TU WIEN Informatics

Planetary Rendering with Mesh Shaders

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Ing. Martin Rumpelnik
Matrikelnummer 01633397

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr. techn. Michael Wimmer
Mitwirkung: Univ. Ass. Dipl.-Ing. Dr. techn. Bernhard Kerbl, BSc

Wien, 24. Februar 2020

Martin Rumpelnik | Michael Wimmer

Informatics

Planetary Rendering with Mesh Shaders

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Ing. Martin Rumpelnik
Registration Number 01633397

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr. techn. Michael Wimmer
Assistance: Univ. Ass. Dipl.-Ing. Dr. techn. Bernhard Kerbl, BSc

Vienna, 24th February, 2020

Martin Rumpelnik   Michael Wimmer

Erklärung zur Verfassung der Arbeit

Ing. Martin Rumpelnik

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 24. Februar 2020

Martin Rumpelnik

Acknowledgements

First and foremost, I want to thank Univ.Ass. Dipl.-Ing. Dr.techn. Bernhard Kerbl, BSc for supporting me during every part of this thesis. Thanks for all the useful tips, corrections and help to create a higher quality result.

I also want to thank Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr. techn. Michael Wimmer for helping me find this topic and my family for supporting me during my studies.

Kurzfassung

Planetare Rendering-Lösungen leiden häufig unter Artefakten oder hohen Renderzeiten, wenn sehr große Landschaften gerendert werden. In dieser Arbeit stellen wir eine Methode vor, die auf Echtzeitanwendungen ausgelegt ist und daher hohe Leistung erzielen soll. Die Methode kann mit einer beliebigen Detailgenauigkeit angewendet werden, was eine stabile Renderzeit unter Hardwareeinschränkungen ermöglicht. Im Gegensatz zu bestehenden Methoden wie Quadtrees und Clipmaps, vermeidet unsere Methode Artefakte, wie plötzliches Aufpoppen von Details oder schwimmend wirkende Landschaften, so weit wie möglich. Die Methode übermittelt rechteckige Bereiche von Zellen um den Betrachter an die neue Geometrie-Pipeline von NVidia, die mit der Turing-Architektur eingeführt wurde. Aufgrund der Funktionen der neuen Pipeline können wir effiziente Entscheidungen bezüglich der Detailgenauigkeit auf der Grafikkarte treffen und höher aufgelöste, kreisförmige Bereiche aus rechteckigen erstellen. Die kreisförmigen Bereiche bieten dem Betrachter eine gleichmäßige Auflösung der Landschaft in alle Richtungen, während niedrige Renderzeiten beibehalten werden.

Abstract

Planetary rendering solutions often suffer from artifacts or low performance when rendering very big terrains with high details. In this thesis, we present a method that targets real-time applications and therefore aims to achieve high performance. The method can be applied with an arbitrary amount of detail, which enables stable performance under runtime or hardware restriction. In contrast to existing methods, like quadtrees and clipmaps, our method avoids artifacts, such as popping or swimming, as much as possible. The method submits coarse, rectangular regions of cells around the viewer to NVidia's new geometry pipeline that was introduced with their Turing Architecture. Due to the capabilities of the new pipeline, we can make efficient level-of-detail decisions on the graphics processing unit (GPU) and produce work to create circular regions from the rectangular ones. These circular regions provide uniform terrain resolution for the viewer in all directions, while maintaining low rendering times.

Contents

CHAPTER 1

Introduction

Planetary rendering is used in various fields and applications that aim to convey contextual or immersive information at the scope of entire worlds. Examples include mapping services, animations, or games that span extensive terrains, such as open-world exploration or flight simulators [CR11]. These applications usually aim to give users a way to navigate planets in real-time and enable observating them from different view points. Games may sometimes be able to use artistic tricks, like fog in the distance to hide sudden terrain changes or geometry reloading when navigating the planet. However, this is not applicable or desired for all fields. Ideally, we want a universal method for rendering the planet that works at all scales, so that any transition from ground to space or vice-versa is visually seamless. In this thesis, we present and evaluate a new method that utilizes current hardware features to efficiently render planet-scale terrains, like the ones shown in Figure 1.1 in real-time, while avoiding noticeable visual artifacts as much as possible.

1.1 Motivation

Real-time planetary rendering solutions always face the challenge to provide vast, high-detail terrains, while maintaining a high frame rate and avoiding swimming or popping artifacts. When present, these artifacts usually manifest as the viewer moves the camera and are particularly distracting when larger triangles are used to represent the planetary terrain. Our goal is a smooth transition between different level-of-detail terrain representations when moving from outer space to the planet's surface and back. Multiple quadtrees projected onto a sphere are a popular approach for rendering large terrains and used by games such as Kerbal Space Program [Ksp] and Elite Dangerous [Eli], as well as in planetary rendering frameworks like Proland [Pro]. However, this method implies highly detailed terrain results in traversing large quadtrees and executing GPU commands for every visible leaf, which can quickly become a bottleneck.

