics applications it is often sufficient to describe positional information in single precision floating point values. However, when simulating and visualizing the full scale of the universe and at the same time globes with sub meter resolution, the issues of low precision float operations easily become predominant unless handled with care.

2.5.1 Floating Point Numbers

A floating point number y is represented by a sign bit s, a significand $c \in N$, a base (also known as the radix) $b \in Z, b > 2$ and an exponent $q \in Z$. See equation 2.3.

$$y = (-1)^s \times c \times b^q \tag{2.3}$$

These numbers are flexible since they can represent big ranges. They have varying accuracy depending on the exponent q since the numbers c and q are stored as integers with a limited number of bits.

In the Institute of Electrical and Electronics Engineers 754 (IEEE 754) standard, the base b is a constant 2 or 10 and the significand c and the exponent q are set according to a well defined scheme [40]. With floating point numbers, operations such as addition and subtraction of values decreases precision due to a drop in the least significant bits [2]. The precision is reduced even more due to accumulating errors [2].

2.5.2 Single Precision Floating Point Numbers

A 32 bit floating point number can accurately represent around 7 significant decimal figures in the IEEE 754 standard [2].

Considering an Earth sized globe with the origin in the center and a radius of 6×10^6 meters, the theoretical vertical resolution is approximately $6 \times 10^{6-7} = 0.6$ meters. However, due to arithmetic operations the resolution reduces even more. Furthermore, these issues get worse when considering rendering of more objects in a scene that is representing not only one planet but the entire solar system. The distance from the sun to the earth is roughly

1.5×10^{11} meters which leaves a positional resolution in the order of $10^{11-7} = 10^4$ meters which is far from acceptable when browsing globes.

2.5.3 Double Precision Floating Point Numbers

A double precision floating point number is able to accurately represent 16 decimal figures [2]. Consider the Voyager 1 spacecraft, which left the solar system and entered interstellar space in August 25, 2012 and currently is at a distance in the order of magnitude of 10^{13} meters. This spacecraft is the farthest any human made object has gone; farther away than all the planets and can still be accurately positioned with sub meter resolution using double precision floating point variables (of course assuming that the positional data is correct).

NASA's Navigation and Ancillary Information Facility (NAIF) is the producer of the SPICE interface [5]. In SPICE, positional information of celestial bodies within our solar system are defined in double precision floating point numbers. Utilizing this precision makes it possible to accurately visualize space crafts in relation to globes.

2.5.4 Rendering Artifacts

Vertex Position Jittering

One noticeable artifact related to the 32 bit floating point limitation of graphics cards is the discrete positioning of vertices when the origin is far away from the rendered object. A clear example of this is shown in figure 2.26 where Jupiter's moon Europa is rendered with single precision floating point operations and the origin is placed at the barycenter of the solar system.

The artifact is shown as discrete positioning of vertices which causes jagged edges of the globe when the positions of the vertices exceeds subpixel precision. When moving the camera, the discrete positioning of the vertices appears to jitter due to the fact that they are transformed to camera space with limited precision.

Cozzi and Ring propose several different solutions to this problem which include rendering "relative to center", "CPU - relative to eye", "GPU - relative to eye" and "GPU - relative to eye - FUN90" [2].

Z-Fighting

Another well known issue related to precision in rendering of highly detailed objects with a large range of possible distances to the camera is z-fighting. Z-fighting appears as triangles or fragments flipping between being positioned

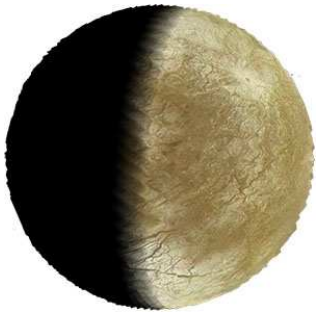


Figure 2.26: Jupiter's moon Europa rendered with single precision floating point operations. The precision errors in the placement of the vertices is apparent as jagged edges even at a distance far from the globe.

in front of or behind each other. Normally the issue appears when it is undecidable what relative depth two objects have, see figure 2.27. When dealing with large distances, undecidable depths can appear for objects such as terrains with bigger relative depths.

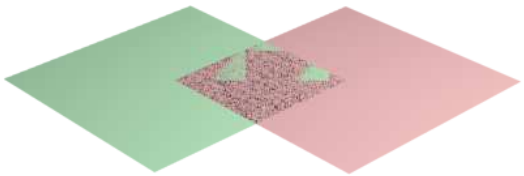


Figure 2.27: Z-fighting as fragments flip between being behind or in front of each other

The problem is due to the inverse relationship that OpenGL implicitly puts on the depth buffer when the perspective division is performed together with the change in precision in the size of the floating point numbers. This causes redundant precision close to the camera and increasingly lower precision for coordinates with big depth values [2].

The problem is already handled in OpenSpace with the use of a linear depth buffer. The depth buffer value is changed by explicitly setting the value of `gl_FragDepth` in all fragment shaders. The value is simply set to the z-coordinate in camera space negated and normalized to fit all distances within the observable Universe.

A negative aspect about setting the value of `gl_FragDepth` explicitly is that early depth testing is not possible. This is OpenGL's method of discarding fragments before the fragment shader has been invoked. Not performing early depth testing leads to lower frame rates if the fragment shader is heavy. Kemen discussed the implications of setting `gl_FragDepth` explic-

itly and concluded that for regular fragment processing, the change in frame rate was not significant for the Outerra software [41].

Other previously proposed methods for handling depth buffer issues include using multiple view frustums and complementary depth buffering [2].

2.6 Caching

Efficient caching of data is crucial in out-of-core, real time data visualization applications. Since the data sets dealt with often can be measured in terabytes or petabytes, caching is most efficient if done in multiple stages.

2.6.1 Multi Stage Texture Caching

There are six levels of caching important to take into consideration for globe rendering softwares. These are:

1. Local GPU texture memory
2. Local RAM
3. Local drive memory
4. Local server
5. Remote server

When implementing a globe rendering system, the local RAM memory caching is of most interest and needs to be implemented explicitly.

2.6.2 Cache Replacement Policies

There are multiple different cache replacement policies, each used for different purposes. Some common policies are: First-In-First-Out, Last-In-First-Out, Least-Recently-Used, Most-Recently-Used, Least-Frequently-Used, Most-Frequently-Used.

When it comes to caching of texture tiles used for globe rendering, the most reasonable policy gets rid of the tiles that have not been used after a significant amount of requests in favor of more recently used ones. Therefore the Least-Recently-Used (LRU) cache replacement policy is typically used [2, p. 386].

LRU Caching

The LRU cache is filled up as new entries are requested. When the cache is full, the least recently used entry will be thrown away when a new entry is to be added. This is implemented with a hash map and a doubly linked list. The hash map maps a unique resource identifier for an entry to a node in the list, which in turn stores the entry. Every time an entry is accessed from the cache, the node storing that entry is placed first in the list. This scheme ensures the least recently used entry will always be located in the back of the list so that it can be thrown out when the cache is full, see figure 2.28.

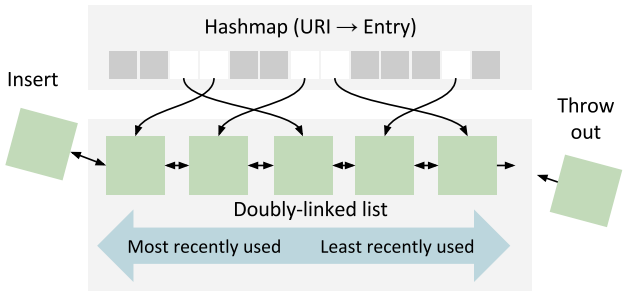


Figure 2.28: Inserting an entry in a LRU cache.

Chapter 3

Implementation

The virtual globe rendering system was implemented as a separate module, programmed in C++ for OpenSpace which, if desired, can be opted out when building the software. The implementation defines a namespace with all the necessary data structures, classes and functionality specifically related to globe rendering. The top level class *RenderableGlobe* with its necessary components is illustrated as a UML diagram in figure 3.1.

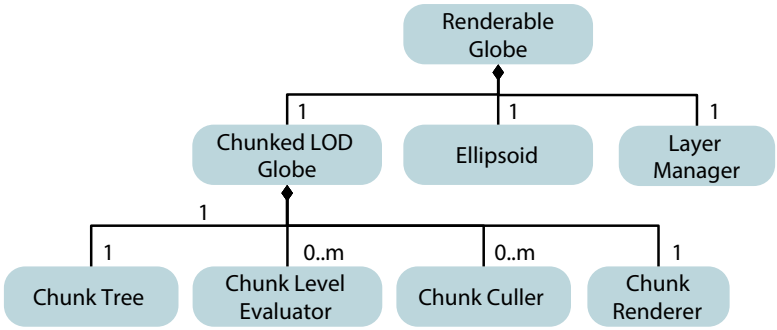


Figure 3.1: Overviewing class diagram of *RenderableGlobe* and its related classes.

The chunked LOD approach implemented is a slightly modified version of what Cozzi and Ring[2, p. 445] uses. The globe is tessellated as an ellipsoid with a geometrical grid using geodetic map projection.

3.1 Reference Ellipsoid

The *Ellipsoid* class was implemented to handle all geographic related calculations. These calculations include conversions between geodetic and Cartesian coordinates and different kinds of projections onto the ellipsoid surface. These calculations are sped up by internal caching of a range of pre-calculated values. Cozzi and Ring provide a complete reference on the implementation [2, p. 17].

The *Ellipsoid* uses several geographically related classes, which were used in multiple places within the implementation. These are:

1. *Angle* - Handles angle related arithmetic, normalization and unit abstraction (degrees and radians)
2. *Geodetic*2 - Represents a 2D geodetic coordinate (latitude, longitude)
3. *Geodetic*3 - Represents a 3D geodetic coordinate (latitude, longitude, altitude)
4. *GeodeticPatch* - Represents a rectangular region in geodetic space

3.2 Chunked LOD

The base of the chunked LOD algorithm revolves around the self updating chunk tree. The chunk tree is a data structure built up of *ChunkNode*s which have the ability to split or merge dynamically. Besides storing four *ChunkNode* children and a reference to its *ChunkedLodGlobe* owner, each *ChunkNode* stores a *Chunk*.

3.2.1 Chunks

As opposed to the definition of chunks in the background section 2.4.1, this implementation of chunks is very lightweight - it does not store any texture or triangle mesh data. Instead, it stores the information needed to query texture tiles from local in-memory caches. In the implementation suggested by Cozzi and Ring, terrain triangle meshes are stored in each chunk. In the case of this implementation however, all terrain is rendered using height mapped vertex grids. Thus there is no need for each chunk to store their own vertex arrays. Instead they can simply share one single instance of a vertex grid within a whole chunk tree. This means that vertices need to be offset by height mapping on the GPU which makes it possibly to dynamically change

height datasets that does not require pre processing before they can be used for rendering.

The most important part of the chunked LOD algorithm is the ability to dynamically select chunk nodes to split or merge. This is done by evaluating all the leaf nodes of the chunk tree. Chunks that are cullable will indicate that they can be merged by their chunk parent node. Chunks that cannot be culled indicate whether they want be split or merged based on a chunk selection algorithm. If all child nodes to a common chunk parent indicate they can be merged, the parent will merge its children. If any chunk reports it want to be split, its chunk node will initialize four new children within the chunk tree.

3.2.2 Chunk Selection

The chunk tree is automatically reshaped depending on the virtual camera. Three different approaches for calculating the error metric were implemented. By letting the user dynamically adjust a LOD scale factor, performance can be weighted against detail level.

By Distance

Letting the error depend on the distance between the closest point on the chunk and the camera d as in Equation 3.1, will lead to more or less constant size of the chunks in screen space.

$$e = l - log_2(\frac{s}{d}), \tag{3.1}$$

where e is the error, l is the current level of the chunk, s is a LOD scaling factor and d is the distance between the closest point of the chunk and the camera. Using this distance as an error metric leads to bigger chunks farther from the camera where less detail is needed.

By Projected Area

Another error metric is the area that the chunk takes up on the screen. The bigger the area, the bigger the error.

The error must not be dependent of the direction of the camera. This is because a chunk rendered on a multi screen display, such as a dome, should not have different errors between two or more screens which might lead to different levels and tearing between screens. Therefore the chunk is projected on a unit sphere and not a view plane which would lead to view direction dependent error metrics.

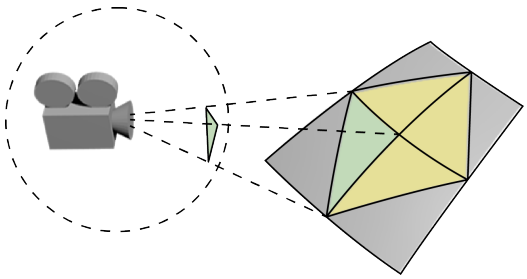


Figure 3.2: Triangle with the area 1/8 of the chunk projected onto a unit sphere. The area is used to approximate the solid angle of the chunk used as an error metric when selecting chunks to render

The projected area is a solid angle approximated by extracting three points on the chunk and projecting them on a unit sphere centered in the camera position. The three points define the closest of the four center triangles on the chunk, see figure 3.2. It is important that the triangle chosen for approximating the projected area can never have two vertices on the same upper or lower most edge. Such triangles (colored gray in figure 3.2) may collapse down to a line and hence have zero area for chunks at the poles, thus cannot be used to approximate the area of the full chunk.

The actual approximated solid angle is the projected triangle area multiplied by eight to accommodate a full chunk. This area is then subtracted by a constant value and scaled by a LOD scale factor to give similar LOD scaling as the distance dependent chunk selection.

3.2.3 Chunk Tree Growth Limitation

The chunk selection algorithms described above determine a suitable level of detail based on the camera. This causes chunks that have too large of an error to split. However, splitting a chunk will not result in higher level of detail unless the corresponding higher tile data (textures, heightmaps, etc) is also currently available for rendering. Therefore, the growth of the chunk tree is limited by checking if there is any tile data there in the first place. This is useful in two scenarios:

1. Rendering of sparse map datasets which contain geographical regions where there are no data

2. Rendering of map datasets which are queried from remote servers; there will always be some delay where the queried map data is not yet available

By enabling limiting the chunk tree growth in these two scenarios, unnecessary chunk rendering calls can be avoided.

3.2.4 Chunk Culling

Given a chunk tree where the level of detail is selected based dynamically based on the camera, each chunk is then tested whether they are visible to the camera. Chunks that are not visible are said to be "cullable", and does not need to be rendered. The two chunk culling algorithms implemented are Frustum Culling and Horizon Culling. They both rely on - and have access to - the minimum and maximum values of the chunk's currently active height maps.

Frustum Culling

Frustum culling is implemented by first calculating a convex bounding polyhedron for the chunk to be tested. The bounding volume is calculated on the fly and takes into account any enabled height maps used to displace the vertices, making sure it fully encapsulates the displaced chunk vertices. The polyhedron is built up of eight vertices which are transformed to normalized device coordinates (NDC) using the view-projection matrix of the camera followed by perspective division. Once in NDC, an axis aligned bounding box (AABB) for the vertices is extracted. This AABB can then be tested against the screen bounds to determine if the chunk is outside the camera's field of view and thus is cullable. This is illustrated in figure 3.3.

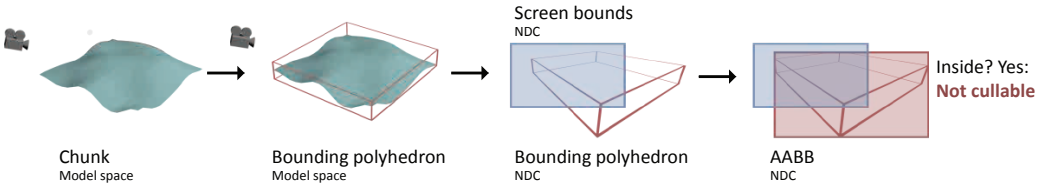


Figure 3.3: Frustum culling algorithm. This chunk cannot be frustum culled.

Horizon Culling

Given a camera position, an object in position $\vec{p}$ with a bounding radius r on the surface of the globe, it can be determined whether the object is completely hidden behind the globe's horizon or not. The calculations are simplified by approximating the globe as a sphere using the minimum radius

r_g of its ellipsoid. Using the minimum radius and not a bigger number ensures that false occluding (chunks marked as invisible actually being visible) is not possible [2, p. 393]. The minimum allowed distance l to the object can be calculated as the distance to the horizon l_h added to the minimum allowed distance to the object from the horizon l_m. See figure 3.4. Once the minimum allowed distance is calculated it can be compared to the actual distance to the object to determine if it is cullable or not.

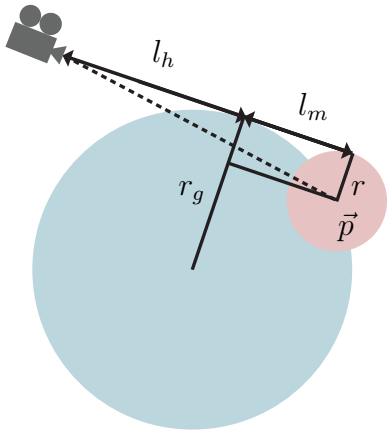


Figure 3.4: Horizon culling is performed by comparing the length $l_h + l_m$ with the actual distance between the camera position and the object at position $\vec{p}$.

