LIU-ITN-TEK-A--15/018--SE

Adaptive rendering of celestial bodies in WebGL

Jonas Zeitler

2015-06-04

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--15/018--SE

Adaptive rendering of celestial bodies in WebGL

Examensarbete utfört i Datateknik
vid Tekniska högskolan vid
Linköpings universitet

Jonas Zeitler

Handledare Stefan Gustavson
Examinator Patric Ljung

Norrköping 2015-06-04

LINKÖPING UNIVERSITY

Abstract

Institute of Technology
Department of Science and Technology

Master of Science

by Jonas Zeitler

This report covers theory and comparison of techniques for rendering massive scale 3D geospatial planet data in a web browser. It also presents implementation details of a few of these techniques in WebGL and Javascript, using the Three.js [1] 3D library.

The thesis project is part of the implementation of Unitea, a web based education platform for interactive astronomy visualizations. Unitea is a derivative of Uniview, which is a fulldome interactive simulation of the universe. A major part of this thesis is dedicated to the implementation of Hierarchical Level of Detail (HLOD) modules for Three.js based on the theory presented by T. Ulrich [2] and later generalized by Cozzi and Ring [3]. HLOD techniques are dynamic level of detail algorithms that represent the surface of objects as accurately as possible from a certain viewing angle. By using space partitioning tree-structures, view based error metrics and culling techniques detailed representations of the objects (in this case planets) can be efficiently rendered in real-time.

The modules developed provide a general-purpose library for rendering planets (or other spherical objects) with dynamic level of detail in Three.js. The library also features connections to online web map services (WMS) and tile services.

Acknowledgements

I would like to thank the entire team at Sciss for giving me the opportunity to contribute to their product and work in a creative environment for cutting edge computer graphics and scientific visualization. I am especially thankful for all the help and guidance received from Urban Lassi, Daniel Hultgren and, of course, Staffan Klashed. I would also like to thank my supervisor Stefan Gustavson and my examiner Patric Ljung for being both supportive and genuinely excited about my thesis work.

Jonas - Stockholm, April 2015

Contents

List of Figures

Chapter 1

Introduction

This first chapter covers the motivation and provides an overview of the thesis project. Objectives and research questions are presented here. A brief introduction to the thesis structure is also outlined below.

1.1 The need for planet rendering on the web

Ever since the now semi-recent internet boom of the late 90's and early 2000's, an ever growing collection of data is available to everyone connected to the web. The question is no longer *if* data is available, but rather *how* to make sense of it. Computer graphics is a powerful tool for visualizing large data sets and it is therefore a natural fit to take computer graphics and applying it in a web browser.

The reason for specifically rendering planets and other astronomic data is to better perceive the *spatial and temporal relations* of the visualized data. This may sound natural, especially for astronomy, but is surprisingly often disregarded. The top ten facts people learn about the universe are probably related to scale or time. For example; The universe is very large (≈ 150 billion light years in diameter), the universe was very hot and very dense in the beginning and it is expanding at an ever accelerating rate. However, things like these are almost always at an inconceivable scale to us. Our only spatio-temporal references are the ones that we experience here on Earth. This is why visualization of astronomic data is such an interesting field. Even though still images may possibly hint about how to perceive the statements above, animations and interactive visualization provide a richer experience and a far deeper understanding for both the spatial and temporal relations of astronomy.

Providing a platform that enables people to explore and understand the universe is a big step

forward from static images and large numbers. Google Earth is one example of this. With more and more data from telescopes and satellites it may be possible to develop similar applications, but this time for the entire universe.

1.2 Problem description

Visualizing geographical terrain and imagery data with dynamic level of detail can be done using several different techniques. Some general (e.g. Ulrich [2]) and some more specific (e.g. Asirvatham and Hoppe [10]). Rendering planets implies handling massive amounts of data efficiently and to selectively render only parts of the data sets. This requires optimized program execution paths and robust data structures to handle very large loads.

1.3 Objectives

The objective of this thesis is to present an efficient way of rendering planets in a web browser. This includes generation of geometry/meshes, texture requests from remote tile servers and optimizations for low performance devices. The implementation is presented as general modules for the Three.js WebGL framework and will be integrated as part of the Unitea astronomy visualization toolkit.

1.3.1 Performance Goals

Since the implementation is targeted towards low performance devices, two simple performance goals have been set, one for desktops/laptops and one for mobile devices:

- Desktop/Laptop: A scene with one planet should run at steady (capped) 60 frames per second in all supported browsers (Chrome 41.0+, Safari 8.0+, Opera 28.0+, Firefox 36.0+) regardless of user interaction pattern and camera view. The minor hardware configuration supported for this performance benchmark is 2,5 GHz Intel Core i7 processor with Nvidia GeForce 750M GPU. The highest supported viewport resolution is 1920x1080 (including retina screens reporting to be that size).
- Mobile: A scene with one planet should run at steady 30 frames per second in all supported browsers (iOS/Android Chrome 41.0+, iOS Safari 8.0+) regardless of user interaction pattern and camera view. The iPad3 is used as threshold device, since it has been known to lack fill rate and runs a weak GPU compared to its retina screen.

Furthermore, a qualitative goal is that the above performance is maintained with geometry/texture detail appropriate for the target device/screen.

1.4 Structure of the thesis

The thesis starts with a background chapter, covering the basic ideas and challenges of planet rendering and general astronomy visualization. The background also describes the current state of the graphics programming API for web browsers and gives a brief introduction to the Unitea project. Specific techniques for terrain rendering, geometry tessellation and data management are described in the *Related work* chapter. The *Implementation* chapter then proceeds to describe details about the development project that is the foundation of this thesis. Results are presented in chapter 5 as measurements and comparisons regarding the execution of the implemented modules. The last two chapters, chapter 6 and chapter 7, summarizes, evaluates and concludes the thesis with remarks on performance and visual quality and pointers as to what could have been done differently.

Chapter 2

Background

2.1 WebGL

Web Graphics Library (WebGL) is an API for rendering 2D and 3D computer graphics content in web browsers. It is based on OpenGL ES 2.0 and provides only the newer shader paradigm and thus lacks support for the fixed-function pipeline (with push/pop matrix stack etc.) of OpenGL 3 and lower. WebGL is supported by the majority of current browsers, both for desktop and mobile devices as shown in Figure 2.1. It uses the HTML5 $canvas$ element, but is not officially part of the HTML5 standard [4].

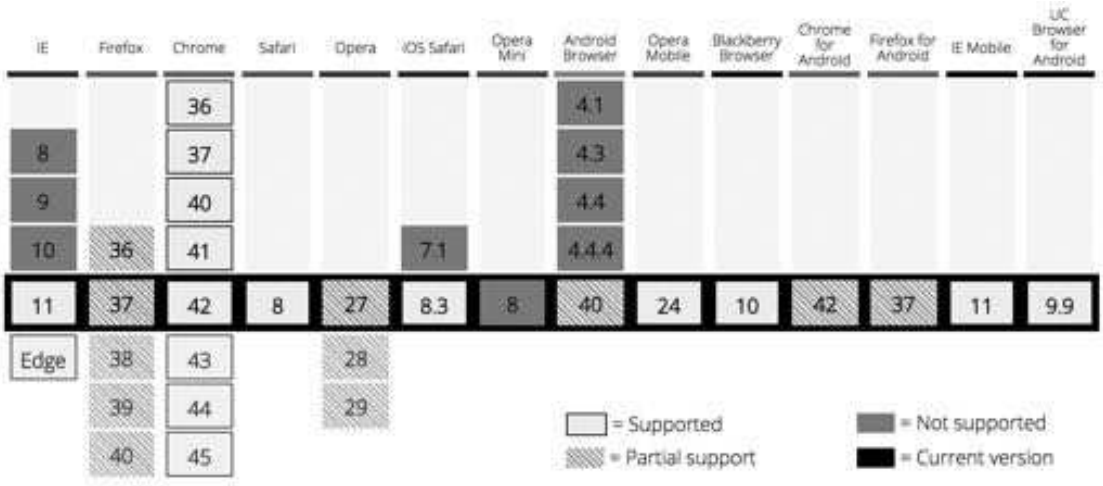

FIGURE 2.1: WebGL browser support [5].

Even though browser support is getting better, WebGL still has a long way before it is on par with its legacy counterpart, OpenGL. For the purposes of this thesis, some restrictions of WebGL should be mentioned.

- Like OpenGL ES, WebGL does not feature the *tessellation* and *geometry shaders* of OpenGL 4.0/3.2 and above.
- *Compute shaders* are not supported, since they became available in OpenGL ES 3.1.
- It is not possible to read data from GPU memory back into CPU memory.

The list can be made much longer. However, the Chrome and Mozilla development teams recently [6] announced that their implementations of *WebGL 2.0* will be available for experimental use in Q2 of 2015. WebGL 2.0 brings many anticipated features, like *gpu queries*, *multiple render targets*, *multisampled renderbuffers*, *transform feedback*, *3D textures* and so on. On top of this, WebGL 2.0 will launch with a shader language that better conforms to the modern GLSL standards. The specification of WebGL 2.0 is based on the OpenGL ES 3.0 specification and consequently does not include compute shaders.

It is safe to say that WebGL 2.0 will be a game changer for computer graphics in browsers. However, the current available version of the API supports enough features to be a powerful tool for planet rendering.

2.2 Scale challenges

As stated in section 1.1, scale is one of the most important features of astronomy visualization. It is also a tricky thing to get right. Since it is, in theory, possible to measure everything in the real world with arbitrary precision from e.g. a single atom ($\approx 10^{-10}$ m) to entire galaxies ($\approx 10^{17}$ m), one might expect a good visualization to reflect that same *level of detail*.

So, can computers capture this infinite precision? Computers use 32-bit floating point format to represent numbers. This is usually called *single* precision. There is also a *double* format, supporting 64-bit precision. 32 bits are enough for representing a number with up to 7 significant digit accuracy.

To put this in perspective; The earth has an equatorial radius of 6378137 m (7 digits). This means that in a scene with *just* the earth it should be possible sample surface data with down to (approximately) one meter between each sample point.

This might be enough precision for many data sets (such as satellite imagery etc.), but it relies on the earth being the only object in the simulated universe. Just adding the moon, which is located approximately 384400000 m from Earth suddenly requires at least 9 digits of accuracy to model one meter in the scene and 32-bit precision is not enough for that.

Javascript is natively using double precision for all numbers, resulting in 15 digit accuracy. This would be more than enough for modeling our entire solar system with meter precision (Neptune is $\approx 4 \cdot 10^{12}$ m from the sun), quite impressive. However, the majority of GPU:s (which are the ones positioning objects in a scene) do not support double precision. This is especially true for mobile devices which are target platforms for this thesis. Consequently, Javascript and WebGL do not allow passing double precision data to any GPU.