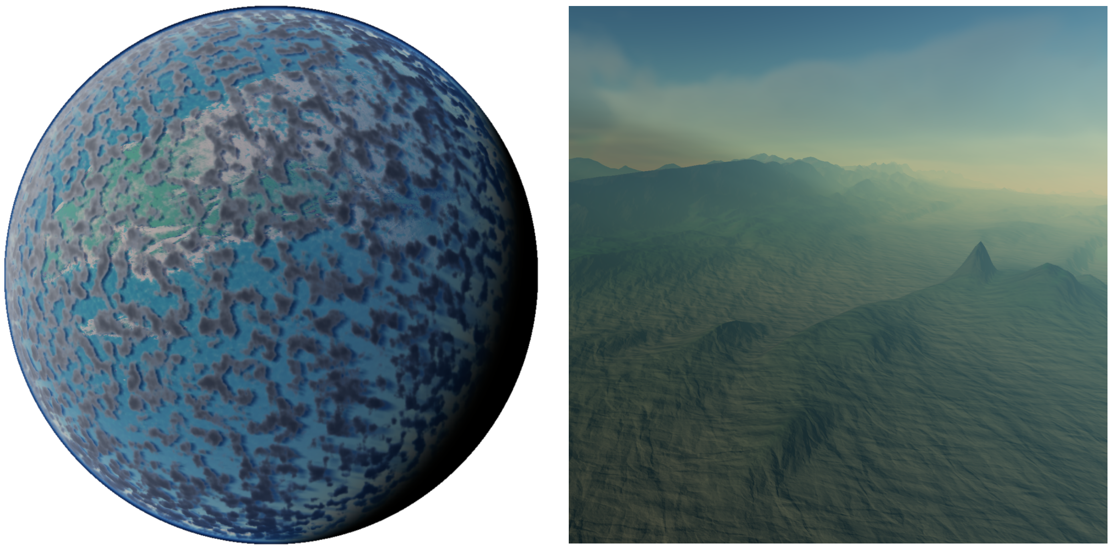


Figure 1.1: Example of an entire planet rendered from space on the left and from the ground on the right. Ideally, the transition from one to the other and vice-versa is seamless and requires no switching of geometry.

With increasing demand for more visual quality, graphic card vendors try to find new ways to increase the capabilities of their hardware. In 2018, NVidia introduced their Turing Architecture, which features a new programmable geometry shading pipeline among its features. The capabilities of this new pipeline make the Turing architecture a unique modality for exploring new techniques and mechanisms that we assume to be usable for planetary rendering.

1.2 The NVidia Turing Architecture

NVidia's Turing architecture [nvi] introduces a new geometry pipeline that offers a compute shader-like programming model to compute and output all necessary information to render meshes with multiple, collaborative threads. The pipeline consists of two stages: task shader and mesh shader. In contrast to compute shaders, task and mesh shaders are part of the graphics pipeline and thus can be executed as part of a conventional rendering pass. The hardware can therefore manage all memory that is passed between the geometry stages and it can be kept on-chip.

According to the Turing architecture specification, task shaders operate similarly to the control stage of tessellation shaders in that they are able to dynamically generate work. Task shaders use a cooperative threading model and instead of taking a patch as input, inputs are user-defined. Their output is passed to mesh shader workgroups in the mesh shader stage (see Figure 1.2). Outputs to the mesh shader are also user-defined as opposed to the fixed conventions that apply to tessellation control shaders, which

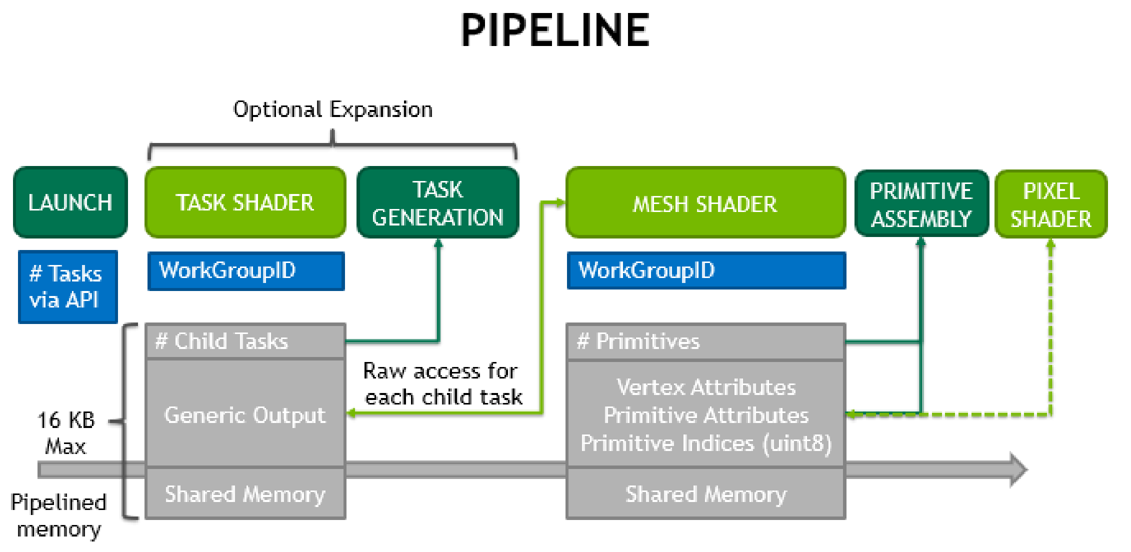


Figure 1.2: The new geometry pipeline introduced with the Turing architecture by NVidia. Task shaders are submitted via a draw call and execute in parallel. Each task shader has access to all user-defined inputs and outputs. The task shader stage is optional if no pre-processing, like early primitive culling, is needed. Mesh shaders are either spawned by task shaders, or draw calls if task shaders are not used. They also execute in parallel and have access to all user-defined inputs and outputs. The mesh shader stage is not optional and generates primitives for the rasterizer. Data passed between both stages is managed by the hardware and kept on-chip. [nvi]

output tessellation descriptors. Mesh shaders produce triangles for the rasterizer and can define work groups that employ multiple collaborative threads to execute their workload. As with task shaders, inputs and outputs are completely customizable. This two-stage approach enables many handy use cases, like early culling and level-of-detail determination on the GPU, directly forwarding results to the rasterizer for rendering[nvi].

1.3 Goal of this Thesis

The goal for this thesis is to present a new method that achieves high performance and enables rendering planetary scale terrain in real-time. Our proposed method uses new Turing architecture features to minimize overhead caused by CPU-GPU communication and exploits the massive parallelism of GPUs. In addition, our method is designed to be scalable and supports adaptive quality settings to, e.g., guarantee frame rates on weaker hardware while minimizing visual artifacts. Furthermore, we also economize on geometry processing cost by enforcing different level-of-detail visualizations based on the viewer's height and generate uniform terrain mesh resolution in all directions around the viewer.