When culling chunks, the closest position on the chunk is used as the position $\vec{p}$ and the bounding height value as r.

3.3 Reading and Tiling Image Data

Fetching the right texture and preparing it for rendering onto chunks is a fairly complicated process. Before digging into the details of this process, there are three concepts that need to be established, as they will be referred to throughout the description of the texture pipeline. These are:

1. *TileIndex* - A tuple of three integers $(level, x, y)$ indexing a specific map region on the globe.

2. *TileStatus* - An enumeration value indicating whether the *Tile* is "OK", "Unavailable", "Out of Range" or whether reading the pixel data triggered an "IO Error". Tiles that have the status "OK" are uploaded to the GPU and can be used for rendering.

3. *RawTile* - A texture carved out to fit the geographical region of a specific chunk, along with meta data. Each *RawTile* is associated with a *TileIndex* and has a *TileStatus*. The *RawTile* class is guarantueed to be thread safe.

4. *Tile* - Like a *RawTile*, but the texture data is uploaded to the GPU and ready to use for rendering (unless it has a status not equal to "OK"). As opposed to *RawTiles*, the *Tile* class rely on an OpenGL context and thus is not thread safe.

All chunk height and texture data are represented using *Tiles*. *Tiles* are created on the client side (i.e. in OpenSpace) on the fly when they are needed. As the pixel data may need to be read from disk or even requested from remote servers, the whole tile preparation pipeline was implemented to be executed on separate threads in order to avoid blocking of the rendering thread with image data reads.

During the iterative process of developing the texture tile pipeline, three layers of abstraction were introduced in order to deal with the fairly high complexity. See table 3.1.

Table 3.1: Abstraction layers used in the texture data pipeline

Layer	**Component**	**Responsibility**	**Input –> Output**
3	*AsyncTileDataset*	Async *RawTile* fetching	*TileIndex* –> *RawTile*
2	*TileDataset*	Tiling, georeferencing, preprocessing	*TileIndex* –> *RawTile*
1	GDAL	Image formats, I/O Operations, georeferencing	pixel region –> pixel data

The subsequent sections of this chapter will cover each abstraction layer in more detail, starting from the bottom and going up the stack.

3.3.1 GDAL

Geospatial Data Abstraction Library (GDAL) is an open source library providing a uniform interface for reading, manipulating and writing geospatial data in the form of raster and vector data models [42]. It provides an interface allowing client code to specify pixel regions within a dataset to read from, independent of the underlying image format. Reading pixel data using the GDAL requires a set of parameters listed below and illustrated in figure 3.5:

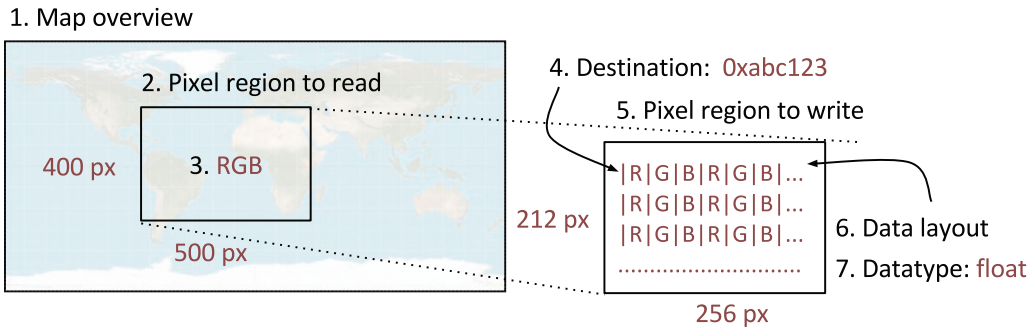


Figure 3.5: The required GDAL RasterIO parameters.

1. A map overview (also known as mip map) to read pixel data from
2. A pixel region within that map overview to read pixels from
3. The raster band(s) (e.g. color channels) to read
4. A pointer to sufficient user allocated memory where GDAL can write the output of the read operation
5. The pixel region to write the output pixel data. The size of the pixel data to be written may differ from the pixel region to read from, in which case GDAL will resample and interpolate the pixel data automatically
6. The layout (i.e. pixel spacing, raster band spacing and line spacing)
7. The data type to retrieve the pixel data in

With the given input parameters shown in figure 3.5, the resulting output would be the pixel data of the requested image region written with the pixel layout parameters to the provided memory block. The output is illustrated in figure 3.6.

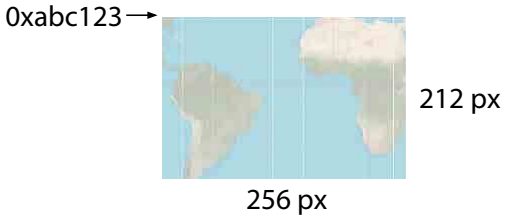


Figure 3.6: Result of GDAL raster IO.

The same interface can also be used to read sparse datasets. A detailed example of how to use GDAL for reading sparse datasets such as local, geo-referenced, image patches of high resolution can be founed in Appendix B.

GDAL also provides the coefficients of a geo-transform which defines the mapping from raster coordinates to georeferenced coordinates. The geo-transform can be inverted to transform a georeferenced region to raster space when specifying the area of an image tile to read.

3.3.2 Tile Dataset

*TileDataset*s carves out *RawTile*s from GDAL datasets based on a *TileIndex*. Along with the pixel data, the served *RawTile*s also contain some metadata. The metadata includes an error code if the read operation failed and some basic information about to read pixel data, such as minimum and maximum pixel values. Figure 3.7 illustrates how the process from *TileIndex* to *RawTile* is carried out. There are three gray subroutines; *Get IO description*, *Read image data* and *Calculate metadata*. These subroutines are explained below.

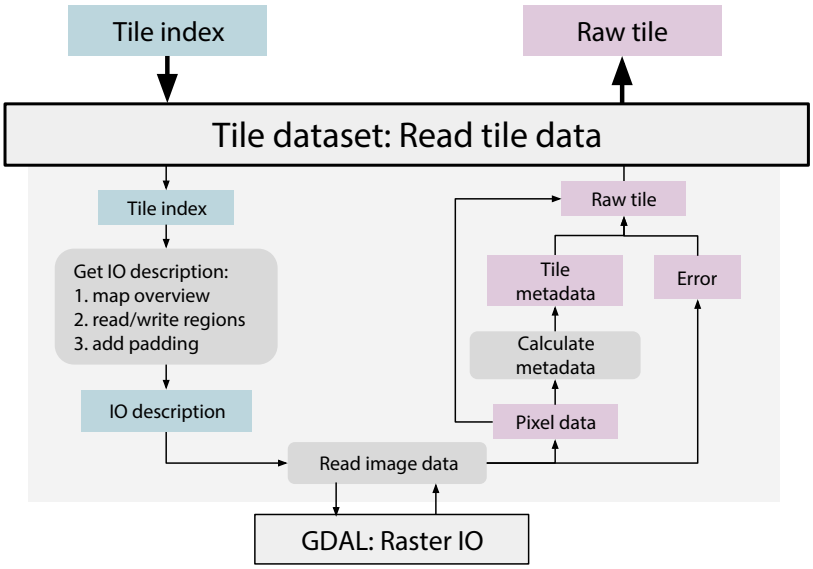


Figure 3.7: The tile dataset pipeline takes a tile index as input, interfaces with GDAL and returns a raw tile

Get IO Description

The IO description contains all tile specific information needed to perform a map read request using GDAL. This includes the pixel region to read and where to store the result. The derivation of this information is summarized in the following steps and illustrated in figure 3.8.

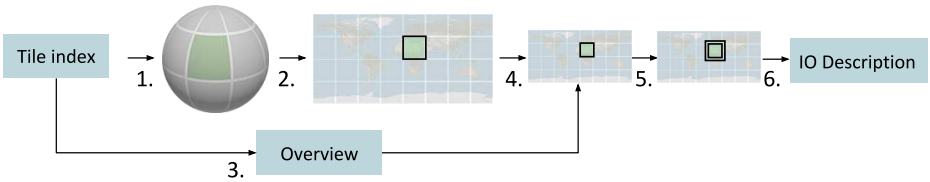


Figure 3.8: Overview of the calculation of an IO description.

1. Calculate a *GeodeticPatch* from *TileIndex*
2. Calculate the pixel coordinates for the patch in raster coordinate space
3. Calculate a suitable map overview
4. Transform pixel region to map overview pixel space
5. Add padding to the down scaled pixel region
6. Collect the information in an IO description object.

In the scheme in figure 3.8, step 1 is performed using equation 3.2,

$$\begin{aligned} \phi_{NE} &= 2\pi y/2^{level} \\ \theta_{NE} &= 2\pi x/2^{level} \\ side &= 2\pi/2^{level}, \end{aligned} \tag{3.2}$$

where ϕ_{NE} and θ_{NE} are the latitude and the longitude of the north east corner of the tile and *side* is the side length of the tile.

Calculating the corresponding GDAL overview is done according to equation 3.4,

$$Overview(level) = N - level - 1 - log_2(size_{tile}/size_{map}), \tag{3.3}$$

where *level* is given by the provided *TileIndex*, N is the total number of overviews in the dataset, $size_{tile}$ is a configurable constant defining the

preferred pixel size of a tile and $size_{map}$ is the size of the full map in pixels. The sizes can be either along the $x-$ or $y-$ axis. In the implementation the $x-$axis is used.

Pixel coordinates can easily be transformed across map overviews using equation 3.4.

$$p_n = p_m \times 2^{m-n} \tag{3.4}$$

Where n is the destination map overview and m is the source map overview. This is used for downscaling the pixel region in step 4, where n is the calculated suitable overview and m is zero (i.e. the full map).

Padding is added to the pixel region in order to perform correct interpolation of pixel values across different tiles later during rendering. However, this may cause the pixel region to extend outside the map region. Therefore the last finalize step also handles wrapping of the pixel region before returning the final IO description. The wrapping used is a CPU implementation of GL_REPEAT.

Tile Meta Data

As mentioned, $TileDataset$s can be configured to calculate some metadata on the fly based on the pixel data that has been read. The metadata includes minimum and maximum pixel values within the pixel data and whether or not the pixel data contains missing-data values. Having access to minimum and maximum values for height layer tiles is required for the culling to be performed correctly since the cullers rely on having bounding boxes for the chunks. The meta data is collected by explicitly looping through all the pixel values on the client side.

Summary

To summarize, the implementation of $TileDataset$ allow reading pixel data from a GDAL dataset corresponding to a specific $TileIndex$, along with metadata unless opted out. The $RawTile$s that are served are padded in order to perform smooth pixel blending between different tiles. $RawTile$s are not available for rendering since the data is not yet on the GPU.

3.3.3 Async Tile Dataset

$AsyncTileDataset$s utilize a shared thread pool and own a $TileDataset$. It defines a concurrent job posting interface for concurrent reads within the $TileDataset$. It has two important functionalities: 1) enqueing tile read jobs

and 2) collect finished *RawTiles*. Reading *RawTiles* on separate threads ensures that the render thread will not be delayed by image data requests, see figure 3.9.

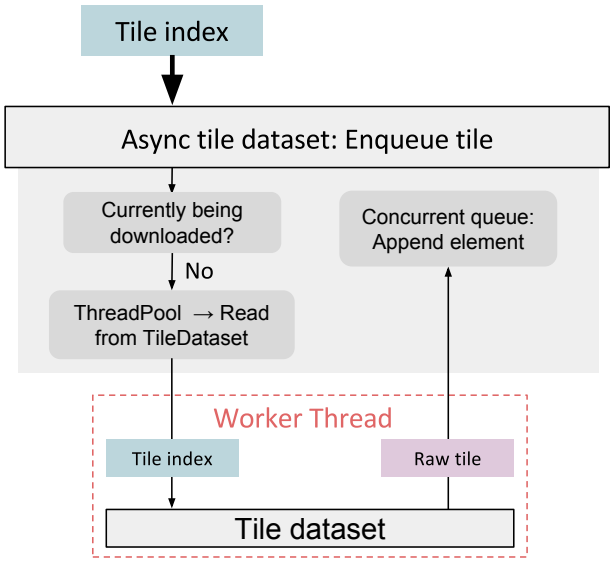


Figure 3.9: Asynchronous reading of Raw tiles can be performed on separate threads. When the tile reading job is finished the raw tile will be appended to a concurrent queue.

Figure 3.10: Retrieving finished *RawTiles*.

The *AsyncTileDataset* internally keeps track of what tile indices it has enqueued and what tile pixel regions are currently being read. If a pixel region for a specific tile index is already enqueued or currently being read, the request is ignored. Figure 3.10 shows how raw tiles can be fetched once they are finished and enqueued.

3.4 Providing Tiles

*Tile*s have three properties: a texture, metadata and a status. The status is used to report any type of problem with the tile. Tiles that have the status "OK" are uploaded to the GPU and can be used for rendering.

Tiles are provided for rendering through *TileProvider*s. They define an interface that allow accessing tiles and meta data about the tiles. There are different types of tile providers, but they must all implement the following functionality:

- GetTile(*TileIndex*) - access the *tile* at the provided *TileIndex*
- GetDefaultTile() - returns a default *Tile* with status "OK"
- GetDepthTransform() - pixel value scale factor (for example converting height map values to meters)
- GetMaxLevel() - the maximum tile level supported by this provider)
- GetNoDataValue() - get value that should be interpreted as "no data"
- CheckTileStatus(*TileIndex*) - check *TileStatus* with no side effects
- Update() - called once per frame, allow for internal updates
- Reset() - full reset of internal state

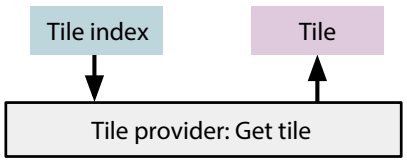


Figure 3.11: Tile provider interface for accessing tiles

The most important functionality is the GetTile ability, which is used by client code to access the tiles. As tile providers may provide tiles of any status, the user of the tile provider is responsible to always check the status of the requested tile before using the tile.

Several implementations of the tile provider interface were developed. They are all described below.

3.4.1 Caching Tile Provider

The *CachingTileProvider* uses an *AsyncTileDataset* to read *RawTile*s as soon as client code tries to access a specific tile. It internally polls the *AsyncTileDataset* every update for finished raw tiles. The *CachingTileProvider* converts the raw tiles into tiles. This is done in the initialization step which is part of the update method that is illustrated in figure 3.13. If no errors of the raw tile are reported, the tile gets the status "OK" and its texture data gets uploaded to the GPU. The tile also gets added to the in-memory "Least Recently Used"-cache. The functionality of accessing tiles is illustrated in figure 3.12.

The uploading of texture data to the GPU needs to be on the rendering thread since there is where the OpenGL context resides. Tiles with data uploaded to texture memory will enable it for use in rendering of a chunk.

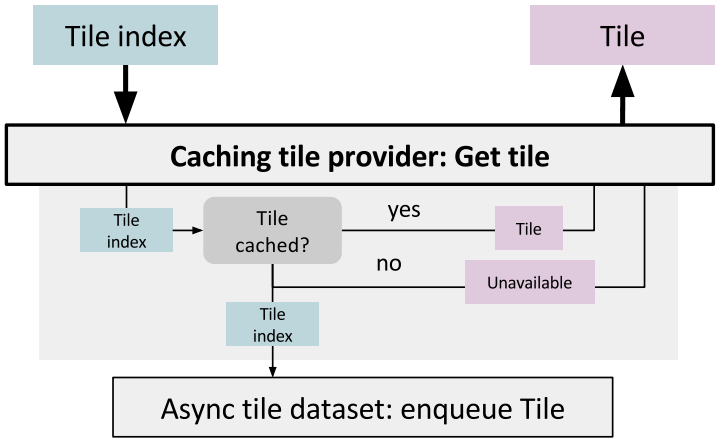


Figure 3.12: Tiles are either provided from cache or enqueued in an asynchronous tile dataset if it is not available

The functionality for internally updating the tile cache is illustrated in figure 3.13.

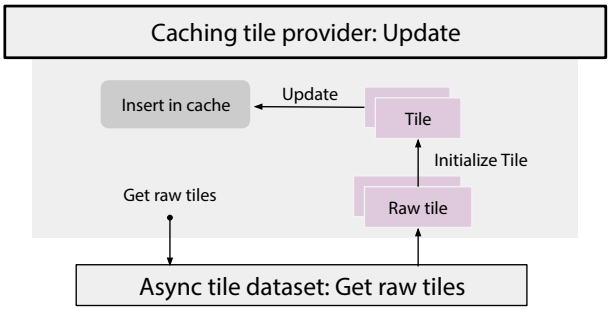


Figure 3.13: The tile cache is updated once per frame

Figure 3.14 demonstrates a typical scenario where a specific tile of index $(3, 4, 2)$ is requested within a sequence of render calls. The first requesting call to that tile will spawn a worker thread in the *AsynchTileDataset*. As soon as the *Tile* is initialized (uploaded to the GPU) and inserted in the cache, it will be accessible on the rendering thread. If the tile is not yet available the *CachingTileProvider* will report that so that the calling function can continue without the use of that specific *Tile*.