2.3 Memory challenges

Just like it is possible to run out of precision, it is possible to run out of *memory*. It is simply not efficient (or even possible) to store an entire universe on today's laptop hard drives and certainly not in RAM or GPU memory. The more detailed satellite imagery data sets are over 10 Petabyte in size ($>$ 10 million GB) and that is just for the earth.

When rendering detailed maps and 3D globe data it is therefore necessary to store only the data that is needed for a particular view. This leads to a need for remote storage. Storage servers like that can expose tile map services and implement standards, such as the Web (Tile) Map Service (WMS/WTMS) standards for delivering geospatial data in chunks or tiles.

2.4 Unitea

Scientific Content for Interactive Systems Sweden AB, or Sciss for short, is the company behind Uniview. Uniview is a massive scale interactive visualization of the universe targeted towards fulldome environments. It is specifically designed to be run on computer clusters with multiple, interconnected, machines feeding multiple projectors or monitors.

The nature of Uniview makes it less than ideal for anything other than very native environments. To provide more portability, the Unitea project was started. Unitea is first an education platform for interactive visualizations but it is also an attempt at pushing the limits of WebGL for low performance devices. A long term goal for Unitea is to support similar interaction patterns and massive scale as Uniview, but with a fraction of the computation power.

Chapter 3

Related Work

3.1 Terrain rendering with level of detail

This section provides a brief introduction to different techniques for rendering massive terrain in real-time frame rates. Note that the *Hierarchical LOD* technique described below in section 3.1.3 is explained more in depth in chapter 4.

Cozzi and Ring [3] propose that LOD terrain rendering algorithms can be split up in three separate subroutines; generation, selection and switching. Generation is the tessellation and labelling part of the algorithm. This is where specific levels of detail are defined as meshes and error thresholds calculated. The selection part is where the algorithm calculates which level of detail to render in a specific view. Selection is often based on camera position and primitive-to-pixel ratios. The last routine, switching, is how terrain rendering algorithms handle swapping or morphing between different levels of detail.

3.1.1 DLOD

Discrete Level of Detail (DLOD) is not really an option for massive terrain rendering, but is a good starting point. DLOD is arguably the most simple LOD algorithm. It is based in representing a model with multiple meshes of different detail. The generation step consists of either automatic decimation of a detailed mesh or artists manually creating low-poly versions of the same model.

Since DLOD consists of separate meshes for a model, the entire model must be replaced in the switching step. For large terrains (such as planets) this is undesirable because the model will usually cover a large part of the scene. Hence, DLOD will not be of any benefit since it selects

models of *uniform* detail. This detail will either be redundant if the camera is far away or not detailed enough if the camera is close to the model.

3.1.2 CLOD and ROAM

As opposed to DLOD, storing discrete versions of a model, Continuous Level of Detail (CLOD) instead stores transformations for decimating certain parts of a detailed mesh. This can be done through a series of edge collapses and vertex splits as proposed by e.g. Garland and Heckbert [7]. CLOD can be adapted to a view based algorithm by modifying Garland and Heckbert's quadric metrics based on the camera position.

Real-time Optimally Adapting Meshes (ROAM) as presented by Duchaineau et. al [8] is an implementation of CLOD that, given a piece of terrain data (e.g. a heightmap), tries to optimize the vertex-to-detail ratio. This is done by sampling a model or heightmap on the CPU and by measuring derivatives between samples. Areas with large differences will simply get more vertices (or, conversely, lower error thresholds). ROAM is quite an old technique presented by Duchaineau et. al in 1997. While it is indeed optimizing the level of detail that can be achieved for a fixed amount of vertices it is a CPU-intensive terrain rendering technique.

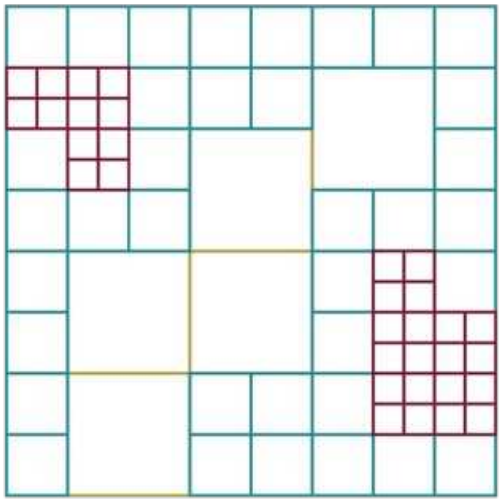

FIGURE 3.1: Continuous LOD algorithms optimize vertex-to-detail ratio.

3.1.3 HLOD

Hierarchical Level Of Detail (HLOD) is a broad class of terrain rendering algorithms. The common denominator for these is a BSP (e.g. a sparse quad-tree) structure that, upon traversal, uses error metrics to decide if more or less detail is needed in a particular region. Each node in the quad-tree is responsible for a surface patch which is a mesh with a fixed resolution and can

split into four child nodes. This technique is less CPU-intensive than ROAM because it can either sample heightmaps on the GPU or generate the patches themselves in advance (similar to DLOD). As graphics cards have become more sophisticated, the performance of draw calls and vertices rendered per draw call has drastically improved. Terrain rendering algorithms do not have to optimize for vertex count in the same way as before; Therefore it is possible to send sufficiently dense meshes to the GPU to sample a heightmap. A popular implementation of HLOD is Chunked LOD, presented by Ulrich [2]. The techniques used in this thesis project are loosely based on both Ulrich's Chunked LOD technique and the HLOD algorithms presented by Cozzi and Ring [3].

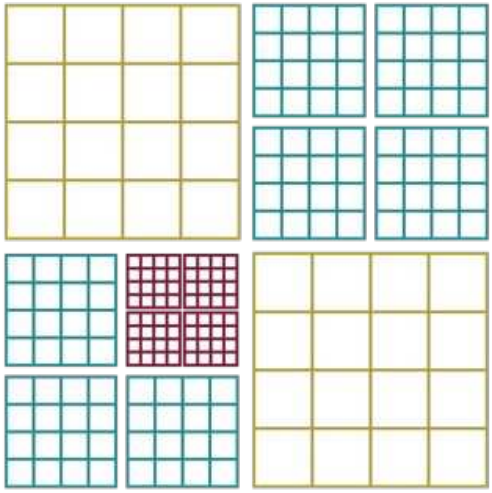

FIGURE 3.2: Hierarchical LOD organizes LOD levels as chunks in a quadtree.

3.1.4 Geometry Clipmaps

Geometry Clipmaps was originally presented by Losasso and Hoppe [9] and then optimized for GPU and published by Asirvatham and Hoppe [10]. The technique is based on *clipmaps*, as presented by Tanner et. al [11]. Clipmaps are a variation of the mipmap concept, exposing a unified interface to sample a texture by combining different resolutions of the same image. Clipmapping can be thought of as *virtual* mipmapping. Mipmaps use the most detailed image to produce lower resolution representations of it. Hence, for large images, the memory footprint will be large even though only a small part of the most detailed texels are actually rendered. In the case of planet rendering, storing a mipmap of the entire planet's imagery is, as stated in section 2.3, not possible. Clipmaps solve this issue by only storing partial textures close to a *clip center* (e.g. the camera) in full detail. This results in a leveled structure of concentric square textures where each texture has the same resolution but covers four times the area of the next, more detailed, level.

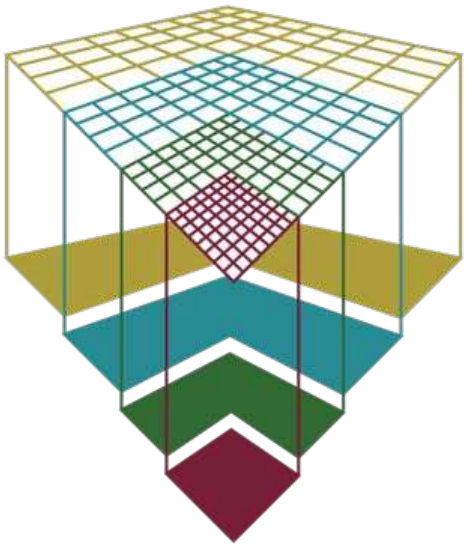

FIGURE 3.3: Geometry Clipmap techniques offset vertices in a static mesh by sampling a dynamic *clipmap*.

The clipmap approach is adapted for geometry using a static mesh that samples height data from a clipmap. The geometry itself also has a clipmap structure to distribute vertices. This results in an algorithm using only one draw call, with a static vertex buffer, to render an arbitrary terrain. Compared to HLOD, which (in WebGL 1.0) needs to render each surface patch in a separate draw call, this is a big performance boost.

3.1.5 GPU Ray Casting

Instead of relying on vertices, heightmaps can be rendered directly using the fragment shader. Dick et. al [12] propose using axis-aligned bounding boxes (AABB:s) instead of the surface patches of HLOD to represent the heightmap tiles. Each AABB is then rendered using ray-marching, i.e. ray-casting in incremental steps from the camera through the bounding box. This approach use almost no geometry, so there is next to no CPU processing involved. However, because there is no analytical solution (as in ray-casing bounding spheres) and no way to dynamically calculate the step size (as with signed distance fields) this is an expensive procedure. To counteract this, Dick et. al propose the use of a *maximum mipmap pyramid*. This data structure splits the heightmap into a full quadtree of different resolutions and effectively reduces the time complexity of the ray-cast from $\mathcal{O}(nm)$ to $\mathcal{O}(n \log(m))$.

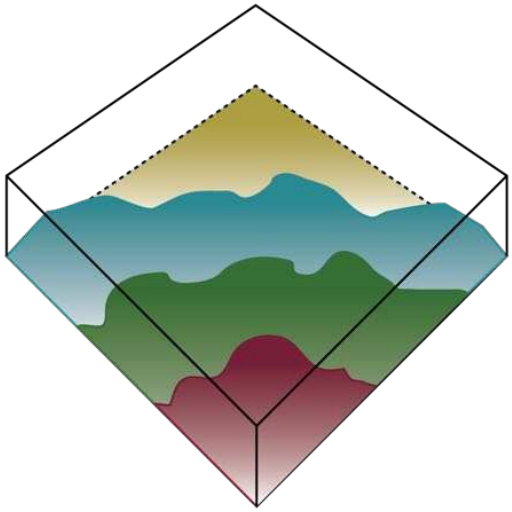

FIGURE 3.4: GPU Ray Casting shifts rendering cost to the fragment shader.

3.2 Sphere tessellation schemes

This thesis approximates planets with spheres. The reason for this is to keep focus on the rendering techniques rather than getting stuck in details. With that said, using World Geodetic System standards like WGS84 (the de facto standard for georeference) is important to model certain aspects of the universe.