CHAPTER 2

State of the Art

There are multiple methods to approach planetary rendering. In this chapter, we want to give a short introduction to quadtrees and clipmaps as well as an overview of less common specializations, such as spherical clipmaps, isocaeder rendering and projective grid mapping for planetary terrains. For clipmaps, we want to focus on the approach by Asirvatham and Hoppe which is a GPU implementation of geometry clipmaps, originated by Losasso and Hope [LH04]. For the terrain quadtree, we will generally refer to the implementation of the rendering research framework "Proland" [Pro]. Our focus on conventional clipmaps and terrain quadtrees is motivated by the fact, that our proposed method shares several similarities with these methods.

2.1 Terrain Quadtree

Terrain quadtrees are used in games such as Kerbal Space Program [Ksp] and Elite Dangerous [Eli], as well as terrain visualization systems such as the Virtual Geographic Information System (VGIS) [KLR$^+$95] and Virtual Reality and Geoinformation System (ViRGIS) [Paj98a] and frameworks like Proland [Pro]. The idea behind using a quadtree as a terrain representation is its adaptive subdivision capability. Every leaf of the tree is rendered as a the grid mesh, so the more a quad is subdivided, the more dense the grid mesh becomes. Height data is applied to all vertices of the subdivided tree by sampling height values from a heightmap, using the position of the vertices. Higher subdivided areas have more vertices and therefore more sample points to provide a higher resolution part of the terrain. The terrains for games are often procedurally generated or modelled by an artist, while visualization systems use real-world height data of actual planets, e.g., the Earth. Since the fast random access memory (RAM) of both the CPU and GPU is limited, terrain data is often preprocessed and streamed in during runtime. For example, VGIS maintains all quadtree levels partitioned as small, fixed-size patches on disk. This leads to duplicated height values on multiple levels due to subsampling of the initial heightmap to create the quadtree. Based on the viewer's orientation to the surface, the visible patches and the resolution needed is determined. Only the fixed-size patches from disk can be accessed and rendered, no further LoD generation is supported. In contrast, ViRGIS keeps a fraction of its entire data in main memory and uses a tiling approach to construct the visible parts of the terrain while the viewer navigates over the terrain. If necessary, adaptive LoD generation on each frame is possible.

Proland also uses a tiling approach and caches loaded tiles for reuse if the viewer moves back to a recently visited location. Unused tiles are evicted from the cache if space runs out and new tiles need to be loaded in. The quadtree is dynamically subdivided, based on the viewer's position projected onto a $2D$ grid to provide more details near the viewer. The distance d from a quad with side length L and (x_q, y_q) for its lower left corner to the viewer (x, y) is calculated as $max(dx, dy)$. The subdivision only happens if this distance d is less than $k \times L$, therefore the height of the actual terrain data does not influence the subdivision. The factor k is called the split distance factor and can be used to influence how soon the subdivision happens. An Example of a k-influenced subdivision can be seen in Figure 2.1. If k is chosen to be greater than 1, we speak of a "restricted quadtree". The original restricted quadtree method was proposed by Von Herzen [VHB87] to triangulate parametric surfaces and later applied to terrain representation by Pajarola [Paj98b] and Lindstrom [LKR$^+$96]. In a restricted quadtree, the difference between the level of two neighbor quads is always 0 or 1. In Proland, the final height values to be applied to the vertices of the subdivided quadtree are sampled on the GPU from a height texture. [Pro].

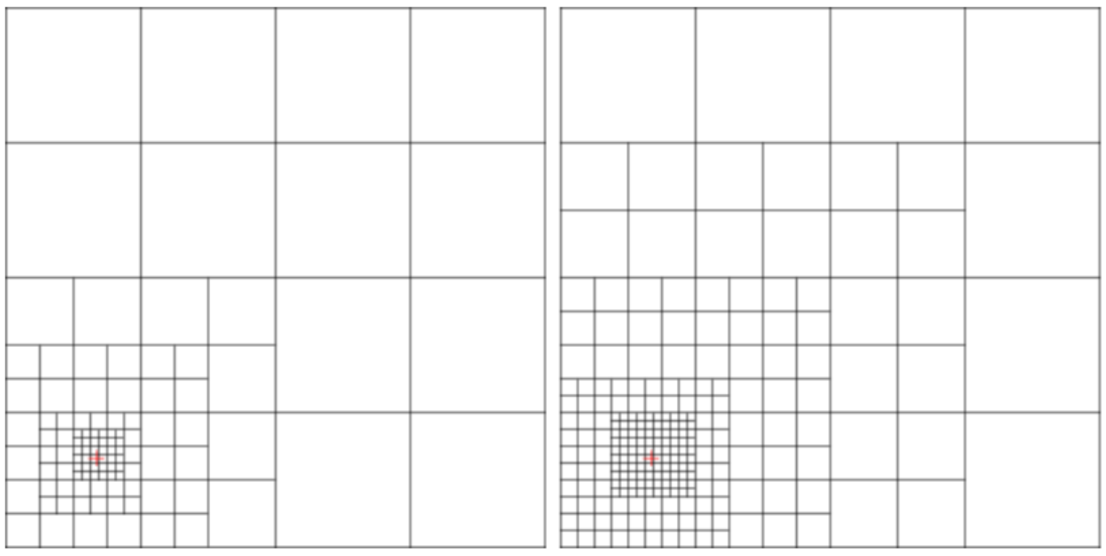


Figure 2.1: A terrain quadtree is subdivided based on the viewer's position (red cross) relative to the terrain. The subdivision of a quad occurs if the distance is smaller than k times its size. The factor k influences how soon the subdivision happens. This example shows the subdivision with $k = 1.1$ on the left and $k = 2.0$ on the right.[Pro]

2.1.1 Planetary Terrain Deformation

Since the quadtree itself is only two-dimensional, a deformation into an actual sphere is needed. In Proland, the planet is composed of six terrain quadtrees that represent the six sides of a cube. Every quadtree can be updated by itself and is then transformed from local space where $z = 0$ to a deformed space, so that each point in the local space corresponds to a point on the sphere as shown in Figure 2.2. The cube is of size $2R * 2R * 2R$, where R is the radius of the sphere.