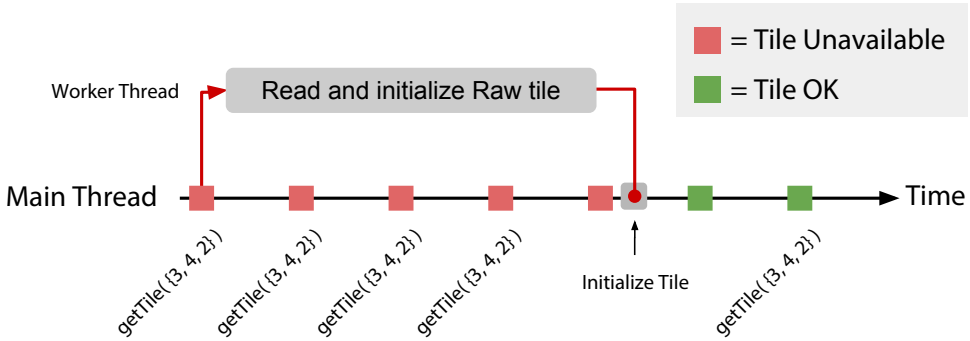


Figure 3.14: Tiles are fetched on demand. The first time a tile is requested, the asynchronous tile dataset will request it on a worker thread. As soon as the tile has been initialized it will have the status "OK" and can be used for rendering

3.4.2 Temporal Tile Provider

In order to incorporate time-resolved map datasets into the rendering scheme, a tile provider for this specific purpose was implemented.

The *TemporalTileProvider* is instantiated with a template URI containing a time placeholder. Information about the supported time range,

time resolution and expected timestamp format to be used in the template URI is also passed during instantiation. In Listing 3.1, all tags starting with `OpenSpace` are used to configure information used to instantiate the temporal dataset. The tag `GDAL_WMS` contains a functional GDAL WMS dataset specification. When requesting the URLs for the dataset, `${OpenSpaceTimeId}` will be replaced with the date and time in the specified format.

Listing 3.1: Temporal WMS dataset specification

```
<OpenSpaceTemporalGDALDataset>
    <OpenSpaceTimeStart>2015-11-24</OpenSpaceTimeStart>
    <OpenSpaceTimeEnd></OpenSpaceTimeEnd>
    <OpenSpaceTimeResolution>1d</OpenSpaceTimeResolution>
    <OpenSpaceTimeIdFormat>YYYY-MM-DD</OpenSpaceTimeIdFormat>
    <GDAL_WMS>
        <Service name="TMS">
            <ServerUrl>
                http://datasetURL/${OpenSpaceTimeId}/${z}/${y}/${x}.jpg
            </ServerUrl>
        </Service>
        ...
    </GDAL_WMS>
</OpenSpaceTemporalGDALDataset>
```

At runtime, the *TemporalTileProvider* checks the global simulation time, quantizes that time with respect to the provided time resolution and lazily instantiates new *CachingTileProvider*s per timeframe within the temporal dataset. An schematic illustration is given in figure 3.15.

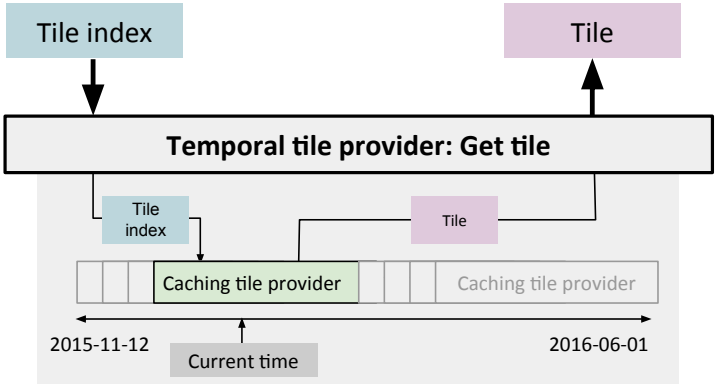


Figure 3.15: Each temporal snapshot is internally represented by a caching tile provider

3.4.3 Single Image Tile Provider

This is a very simple implementation of the tile provider interface which only serves the same tile for every tile index. This tile provider was used for

testing and debugging alignment and padding between tiles, see figure 3.16.

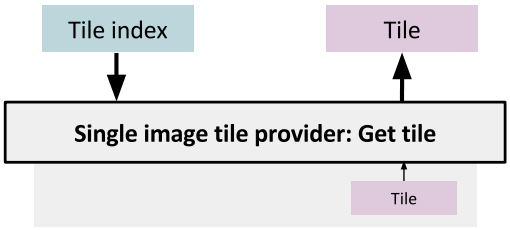


Figure 3.16: Serving single tiles is useful for debugging chunk and texture alignment

3.4.4 Text Tile Provider

The ability to serve tiles with text rendered on the fly was implemented as a general debugging utility. The tiles are generated on demand by rendering text to textures and cached using the same LRU caching mechanism as the *CachingTileProvider*.

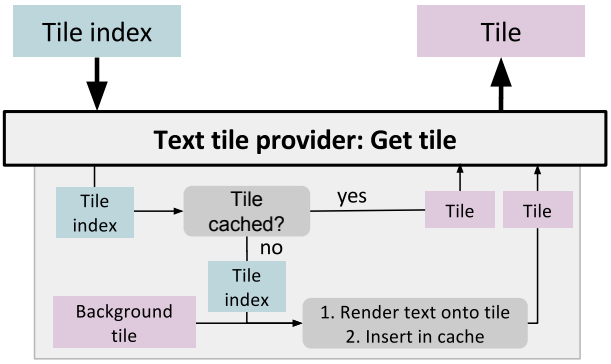


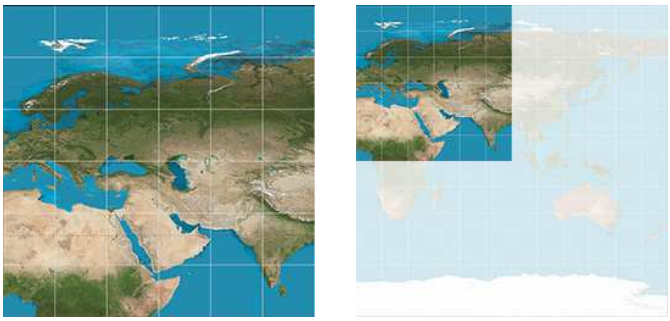
Figure 3.17: Serving tiles with custom text rendered on them can be used as size references or providing other information. The tile provider is internally holding a LRU cache for initialized tiles

Two different types of *TextTileProvider*s were implemented. *TileIndexTileProvider* serves tiles where each tile has its tile index rendered onto it and *SizeReferenceTileProvider* uses the globe's reference ellipsoid to render a size reference in meters or kilometers onto its tiles.

3.5 Mapping Tiles onto Chunks

In the implementation of chunks suggested by Cozzi and Ring [2], chunks store the data they need for rendering by themselves. That means that as soon as a chunk has been fully initialized, it has everything it needs to be rendered. However, when dealing with multiple map datasets potentially residing on distant servers, there is no guarantee that all the tiles needed for a chunk to be rendered are available.

When an accessed *Tile* does not have the status "OK", it can not be used for rendering. The second best thing to try is to check the parent of the tile. A parent tile is guaranteed to always cover a larger geodetic map region that includes its children's subregions, but with lower number of pixels per geodetic degree. Figure 3.18 demonstrates this with a simple example.



(a) Requested tile

(b) Requested tile's region shown in its parent tile

Figure 3.18: Only the highlighted subset of the parent tile is used for rendering the chunk. Figure adapted from [13]

In figure 3.18, it is realized that in order to use a parent tile for rendering, the texture coordinates used to sample the parent tile needs to be adjusted to represent the same geographic region. This shows the need of a higher level concept than just *Tile*s, which leads to the introduction of *ChunkTile*s.

3.5.1 Chunk Tiles

A *ChunkTile* represents a *Tile* that corresponds to a specific *Chunk*. It stores a *Tile* which is guaranteed to have the status "OK" along with a transform defined by a scaling and translation component. The transform is used to map texture coordinates of the chunk into its corresponding geodetic region within the tile.

The algorithm used for selecting the highest resolution *ChunkTile* from a tile provider is described by the pseudo code in Listing 3.2.

Listing 3.2: Selecting optimal Chunk Tiles

```
ChunkTile TileSelector::getChunkTile(TileProvider* tp, TileIndex ti){
  TileUvTransform uvTransform;
  uvTransform.uvOffset = glm::vec2(0, 0);
  uvTransform.uvScale = glm::vec2(1, 1);

  while (ti.level > 1) {
      Tile tile = tp->getTile(ti);
      if (tile.status == Tile::Status::OK) {
          return { tile, uvTransform };
      }
      else {
        ascendToParent(ti, uvTransform);
      }
  }
  uvTransform.uvOffset = glm::vec2(0, 0);
  uvTransform.uvScale = glm::vec2(1, 1);
  return { tp->getDefaultTile(), uvTransform };
}
```

The subroutine `ascendToParent` returns an updated transform which maps texture coordinates to the same geodetic region within the next parent tile. The routine is described in Listing 3.3.

Listing 3.3: Ascend to parent

```
void TileSelector::ascendToParent(TileIndex& tileIndex, TileUvTransform&
    uv) {
    uv.uvOffset *= 0.5;
    uv.uvScale *= 0.5;

    if (tileIndex.isEastChild()) {
        uv.uvOffset.x += 0.5;
    }

    // In OpenGL, positive y direction is up
    if (tileIndex.isNorthChild()) {
        uv.uvOffset.y += 0.5;
    }

    tileIndex.toParent();
}
```

As opposed to regular tiles, chunk tiles can always be used for rendering since they by definition always have the status “OK”.

3.5.2 Chunk Tile Pile

A *ChunkTilePile* represents a range of *ChunkTile*s across multiple mip levels. They contain the information needed to perform the LOD switching later described under section 3.7.5. Retrieving a *ChunkTilePile* simply requires

a tile index of the highest desired mip level (highest LOD) and the number of desired *ChunkTile*s in the pile as described in Listing 3.4.

Listing 3.4: Instantiating a Chunk Tile Pile

```
ChunkTilePile TileSelector::getChunkTilePile(TileProvider* tp, TileIndex
    ti, int n){
  ChunkTilePile chunkTilePile;
  for (int i = 0; i < n; ++i){
    chunkTilePile.push_back(getChunkTile(tp, ti));
    ti.toParent();
  }
  return chunkTilePile;
}
```

As an example: Assuming that the texture data is available in local memory, invoking the getChunkTilePile method with index $\{x : 2240, y : 4824, level : 13\}$ and a chunk tile pile size 3, would return the *ChunkTilePile* represented in figure 3.19.

(a) Tile (b) Parent 1 (c) Parent 2

Figure 3.19: The image data of a given chunk tile pile. Only the highlighted subset of the parent tiles are used for rendering the chunk. Figure from [28]

As chunk tiles are guaranteed to be good for rendering on the GPU (since their tiles are guaranteed to have the status "OK"), all *ChunkTilePile*s are guaranteed to be good for rendering as well.

3.6 Managing Multiple Data Sources

The *LayerManager* maintains and organizes different types of texture data sources into groups. This is required as different types of texture datasets are used for different purposes and rendered in different ways. The *LayerManager* owns seven *LayerGroups*; these are:

1. HeightLayers
2. ColorLayers
3. ColorOverlays
4. GrayScaleLayers
5. GrayScaleOverlays
6. NightLayers
7. WaterMasks

3.6.1 Layers

Much like in image editing softwares, a layer in this context represents a raster of pixels, which among other internally ordered raster of pixels, are used to produce a final result. A *Layer* consists of three things:

1. A *TileProvider* - Used for accessing tiles of texture data.
2. A collection of render settings for real time image processing. The parameters implemented at this stage are *gamma*, *multiplier* and *opacity*.
3. A simple boolean property which specifies whether the layer is enabled or disabled. Disabled layers are not used in rendering.

*Layer*s allow for retrieving of *ChunkTilePile*s, as described in section 3.5.2. The class hierarchy and data flow is illustrated in figure 3.20.

3.6.2 Layers on the GPU

The data hierarchy defined by *LayerManager* can almost be reproduced on the GPU by using GLSL in a near object-oriented approach. In order to handle the data mapping between the CPU and GPU, a CPU representation of the GPU data hiearchy was implemented. This allows for easily updating the values of uniform variables and map uniform names to uniform locations.

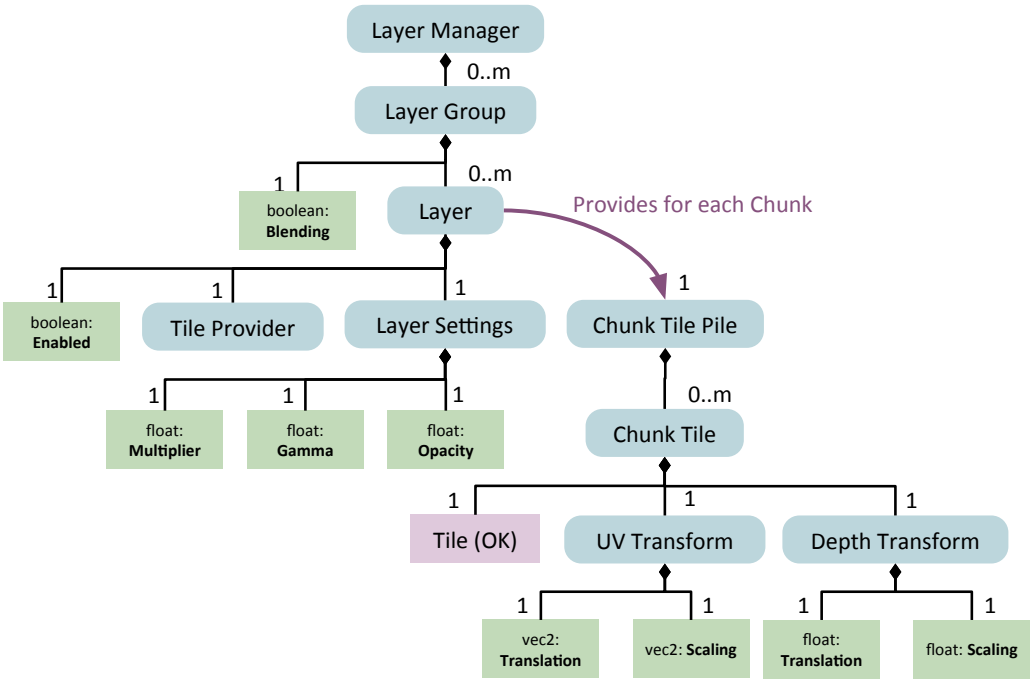


Figure 3.20: UML diagram of the *LayerManager* and its related classes

CPU to GPU Data Mapping

There are some key differences between the *LayerManager* data structure on the CPU and its corresponding GPU-synchronized representation - *GPULayerManager*. These are:

- The GPU representation does not store any of the actual *Layer* data - only the uniform locations within a shader program so that it knows where to upload the layer data to the GPU.
- Render calls are done on a per chunk basis. This means that all texture data to be used within a single render call is contained in a single *ChunkTilePile* for each *Layer*. Thus, the *provides*-relation between *Layer* and *ChunkTilePile* in figure 3.20 simply becomes a *has*-relation between a *GPULayer* and a *GPUChunkTilePile*.
- Layers that are disabled are not rendered and consequently not uploaded to the GPU. This means that all layers on the GPU are enabled, thus they do not need to store that information.

Incorporating these few differences yields a similar class hierarchy, as illustrated in figure 3.21.

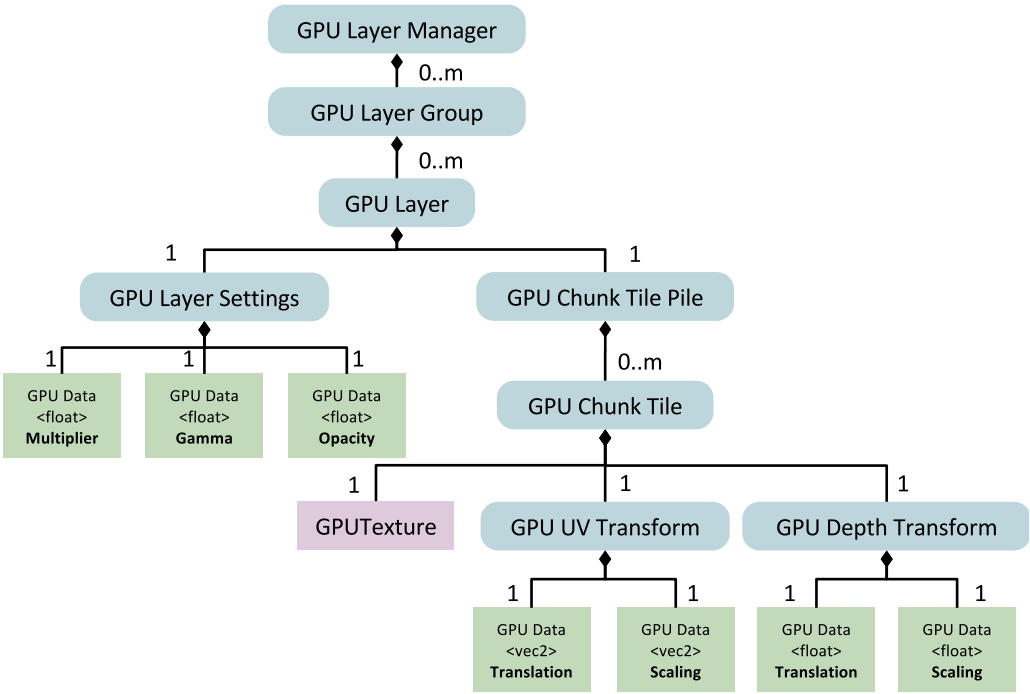


Figure 3.21: UML structure for corresponding GPU-mapped hierarchy. The class `GPUData<T>` maintains an OpenGL uniform location

The leaf classes within the class hierarchy store one uniform location each and represent one GLSL uniform value. The declaration of the layer uniforms within the shader code is defined so that it matches the exact same hierarchy. This is implemented by mapping C++ classes to GLSL structs and representing *one-to-many* relationships as structs with either a range of conformally named fields or plain GLSL arrays.