There are several different ways of approximating a sphere with vertices. Depending on the use case, some are better suited than others. For planet rendering, there are many options. This section covers three of these tessellation schemes.

3.2.1 Tessellation from spherical coordinates

Perhaps the most natural way of tessellating a sphere is to define vertices using spherical coordinates. Creating vertices by iterating linearly through ϕ and θ results in non uniform vertex density, oversampling the poles of the sphere. Figure 3.5 shows spheres approximated with different step sizes along ϕ and θ.

FIGURE 3.5: Tessellation from spherical coordinates. From the left; 4 width segments/2 height segments, 8 width segments/4 height segments, 16 width segments/8 height segments.

3.2.2 Quad Sphere tessellation

A *quadrilateralized spherical cube*, more commonly referred to as a quad sphere or quad cube, is a cube with arbitrary subdivided faces where each vertex is projected onto a sphere. This tessellation scheme results in more uniform vertex density than the spherical coordinate tessellation as shown in Figure 3.6. Mapping points in spherical coordinates back into cube coordinates is simple and GPU:s have support for this cube mapping built in.

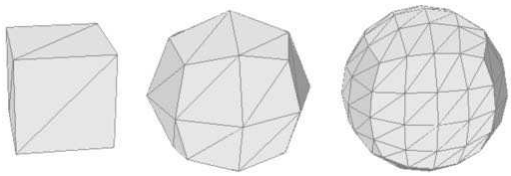

FIGURE 3.6: Quad Sphere tessellation. From the left; 1 segment per face, 4 segments per face, 16 segments per face.

3.2.3 HTM Sphere tessellation

Hierarchical Triangular Mesh (HTM) is a technique for approximating spheres with uniform vertex density. This is done using the subdivision algorithm below;

- Start with a base primitive, like a tetrahedron or octahedron, where each vertex has the same distance to the center of the primitive.
- Subdivide each triangular face into four new faces
- Offset the new vertices to have the same distance to the center as the base vertices.
- Repeat the subdivision to desired detail.

FIGURE 3.7: HTM Sphere subdivision. From the left; starting primitive (octahedron), 1 subdivision, 3 subdivisions.

3.3 Data Mapping

The choice of tessellation scheme is much dependant on the format of the available data. Data formats, corresponding to the tessellation schemes in section 3.2, are described in this section.

3.3.1 Cylindrical Projection Mapping

Using the geographic longitude and latitude coordinates is, by far, the most common way of mapping planet data to store in an image. This mapping scheme is called *Cylindrical Projection*. Cylindrical Projection maps data measured in $(long, lat)$ or (ϕ, θ) coordinates to the respective x and y dimensions of an image. A standard world map, as shown in Figure 3.8, is an example of a cylindrical projection.

FIGURE 3.8: Example of an (equirectangular) cylindrical projection.

There are two concerns when using cylindrical projections. The first is the same oversampling near the poles as in spherical coordinates tessellation. This stems from the fact that all texels in both the uppermost and lowermost rows of the image are mapping the same point of a sphere. Oversampling results in discontinuities close to the poles and are visible as *polar pinch* artifacts.

The second problem is discontinuity in the antimeridian, e.g. where the left and right side of the map gets folded together. If there are edges in the geometry crossing the antimeridian, texture sampling along those edges will get complicated. However, all of the tessellation schemes above can be setup to never have edges crossing the antimeridian.

3.3.2 Cubemaps

Cubemaps are widely used in computer graphics and provide an easy way of mapping imagery to a cube. Skyboxes and environment maps often use cube maps even though the geometry they project onto is usually ellipsoidal or curved in some way.

The quad sphere in section 3.2 is particularly well suited for sampling from a cubemap. The vertex positions, before projecting the cube onto a sphere, can be used directly to sample a cubemap.

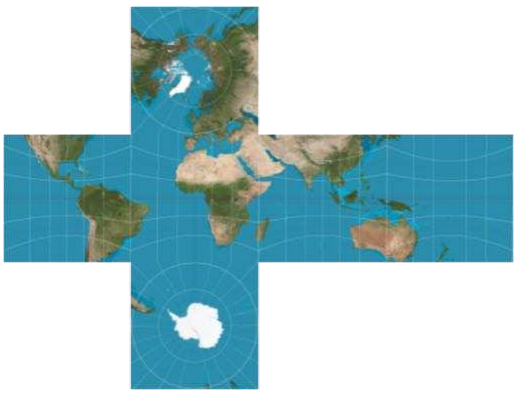

FIGURE 3.9: The world map in Figure 3.8 reprojected into a cubemap.

3.3.3 TOAST

A problem with cubemaps is that they are non-trivial to store in an efficient manner. Either each individual side must be stored separately or the entire ”cross” is stored with a lot of empty texels. Image compression makes the storage impact negligible, but decoding and reading such an image into GPU memory takes a lot of redundant processing.

Tessellated Octahedral Adaptive Subdivision Transform (TOAST), as presented in Microsoft’s WorldWide Telescope [13], is a projection scheme which aims to pack spherical data into square images. The resulting images has the north pole mapped to the center, south pole split into the four corners and the equator following the diagonals between the midpoints of each edge (which produces the ”toast” structure). By using the repeat texture wrap mode in both dimensions on the GPU, the TOAST map produces smooth interpolation across texture borders. A bonus with the mapping scheme is that no continent (apart from Antarctica) on the world map has discontinuities over texture edges.

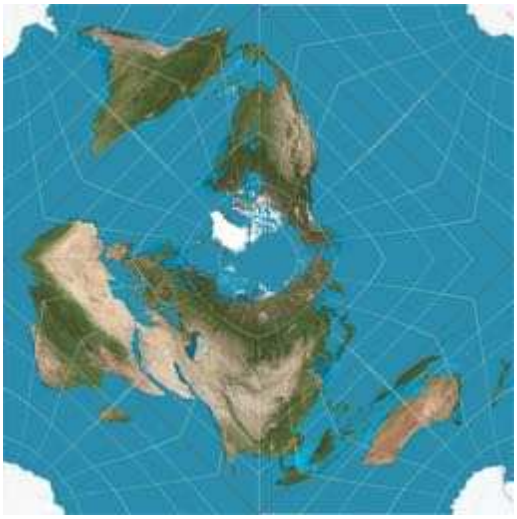

FIGURE 3.10: The world map in Figure 3.8 reprojected into a TOAST map.

The TOAST projection scheme can be used to map images to octahedron-based HTM Spheres. An image is mapped to an octahedron by UV-mapping its vertices to the middle, corners and edge midpoints of the TOAST map. Subdivided vertices are then recursively assigned the mean of its two vertex neighbors' texture coordinates.

3.4 EPSG standards

As stated above, cylindrical projection is the most common format for data mapping. It is also mentioned, in passing, that a specific version is used in Figure 3.8, namely *equirectangular projection*. This projection is one of many ways to transform the measured data into cylindrical, cubemap or TOAST map coordinates. The mapping examples above (figs. 3.8 to 3.10) all use equirectangular projection which maps each data coordinate (measured in $long/lat$) to a regular grid with *equal spacing* between both $latitude$ and $longitude$ coordinates.

While equirectangular projections are well suited for mapping data back onto a sphere or ellipsoid, they are not ment to be displayed as-is. The reason for this is that while the $long/lat$ ratio is preserved in the projection, distances are not equally scaled. A common example of this is buildings that are square in reality will be distorted by the projection, making them compressed towards the equator and stretched towards the poles. Therefore, to preserve relative scale, it is possible to use another projection scheme called *Mercator projection*. This is the most common way of presenting cylindrical projections as 2D maps. Figure 3.11 shows an image of the world mapped with mercator projection. Notice, however, that it is the *relative* scale that is preserved. Just like in the equirectangular case, features still get enlarged near the poles and compressed near the equator.

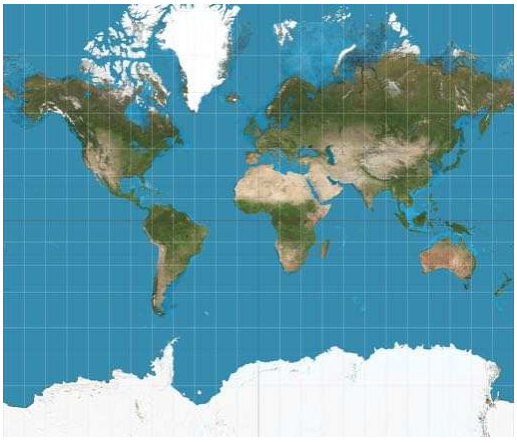

FIGURE 3.11: Example of a mercator cylindrical projection.

There are thousands of these different projection schemes that have been standardised with their own Spatial Reference System Identifier (SRID). The most prominent identifier is the *EPSG*-system by the International Association of Oil & Gas Producers [14]. The equirectangular projection has SRID EPSG:4326 and a special case of mercator projection (used by google and microsoft) called *Pseudo-mercator* has the identifier EPSG:3857.

3.5 Tiling schemes and online map services

Web Map Services (WMS) and Tile Map Services (TMS/WTMS) are online map services that expose standardised interfaces to the user. Through such an interface, users can request specific *map tiles*. There are different interface standards employed by the map services, three of the most common ones are explained below.

3.5.1 WMS

The *Web Map Service* standard, maintained by the Open Geospatial Consortium [15], is a tile service standard that lets users request arbitrary map tiles that are dynamically generated from a large dataset. Such a request is shown in Figure 3.12 along with the corresponding (equirectangular) tile response.

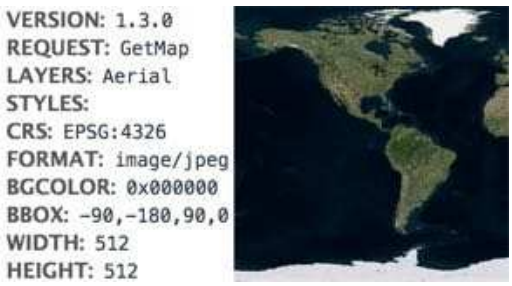

FIGURE 3.12: Example of WMS request parameters (left) and response (right). Image from the OnTerra WMS for Bing maps [16].

The `BBOX` parameter lets users send requests specifying the exact (N, W, S, E) edges of tiles.

3.5.2 TMS

While WMS interfaces permit requesting of dynamic map tiles, they need backends to pre-process the response tiles. This may cause latency problems, especially for interactive applications. The Tile Map Service (TMS) standard, managed by the Open Source Geospatial Foundation community [17], is an alternative to WMS that use *static* (pre-rendered) tiles. The tiles

are requested with *row* and *column offsets* as well as a *zoom level*. The TMS tile structure is exemplified in Figure 3.13 where the rows and columns of the first three zoom levels are shown.