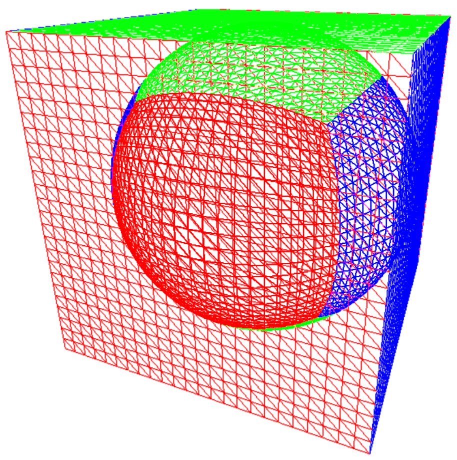


Figure 2.2: With spherical deformation, a planet can be created by deforming the 6 sides of a cube. Each of the sides represents a terrain quadtree.[Pro]

To transform a point $\vec{p} = (x, y, z)$ in local space to a deformed point q in deformed space, we need to first construct a Point $\vec{p'} = (x, y, R)$ on the top or "north" face of the cube (green in Figure 2.2). The point q is then defined as

$$q = (R + z) * P/||P||$$

This deformation also holds for the other terrains, except that the point also needs to be rotated to the corresponding cube face by rotating in multiples of 90° around X, Y and Z axes.

2.2 Clipmaps

Clipmaps are nested regular grids that are shifted as the viewer moves. Terrain height values are sampled from a pre-filtered mipmap pyramid of L levels, as shown in Figure 2.3. For complex terrains, the full pyramid can be too large to fit into memory, so only a square "window" of at most $N \times N$ samples, where N is the desired resolution, within each level is cached. These windows are centered at the viewer's position, as shown in

Figure 2.4. Only the finest level is a complete grid, while the other levels form "rings", since the center portion is already filled by finer resolution levels. If the viewer moves, the rings are shifted and updated with new data. The challenge this method brings is to hide popping artifacts due to boundaries between two neighboring resolution levels. To address this, a transition region near the border to the next level is introduced. At these borders, the geometry and textures are smoothly morphed to the next level by interpolation [PF05].

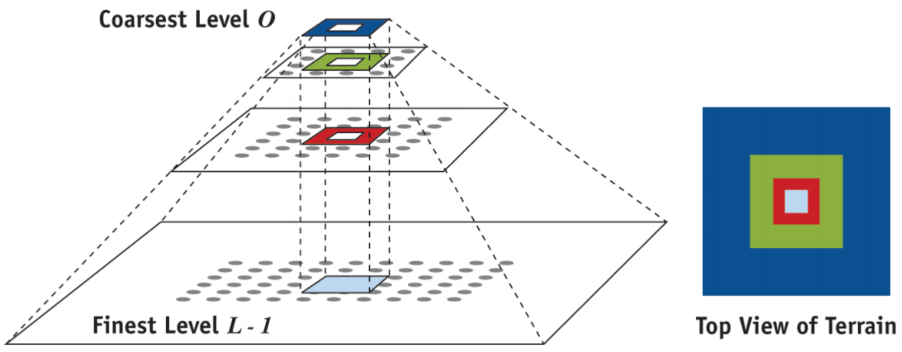


Figure 2.3: Clipmaps represent a square window from a pre-filtered pyramid of different levels. Depending on the viewer's distance to the ground, a set of these nested rings is used and centered at the viewer's position. Only the finest level is a full quad, the other levels leave a hole which is filled by finer resolutions.[PF05]

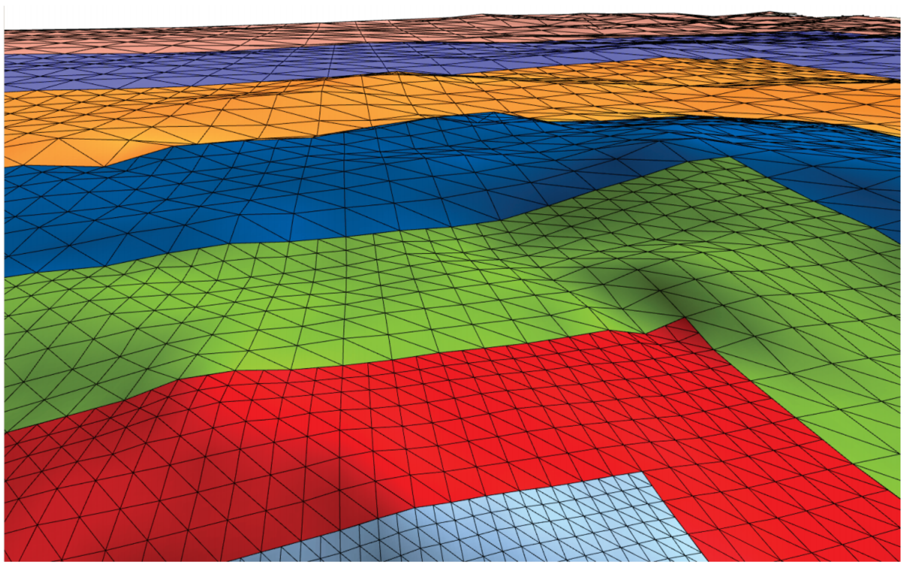


Figure 2.4: Example of a terrain rendered using clipmaps. Every colored ring represents a different clipmap level k where the size of the grid is 2^k. The further away the ring, the coarser the resolution gets.[PF05]

Depending on the viewer's height above the ground, a set of active levels are selected from all L levels for updating and rendering. This is to avoid high-frequency artifacts that are caused by oversampling the finest level if the viewer is far away. The mesh of each ring is split into multiple rectangles that are displaced in the vertex shader to form the ring. Processing these rectangles separately allows for frustum culling if a ring is not completely visible and saves memory because only small geometry patches need to be stored. The height values for each level are stored in single-channel floating point textures. In transition regions, height displacements are interpolated between the values from the next finer and next coarser level [PF05].