Updating the GPU Data

With this 1:1 data mapping between the CPU and GPU, all the GPU-prefixed classes were given two responsibilities.

1. Bind its uniform locations to GLSL variable names within a provided shader program
2. Set its uniform values within a provided shader program. That is, a *GPUChunkTile* should be able to set its uniform values from a regular *ChunkTile*

The leaf classes automatically takes care of both binding its uniform location to specific GLSL variables and setting their uniform values. The compound GPU-prefixed classes (non leaf classes) more or less simply propagates the method calls to all its dependents. When binding an object to a variable name, the GLSL variable identifiers are built up successively during the propagation down to the leaf classes. *GPULayer* is used as an example in Listing 3.5.

Listing 3.5: Bind a GPU Layer to a Layer on the CPU

```
void GPULayer::bind(ProgramObject* p, std::string nameBase, int pileSize)
    {
  this->gpuChunkTilePile.bind(p, nameBase + "pile.", pileSize)
  this->gpuRenderSettings.bind(p, nameBase + "settings.")
}
```

In Listing 3.5, it can be concluded that each layer in the GLSL code must be a struct with a member called “pile” and a member called “settings”. An example of a fully resolved identifier is:

```
"ColorLayers[0].pile.chunkTile0.tileUvTransform.scaling"
```

When setting the values, the GPU representation of the data is updated based on the currently available *Layer* data and the update is propagated down to all the leaf classes. This is exemplified using code from the *GPULayer* class defined in Listing 3.6.

Listing 3.6: Set all GPU Layer variables from a CPU Layer

```
GPULayer::setValue(ProgramObject* p, Layer& l, TileIndex ti, int n){
  ChunkTilePile chunkTilePile = l.getChunkTilePile(ti, n);
  this->gpuChunkTilePile.setValue(p, chunkTilePile);
  this->gpuRenderSettings.setValue(p, l.renderSettings());
}
```

3.7 Chunk Rendering

Rendering of chunks uses both vertex and fragment shaders. As the globe geometry is always represented as height maps, the same vertex geometry - a uniform grid - is used as a basis for rendering all chunks.

3.7.1 Grid

The vertex grid used for rendering a chunk has square size. It is defined by the number of segments N along its sides, which is used to build up a two dimensional array of vertex coordinates. These vertex coordinates can be used both for texture fetching and as interpolation parameters in an inverse geographic projection to place each vertex in 3D space. The grid also contains the extra strip of vertices defining the skirt of the chunk. Figure 3.22 shows how a the skirt region have uv-coordinate values less than 0 or greater than 1 in either the u or the v dimension.

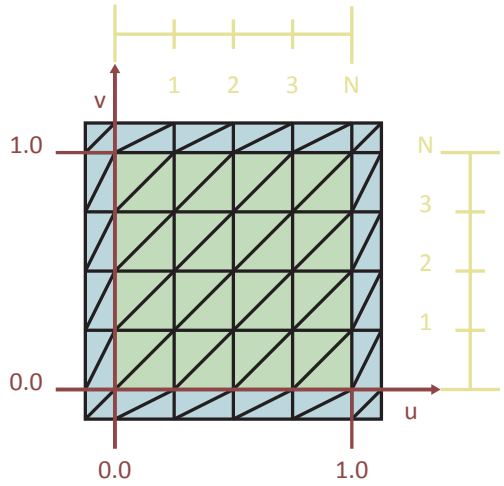


Figure 3.22: Grid with skirts with a side of $N = 4$ segments. Green represents the main area with texture coordinates $\in [0, 1]$ and blue is the skirt of the grid.

3.7.2 Vertex Pipeline

Rendering the grid includes performing displacement mapping of the vertices based on active heightmaps as well as offsetting skirt vertices downwards. Two different methods for rendering the vertex grid were implemented. The first method performs these operations in model space, whereas the second one performs these operations in camera space.

Model Space Rendering

The vertices of the grid is transformed into geographic space on the GPU using a transform derived from the chunk's tile index. From geographic space, an inverse projection transform is applied to place the vertices in the Chunk's Cartesian model space. From this space, the vertices are then displaced according to active height maps. This approach yields a limited precision of the vertices as they are defined relative to the center of the globe. However, it provides high accuracy as the vertices are positioned exactly on the globe in geodetic coordinates. Figure 3.23 illustrates inverse projection transform.

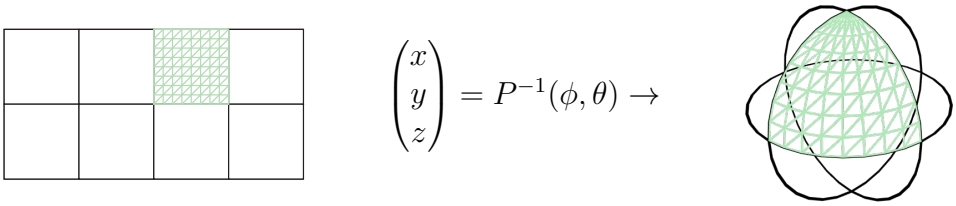


Figure 3.23: Model space rendering of chunks is performed with a mapping of vertices from geodetic coordinates to Cartesian coordinates.

Figure 3.24 shows the pipeline for rendering a chunk using the model space method.

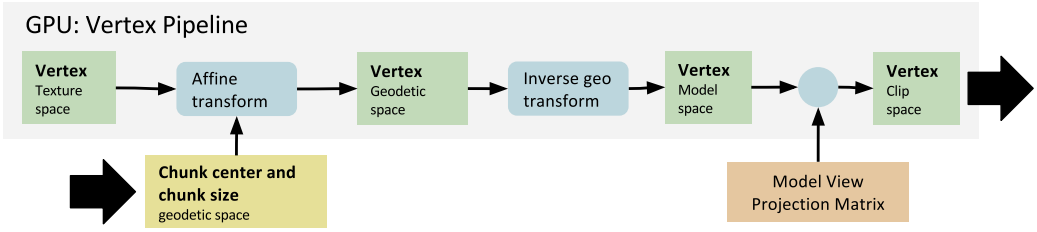


Figure 3.24: Vertex pipeline for model space rendering. Variables on the CPU are defined in double precision and cast to single precision before being uploaded to the GPU.

Camera Space Rendering

Camera space rendering is performed by transforming the corner points of the chunk to camera space using double precision arithmetics. Once in camera space, the points are cast to single precision and uploaded to the GPU. All the other vertices within the grid is then positioned on the GPU using bilinear interpolation of the corner points in camera space. The bilinnear interpolation means the grid will be rendered without curvature. However, having the points in camera space solves the proposed precision issues since the precision will increase when the points are closer to the camera. This concept is illustrated in figure 3.25 where the vertex position vector precision increases as vertices are closer to the camera.

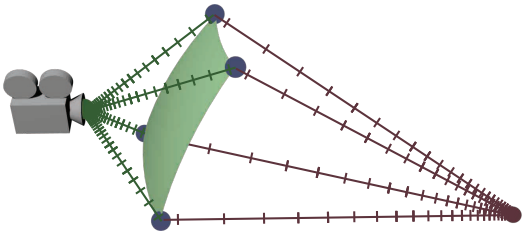


Figure 3.25: Interpolating vertex positions in camera space leads to high precision in the representation of vertex positions close to the camera compared to positions defined in model space.

Figure 3.26 shows a flowchart of the local vertex rendering pipeline.

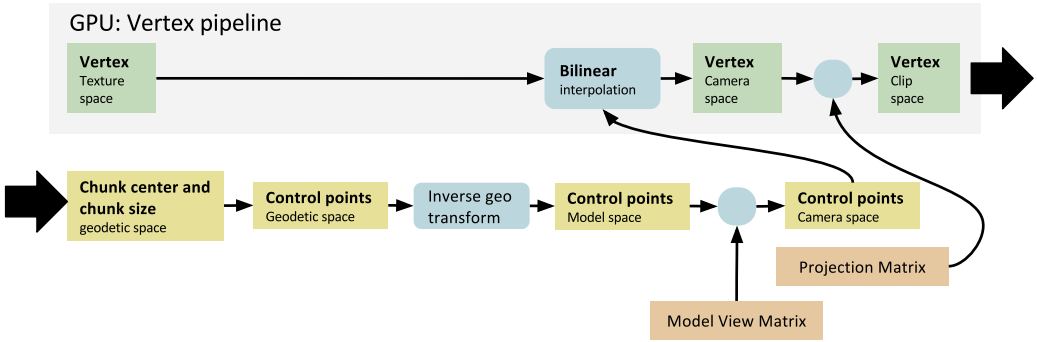


Figure 3.26: Vertex pipeline for camera space rendering. Variables on the CPU are defined in double precision and cast to single precision before being uploaded to the GPU

Combination

Both methods are used within the rendering system. Using the camera space rendering with bilinear interpolation method instead of model space rendering is done for chunks at level 10 and higher. This means that chunks with a latitudinal and longitudinal curvature of $360/2^{10} \simeq 0.3516$ degrees and less are rendered as flat surfaces.

3.7.3 Fragment Pipeline

The purpose of layered texture rendering is to provide the user an ability to toggle different kinds of layers for rendering a chunk. For each fragment, the final color is calculated by using the active layer groups. This is done using the following step in the fragment shader which is the same for the camera space rendered chunks as it is for the model space rendered chunks:

1. **For all ColorLayers**: Sample RGB, apply layer settings and update fragment RGBA using alpha blending
2. **For all GrayscaleLayers**: Sample the grayscale value R, apply layer settings and update fragment V in color space HSV using alpha blending
3. **For all GrayscaleOverlays**: Sample grayscale value R and A, apply layer settings and update fragment RGBA using alpha blending
4. **For all WaterMasks**: Sample A, apply layer settings and add A as a specularity component for the fragment
5. **For all NightTextures**: Sample RGB, apply layer settings and update fragment RGBA using alpha blending in shaded regions of the globe
6. Perform shading by darkening shaded regions
7. Add simple atmosphere color to RGB
8. **For all ColorOverlays**: Sample RGBA, apply layer settings and update fragment RGBA using alpha blending

All of these steps are optional and can be toggled by activating or deactivating layers or specifying whether or not to perform shading or use atmosphere for rendering. Sampling the color from a layer group is done by looping through all layers in that group, see Listing 3.7. Listing 3.8 and 3.9

shows how layer weights and texture transforms for chunk tile piles are used for sampling of the tile textures. Layer settings is performed for all raster bands within all layers. The application of layer settings is shown in Listing 3.10.

Listing 3.7: Setting color for a given layer group. Example using ColorLayers

```
#for i in 0..#{lastLayerIndexColorLayers} {
    vec4 colorSample = getTexVal(ColorLayers[#{i}].pile,
      levelWeights, uv);
    colorSample = performLayerSettings(colorSample,
      ColorLayers[#{i}].settings);
    color = blendOver(color, colorSample); // Alpha blending
  }
#endfor
```

Listing 3.8: Getting texture value by blending a chunk tile pile

```
vec4 getTexVal(ChunkTilePile chunkTilePile, LevelWeights w, vec2 uv){
  return w.w1 * getTexVal(chunkTilePile.chunkTile0, uv) +
    w.w2 * getTexVal(chunkTilePile.chunkTile1, uv) +
    w.w3 * getTexVal(chunkTilePile.chunkTile2, uv);
}
```

Listing 3.9: Using texture transform and sampling a texture

```
vec4 getTexVal(ChunkTile chunkTile, vec2 tileUV){
  vec2 samplePosition = TileUVToTextureSamplePosition(chunkTile, tileUV);
  vec4 texVal = texture(chunkTile.textureSampler, samplePosition);
  return texVal;
}
```

Listing 3.10: Perform layer settings

```
float performLayerSettings(float currentValue, const LayerSettings
    settings) {
  float newValue = currentValue;
  newValue = sign(newValue) * pow(abs(newValue), settings.gamma);
  newValue = newValue * settings.multiplier;
  newValue = newValue * settings.opacity;
  return newValue;
}
```

3.7.4 Dynamic Shader Programs

The ability to dynamically toggle layers with dynamic blending requires multiple sampling of textures in the fragment shaders. To avoid unnecessary processing when not all layers are in use there is a need to dynamically recompile the shader programs as layers are toggled. We use the GLSL preprocessor in the General Helpful Open Utility Library (GHOUL) [43] to preprocess all shader programs when the number of layers in use change.

Table 3.2: Data used for a layer group for preprocessing of layer shader programs

Layer Group Preprocessing Data
Number of layers: integer
Blending enabled: boolean

Table 3.3: Data used for a renderable globe for preprocessing of layer shader programs

Layer Shader Preprocessing Data
0..* Preprocessing data: Layer Group Preprocessing Data
0..* Key-value pairs: pair of strings

We define a layer shader program to be a GLSL program object using vertex and fragment shading. Moreover, it fits the pipeline of rendering a chunk with a specific number of layer groups and a varying number of layers for each group.

To decide whether or not a layer shader program needs to be preprocessed and re-compiled, the relevant Layer Shader Preprocessing Data is saved for the layer shader program and compared to the updated preprocessing data to determine if it has been changed by the user. Table 3.3 shows what data is needed to preprocess layer shader programs. Key-value pairs can be used to set properties that are used on a per-globe basis. Examples are **performShading** : *true* or *false*, **useAtmosphere** : *true* or *false*. Letting the preprocessor handle these properties avoids the need of uniform variable uploading and GLSL if-statements. The preprocessor also handles unrolling of for-loops which go through all layers in a group, see Listing 3.7.

3.7.5 LOD Switching

Instead of the time based switching routine proposed by Cozzi and Ring [2], we perform a distance based switching which works on a per fragment basis for textures and on a per vertex basis for height maps. The method is implemented in the vertex and fragment shaders and uses linear interpolation between levels. The blending technique uses the concept of Chunk Tile Piles as seen in section 3.5.2. A Chunk Tile Pile contains three chunk tiles of different level to achieve level blending which is used to avoid popping artifacts.

When rendering a specific fragment of a chunk, blending between up to three chunk levels can be used as a part of the switching in the chunked LOD

algorithm. Assuming that the level of Tile 0 is equal to the level of the chunk to render and that Chunk Tile 1 and Chunk Tile 2 are 1 and 2 levels lower respectively, three level weights can be determined based on the distance to the fragment to achieve smooth transitions between levels. The three level weights are based on an interpolation parameter t as in Listing 3.11.

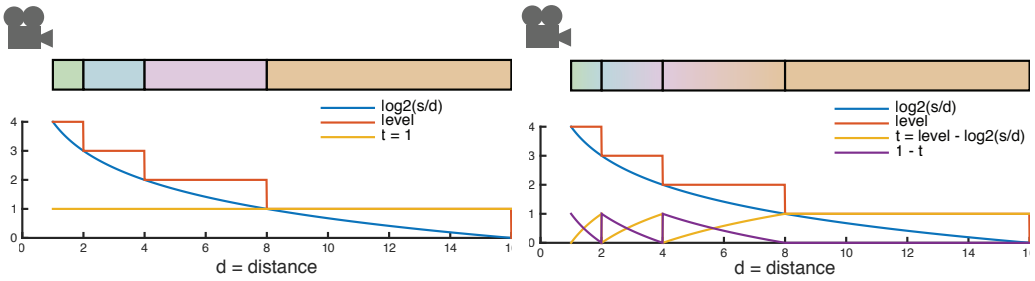
Listing 3.11: Calculate level weights for three chunk tiles using an interpolation parameter calculated based on the distance between the camera and the fragment

```
LevelWeights getLevelWeights(float t){ // level interpolation parameter
  LevelWeights levelWeights;
  levelWeights.w1 = clamp(1 - t, 0 , 1);
  levelWeights.w2 = clamp(t, 0 , 1) - clamp(t - 1, 0 , 1);
  levelWeights.w3 = clamp(t - 1, 0 , 1);
  return levelWeights;
}
```

The interpolation parameter t is calculated using the distance to the camera d and the current level of the chunk l as in equation 3.5.

$$t = l - log_2(\frac{s}{d}), \tag{3.5}$$

where s is a scale factor determining how much of the higher level tile to use in the blending. In the ideal case, the interpolation parameter should be a value between 0 and 1. However, adjacent chunks can have a difference in level greater than 1 which in turn leads to interpolation parameters with larger values. This is why it is not enough to use two Chunk Tiles per Chunk Tile Pile to guarantee no popping. Popping can also occur if the LOD scale factor s is small enough for adjacent chunks to have even bigger difference in level. Figure 3.27 shows how the interpolation parameter from the desired level is used for blending between tiles of different level.



(a) Blending disabled (b) Blending enabled

Figure 3.27: Blending on a per fragment basis. The level interpolation parameter t is used to calculate level weights $w_1 = 1 - t$ and $w_2 = t$, in this case using two chunk tiles per chunk tile pile.