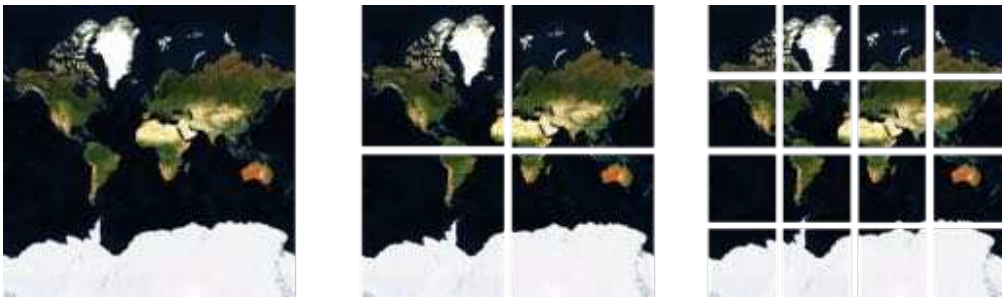

FIGURE 3.13: Example of how TMS splits data into rows and columns of tiles. Images from Mapbox [18].

Many open source services use TMS. Google maps uses an interface similar to the TMS standard.

3.5.3 Virtual Earth Tile Service

Virtual Earth, one of Microsoft's map services, use a variation of the TMS standard. This variation is based on a quadtree rather than flat rows and columns and may therefore be more appropriate to work with in different scenarios.

Each tile in the Virtual Earth scheme is assigned a *quadkey* and can split into four subtiles with quadkeys all being derivatives of the parent quadkey. For example, tile `[0]` will split into tiles `[00]`, `[01]`, `[02]` and `[03]` as exemplified in Figure 3.14.

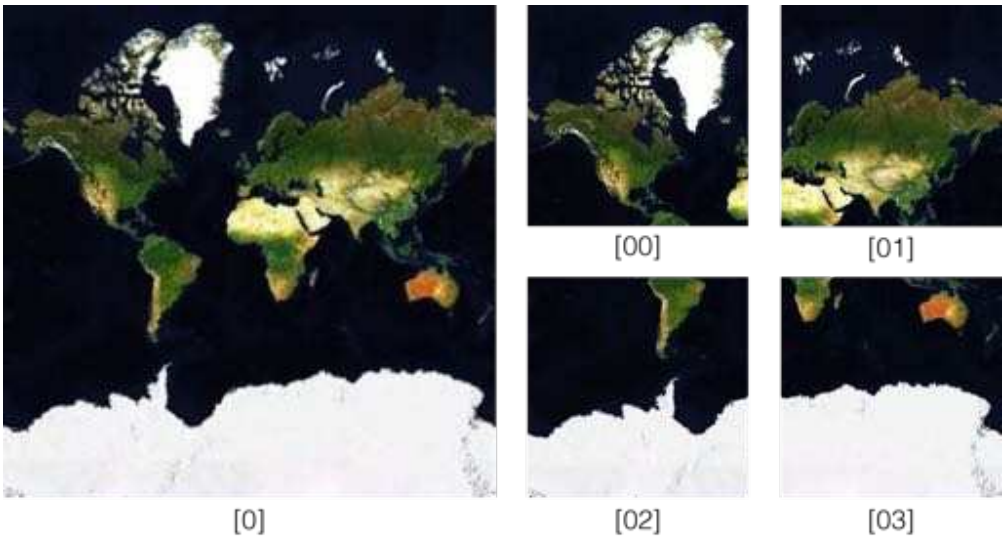

FIGURE 3.14: Example of how Virtual Earth Tile Services represent data in a quadtree of tiles.

Chapter 4

Implementation

This section describes, in depth, the implementation of LOD techniques for planet surface and terrain rendering in WebGL. The methods used are Hierarchical/Chunked LOD with out-of-core rendering from WMS and TMS data sources.

4.1 Geometry, data mapping and terrain rendering techniques

Chapter 3 describes different techniques for representing sphere meshes and ellipsoid surface data. This implementation chooses a subset of these techniques for reasons listed below.

- Terrain rendering technique: HLOD/Chunked LOD - Offers most control over performance/accuracy and does not require the modern Open GL 4.0+ pipeline. Geometry clipmaps were thoroughly tested before deciding that these were too complex and GPU dependant to get working on the WebGL pipeline.
- Data mapping: Cylindrical projection mapping - All WMS and TMS that have been used in this project expose data using cylindrical projection mapping. Either as epsg:4326 or epsg:3857. This implementation tries to keep images in the format they are delivered to avoid resampling artifacts.
- Geometry representation: Tessellation from spherical coordinates - Fits well into the WMS standards and can be used to have each LOD patch only be dependant on one texture tile since geometry and textures will be aligned to each other. This tessellation scheme was also determined to be the easiest to work with.

In other words, these choices are much based on how much flexibility and ease of use that the different techniques offer.

4.2 Quadtree structure

As stated in section 3.1, the base of Chunked LOD is a sparse quadtree. At the root of the tree is a node containing a simplified version of the entire model, i.e. the entire planet in this case. Nodes can then split in four, each with the same number of vertices as its parent. This effectively quadruples the number of vertices incident on a given area and, consequently, increases the level of detail.

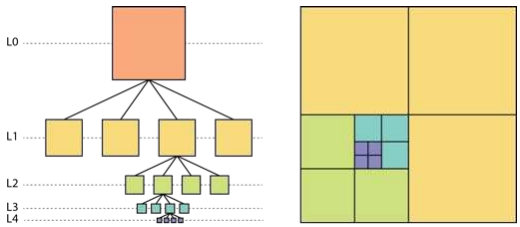

FIGURE 4.1: Surface patch quadtree.

Figure 4.1 illustrates the quadtree structure. Each node, or quad, in the image holds some data. These *chunks* of data, representing the different levels of detail, are what gives chunked LOD its name. Cozzi and Ring [3] defines that, after the *generation* step, chunks should conform to three criteria:

- *Rectangular shape* - Since each chunk is an atomic entity and should not operate on other chunks it needs to *fit* with adjacent chunks. A straight forward way to guarantee this is to define the chunks as a grid of rectangular meshes. Hence, each chunk must have (at least) a vertex in each corner and edges between these corner vertices.
- *Geometric error* - Cozzi and Ring also suggest storing error metrics inside the chunks themselves. Note that this implementation differs from that and instead relies on the view-plane projection described in section 4.4.1. However, Cozzi and Ring's approach is more generalized.
- *Bounding volume* - Both visibility and error metrics calculation rely on a bounding volume for the chunks. Implementations with both regular AABB:s and bounding boxes that fit to the patches have been evaluated.

Another design decision that went into this implementation is to let chunks be responsible for their own textures. Figure 4.2 exemplifies the resulting layout of a chunk.

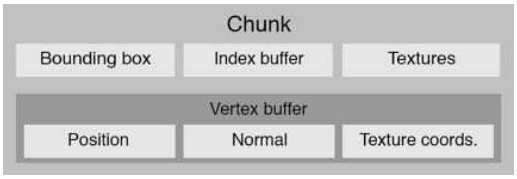

FIGURE 4.2: Chunk module layout.

4.3 Tree traversal

The quadtree is traversed every simulation step visiting each node in the tree. This update procedure is the *selection* part of HLOD. Here, visibility and error metrics are calculated and both the scene and quadtree are updated. The full decision tree is illustrated in Figure 4.3.

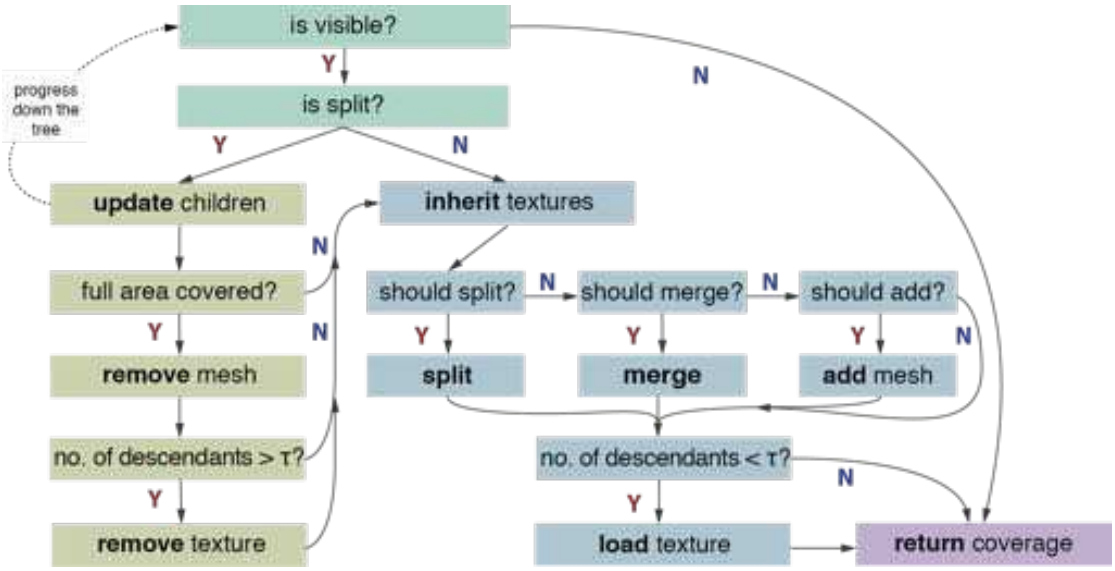

FIGURE 4.3: The selection procedure, updating the quadtree.

The traversal starts at the root node and then recursively descends down the tree. This is a *depth first* traversal meaning that one branch will be fully updated before starting on the next branch. Each node has a number of operations associated with it, these are marked in bold in Figure 4.3 and will be described in detail below.

4.4 Splitting and merging

A core part of the selection algorithm is splitting and merging chunks, this is what creates the dynamic level of detail. Whether a chunk should split or merge is decided through an error metric calculated with respect to the camera. The chunk's bounding box is projected onto the camera's viewplane, the area of the projection (the error metric) is calculated and compared to a

threshold. Note that only one global threshold is necessary since the limit of the projected area, and hence the perceived detail, will always be compared in camera coordinates regardless of a chunk's scale in the underlying scene.

4.4.1 Viewplane projection

Projecting the bounding boxes into the viewplane is the same three step model-view-projection operation that is done for every vertex in every frame on the GPU. In eq. (4.1), a single vertex is projected into the view plane.

$$\vec{p}_{proj} = PVM\vec{p} \tag{4.1}$$

The interesting metric is, however, the *visible* projected area of a bounding box and since eq. (4.1) does not perform clipping, $\vec{p}_{proj}$ may be outside of the camera frustum. Efficient triangle clipping is both complex and produces (up to 4) more triangles to calculate the area for. For the interested reader the general polygon clipping algorithm presented by Sutherland and Hodgman [19] may be a good starting point.