2.3 Spherical Clipmaps

Spherical clipmaps are a three-dimensional extension to the basic clipmaps approach. Instead of rendering concentric, rectangular grids, the main idea of spherical clipmaps is to take the viewer's position on the planet and render "rings" with different level of detail around it. The finest resolution is at the viewer's position and gets coarser, the further away the ring is. However, instead of rendering rectangular grids, actual circular rings are used. The idea behind this is to always have a consistent amount of detail surrounding the viewer at all times and in all viewing directions (see Figure 2.5).

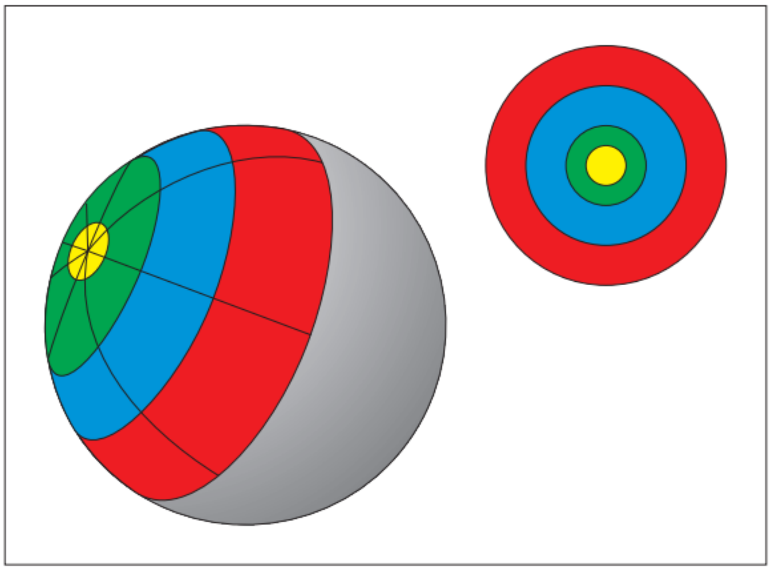


Figure 2.5: Spherical Clipmaps generate circular rings instead of rectangular rings that cover roughly one hemisphere. The method partially relies on the fact that a viewer can never see more than one half of a sphere.[CH06]

To render the spherical clipmaps, the hemisphere centered at the viewer's position is discretized into quads which are then further partitioned into triangle pairs. Spherical coordinates are then used to sample the height values from textures. Compared to clipmaps, this method comes with the advantage that the resulting geometry will be gapless, hence no T-junction prevention method is necessary, because the constant discretization implies a seamless geometry transition. The disadvantage of this method is that there is no direct correspondence between the height map entries and the vertices. Depending on the sampling rate for the different pre-filtered heightmaps, aliasing can occur if the heightmap is resampled at a different rate by the underlying geometry, leading to severe swimming artifacts. Values inside the triangles of the geometry are interpolated, which is bad for good reconstruction of the original height map. Matching the source sample rate means, finer geometry is needed. Lowering the source sample rate to match the geometry means lower visual details of the terrain [CH06].

2.4 Icosaeder Rendering

Icosaeder rendering as proposed by Kooima *et al.* [KLJ$^+$09] uses a low-poly icosaeder to produce a spherical surface by recursive tessellation. The method consists of three phases. First, the visibility and level-of-detail determination with a coarse pre-subdivision is done on the CPU. The pre-processed mesh is then passed to the GPU to do a finer subdivision and generate the final geometry to render.

The CPU produces a coarse triangulation of the visible portion of the sphere. It starts with an icosahedron and does recursive subdivision until a coarse triangulated sphere is produced as seen in Figure 2.6. The icosahedron is chosen as a base polygon because of the uniformity of its triangulation, as it is the most complex of the Platonic solids. From this coarse sphere, triangles are then added to a tree if they are inside the view frustum. As the view changes from frame to frame, triangles are added to or removed from the tree. The geometry generation then takes the coarse triangulation tree from the CPU and produces a high resolution triangulation on the GPU by further recursive subdivision to accurately represent the terrain via height maps. Finally, the geometry is shaded using deferred texturing.

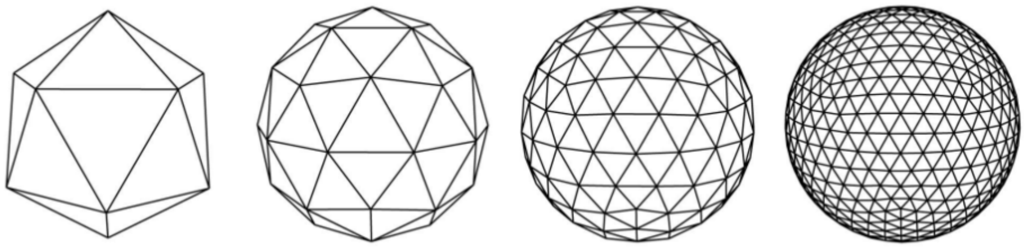


Figure 2.6: Recursive tessellation of a low-poly icosaeder on the left to produce the final subdivided, spherical surface on the right.[KLJ$^+$09]

2.5 Projective Grid Mapping

Projective Grid Mapping for planetary terrain is a hybrid technique that uses ray casting and rasterization to create a view-dependent mesh in GPU memory. It was introduced by Mahsman *et al.* [Mah10] because it allows for vertex creation on the GPU, hence there are no vertex transfers between CPU and GPU. The method also comes with gradual detail transition when moving away from the viewer, as it is inherently view-dependent.