3.8 Interaction

An interaction mode specifically for globe browsing had to be implemented to deal with the vast scale differences and limitations of camera movements required for browsing globes.

Representing the camera state as a three dimensional Cartesian vector for the position and a quaternion for the rotation made it possible to achieve interaction solving the proposed objectives. Basic linear algebra was used on vectors such as the geodetic normal of the globe in the camera position geodetically projected on the surface, camera position and direction as well as the height offset of the terrain.

The horizontal movement interaction speed was set to be proportional to the distance from the camera to the terrain surface. That way the user can come in close and slowly hover across the detailed terrains as well as quickly moving from one continent to the other by increasing the height to the surface.

The camera rotation quaternion can be decomposed in to two rotations; one directing the camera in the direction of the geodetic normal and one directing it in the remaining view direction with a given roll. This way it is possible to both travel across the surface and keeping the horizon angle as well as looking around at the spot.

The camera is always pushed up above the surface of the terrain to avoid penetrating it. The minimum height given in meters can be set together with other parameters such as sensitivity and friction.

Chapter 4

Results

Two types of results are presented in this chapter: screenshots showing different visual aspects of the system and benchmarks focusing on the achieved performance.

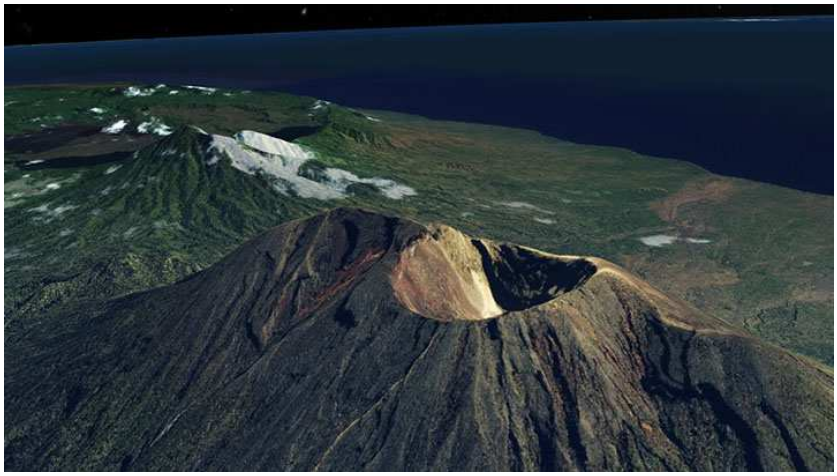
4.1 Screenshots

Screenshots from the implemented system are presented and described in each subsection below. Figure 4.1 shows the Earth shaded using a simple implementation of an atmosphere together with a daily image of surface reflectance.

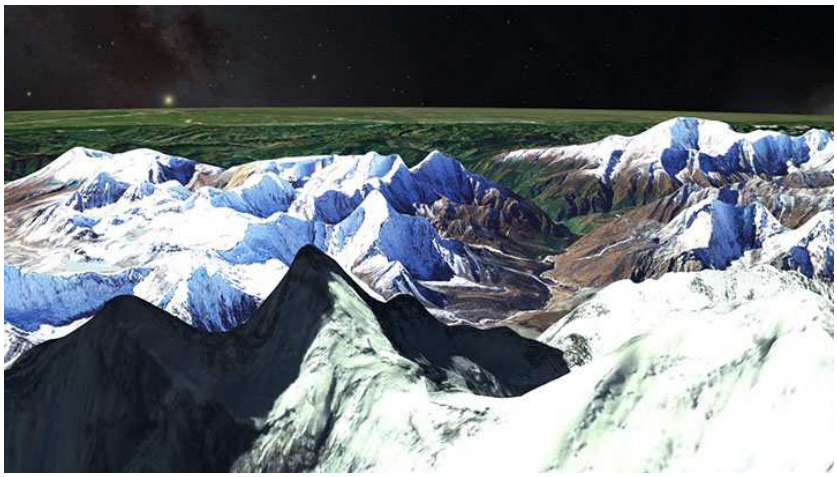
Figure 4.1: Shaded Earth rendered with NASA GIBS VIIRS daily image [17]

4.1.1 Height Mapping

Figure 4.2 shows the Earth rendered with a height dataset displacing the chunks' vertices according to the geometric shape of the ground.



(a) The volcano Gunung Agung on Bali.



(b) Looking south from Mount Everest

Figure 4.2: Earth rendered with ESRI World Elevation 3D height map [44]. Color layer: ESRI Imagery World 2D [28]

4.1.2 Water Masking

Figure 4.3 shows the Earth rendered with a water mask layer enhancing specular contrasts between land and water.

(a) Sun's specular reflection in the Amazon River.

(b) Sun's specular reflection around Bali, Indonesia.

Figure 4.3: Shaded Earth using water mask texture. Color layer: ESRI Imagery World 2D [28]

4.1.3 Night Layers

Figure 4.3 shows the Earth rendered with a night layer showing light from cities, which is only rendered on the night side of the planet.

(a) Earth at night

(b) New Deli at night

Figure 4.4: Night layers are only rendered on the night side of the planet. Night layer: NASA GIBS VIIRS Earth at night [17]

4.1.4 Color Overlays

To demonstrate color overlays, the three examples of figure 4.5 show how different tile providers produce references useful for debugging or locationing.

(a) Color Overlay: tile indices

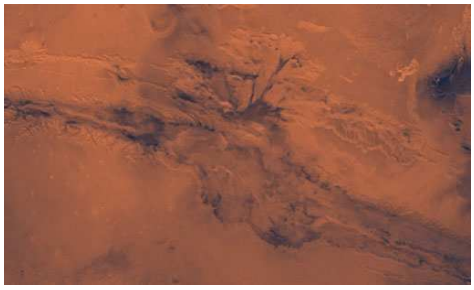
(b) Color Overlay: size reference

(c) Color Overlay: reference features and reference labels [17]

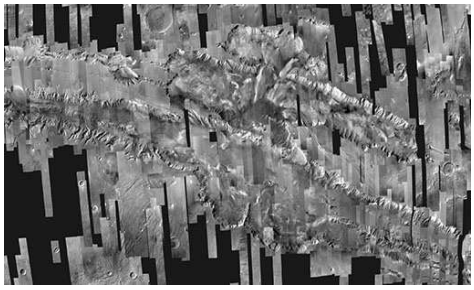
Figure 4.5: Earth rendered with different color overlays used for reference. Color layer: ESRI Imagery World 2D [28]

4.1.5 Grayscale Overlaying

In figure 4.6, the great canyon Valles Marineris on Mars is rendered with a low resolution color layer combined with a higher resolution grayscale overlay.



(a) Color layer: Viking MDIM [19]



(b) Grayscale layer: CTX Mosaic

(c) CTX Mosaic HSV blended on top of Viking MDIM [19]

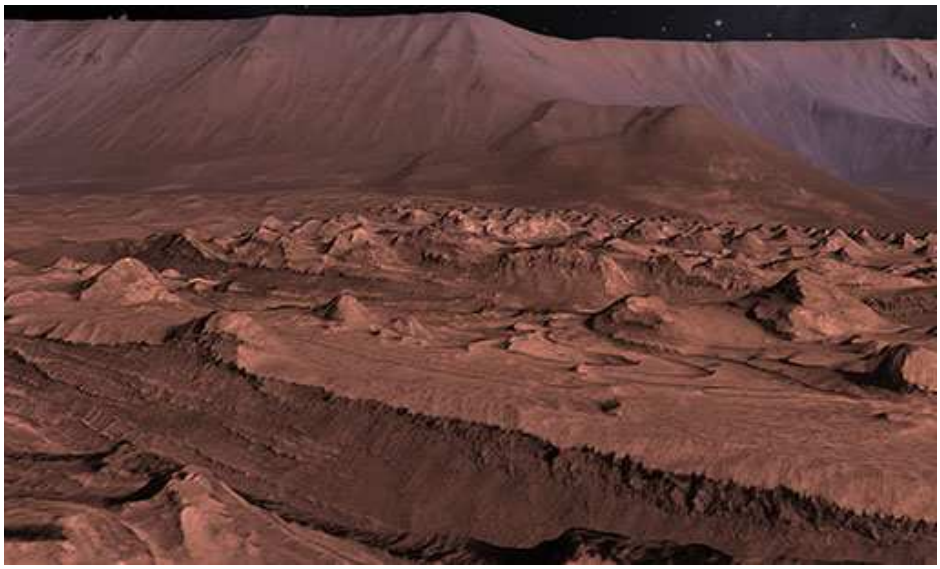
Figure 4.6: Valles Marineris, Mars with different layers

4.1.6 Local Patches

Figure 4.7 shows the approach of local DTM patches in West Candor Chasma.

(a) Local Patch: CTX DTM patch

(b) CTX mosaic, CTX DTM patch and 25 cm per pixel DTM patch

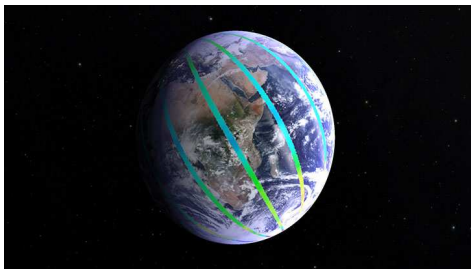


(c) CTX mosaic, CTX DTM patch and HiRISE 25 cm per pixel DTM patch closeup

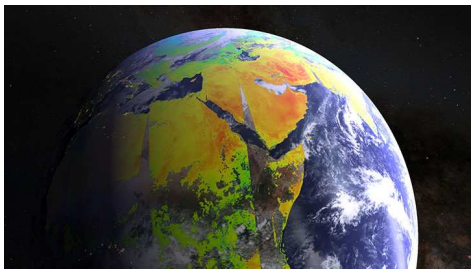
Figure 4.7: Local DTM patches of West Candor Chasma, Valles Marineris, Mars. All figures use color layer: Viking MDIM [19] and height layer: MOLA [45].

4.1.7 Visualizing Scientific Parameters

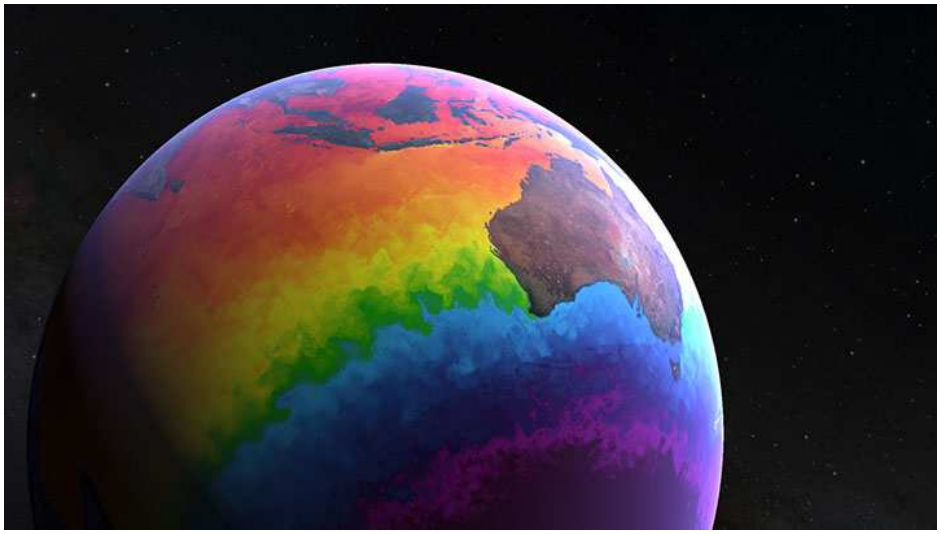
Using the vast amount of datasets provided by NASA GIBS [17], several scientific parameters measured by the Terra and Aqua satellites among others could be visualized as a result of layered texture rendering. Three examples are shown in figure 4.8.



(a) Ozone (measured in streaks)



(b) Land temperature

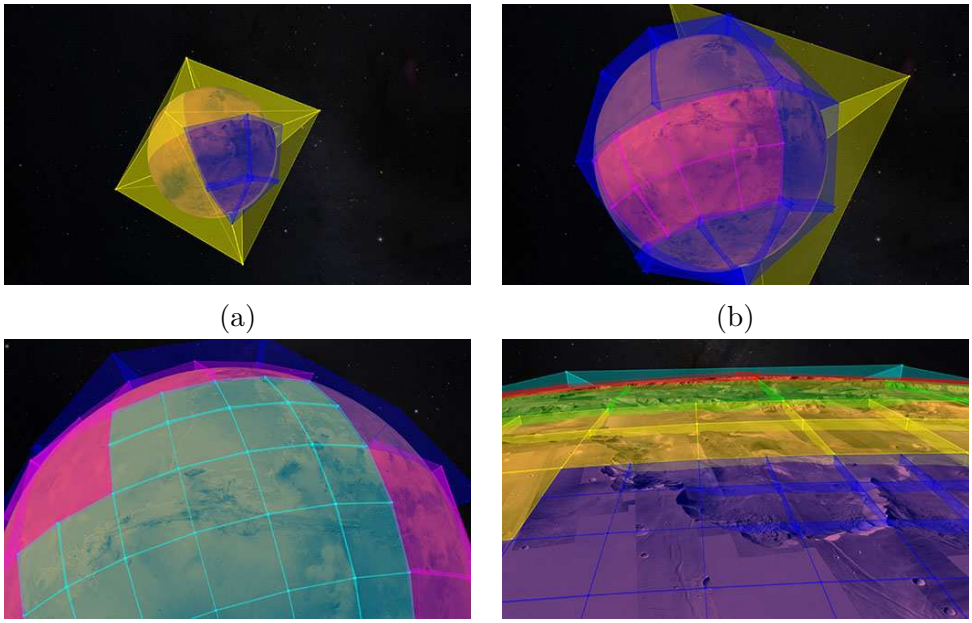


(c) Sea surface temperature

Figure 4.8: Visualization of scientific parameters on the globe. All these datasets are temporal and can be animated over time. Datasets from [17].

4.1.8 Visual Debugging: Bounding Volumes

Bounding polyhedra are calculated on the fly for each chunk, taking onto consideration any currently enabled height dataset. These bounding volumes are then used for by culling algorithms. Figure 4.9 shows how smaller, more planar chunks can be encapsulated more tightly than large chunks by polyhedron shapes of eight vertices converging to rectangular blocks for high chunk levels.



(a) (b)

(c) (d)

Figure 4.9: Rendering the bounding polyhedra for chunks at Mars. Note how the polyhedra start out as tetrahedra for the largest chunks in 4.9a and converge to rectangular blocks as seen in 4.9d.

4.1.9 Visual Debugging: Camera Frustum

Figure 4.10 shows a camera frustum visualized and the affect of frustum culling on chunks together with the difference in skirt length between levels.

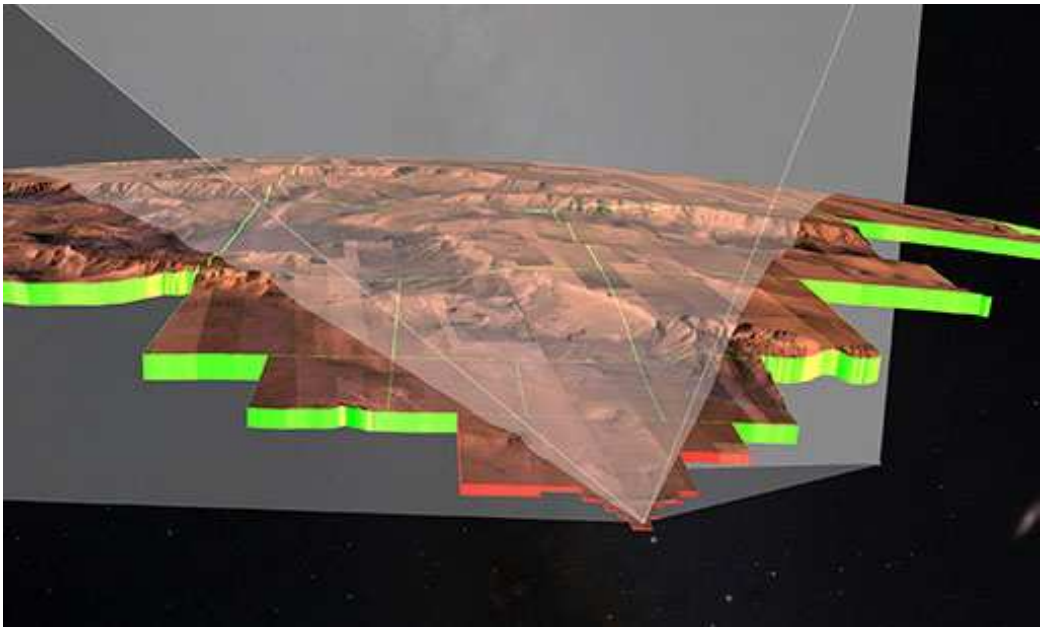


Figure 4.10: Chunks culled outside the view frustum. The skirt length of the chunks differ depending on the level. The figure also shows how some chunks are rendered in model space (green edges) and some in camera space (red edges).

4.2 Benchmarking

This section cover a range of benchmarks diagnosing the behavior and performance of the system. The specifications for the computer used for benchmarking is defined in table 4.1.