Instead of trying to clip the projected triangles from eq. (4.1), the projection itself can be changed from projecting onto a plane to a spherical projection around the camera as shown in eq. (4.2) and Figure 4.4. Spherical projection is simple, once a point can be expressed in camera coordinates it just has to be normalized to be on the surface of a unit sphere around the camera.

$$\vec{p}_{sproj} = normalize(VM\vec{p}) \tag{4.2}$$

This is what Uniview does, since the view-surface inside a dome theatre is better approximated as a hemisphere than a plane. This implementation is not targeted towards curved dome theatres but the spherical projection results in a good approximation of relative bounding box visible area which is useful as an error metric.

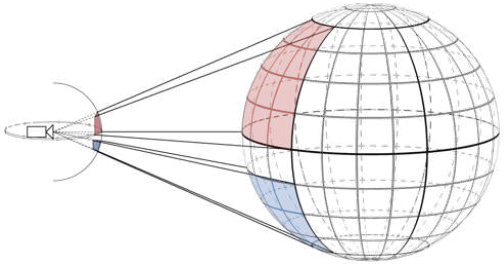

FIGURE 4.4: Projecting points from the scene to a sphere around the camera.

To get even more accurate results and speed up this operation, the bounding boxes can be back-face culled. This is done by checking the sign of the face normal ($\hat{n}_z$) after transforming a face into camera space and then discard faces where $\hat{n}_z < 0$.

Putting it all together, a procedure for determining if a patch should split or merge can be formulated:

```
1  SpherePatch.prototype.shouldSplit = function () {
2      if (this.isSplit) return false;
3
4      var visibleArea = this.projector.getProjectedArea(this.boundingBox);
5      var errorMetric = visibleArea*SpherePatch.SPLIT_FACTOR;
6
7      var canSplit = this.level < this.master.getMaxLodLevel();
8      var shouldSplit = SpherePatch.SPLIT_TOLERANCE < errorMetric;
9
10     return canSplit && shouldSplit;
11 };
```

LISTING 4.1: Determining if a surface patch should split.

A similar procedure, using the technique in listing 4.1, can be formulated for a `shouldMerge` as well.

4.5 Surface patch visibility calculation

Manually culling patches that are occluded or outside the frustum are important operations in HLOD algorithms. This lowers the cost of rendering surface patches to the screen and even more importantly; it drastically lowers the number of patches that has to be traversed in the quadtree.

4.5.1 View-frustum culling

View-frustum testing is an efficient way of culling geometry that is not visible in the current frame. A frustum test consists of intersection tests for each of the view frustum's six bounding planes (near, far, top, bottom, left and right) against the boudning box of a patch. For this test, the regular AABB is used since it is cheaper to test against.

Both of the two visibility tests benefit greatly from the quadtree structure. Since the quadtree guarantees that the children of a given patch fit inside the area of their parent, their bounding boxes will fix inside the bounding box of the parent too. Hence, if a parent is not visible, the

traversal stops and no redundant update calls/visibility tests are made. This visibility-termination is also shown in Figure 4.3 above.

4.5.2 Horizon culling

An additional visibility test is available to further boost performance. Occlusion culling is common in computer graphics applications to remove geometry that is hidden behind other geometry. In the case of planet rendering, the planet always occludes itself. For simpler objects, normal back-face culling on the GPU might be sufficient but since HLOD deals with much larger and much more complex data structures, it is efficient to utilize the quadtree structure here as well.

An efficient way of testing if a point is above or beyond the horizon is to make *plane-* and *cone tests*. These two tests were presented by Ring [20] as optimizations to the horizon culling proposed by Cozzi and Ring [3]. Referring to Figure 4.5, a plane test consists in calculating the plane which intersects a sphere at the horizon for a given camera position and then testing if a point is in front of the plane. A cone test then determines if the point is inside or outside the cone that is tangential to the horizon.

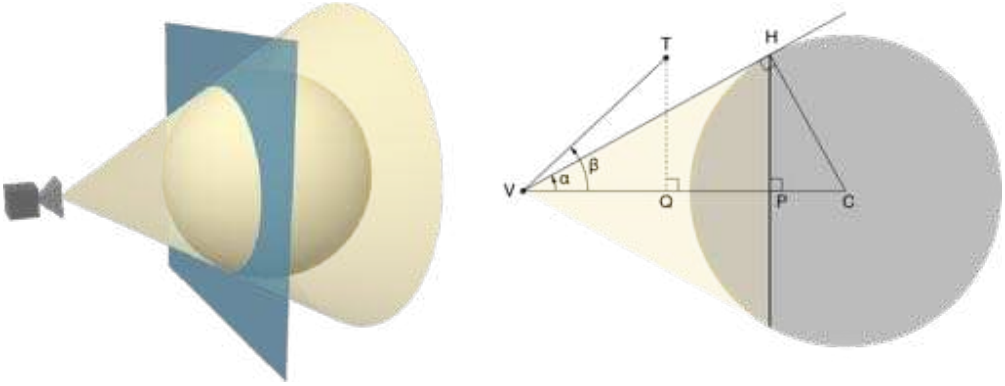

FIGURE 4.5: Plane and cone for horizon culling.

In Figure 4.5; V is the camera position, C is the center of a sphere, H is the horizon point, T is the target point to test, P is the projection of $\vec{VH}$ onto $\hat{VC}$ and Q is the projection of $\vec{VT}$ onto $\hat{VC}$.

Since $\triangle VHC$ is a right triangle, the Pythagorean theorem states that:

$$||\vec{VH}||^2 + r^2 = ||\vec{VC}||^2 \Leftrightarrow ||\vec{VH}||^2 = ||\vec{VC}||^2 - r^2 \tag{4.3}$$

$\triangle VHC$ and $\triangle CPH$ are similar triangles that share an angle at C. Therefore, by similarity:

$$\frac{||\vec{PC}||}{r} = \frac{r}{||\vec{VC}||} \Leftrightarrow ||\vec{PC}|| = \frac{r^2}{||\vec{VC}||} \tag{4.4}$$

The orthogonal distance, $||\vec{VP}||$, from the viewpoint V to the plane P can be expressed as:

$$||\vec{VP}|| = ||\vec{VC}|| - ||\vec{PC}|| = ||\vec{VC}|| - \frac{r^2}{||\vec{VC}||} \tag{4.5}$$

It is now possible to formulate an inequality to determine if a point T is in front of the plane:

$$\vec{VT} \cdot \hat{VC} < ||\vec{VC}|| - \frac{r^2}{||\vec{VC}||} \Leftrightarrow \vec{VT} \cdot \vec{VC} < ||\vec{VC}||^2 - r^2 \tag{4.6}$$

Determining if T is inside the cone is the same thing as testing if the angle from the camera towards the point is smaller than the angle towards the horizon, i.e. if $\beta < \alpha$ in Figure 4.5. For angles in the range $[-\pi, \pi]$ this is the same as $cos(\beta) > cos(\alpha)$. The right hand side of the inequality can be rewritten as:

$$cos(\alpha) = \frac{||\vec{VH}||}{||\vec{VC}||} \tag{4.7}$$

By substituting with a dot product, the left hand side can be rewritten as:

$$cos(\beta) = \frac{\vec{VT} \cdot \vec{VC}}{||\vec{VT}||||\vec{VC}||} \tag{4.8}$$

Substituting both eq. (4.7) and eq. (4.8) into the inequality and simplifying results in:

$$\frac{(\vec{VT} \cdot \vec{VC})^2}{||\vec{VT}||^2} > ||\vec{VH}||^2 \tag{4.9}$$

The reason for squaring both sides is because the inequality still holds without the square root to get the actual lengths of the vectors and is therefore an optimization. $\vec{VH}$ is still unknown, but substituting by eq. (4.3) results in the final cone test inequality:

$$\frac{(\vec{VT} \cdot \vec{VC})^2}{||\vec{VT}||^2} > ||\vec{VC}||^2 - r^2 \tag{4.10}$$

Note that the majority of eq. (4.10) is already calculated in eq. (4.6), so using these two tests together makes for an efficient horizon culling algorithm.

4.6 Texture queries

With the selection procedures from section 4.4 and section 4.5 in place it is possible to start fetching data for the patches that will be rendered. Two types of textures are required to render the planet; a surface color texture and a surface height texture. These are requested through a texture provider pipeline. Figure 4.6 shows an overview of how textures are loaded into the application.

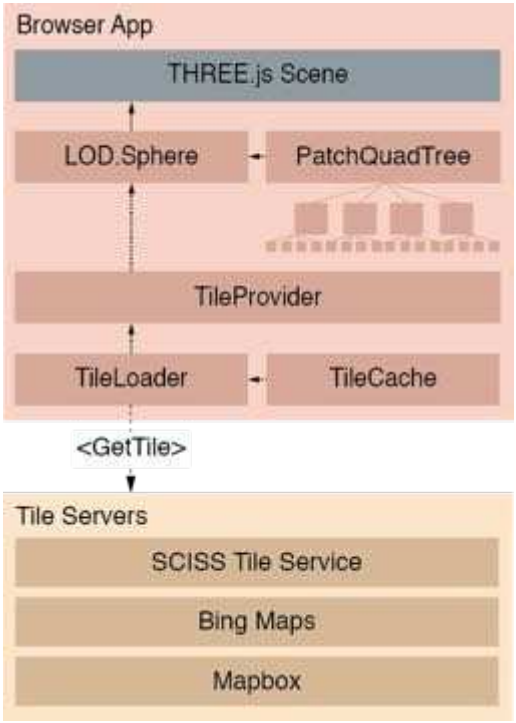

FIGURE 4.6: Simplified architecture of the texture provider pipeline.

Each `LOD.Sphere` depend on two `TileProvider`:s. One for surface color tiles and one for terrain tiles. Textures are not loaded directly into the GPU but are instead cached until needed for rendering. Details about the caching policies are found in section 4.8. Each texture is associated with its corresponding node in the quadtree. The update loop initiates texture requests depending on which patches are selected for rendering.

4.7 Texture inheritance

Since textures are requested the moment a patch should be rendered the textures will not be available for several frames. Therefore, the patch must find other textures to use until its own textures have finished loading. This is done by traversing up the quadtree, visiting ancestors and checking for loaded textures.