A regular grid of points is stretched across the viewport and every point is projected onto the terrain along its eye ray. The intersection point of the ray and the terrain is then used as a target location in a height map. After retrieving the height from the height map, the point is vertically displaced along the base plane normal as seen in Figure 2.7. The resulting displaced grid can then be submitted as a triangle mesh for rasterization [Mah10]. While generally artifact-free, depending on the grid and screen size, this method has image-order complexity and is likely to be too inefficient for future higher-resolution displays (>8k).

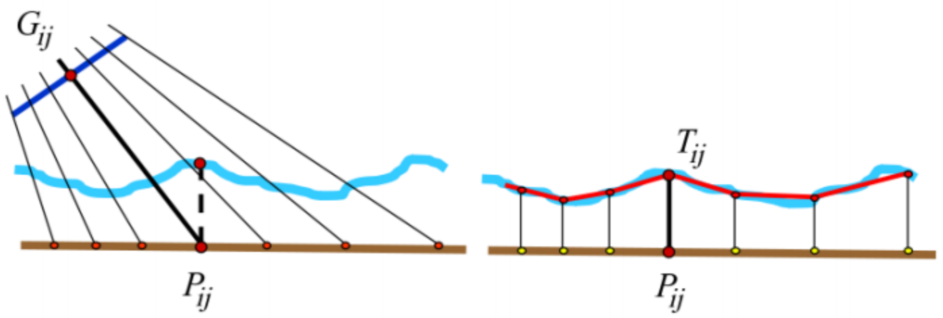


Figure 2.7: A point G_{ij} from a regular grid is projected through the viewport (dark blue) along its eye ray onto the base plane (brown). The intersection point P_{ij} is used to retrieve the height value from a height map to vertically displace P_{ij} from the base plane (brown) to its final position T_{ij}.[Mah10]

CHAPTER 3

Methodology

The main requirement for our method is to achieve high performance because we target real-time applications and want to render planetary-scale terrains. To support a wide range of GPU models (including weaker models with limited geometry processing performance), the resolution of the terrain needs to be parameterizable, while avoiding artifacts as much as possible. The proposed method is mainly inspired by Proland's [Pro] quadtree method for planetary rendering and Graphics Processing Unit (GPU) GPU clipmaps [PF05]. As with Proland, we maintain six seperate terrains and treat them as the six sides of a cube. However, we do not want to use quadtrees to represent the terrains, since they are held and maintained on the CPU. We aim to avoid the cost of updating and rendering this adaptive spatial structure by leveraging GPU parallelism to minimize CPU-GPU communication. To support level-of-detail (LoD) rendering, we also want to produce nested concentric regions, similar to clipmaps. Instead of rectangular regions however, we aim to produce round regions to have uniform terrain detail in all directions.

3.1 Challenges

Spherical clipmaps would give us circular regions, but cause sever swimming artifacts, i.e., geometry that moves with the viewer. Even when using snapping, the geometry moves in steps as big as the side length of a grid cell, so a moved vertex snaps to the next position and usually lands in a position where no other vertex was before relocation. Hence, vertices are moving the geometry does not perfectly line up when moving from frame to frame. To avoid this, sampling with high resolution geometry equal to the area of each pixel projected into world space is possible, similar to projective grid mapping [Mah10]. But this means that rendering is bound to high density sampling, which can get slow and therefore reduce applicability for real-time. To avoid swimming artifacts, icosahedron rendering [KLJ$^+$09] could be used, since the mesh is clearly defined at arbitrary levels of detail over the entire sphere and does not move. However, the implied recursive subdivision can become prohibitive to performance as the viewer moves closer and closer to the planet. We also cannot use uniform icosahedron subdivision, as we want to support different LODs as the viewer moves toward the planet or away from it. If we were to subdivide the whole icosahedron, we would have the same level of detail on every part of the sphere. This implies that even invisible portions of the sphere have the same resolution as visible ones, which would result in a large number of redundant shader calls for vertices that are not visible.

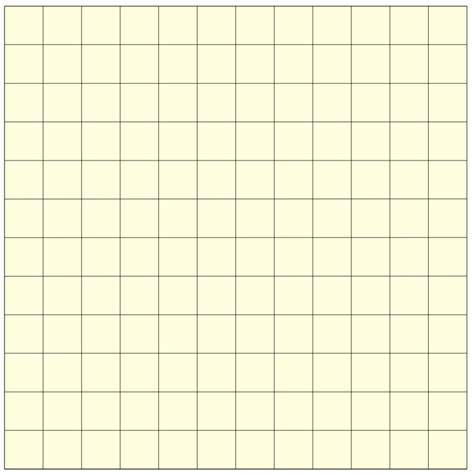
While cube or icosahedron subdivision via hardware tessellation seems like a potential solution for real-time planetary rendering, it is limited to 64 segments per edge of the base mesh. If we want to fly from outer space to the planet's surface, we will need more segments per edge to get a good amount of detail in every level. This limit can be overcome by using feedback loops, but that requires extra rendering passes, which take time. Another solution is to have fixed geometry like a grid and switch out tessellated geometry patches with more complex, pre-subdivided ones. The problem with this is, that hardware tessellation is not recursive. While the shape will stay the same, diagonals of quads in the base grid will be flipped if we combine patches with different tessellation level. This would produce visible artifacts as we introduce higher resolution quads [KB14].