Table 4.1: Computer used for Benchmarking

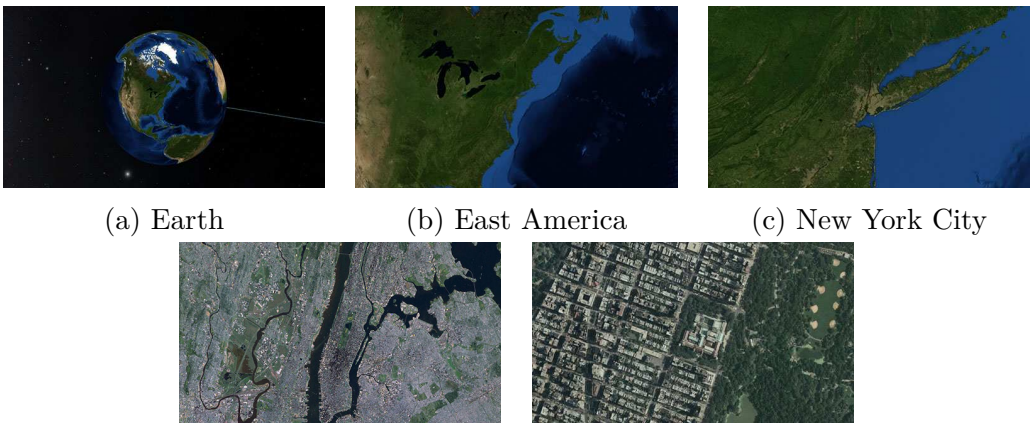
Computer Model:	MacBook Pro (15" early 2011)
Processor:	2GHz Intel Core i7
RAM:	8 GB 1333MHz DDR3 RAM
Graphics:	AMD Radeon HD 6490M 256 MB

4.2.1 Top Down Views

The chunk rendering algorithm was evaluated for a top down view at different distances to the ground. The settings for the evaluation are presented in Table 4.2. The camera views for the evaluation points are shown in figure 4.11 and the results are presented in figure 4.12.

Table 4.2: Globe rendering settings for top down view benchmark

Globe:	Earth
Map datasets:	HeightLayers = [GCS_Elevation [44]]
	ColorLayers = [ESRI Imagery World 2D [28]]
LOD Scale factor:	10.0
Chunk Selection:	By distance
Culling:	Frustum culling, Horizon culling
Level blending:	Enabled
Camera View:	Facing down, varying altitudes



(a) Earth (b) East America (c) New York City

(d) Manhattan (e) Central Park

Figure 4.11: Top down views of Earth at different altitudes

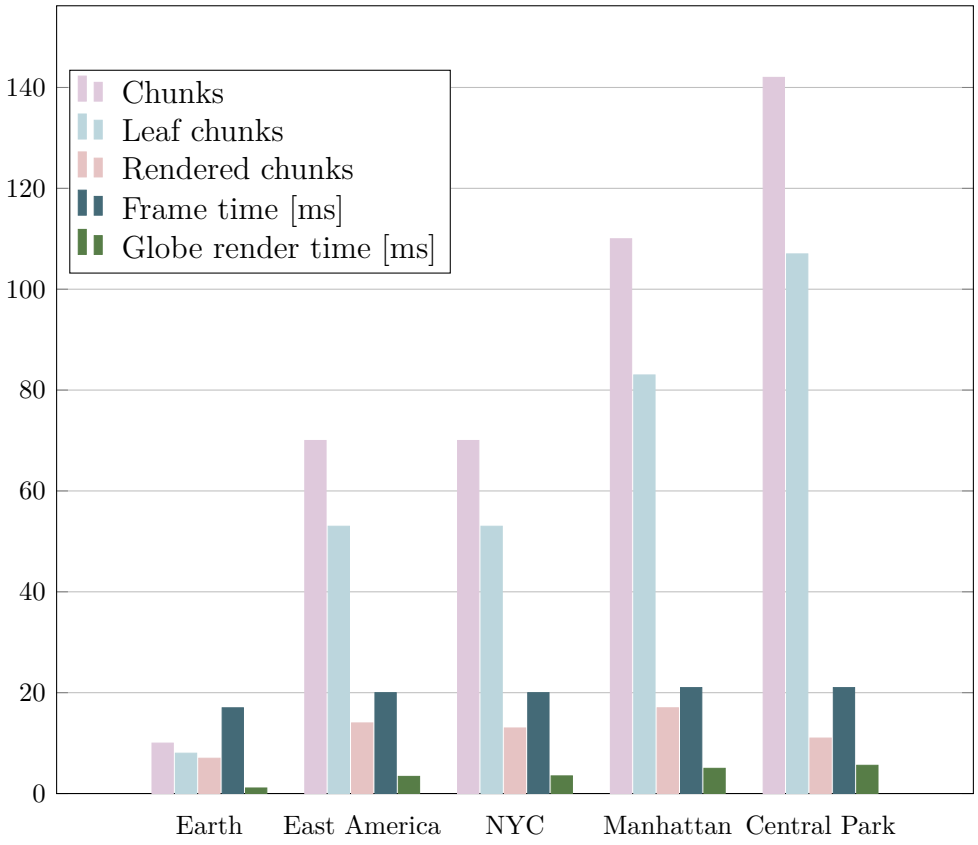


Figure 4.12: As the camera descends towards the ground looking straight down, the chunk tree grows but the number of rendered chunks remains relatively constant due to culling.

4.2.2 Culling for Distance Based Chunk Selection

The two culling algorithms frustum culling and horizon culling were evaluated in combination with the distance based chunk selection algorithm. The settings used are provided in table 4.3. Figure 4.13 shows the camera view evaluated. Figure 4.14 shows an overview of the rendered chunks with the results shown in figure 4.15.

Table 4.3: Globe rendering settings for evaluation of culling with distance based chunk selection

Globe:	Earth
Map datasets:	HeightLayers = [GCS_Elevation [44]]
	ColorLayers = [ESRI Imagery World 2D [28]]
LOD Scale factor:	7.8
Chunk Selection:	By distance
Culling:	**Evaluated**
Level blending:	Enabled
Camera View:	Looking towards western horizon
Location:	New York, New Jersey

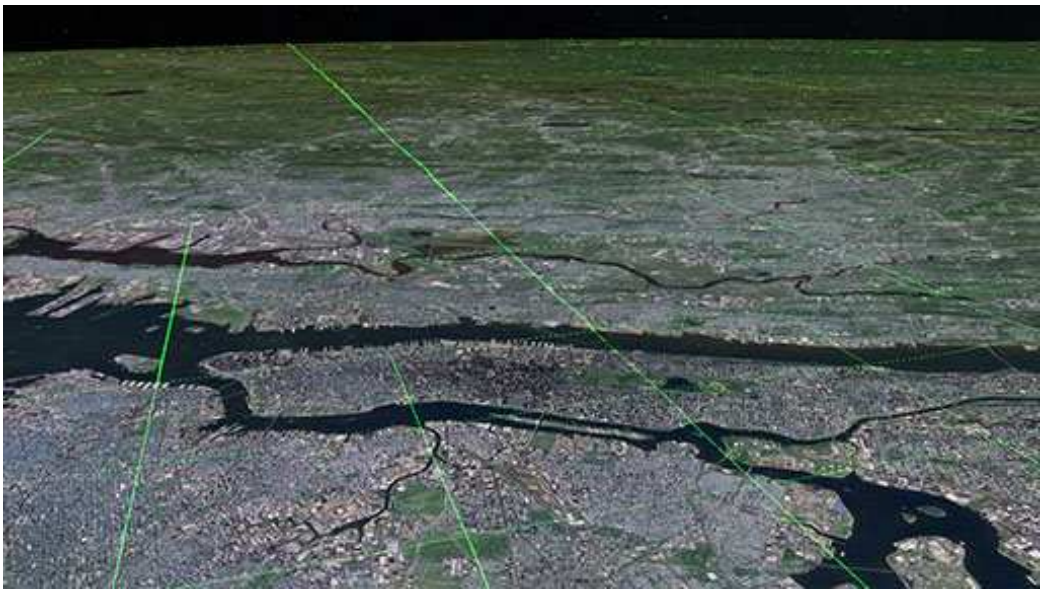
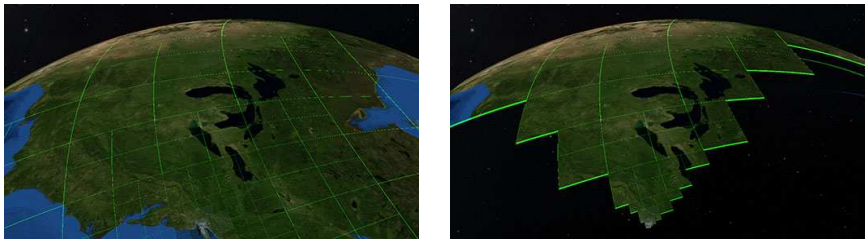
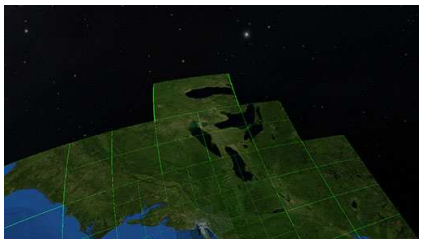


Figure 4.13: Chunks yielded by the distance based chunk selection algorithm. Brooklyn, Manhattan and New Jersey is seen in the camera view.

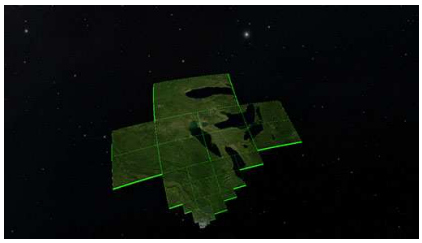


(a) No culling

(b) Frustum culling



(c) Horizon culling



(d) Frustum and horizon culling

Figure 4.14: Culling of chunks with distance based chunk selection

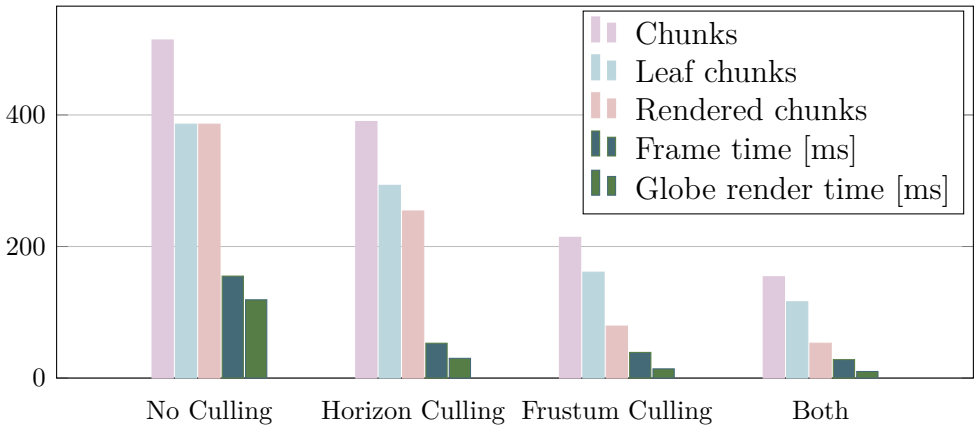


Figure 4.15: The number of chunks effected by culling

4.2.3 Culling for Area Based Chunk Selection

The two culling algorithms frustum culling and horizon culling were evaluated in combination with the area based chunk selection algorithm. The settings used are provided in table 4.4. Figure 4.16 shows the camera view evaluated. Figure 4.17 shows an overview of the rendered chunks with the results shown in figure 4.18.

Table 4.4: Globe rendering Settings for evaluation of culling with area based chunk selection

Globe:	Earth
Map datasets:	HeightLayers = [GCS_Elevation [44]]
	ColorLayers = [ESRI Imagery World 2D [28]]
LOD Scale factor:	7.8
Chunk Selection:	By projected area
Culling:	**Evaluated**
Level blending:	Enabled
Camera View:	Looking towards western horizon
Location:	New York, New Jersey

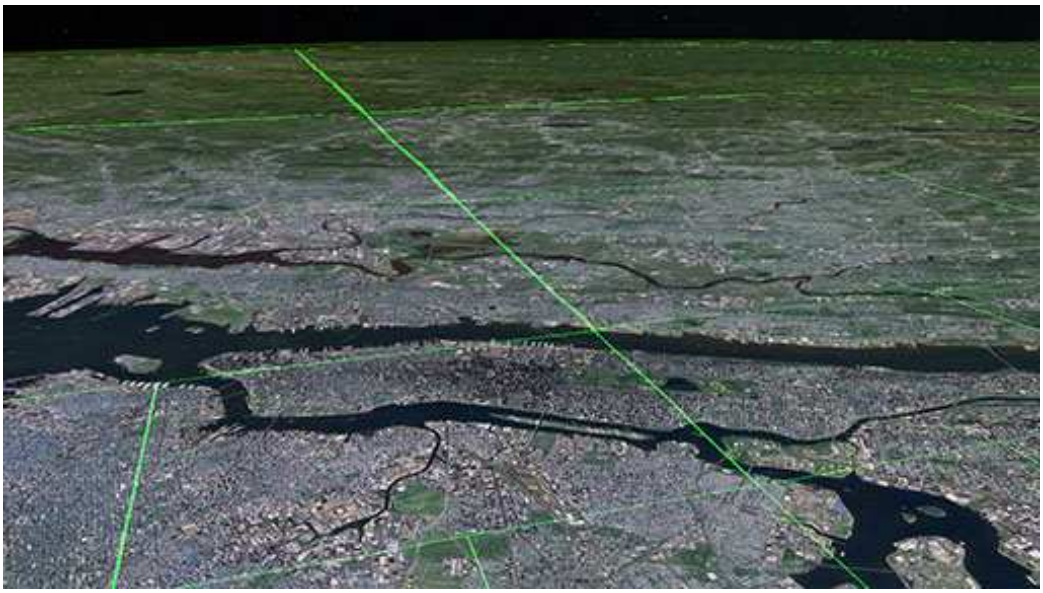
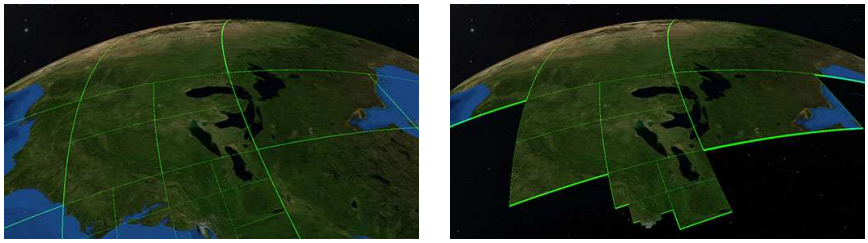


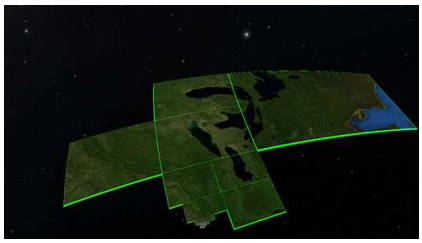
Figure 4.16: Chunks yielded by the projected area based chunk selection algorithm. Brooklyn, Manhattan and New Jersey is seen in the camera view.



(a) No culling

(b) Frustum culling

(c) Horizon culling



(d) Frustum and Horizon culling

Figure 4.17: Culling of chunks with area based chunk selection

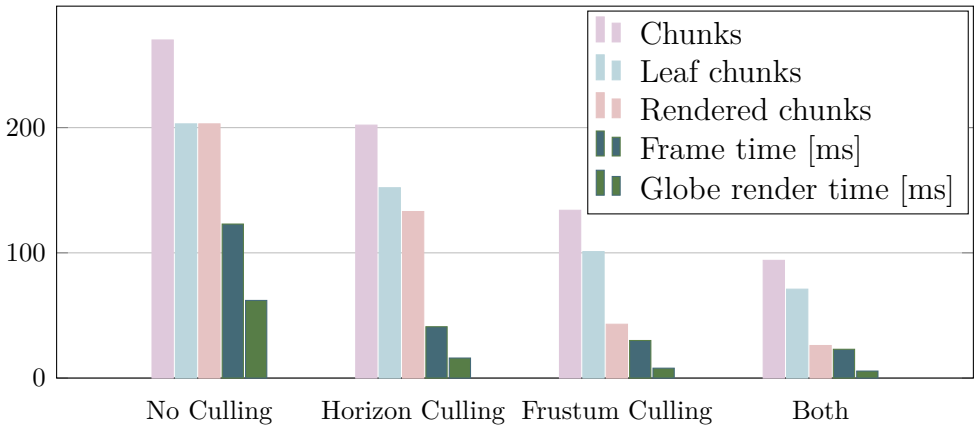


Figure 4.18: The number of chunks effected by culling

4.2.4 LOD: Distance Based vs. Area Based

The benchmark results in section 4.2.2 and 4.2.3 were compared and evaluated in relation to each other. In figure 4.19 the two approaches are compared where both frustum culling and horizon culling were enabled.