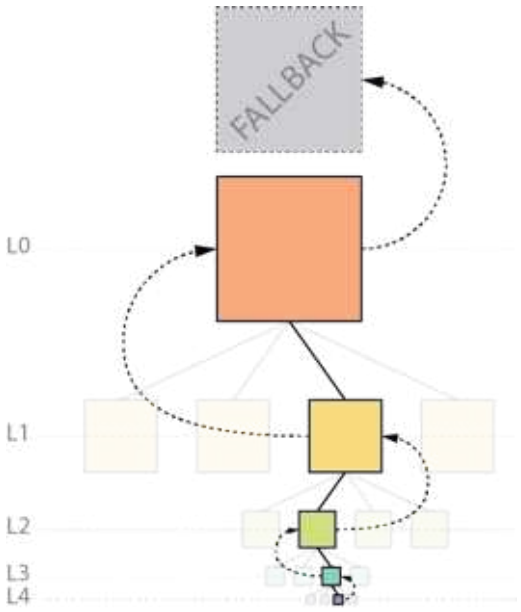

FIGURE 4.7: Texture inheritance lookup.

If the operation finds a loaded texture in an ancestor node, the alignment and crop must be calculated. This is done by maintaining the alignment each node has to its parent, i.e. if a node is a top-right child its parent alignment vector would be $[0.5, 0]$ assuming that the anchor is in the top left corner of a patch. Listing 4.2 shows how this alignment can be used to generate an anchor and extent in an inherited texture. Note that since both surface patches and texture tiles have the same resolution in both of their dimensions, only one `extent` value needs to be calculated.

```
1  SurfacePatch.prototype.getTextureFromAncestor = function (texType) {
2    var ancestor = this;
3    var texAnchor = new THREE.Vector2(0, 0);
4    var texExtent = 1;
5
6    while (!ancestor.textureLoaded[texType]) {
7      texExtent *= 0.5;
8      texAnchor.multiplyScalar(0.5).add(ancestor.parentAlign);
9      ancestor = ancestor.parent;
10     if (!ancestor) return false; // use fallback
11   }
12
13   return {
14     anchor: texAnchor,
15     extent: texExtent,
16     texture: ancestor.texture[texType],
17   };
18 };
```

LISTING 4.2: Texture inheritance lookup procedure.

4.8 Caching texture tiles

Latency in texture requests is perhaps the largest bottleneck when rendering planets. Therefore, well structured caching of texture tiles is an important optimization. Caching is done as an in-memory cache on the CPU side. The cache uses a *Least Recently Used* (LRU) replacement policy for deciding which tiles to keep and which to replace. According to Hultgren [21], LRU is an efficient general-purpose replacement policy for tiled WMS data.

The underlying data structures of the cache are a doubly-linked list that acts as a queue for texture tiles and a hashmap with values referencing items in the list. This structure guarantees $\mathcal{O}(1)$ time complexity for all operations on the cache. Figure 4.8 illustrates how these two data structures are connected to form an efficient cache.

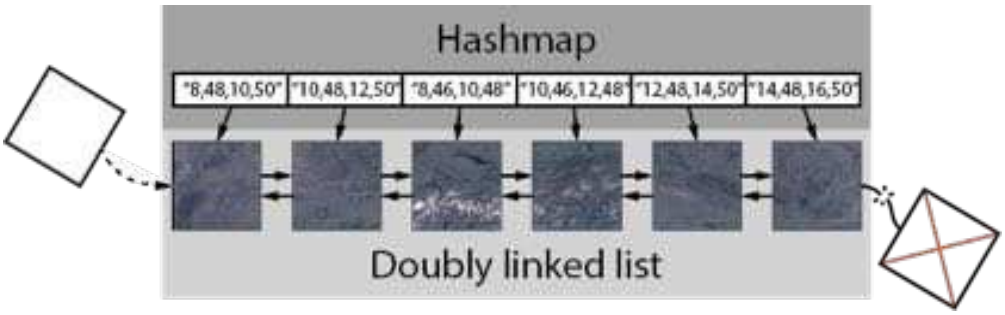

FIGURE 4.8: A tile cache structured as a doubly linked list and a hashmap.

The hashmap provides constant time access to any item in the cache and the linked list insert/remove operations are of constant time complexity as well. Every `insert` operation will check if the cache is full and, if so, remove the last item in the list to make room for the new. Every `find` operation will, if found, promote the found item to the front of the list. This effectively models an LRU replacement policy for the cache.

4.9 Mercator to equirectangular transform

The majority of online map services use the pseudo-mercator (EPSG:3857) format to expose geographic data. This is problematic for planet rendering because, as shown in Figure 4.9, the mercator projection does not produce accurate results when mapped directly onto a sphere.

It is, however, possible to *reproject* the mercator tiles into an equirectangular projection. In order to reproject a tile, its *latitude* bounds must be known. Its longitude bounds will be the same in an equirectangular projection. The latitude bounds then needs to be converted into *mercator angles*:

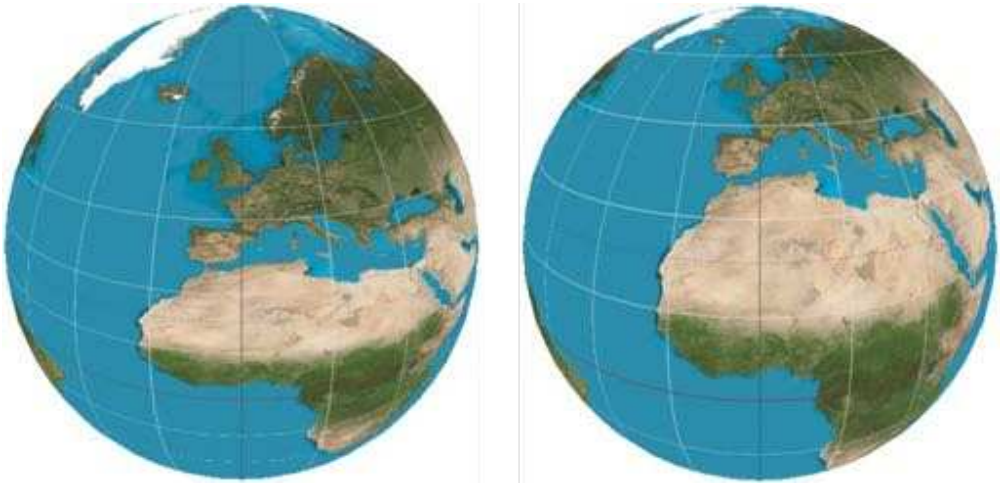

FIGURE 4.9: Mapping a mercator texture (left) and an equirectangular texture (right) onto spheres.

$$\theta_M = \frac{ln(\frac{1+sin(\theta_{lat})}{1-sin(\theta_{lat})})}{2} \tag{4.11}$$

The equation above has singularities for $\theta_{lat} = \frac{\pi}{2} + n\pi$ where $n \in \mathbb{Z}$, i.e. in the poles of a globe. However, the epsg:3857 standard solves this by restricting tile latitudes to a range of $-85.06° \leq \theta_{lat} \leq 85.06°$.

Since eq. (4.11) maps latitude non-uniformly, it needs to be done for every point along the tile's latitude extent. Since this is a performance heavy operation, it is implemented as a separate render-to-texture-quad pass on the GPU. This setup also makes it possible to tweak the accuracy-to-performance ratio using different tessellations of the underlying quad's vertices. The quad only needs to be tessellated along the y-axis since the x-axis (longitude) will map $1:1$ between the mercator and equirectangular projection. Ring [22] suggests (from empirical observations) using a 2×64 plane as base quad for the reprojection pass. The mercator angles and resulting texture coordinates are then calculated in the vertex shader (using eq. (4.11)) and sent to the fragment shader to sample the mercator texture tile.

```
1  uniform float mercatorExtent;
2  uniform float southMercatorAnchor;
3  uniform float northLat;
4  uniform float southLat;
5
6  attribute vec2 uv;
7
8  varying vec2 reprojectedTexCoords;
9
10 // 85.06 degrees in radians
11 #define MAX_LAT 1.484577
```

```
12 void main() {
13     float lat = southLat*(1.0 - uv.y) + northLat*uv.y;
14     lat = max(min(lat, MAX_LAT), -MAX_LAT);
15
16     float sinLatitude = sin(lat);
17     float mercatorY = 0.5*log((1.0 + sinLatitude)/(1.0 - sinLatitude));
18     float localMercatorY = mercatorY - southMercatorAnchor;
19
20     reprojectedTexCoords = vec2(uv.x, localMercatorY/mercatorExtent);
21
22     // calculate position as usual
23 }
```

LISTING 4.3: Vertex shader for sampling a mercator texture from equirectangular coordinates.

The above procedure can also be used to combine multiple mercator tiles into equirectangular tiles as seen in fig. 4.10. It runs in real-time framerates so it can be used to reproject tiles the moment they are needed.

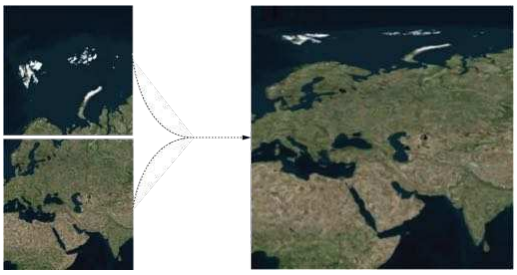

FIGURE 4.10: Reprojecting two mercator texture tiles into one equirectangular tile.

4.10 Skirts

Even when satisfying the *generation* step geometry requirements outlined in section 4.2, there is going to be visible cracks between surface patches. This stems from the fact that when displacing two neightboring patches using two heightmaps of different resolution the vertices along the shared edge will not align. Figure 4.11 shows a situation where cracks have formed.

Figure 4.11 also shows how to remedy the situation with *skirts*. Skirts do not fix the height discontinuities but is an easy way to hide them so that the model does not look like it is breaking apart.

Skirts are implemented as a one-vertex padding on the perimeter of each surface patch. The vertices are extruded inwards, towards the planet origin. Texture coordinates for the skirt are simply copied from the neighboring ”real” border vertices. Figure 4.12 shows an example of how this improves the visual result in the HLOD implementation.

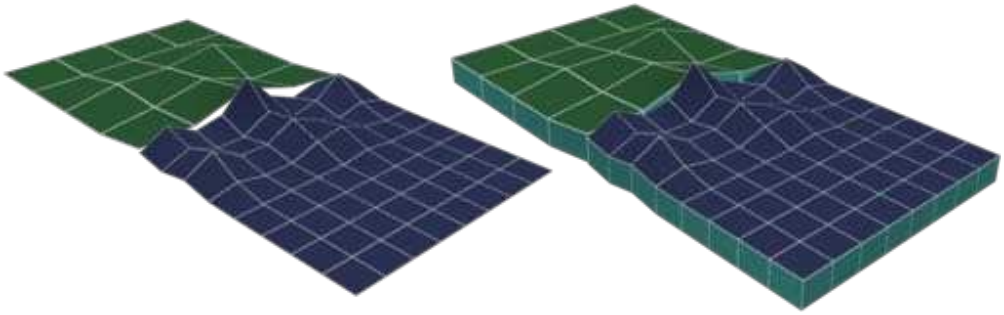

FIGURE 4.11: Cracks between surface patches (left) are hidden with skirts (right).