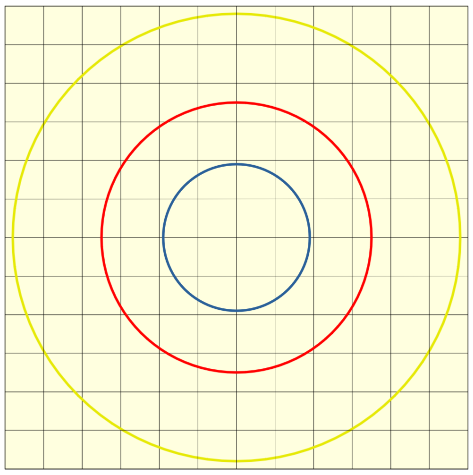
In order to arrive at a truly scalable, artifact-free method for planetary rendering, we introduce a new method that overcomes the above issues. In the following sections, we outline the main concepts of our approach and discuss how the individual steps map to the revised NVidia Turing geometry pipeline.

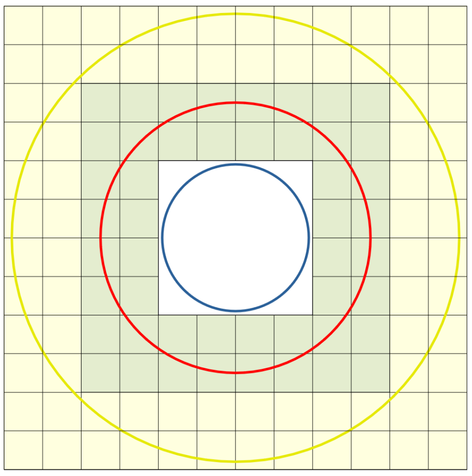
3.2 Concept

This section explains the main concept behind our method before we explain how the individual steps map to the hardware. For the underlying geometry of the terrain, we decided to stay away from tessellation and use a non-moving grid. To avoid mentioned artifacts with tessellation, we aim to keep the topology the same for every higher resolution quad we introduce. Note that the terrain grids on the six sides of the cube are independent, so for the rest of this section we will only look at one of the six terrains without loss of generalization. Similar to the quadtree method, we want to replace each grid cell with four smaller grid cells if the viewer moves towards the planet to get higher resolution near the viewer. As the cells below the viewer are replaced, concentric nested regions, as seen in 3.1c, are created. Every LoD level is always half the resolution of its next-inner one and vice versa. However, instead of rectangular regions, we want to insert and remove circular regions to maintain a uniform terrain resolution in all directions.

Consider the rectangular grid in Figure 3.1a. If a viewer moves towards the center of the grid, we need to subdivide it. In Figure 3.1b we determined three circles, centered at the middle of the grid, that we want to represent our LoD levels with. The yellow circle should cover the coarsest LoD level, while the blue circle should cover the finest LoD level. To create higher-detailed layers for each LoD level, we can mark cells that


(a) A rectangular grid with only a single LoD level.


(b) Circles are drawn that represent 3 LoD levels.


(c) The bounding area (green) for LoD level *L* (red) is determined.

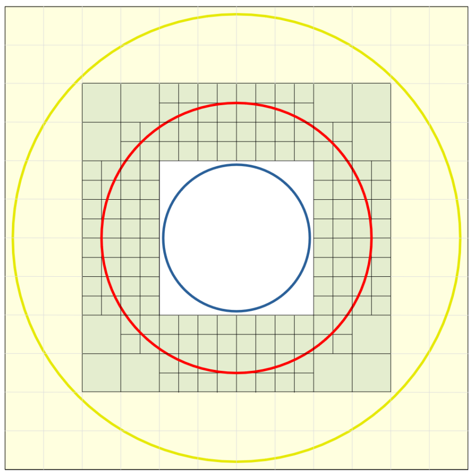
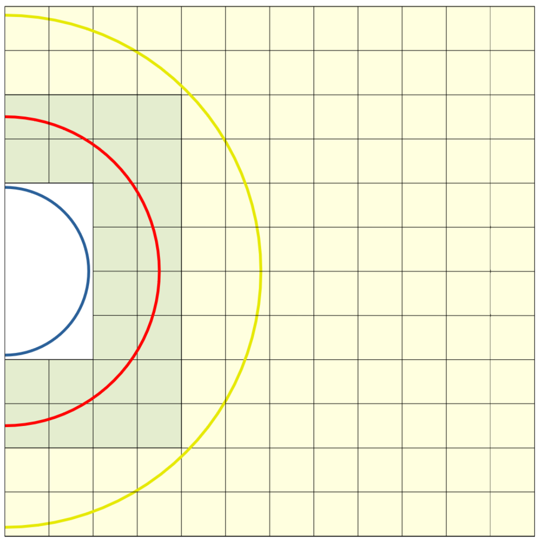

(d) Cells that overlap the circle of *L* (red) are subdivided.

Figure 3.1: Consider the rectangular grid in Figure 3.1a without LoD levels. Circles that represent LoD levels are drawn in Figure 3.1b as a viewer moves towards the center of the grid. The level of detail (LoD) decision is made based on the enclosing circle of each LoD level. To determine the resolution for LoD level *L* (red) in Figure 3.1c, a bounding area (green) is marked to avoid testing cells that clearly lie outside of the circle of *L*. If a cell in Figure 3.1c lies between the circle of *L* (red) and *L-1* (yellow), it remains as one cell. If it lies between the circle of *L* (red) and *L+1* (blue), it is subdivided into four smaller cells (double the resolution). Cells that lie in the circle of *L+1* (blue) and outside of *L-1* (yellow) are rejected.

overlap with the corresponding circle to produce circular regions. The finer the grid is, the more the marked cells will look like circles. To avoid testing every cell for every LoD, a bounding area for each LoD circle is created as seen in Figure 3.1c. The LoD tests of the cells are independent operations, so they are a perfect fit for being performed in parallel. We want to minimize CPU-GPU communication, so in an ideal case we can submit only as many cells as needed and keep them and their results from the LoD test in GPU memory for rendering.