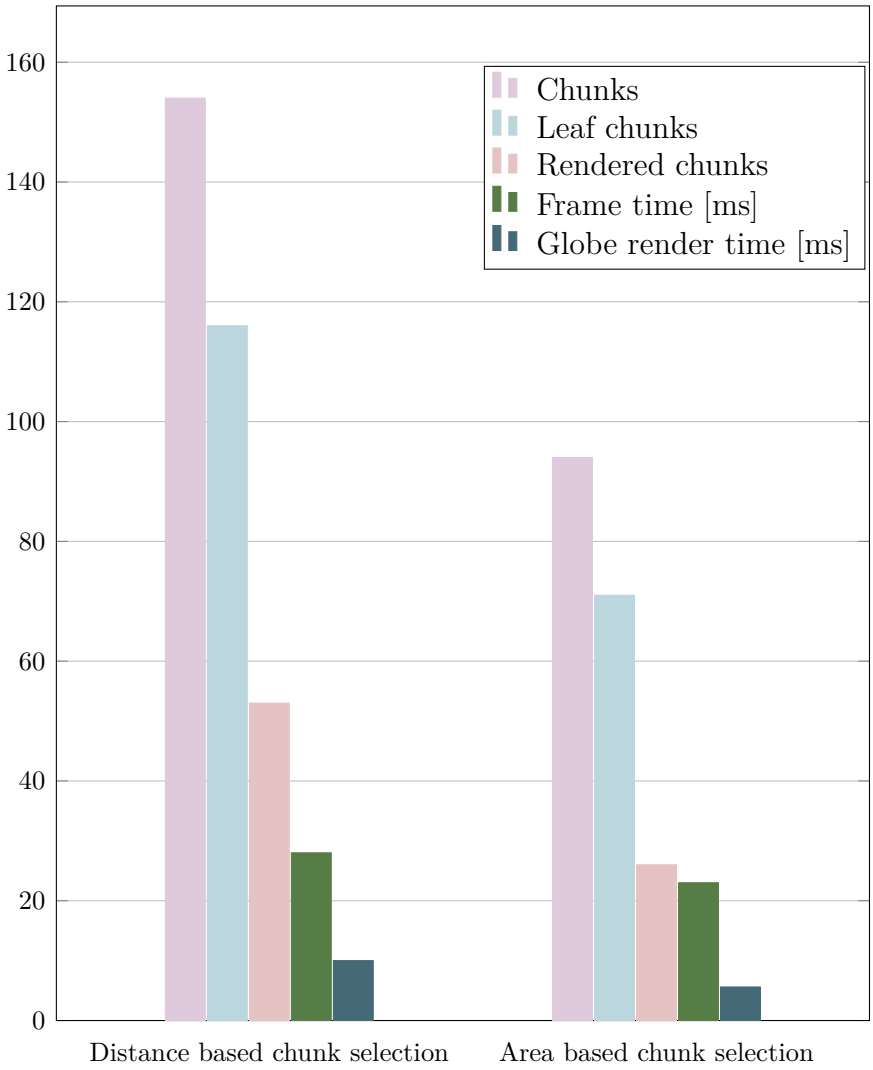
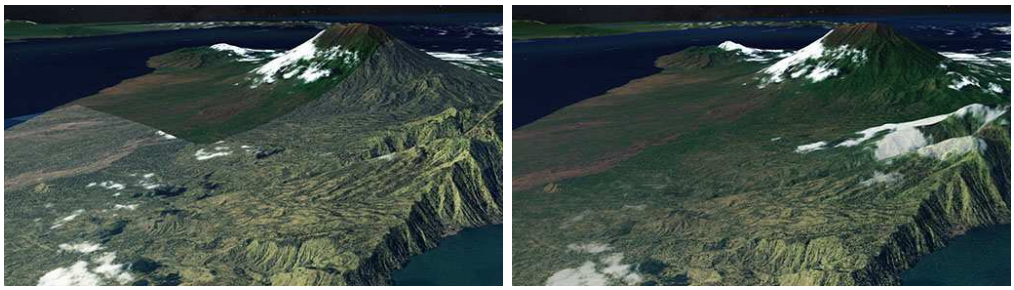


Figure 4.19: Comparison of distance based and area based chunk selection

4.2.5 Switching Using Level Blending

The visual result of using the distance based level blending is shown in figure 4.20.



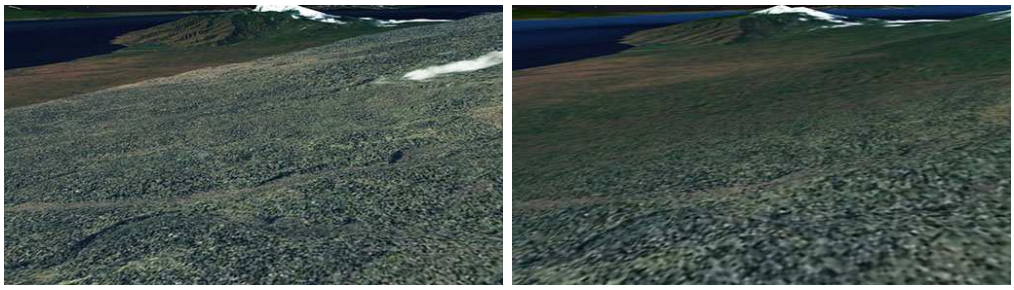
(a) Blending disabled (b) Blending enabled

Figure 4.20: Comparison of using level blending and no blending. Level blending hides edges the underlying chunks

Table 4.5 shows the settings used to compare having level blending enabled and disabled. To show the penalty in texture detail that is paid for using level blending, the LOD scale factor was set low. Figure 4.21 shows that the texture resolution becomes low enough to see the individual pixels when having a low LOD. The results in table 4.6 also show that figure 4.21a contains more visual information than 4.21b.

Table 4.5: Globe rendering settings used in benchmark

Globe:	Earth
Map datasets:	HeightLayers = [GCS_Elevation [44]]
	ColorLayers = [ESRI Imagery World 2D [28]]
LOD Scale factor:	4.65
Chunk Selection:	By distance
Culling:	Frustum culling, Horizon culling
Camera View:	Horizon



(a) Blending disabled (b) Blending enabled

Figure 4.21: Comparison of level blending and no blending. The LOD scale factor is set low to show the resolution penalty of using blending

Table 4.6: Level blending benchmarks. Comparing the disk space of a compressed JPG image of both cases works as a heuristic to the information contained in each image

Figure 4.21	**No blending**	**Level blending**
Samples per fragment	1	3
Mean globe render time	10.3 ms	10.1 ms
Mean frame time	40 ms	49 ms
Screenshot jpg size	1,1 MB	0,7 MB

4.2.6 Polar Pinching

The impact of polar pinching on the chunk tree was evaluated using both distance based chunk selection and area based chunk selection. The results are presented in figure 4.22.

Table 4.7: Globe rendering settings for evaluation of polar pinching

Globe:	Earth
Map datasets:	HeightLayers = [GCS_Elevation[44]]
	ColorLayers = [ESRI Imagery World 2D [28]]
LOD Scale factor:	10.0
Chunk Selection:	**Evaluated**
Culling:	Frustum culling, horizon culling
Camera View:	Facing down
Location:	Equator (at Quito, Ecuador) and North Pole

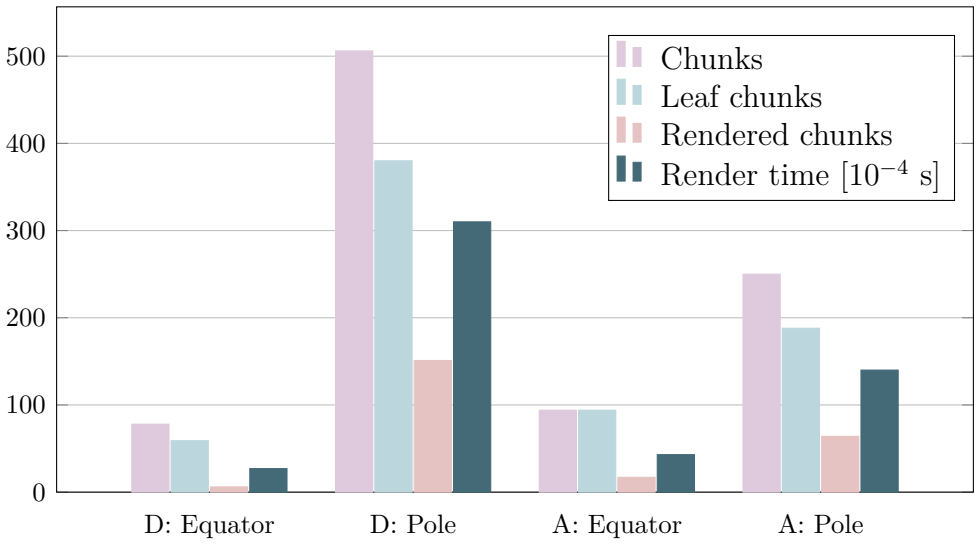


Figure 4.22: Comparison of distance based and area based chunk selection at the Equator and the North Pole. **D** = distance based, **A** = area based

4.2.7 Benchmark: Interactive Globe Browsing

The purpose of the globe browsing is to be able to interactively explore virtual globes and map datasets. In order to capture how the system behaves over time, a user interaction sequence was evaluated. The sequence is described step by step below and illustrated in Figure 4.23.

1. Camera views Earth from space
2. Descend down to Naturpark Karwendel, north of Innsbruck, Austria
3. Tilt camera up, view northern horizon
4. Turn camera 180 degrees, view southern horizon
5. Tilt camera down

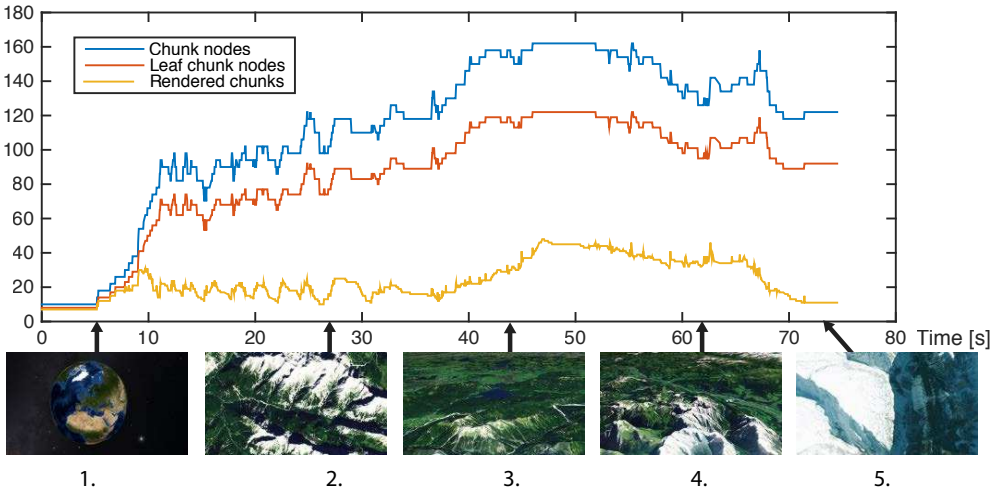
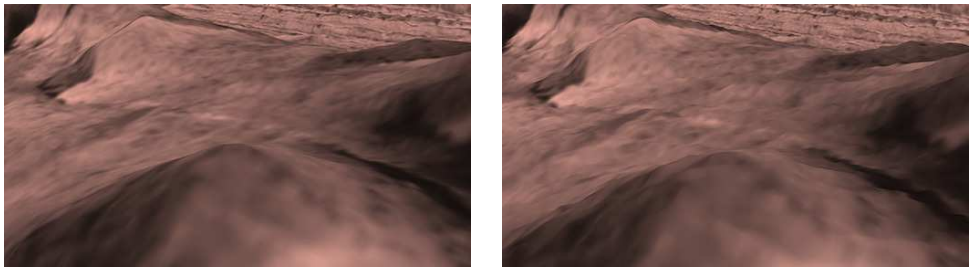


Figure 4.23: Chunk tree over time when browsing the globe

4.2.8 Camera Space Rendering

Comparing the use of camera space rendering and model space rendering, vertex jittering is apparent when interacting close to the surface. Figure 4.24 shows how vertex jittering is apparent when browsing a HiRISE patch close to the surface of Mars. The visual result in the image is not as striking as the artifact that appears when the camera is moving. Then the vertices are jittering only in the case of model space rendering and not for camera space rendering.



(a) Camera space rendering (b) Model space rendering

Figure 4.24: Vertex jittering of model space rendering

Chapter 5

Discussion

In this chapter, the resulting implementation and design decisions made throughout the research phase and development phase will be discussed along with the results presented in chapter 4.

5.1 Chunked LOD

The performance related results of the benchmarks presented in section 4.2 are discussed below.

5.1.1 Chunk Culling

From Figure 4.11 and 4.23 it is noted that the chunk tree grows as the ground facing camera comes closer to the surface, but the number of rendered chunks remain relatively constant. This is the desired result of the frustum culling algorithm, which efficiently culls most of the chunk tree as the camera looks straight down towards the ground. In Figure 4.11 we can also see that the total globe rendering time does not only depend on the number of rendered chunks. Even though the render call done for each rendered chunk is quite heavy (including accessing texture data and setting uniforms on the GPU), other things such as chunk selection and performing culling (done for each leaf node) are operations that also affect the render time.

In figure 4.15, the culling algorithms are evaluated and compared for a camera view looking at the horizon. Looking at the horizon means a much larger geographical area is visible to the camera compared to looking straight down towards the ground as in the case discussed above. This means that chunk culling algorithms can not be expected to increase the rendering performance as much as in the case above. However, it can be seen that

by using both the culling approaches, the globe rendering time is reduced approximately by a factor 10. When comparing the two, we see that the frustum culling algorithm does a much better job reducing the number of rendered chunks than than horizon culling. This is intuitively explained by the fact that horizon culling can only cull chunks far away, where the selection algorithm prefers choosing a few big, low resolution chunks any way. The frustum culling algorithm, on the other hand, may cull chunks that are both far away and very close to the camera. However, there is no practical reason for not using both the algorithms in combination at any time. The overhead of running both algorithms is paid back in rendering time as seen in figure 4.15.

5.1.2 Chunk Selection

In order to efficiently render camera views including facing the horizon, parameter tuning and optimization should not focus on chunk culling.

Instead, the selection algorithm need to be carefully considered. In figure 4.19, the two implemented chunk selection algorithms are compared to each other for such a camera view. We see that the algorithm based on the projected area is significantly faster than the distance based approach. The explanation for this is found by looking at the figures 4.13 and 4.16 showing the corresponding view for the distance based and projected area based algorithm respectively. The projected area based algorithm prioritize high level of detail close to the camera to a larger extent than the distance based approach, which is also how the algorithm gains its performance. By looking at the closest rendered chunks however, we see that the same level of detail was selected for both algorithms. Considering the relatively low difference in the visual appearance, one conclusion is that the projected area based algorithm is an efficient way to select chunks for rendering, especially for camera views looking at the horizon. Visual penalties will mainly be apparent when high detail is desired at distances far from the camera. Such cases can be when tall mountains are visible near the horizon.

5.1.3 Chunk Switching

The mipmapped texture data within a single dataset does not have to be strictly down sampled versions of the original size. The dataset may be combined from multiple different data sources such as satellite imagery and aerial photographs. Thus, two adjacent mip levels within a map dataset may look very different from one another. This is one motivation for using level blending as a main approach for chunk switching.

As seen in the results of section 4.2.5, the level blending removes the otherwise visible edges between chunks of different level. However, the penalty for using level blending is not directly the increase in render time (which is insignificant) but rather the loss of image quality, as seen in table 4.6. Enabling the level blending algorithm for this specific case reduces the JPG compressed version of the screenshot from 1,1 Mb to 0,7 Mb; that is a reduction of 37%. In other words, using level blending can be seen as trade off between poorer texture resolution versus popping artifacts and visible edges. In the end it all boils down to a tradeoff in render time as the LOD scale factor can be increased to compensate the overall loss in quality for the rendered globe. The quality per rendered chunk, however, is undoubtedly reduced notably.

As mentioned in section 5.1.2, the projected area based selection approach was concluded to be more efficient than the distance based. However, using the area based approach comes with the drawback of not being able to correctly perform distance based chunk switching using level blending on a per-fragment basis. The convenience of using a simple distance based chunk selection algorithm is that the exact same calculations can be used in a fragment shader to calculate the interpolation parameter for the different mip levels of the texture datasets. Calculating a correct interpolation parameter corresponding to the projected area based chunk selection algorithm is a harder problem, not considered in the scope of this thesis. Conceptually, one approach could be to use bilinear interpolation based on four blending control points in each corner of the chunk. These values, however, would inherently depend on the LOD level of neighboring chunks, which makes the problem non-trivial.

5.1.4 Inconsistent Globe Browsing Performance

Using quad trees (chunk trees) is a simple approach for hierarchically dividing a region into smaller subregions. However, the way the chunk trees are structured may cause different geographical regions to correspond to different types of chunk tree structures. That is, two different chunks on the same chunk level can correspond to large differences in the total number of nodes in the chunk tree. As an example: if the camera is looking at a small region just north of the equator, the whole southern hemisphere can be efficiently culled. However, if the camera moves just a few meters south, the chunk tree needs to suddenly consider a lot more chunks than previously. This effect will cause the performance of the rendering system to be affected by location. This would explain the same number of chunk nodes in figure 4.12 for East America and New York.

From Figure 4.23 we can see that the chunk tree grows as the camera views the horizon, as do the number of rendered chunks. When turning the camera 180 degrees, the chunk tree shrinks temporarily. This is explained by the fact that the chunk tree growth is limited by the available tile data. Rotating half a revolution causes a lot of previously unseen chunks to no longer be culled by the culling algorithms. The tree grows again as the map tiles for these chunks are being fetched.