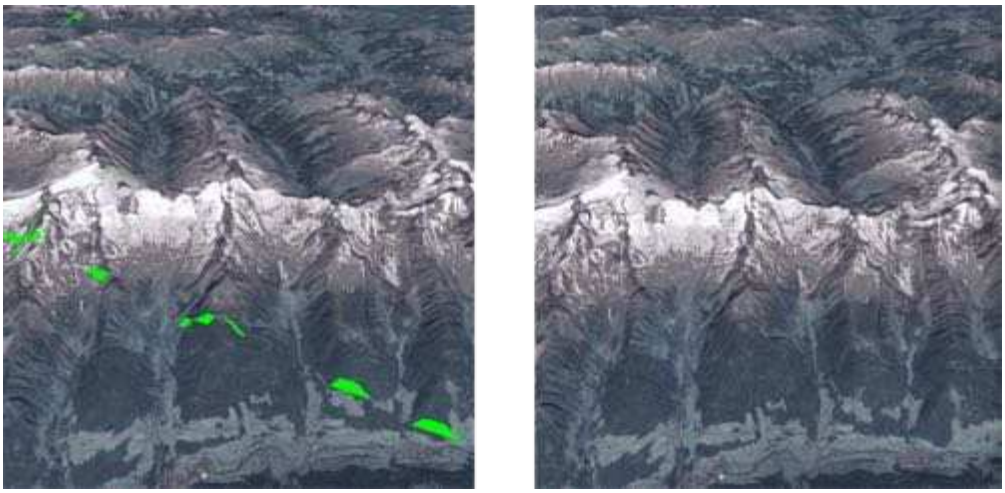

FIGURE 4.12: Implementation of hiding cracks (left) with skirts (right).

4.11 Optimizing tree traversal

A high level goal is to keep the tree traversal as an operation with linear time complexity, i.e. visiting each node only once. However, each inner node needs information both about the state of its ancestors and of its descendants. This suggests a time complexity of $\mathcal{O}\left(n^2\right)$ worst case since each node must visit their entire branch of the quadtree.

The reason for nodes to keep track of every other node in the branch is so they can inherit textures from ancestors (see listing 4.2) and know the coverage and visibility of descendants (see fig. 4.3). To keep time complexity of the traversal linear, this state information can instead be accumulated and propagated up or down. Listing 4.4 shows a recursive update function for nodes in the quadtree which propagates information between tree levels.

```
1 SurfacePatch.prototype.update = function (ancestorsState) {
2
3     // ...
4
5     if (this.texture.isBetterThan(ancestorsState.texture)) {
6         ancestorsState.texture = this.texture;
```

```
7           ancestorsState.textureAlignment = this.textureAlignment;
8       } else {
9           this.texture = ancestorsState.texture;
10          this.textureAlignment = ancestorState.textureAlignment;
11      }
12
13      var descendantsState = this.updateChildren(ancestorsState);
14
15      var nodeState = {};
16      nodeState.isCovered = descendantsState.isCovered;
17      if (nodeState.isCovered) {
18          this.remove();
19      }
20
21      return nodeState;
22  };
```

LISTING 4.4: Propagating states from ancestors and descendants

Merging the texture inheritance procedure from listing 4.2 into the accumulative function above makes the tree traversal linear.

Chapter 5

Results

This thesis project has resulted in a library, compatible with Three.js, for modeling textured spheres with heightmaps and hierarchical level of detail. The library has the following features:

- Sphere meshes, tessellated from spherical coordinates with continuous, dynamic, level of detail based on camera position.
- Out-of-core rendering of height and color data from epsg:4326 WMS and TMS sources. Experimental support for epsg:3857 TMS sources.
- Client side caching of texture tiles.
- View-optimized meshes and quadtree traversal.
- Dynamic, user defined, error thresholds and split factors for use on devices with varying performance.

5.1 Maintainability and integration

The library is designed following the CommonJS code conventions [23] and built for web browsers using Browserify. CommonJS is a dependency management system that allows developers to separate Javascript code into atomic modules and Browserify is a build system for CommonJS modules. The library is unit tested with NodeJS (native Javascript implementation) and Mocha (Javascript testing framework).

5.2 Device performance

Since this project is targeted towards low performance devices the implementation has been tested on a range of mobile devices as well as laptops with varying performance specifications.

The ”MBP15” device listed below is a MacBook Pro 15-inch running OSX Yosemite 10.10.3 with 2.5GHz quad-core Intel Core i7 processor, Retina display, 16GB RAM and NVIDIA GeForce GT 750M with 2GB GDDR5 memory. The ”Dell XPS15” device listed below is a laptop running Windows 8.1 with 2.3GHz quad-core Intel Core i7 processor, 16GB RAM and NVIDIA GeForce GT 750M with 2GB GDDR5 memory.

Device	**Browser**	**Min FPS**	**Avg. FPS**
MBP15	Chrome 41.0	60	60
MBP15	Opera 28.0	60	60
MBP15	Firefox 36.0	60	60
MBP15	Safari 8.0	49	60
Dell XPS15	Chrome 41.0	60	60
iPhone5S	Chrome 41.0	7	32
iPhone5S	Safari 8.0	9	48
iPad3	Chrome 41.0	1	8
iPad3	Safari 8.0	3	25
iPad Air 2	Chrome 41.0	11	33
iPad Air 2	Safari 8.0	20	54
Sony Xperia Z3	Chrome 41.0	5	26
Sony Xperia Z3	Android Browser 37	12	35

TABLE 5.1: Browser and device performance.

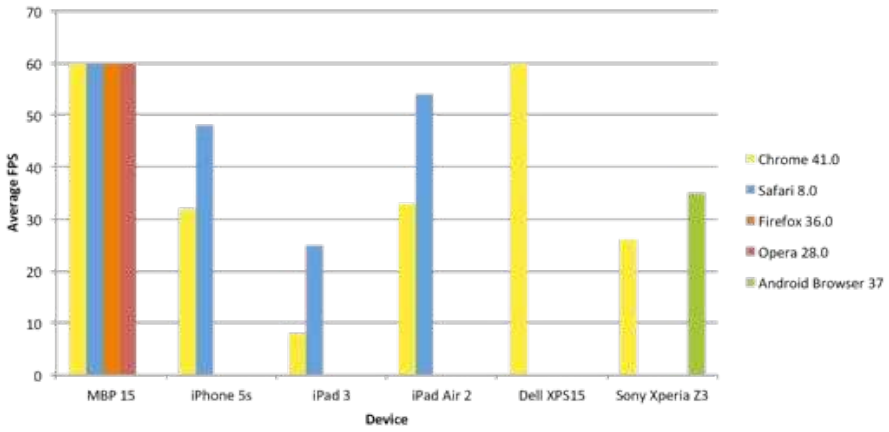

FIGURE 5.1: Comparison of average FPS in browsers and devices from table 5.1

Table 5.1 and fig. 5.1 shows a device test of the implemented features. Test conditions were the following:

- Dynamic level of detail sphere with both height and color maps from epsg:4326 WMS. Max quadtree depth for this dataset is 16 levels.
- Patch tessellation: 32×32 vertices/patch (+ skirts).
- Patches split when $projectedArea > \frac{screenArea}{5}$
- Camera path: Start $3\times$ earth radii from earth center, zoom in on the alps, tilt to see horizon and then rotate 180° in the surface plane.

Memory footprint is steady at around 30 MB depending on current view. Figure 5.2 Shows screen dumps from the test that generated the results above.

FIGURE 5.2: Camera path generating the data in tables 5.1 to 5.3.

5.3 Optimizations

The impact of the manual culling described in section 4.5 has been measured. The results are presented in table 5.2 and fig. 5.3. Data in the table is based on the same test setup and camera path as in section 5.2.

Culling	**Nodes visited**	**Draw calls**
Frustum + Horizon	mean: 95, max: 162	< 60
Horizon	mean: 134, max: 226	< 170
Frustum	mean: 113, max: 192	< 74
No culling	mean: 153, max: 247	< 185

TABLE 5.2: Culling performance impact.

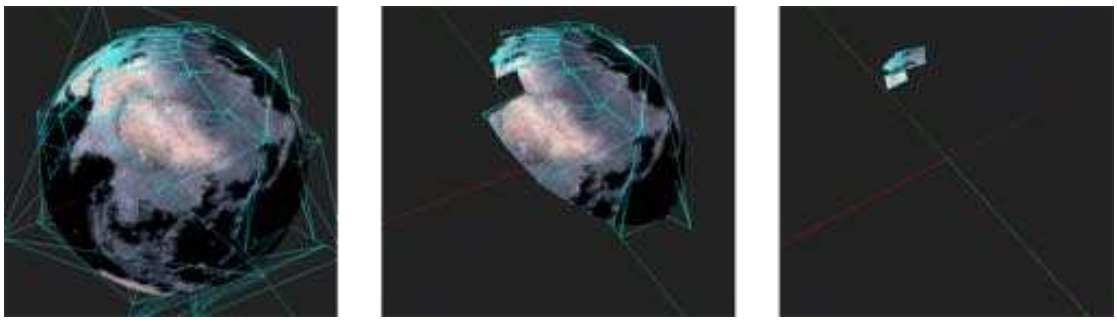

FIGURE 5.3: Culling implementations using a camera at $[lon, lat] = [45^\circ, 45^\circ]$ and approx 1% of sphere radius above the surface, facing towards the north pole. From the left; no culling, frustum culling, frustum + horizon culling.

5.4 Call stack dissection

The implementation has been profiled using the Chrome 41.0 profiler on the MBP15 device. Profiling gives several hints to the performance impact of individual function calls in the application. Table 5.3 shows a profile snapshot of the six heaviest operations from running the camera test path in section 5.2.

Operation	**% of processing time**
draw	22.56%
gl texture upload	17.24%
geometry tessellation	3.94%
bounding box projection	3.49%
quadtree traversal	2.02%
frustum culling	1.18%

TABLE 5.3: Function call performance.

Chapter 6

Discussion and future work

6.1 Visual quality

Visual quality of the final renderings is what can be expected from plain HLOD. Probably the most important goal for any visualization is to accurately convey information to users and the implementation does a good job of just displaying the plain WMS and TMS data. However, there are many things that could be improved in regard to visual quality.

6.1.1 Swapping

Chapter 4 decribes the generation and selection processes from section 3.1 but *swapping* has been left out. Not handling swapping results in popping artifacts when a surface patch swaps between different heightmap resolutions. This can be mitigated with temporal interpolation between old and new heightmap values. The reason for not adding this morphing behaviour is that performance has been the focus for the implementation and, as seen in the tables above and discussed below, performance is not yet at a satisfactory level.

Geometry Clipmaps is probably the best terrain rendering technique to deal with inter-resolution morphing simply because the entire method is based in blending different textures together. Another technique, used by Google Earth, is to apply a blurring depth of field shader to surface patches that are currently loading more high resolution data.

6.1.2 Surface shading

Surface normal calculations, atmosphere scattering, specular reflections and other visual effects has not been implemented. This would certainly improve the visual result but has not been part of the scope in this thesis project.

6.2 Performance

As seen in table 5.1, performance is not yet on par with the expectations stated in section 1.3.1. This is partially due to the devices, especially iPad 3, lacking rendering performance. However, referring to table 5.3, the `draw` and `texture upload` operations are responsible for more than a third of the application's processing time. There *are* many draw calls in the implementation as seen in 5.2 but with batch rendering and/or instancing (as in WebGL 2.0) it should be possible to improve this metric. The number of texture uploads could be improved by better caching, in particular a GPU texture cache could be implemented to increase performance.

Note, however, that the camera moves relatively fast in the tests above. Normal (human) use of an application integrating this library will, on average, experience higher framerates than the ones reported in table 5.1.

6.3 Latency workarounds and pre-fetching

As stated in section 4.7 textures are loaded asynchronously from the simulation/render loop. The latency in the texture requests means that textures will not be directly available for rendering. This could be improved by trying to predict and pre-fetch textures once the important textures, i.e. the ones rendered on screen, are loaded. Such a predcition algorithm can be based in both the quadtree (fetching textures from, say, a level below the current leafs) and a camera velocity vector. When implementing pre-fetching, it is also possible to sort tile requests by order of importance. Cozzi and Ring [3] suggest using a request queue, similar to the texture cache in section 4.8, for this.

Some experiments has also been conducted with limiting the number of concurrent requests. This may sound like a strange optimization, but there are two reasons for it. First, most browsers will throttle http-requests when there are too many concurrent requests. Chromium, for example, has a built in anti-DDoS throttling mechanism that does this. In most browsers it can be turned off, but that means users have to manually flip a flag in their browsers. Second, limiting the number of concurrent requests has an impact on request *order*. Enforcing a limit of, say,

10 concurrent requests will force the `tileProvider`:s to retry ”failed” requests and thus, the most frequently requested/re-tried tiles will be loaded first. Limiting requests may be especially useful when the camera is close to the surface and moves fast. In such a case, requesting very detailed textures is redundant because their corresponding surface patches will already be out of view when the tiles have finished loading. For this to work well it would be necessary to refactor the quadtree traversal to be breadth first, i.e. requesting large tiles before smaller. This refactor may have benefits in the current implementations as well.

6.4 Scale

The scale challenges mentioned in section 2.2 were not brought up in chapter 4. This is because implementing solutions for scale transforms should be *application wide*. While the chunked spheres in this thesis can be used as a complementary library to Three.js, it assumes that the user defines rules for scenes, cameras and renderers themselves. By letting the library expose modules that follows the Three.js conventions for `THREE.Object3D` and `THREE.Geometry` etc. it will be possible to implement appropriate solutions for the scale and precision problems such as Power Scaled Coordinates by Hanson et. al [26], octree-based relative to eye/center (RTE/RTC) rendering techniques and logarithmic depth buffers.

6.5 Terrain rendering techniques

6.5.1 Enabling Geometry Clipmaps

The two main reasons for not using Geometry Clipmaps are that the technique needs an API that allows very detailed control of the GPU and, more importantly, Geometry Clipmaps does not map well onto a sphere. This is unfortunate, because it is a very useful terrain rendering technique for flat base geometry. There are, however, techniques for enabling Geometry Clipmap rendering on spherical surfaces. Perhaps the most promising one is the spherical clipmap method proposed by Clasen and Hege [24]. Their implementation shows that Geometry Clipmaps can be used in out-of-core rendering of ellipsoidal geometry in interactive framerates. One drawback with their approach is the heavy use of trigonometric functions to transform the clipmap into spherical coordinates. This both has a high processing cost and has precision issues.

Another approach, based on *hexagonal* clipmaps as pesented by Bhattacharjee and Narayanan [25] may be a better fit to the use cases in Unitea. Bhattacharjee and Narayanan’s implementation use HTM tessellation and TOAST maps to achieve continuous level of detail on uniform spherical geometry.

It is hard to tell how well these two approaches integrate with out-of-core rendering since all implementations are very experimental. With out-of-core rendering and sphere mapping solved, Geometry Clipmaps is probably the most efficient of the planet terrain rendering techniques listed in section 3.1.

6.5.2 Improving the HLOD implementation

One of the highest priority issues of this thesis' implementation of HLOD is the fact that both terrain tiles and imagery tiles are tightly coupled to surface geometry. This is because the WMS tile data used at Sciss is split in the exact same way for both heightmaps and colormaps. The implementaiton assumes this relationship is always true, which it generally is not, regardless of data source.

Refactoring this coupling should be a top priority for future work on the library. It may be useful to align geometry to heightmap tiles, but the imagery tiles need to be sampled independently from the surface patch bounds.

Chapter 7

Conclusions

This thesis has described how to implement continuous level of detail for planetary terrain rendering in web browsers. The implementation supports out-of-core rendering of imagery and heightmaps. Integration into Unitea has been successful, but is not yet released. With the improvements proposed in section 6.2, it will hopefully be in use later this year.

Since performance in `draw` and `glTexture2D` are the bottlenecks of the application and these must be carried out by the main process, parallelizing with web workers should provide only limited speedups. Efforts should instead be focused on optimizing the current, single threaded, solution by implementing a GPU texture cache as proposed above (less GPU bus traffic) or rendering larger surface patches (fewer draw calls). Revisiting Geometry Clipmaps is also recommended since that technique has a more predictable render cost that should result in fewer framerate drops.

WebGL 2.0 will provide some tricks to improve performance by exposing more detailed control of the graphics pipeline. Things like batch rendering will be more efficient with instancing in the specification. The new uniform buffer objects will also, most likely, improve texture-to-gpu load times.

As discussed in chapter 6, HLOD may not be the most efficient terrain rendering algorithm for the purpose of planet rendering in web browsers. However, it is an intuitive approach that produces good visual results and can adapt to different performance and accuracy requirements. According to Ring [3], most commercial virtual globes use HLOD, but we have yet to see an implementation for mobile devices running in interactive framerates in WebGL.

Bibliography

[1] R. "Mr.doob" Cabello, *Three.js*. Available from: <http://threejs.org/> [7 April 2015].

[2] T. Ulrich 2000, *Rendering Massive Terrains using Chunked Level of Detail Control*, tutorial notes for: *Super-size it! Scaling up to Massive Virtual Worlds*, In SIGGRAPH Course Notes, Vol. 3 (published 2002)

[3] P. Cozzi & K. Ring 2011, *3D Engine Design for Virtual Globes*. A K Peters/CRC Press.

[4] World Wide Web Consortium, *HTML5 specification*. Available from: <http://www.w3.org/TR/html5/> [8 April 2015].

[5] caniuse.com, *WebGL browser support*. Available from: <http://caniuse.com/webgl> [8 April 2015].

[6] The Khronos Group, *WebGL Meetup at GDC 2015*, Recording available from: <https://youtu.be/zPNM3yOsP0I> [9 April 2015].

[7] M. Garland and P. Heckbert. *Surface simplification using quadric error metrics*. SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques, pp. 209–216.

[8] M. Duchaineau, M. Wolinsky, D. Sigeti, M. Miller, C. Aldrich, M. Mineev-Weinstein 1997, *ROAMing terrain: real-time optimally adapting meshes*. VIS '97, Proceedings of the 8th conference on Visualization '97, pp. 81-88.

[9] F. Losasso & H. Hoppe 2004, *Geometry clipmaps: terrain rendering using nested regular grids*. ACM Trans. on Graphics (SIGGRAPH), vol. 23, no. 3.

[10] A. Asirvatham & H. Hoppe 2005, *Terrain Rendering Using GPU-Based Geometry Clipmaps*. GPU Gems, vol. 2, pp. 27-45

[11] C. Tanner, C. Migdal & M. Jones 1998, *The Clipmap: A Virtual Mipmap*. ACM SIGGRAPH, Proceedings, pp. 151-158.

[12] C. Dick , J. Krüger , R. Westermann 2009, *GPU Ray-Casting for Scalable Terrain Rendering*. Proceedings of Eurographics 2009

[13] Microsoft Corporation, *WorldWide Telescope Projection Reference*. Available from: <http://www.worldwidetelescope.org/docs/worldwidetelescopeprojectionreference.html> [4 May 2015].

[14] International Association of Oil & Gas Producers, *EPSG Coordinate Systems definition and reference*. Available from: <http://www.epsg.org/> [20 April 2015].

[15] Open Geospatial Consortium, *OpenGIS web map service (wms) implementation specification*. Available from: <http://www.opengeospatial.org/standards/wms> [20 April 2015].

[16] OnTerra Systems LLC 2009, *OnTerra Web Map Service*. Available from: <http://register.onterrasystems.com/wms.aspx> [20 April 2015].

[17] Open Source Geospatial Foundation, *OSGeo Tile Map Service (tms) unofficial specification*. Available from: <http://wiki.osgeo.org/wiki/Tile_Map_Service_Specification> [20 April 2015].

[18] Mapbox Inc. *Mapbox* Available from: <https://www.mapbox.com/> [20 April 2015].

[19] I. Sutherland, G. Hodgman, *Reentrant Polygon Clipping*. Communications of the ACM, vol. 17, pp. 32–42, 1974.

[20] K. Ring 2013, *Horizon Culling*. Available from: <https://cesiumjs.org/2013/04/25/Horizon-culling/> [24 April 2015].

[21] D. Hultgren 2012, *Evaluation of Page Replacement Algorithms in a Geographic Information System*. Master's thesis, Royal Institute of Technology.

[22] K. Ring 2013, *Rendering the Whole Wide World on the World Wide Web*. Guest lecture in CIS 565: GPU Programming and Architecture. University of Pennsylvania 12/2013.

[23] K. Drangoor, *CommonJS specification*. Available from: <http://wiki.commonjs.org/> [29 April 2015].

[24] M. Clasen & H-C. Hege 2006, *Terrain Rendering using Spherical Clipmaps*. Eurographics/IEEE-VGTC Symposium on Visualization.

[25] S. Bhattacharjee & P. J. Narayanan 2008, *Hexagonal Geometry Clipmaps for Spherical Terrain Rendering*. SIGGRAPH Asia, poster.

[26] A. Hanson, C. Fu, E. Wernert 2000, *Very Large Scale Visualization Methods for Astrophysical Data*. Eurographics 2000, pp 115-124.