5.2 Ellipsoids vs Spheres

Representing globes as ellipsoids and not as spheres was a decision settled early in the development process. The accuracy that the ellipsoidal model gives is important to be able to show near Earth orbits and rocket launches where the space crafts need to be positioned accurately relative to the WGS84 reference frame which is used in the SPICE library. Other globes such as the planet Saturn has a clearly visible ellipsoidal shape due to its angular momentum. This can now be shown in OpenSpace by configuring the planet to have the correct radii.

5.3 Tessellation and Projection

The choice of using geographic tessellation for the ellipsoid was a direct result of using equirectangular geodetic map projections. The equirectangular geodetic map projection is very common for many public map datasets and was therefore a given choice to be able to visualize a vast amount of data on the globes surfaces.

The drawbacks with oversampling at the poles giving undesired visual artifacts was considered, but in the end prioritized down due to time restrictions. However, due to the fact that the issue of polar pinching was not handled explicitly, the resulting issue of performance drops close to the poles can be seen in section 4.2.6.

The most reasonable solution to the polar issues would be to use polar caps similar to Dimitrijević and Rančić's solution [24]. Even though the LOD algorithm is different, the similarities in tessellation and map projection would mean that the implementation would have similar characteristics. Polar caps will however not solve all problems related to poles in geographical grid tessellations. The first time the software wants to render chunks on a polar cap near the pole, it will still need to request all the same tiles that a it would using a standard geographical grid tessellation. If it later

is able to cache all reprojected tiles, they will be faster to render next time they are requested. The requests together with the re-projections will make it slower to visualize tiles at the caps, this could be a big limitation for datasets that needs to be accessed fast such as temporal datasets used for animating textures. Many of the GIBS datasets however are available in both equirectangular format as well as polar stereographic (EPSG:3413) for the North Pole and Antarctic polar stereographic (EPSG:3031) for the South Pole. Hence re-projection would not be needed if the polar caps are defined with stereographic projections.

5.4 Chunked LOD vs Ellipsoidal Clipmaps

Both Ellipsoidal clipmaps and chunked LOD were thoroughly considered as the overall LOD algorithm to used. Both methods were implemented as proofs of concepts before deciding on using the chunked LOD approach.

The clipmap implementation was aborted as the chunked LOD was significantly more straight forward to implement and early on started yielding visible results. Moreover, having decided on using equirectangular tessellation and map projection, the ellipsoidal clipmap approach would completely depend on an working implementation of polar caps, which would have extended the implementation time further with the need of considering both re-projection and edges.

Even though geometry clipmaps would pose some very different rendering challenges than the chunked LOD approach, many components developed for the chunked LOD approach could be shared between the two approaches. The commonly used components would include the entire texture data pipeline and layer structure along with all the geometric related calculations, the interaction mode and various helper classes. The main difference is the rendering pipeline even though it also has similarities; for example, using the model space and camera space vertex rendering schemes could be done for geometry clipmaps just as well as for chunk grids.

5.5 Parallel Tile Requests

Requesting tiles from a given remote dataset could in theory be performed in parallel but GDAL puts some restrictions on open datasets. GDAL can internally request tiles in parallel when performing the RasterIO operation. However, GDAL does not guarantee thread safe reading within all its formats. Reading from one open dataset on several threads in parallel often results

in corrupt tiles. If the parallelization is limited to be performed on separate overviews however, GDAL's data writing and caching would hopefully not lead to race conditions between threads since overviews are completely separated from each other. Another method to speed up GDAL requests could be to open several GDAL dataset per tile dataset. However this will lead to internal cache misses within GDAL that most likely results in even slower requests and unnecessary memory usage.

5.6 High Resolution Local Patches

The software was implemented to be able to handle highly detailed datasets down to 25 centimeters per pixel as shown in section 4.1.6 rendering HiRISE patches. Using the combination of HiRISE patches, CTX patches and a global terrain model with textures, the detail level together with the vast scale of surrounding canyon walls resulted in a combined view as close as possible to the real landscapes of Mars.

As described under appendix B however, reading local patches used to render in OpenSpace requires extensive amount of preprocessing. Another issue is that a very sparse dataset specified with a virtual dataset (described in appendix B) currently poses on OpenSpace is that empty tiles outside of the defined region are still initialized which makes the software run slower. The preprocessing and the unnecessary initialization makes reading and rendering of local patches more of a proof of concept and will be left for further development as future work.

Chapter 6

Conclusions

From the results and experimentation of this globe rendering system, the following conclusions have been drawn:

- Chunked LOD is the most straight forward approach for out-of-core rendering of large scale map datasets where accuracy is important.
- Using a combination of frustum culling and horizon culling, the rendering time can be kept relatively constant with respect to distance to the ground when camera is facing straight down.
- A chunk selection algorithm based on the chunks projected area is more efficient than one based on only the chunk's distance to the camera. The projected area based approach is better in terms of resulting visual appeal versus rendering performance.
- Distance based level blending can be done on the fragment shader to significantly reduce the appearance of chunk edges successfully when used in combination with distance based chunk selection. However, in order to perform accurate level blending using a area based chunk selection algorithm, another approach for the fragment shader must be considered.
- The number of chunk nodes in the tree depends not only on the chunk selection algorithm per se, but also on the geographic location of the camera due to the underlying quad tree.

Chapter 7

Future Work

We will discuss the most relevant features that could be added to the globe browsing module in the future. There are possibilities for optimizations as well as a need for features, required from a fully versatile globe browsing system, that easily extends beyond the scope of this thesis.

7.1 Parallelizing GDAL Requests

Given the discussion about parallelizing of GDAL WMS requests under section 5.5, testing parallelizing on a per overview level would be reasonable to maximize efficiency in tile requests. The best thing, of course, would be if the developers of GDAL made sure that WMS requests would be thread safe.

7.2 Browsing WMS Datasets Upon Request

The focus was not put on user interfacing, but instead the techniques required for rendering. A given future feature however would be to provide the users of the software the ability to specify datasets to read during runtime. This could be done using the GetCapabilities operation for WMS or WMTS providers to be able to list all available datasets behind a service. Then users could select a specific dataset to load on demand.

7.3 Integrating Atmosphere Rendering

Integrating a sophisticated atmosphere rendering technique for globe browsing would make the complete experience more rewarding. Using atmospheric

parameters defining phenomena such as Rayleigh scattering and Mie scattering estimated for globes around our solar system, the representations of other worlds would be more accurate. With the implemented system, these estimations could be based on public, real world weather data. Atmospheres also works great for generating depth cues which enhances the illusion of true scale within the visualization of globes.

7.4 Local Patches and Rover Terrains

The ability to read and render local patches will be developed further to ensure easy integration of geographically smaller datasets. A goal in the future is to be able to read and render even higher resolution imagery than demonstrated in this thesis. Rendering Mars rover terrain models, on top of HiRISE patches, on top of CTX patches, on top of a global terrain datasets would transport audiences and users of the software even closer to the real surface of Mars with all the detail in imagery currently available through research of the planet's geology.

7.5 Other Features

There are other features that naturally can be incorporated to make the software more usable.

A switching algorithm implemented on the shader corresponding to the area based chunk selection algorithm can be researched and explored further.

Another switching approach than the level blending approach introduced can be to temporally interpolate between levels as chunks are split. This will decrease the number of textures needed for switching but will still lead to visible edges between chunks of different levels.

The skirt lengths can be optimized to decrease the number of fragments rendered. This can be done if each chunk has knowledge of the adjacent chunks height data at the edges. To avoid potential cracks that might appear when a chunk of high level is adjacent to a chunk of low level, skirt lengths can be set to be proportional to the geodetic size of the available height map and not the chunk itself.

Annotations such as country names can currently only be rendered using pixel based layers. Introducing vector formats for layers would make it possible to display annotations such as text without the discrete changes of different levels.

Other types of tile providers can also be implemented. Examples are:

text tile provider for visualizing a lat-long grid, temporal tile provider using local image data.

7.6 Other Uses of Chunked LOD Spheres in Astro-Visualization

Rendering of ellipsoidal or spherical models is not only useful for globes. Being able to browse the night sky using an inverted sphere also requires a LOD approach to be able to visualize the most distant objects in high level of detail. World Wide Telescope uses such a technique to render photographs of distant galaxies when pointing the camera to the sky. The edge of our observable Universe could also be rendered as a sphere textured with a mapping of the cosmic microwave background radiation.

References

[1] *OpenSpace*,
http://openspaceproject.com/, (accessed 2016-11-16)

[2] Patrick Cozzi and Kevin Ring, *3D Engine Design for Virtual Globes*, CRC Press, 1st edition, 2011.
http://www.virtualglobebook.com (accessed 2016-08-03)

[3] *Google Maps*,
http://maps.google.se, (accessed 2016-08-17)

[4] National Oceanic and Atmospheric Administration, *Science On a Sphere*,
http://sos.noaa.gov/, (accessed 2016-08-17)

[5] NASA's Navigation and Ancillary Information Facility, *An Overview of SPICE, NASA's Ancillary Data System for Planetary Missions*, 2016.
http://naif.jpl.nasa.gov/pub/naif/toolkit_docs/Tutorials/pdf/individual_docs/03_spice_overview.pdf, (accessed 2016-11-16)

[6] SCISS AB, *Uniview*,
http://sciss.se/uniview, (accessed 2016-08-17)

[7] World Wide Telescope, *American Astronomical Society*,
http://www.worldwidetelescope.org, (accessed 2016-08-17)

[8] Evans & Sutherland, *Digistar*,
http://www.es.com/Digistar, (accessed 2016-08-17)

[9] Sky-Skan, *DigitalSky*,
https://www.skyskan.com/products/ds, (accessed 2016-08-17)

[10] Brano Kemen and Laco Hrabcak, *Outerra Website*,
http://www.outerra.com, (accessed 2016-08-17)

[11] Vladimir Romanyuk, *Space Engine Website*, http://en.spaceengine.org, (accessed 2016-08-17)

[12] Sasha Hinkley, *Directly Imaging Exoplanets*, Astrophysics Group, University of Exeter, 2016 https://palereddot.org/directly-imaging-exoplanets/, (accessed 2016-08-18)

[13] *Map Projection Images*, https://en.wikipedia.org/wiki/List_of_map_projections, (accessed 2016-11-25)

[14] Open Geospatial Consortium Inc, *OpenGIS Web Map Server Implementation Specification*, 2006.

[15] Open Source Geospatial Foundation, *Tile Map Service Specification*, https://wiki.osgeo.org/wiki/Tile_Map_Service_Specification, (accessed 2016-08-17)

[16] Open Geospatial Consortium Inc, *OpenGIS Web Map Tile Service Implementation Standard*, 2010.

[17] Global Imagery Browse Services *GIBS Available Imagery Products*, https://wiki.earthdata.nasa.gov/display/GIBS/GIBS+Available+Imagery+Products, (accessed 2016-11-27)

[18] ESRI, *Arcgis Online*, https://www.arcgis.com, (accessed 2016-08-17)

[19] U.S. Geological Survey, *Mars Viking MDIM21 ClrMosaic global 232m*, http://astrogeology.usgs.gov/search/map/Mars/Viking/MDIM21/Mars_Viking_MDIM21_ClrMosaic_global_232m, (accessed 2016-11-23)

[20] *Online Map Viewer for MRO CTX Patches*, http://viewer.mars.asu.edu/viewer/ctx#T=0, (accessed 2016-11-23)

[21] Lunar & Planetary Laboratory *HiRISE Digital Terrain Models*, http://www.uahirise.org//dtm/, (accessed 2016-11-23)

[22] NASA Jet Propulsion Laboratory, *Onmoon Web Interface*, California Institute of Technology http://onmoon.jpl.nasa.gov, (accessed 2016-08-17)

[23] Isaac Newton, *Philosophiae naturalis principia mathematica*, J. Societatis Regiae ac Typis J. Streater, University of California, 2011. https://books.google.se/books?id=-dVKAQAAIAAJ, (accessed 2016-11-11)

[24] Aleksandar M. Dimitrijević and Dejan D. Rančić *Ellipsoidal Clipmaps - A planet-sized terrain rendering algorithm*, In Proceedings of the Eighth Joint Eurographics / IEEE VGTC conference on Visualization (EUROVIS'06), Comput. Graph. 52, C (November 2015), 43-61, 2015. doi: http://dx.doi.org/10.1016/j.cag.2015.06.006, (accessed 2016-11-01)

[25] Hierarchical Triangular Mesh, *Sloan Digital Sky Survey*, 2007. http://www.skyserver.org/HTM/ (accessed 2016-11-12)

[26] European Petroleum Survey Group, http://www.epsg.org, (accessed 2016-11-11)

[27] Spatial reference list, http://spatialreference.org/ref/epsg/, (accessed 2016-08-10)

[28] ESRI, ArcGIS Online, *ESRI Imagery World 2D WGS84*, https://www.arcgis.com/home/item.html?id=37b8a8b1014747deb5e15472d06d0da9, (accessed 2016-11-25)

[29] ESRI, ArcGIS Online, *World Imagery*, https://www.arcgis.com/home/item.html?id=10df2279f9684e4a9f6a7f08febac2a9, (accessed 2016-11-25)

[30] Sarah E. Battersby, Michael P. Finn, E. Lynn Usery and, Kristina H. Yamamoto, *Implications of Web Mercator and Its Use in Online Mapping*, Cartographica 49:2, 2014, pp. 85-101, 2014. doi: http://dx.doi.org/10.3138/carto.49.2.2313, (accessed 2016-11-11)

[31] Jonathan Fay, *WorldWide Telescope Projection Reference*, http://www.worldwidetelescope.org/docs/worldwidetelescopeprojectionreference.html, (accessed 2016-11-16)

[32] NASA, Jet Propulsion Laboratory, *HEALPix*, http://healpix.jpl.nasa.gov/index.shtml, (accessed 2016-11-23)

[33] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. *ROAMing Terrain: Real- Time Optimally Adapting Meshes*, In Proceedings of the 8th

Conference on Visualization #97, pp. 81-88. Los Alamitos, CA: IEEE Computer Society, 1997,

[34] Brano Kemen, *Procedural terrain algorithm visualization*, 2009, `http://outerra.blogspot.se/2009/02/procedural-terrain-algorithm.html` (accessed 2016-11-16)

[35] Terragen - Planetside Software, *Terragen Wiki - Procedural Data*, 2013, `http://planetside.co.uk/wiki/index.php?title=Procedural_Data`, (accessed 2016-11-16)

[36] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones, *The clipmap: a virtual mipmap*, In Proceedings of the 25th annual conference on Computer graphics and interactive techniques (SIGGRAPH '98), New York, NY, USA, 151-158, 1998.
doi: http://dx.doi.org/10.1145/280814.280855, (accessed 2016-11-01)

[37] Frank Losasso and Hugues Hoppe *Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids*, ACM Trans. Graphics (SIGGRAPH), 23(3), 2004.

[38] Malte Clasen and Hans-Christian Hege, *Terrain rendering using spherical clipmaps*, In Proceedings of the Eighth Joint Eurographics / IEEE VGTC conference on Visualization (EUROVIS'06), Aire-la-Ville, Switzerland, 2006.
doi: http://dx.doi.org/10.2312/VisSym/EuroVis06/091-098, (accessed 2016-11-16)

[39] Mark Segal and Kurt Akeley, *The OpenGL Graphics System: A Specification (Version 4.0 (Core Profile) - March 11, 2010)*, CRC Press, 1st edition, 2010.

[40] The Institute of Electrical and Electronics Engineers *IEEE Standard for Binary Floating-Point Arithmetic*, The Institute of Electrical and Electronics Engineers, Inc, New York, 1985.
`http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=30711`, (accessed 2016-11-18)

[41] Brano Kemen, *Maximizing Depth Buffer Range and Precision*, Article for the Outerra software, 2012.
`http://outerra.blogspot.se/2012/11/maximizing-depth-buffer-range-and.html` (accessed 2016-11-14)

[42] The Open Source Geospatial Foundation, *GDAL - Geospatial Data Abstraction Library*, GDAL Website,
`http://www.gdal.org`, (accessed 2016-11-16)

[43] *The General Helpful Open Utility Library*,
`https://github.com/OpenSpace/Ghoul`, (accessed 2016-11-23)

[44] Lucian Plesea *ESRI World Elevation 3D Equirectangular Reprojection*,
`http://198.102.45.23/arcgis/rest/services/worldelevation3d/terrain3d?request=GetCapabilities`, (accessed 2016-11-25)

[45] US Geological Survey *Mars Orbiter Laser Altimeter (MOLA) DTM*,
`http://astrogeology.usgs.gov/search/map/Mars/GlobalSurveyor/MOLA/Mars_MGS_MOLA_DEM_mosaic_global_463m`,
(accessed 2016-12-12)

Appendices

Appendix A

General Use of Globe Browsing in OpenSpace

https://github.com/OpenSpace/OpenSpace/wiki/Concepts-Modules-GlobeBrowsing-GeneralUse

Appendix B

Build Local DEM Patches to Load With OpenSpace

https://github.com/OpenSpace/OpenSpace/wiki/Concepts-Modules-GlobeBrowsing-BuildLocalDEMPatchestoLoadWithOpenSpace