To determine how many cells we need to submit, let us consider the red circle in Figure 3.1c as LoD level L. The green area by itself is a 8×8 subgrid, but we do not need to test all 64 cells as there is a 4×4 region inside of L (red), which is covered by a finer LoD $L{+}1$ (blue). Subtracting the inner 4×4 subgrid from the green 8×8 one equals the $64 - 16 = 48$ cells we actually need to submit to the GPU.

The number of cells to submit cannot be calculated once and reused later. It is possible that the count is different every frame because the viewer moves away from the initial location. If the viewer position changes, i.e., the viewer moves to towards the left side or towards the bottom right corner, parts of the LoD circle and its bounding area can be outside of the terrain (see Figure 3.2). These cut-off parts are not lost, but cover another adjacent side of the cube.


(a) The viewer moved to the left edge of the grid, which cuts off half of each LoD circle and also divides the cell count per LoD level by two.

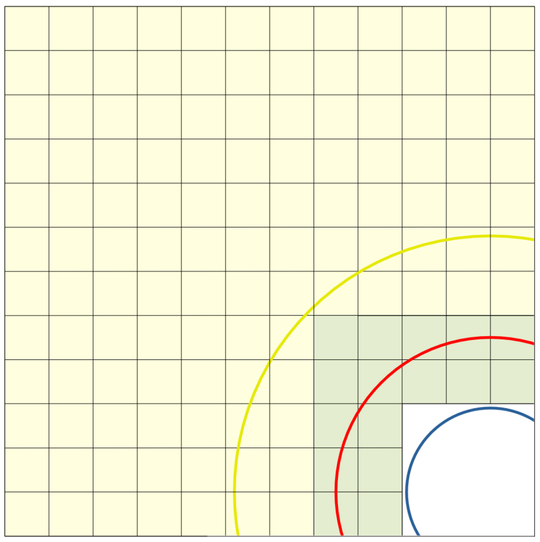

(b) The viewer moved to the bottom right corner of the grid, which removed roughly three quarters of each LoD circle and the number of cells considered for each one.

Figure 3.2: In Figure 3.2a the viewer moved to the left edge of the grid and in Figure 3.2b the viewer moved to the bottom right corner. In both cases, the LoD circles appear cut-off but continue on other adjacent sides of the cube.

In order to maximize parallelism and best exploit the capabilities of the GPU, every thread should handle and test one cell. Depending on the LoD, a choice about subdivision is made, hence generating dynamic workload: If a cell lies between the circle of *L* (red) and *L-1* (yellow), it remains one cell. If it lies between the circle of *L* (red) and *L+1* (blue), it is subdivided into four smaller cells, effectively doubling the resolution. Note that if the bounding area for *L* was chosen too large and we submitted too many cells, we need to reject the cells that lie in the circle of *(*blue) and outside of *L-1* (yellow). Figure 3.1c shows the result of all LoD tests for a given level *L*. All cells that got split into four smaller ones produce a circular region with double the resolution compared to the surrounding ones.

The results of the LoD test are passed on to add height and final shading to the terrain. Depending on the tests results, cells are either rejected, split into four smaller ones or stay the same. Therefore, for every LoD level, an arbitrary amount of work can be created for further processing. If we used compute shaders for the LoD test, we would now need to issue a separate render pass to actually render the terrain. With the Turing architecture, this is no longer necessary, since NVidia [nvi] introduced task and mesh shaders. Task shaders are able to dynamically generate work and invoke mesh shaders, which for their part produce triangles for the rasterizer. Both stages are integrated into the default graphics pipeline, so memory passed between both stages can be kept on-chip. These new stages are a perfect fit for our problem: Every frame, we need to work on an arbitrary number of cells to produce an adequate amount of detail for rendering the individual rings in our terrain mesh. Hence, the steps we need to perform for every LoD and every cube terrain side are as follows:

1. Level of Detail Bounding

   - Find a rectangular bound on the grid for the number of cells covered

   - Submit one task shader for each grid cell covered by the rectangular bound

2. Level of Detail Classification

   - Use the task shader to classify each cell based on the exact LoD ring overlap

   - Invoke either one, four or zero mesh shader workgroups for each input cell

3. Geometry Output

   - Output the actual geometry from the mesh shader

Figure 3.3 shows the result of four different circular LoD levels with the viewer at the center of the terrain. For the reminder of this chapter, we will look at the three main steps for our method in more detail.

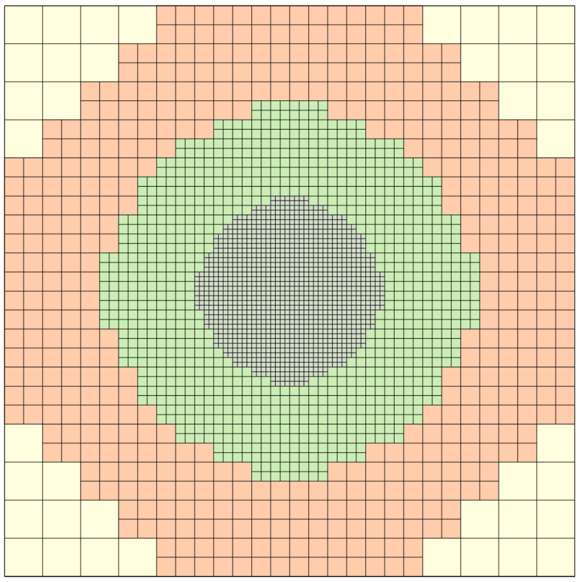


Figure 3.3: The result and desired output of four different circular LoD levels with the viewer at the center of one terrain. Each LoD level is exactly half the resolution of the next-inner one. As the grid gets finer with each level towards the center, the circular appearance is more and more visible.

3.3 Level of Detail Bounding

The final grid terrain consists of multiple concentric regions that represent the different LoD layers. Every inner LoD has double the resolution and is also exactly half the size of its next outer level. Compared to clipmaps we do not want to render rectangular but circular regions. Circular regions always have the same level of detail in all directions for a particular distance from the viewer, which is not true for rectangular regions. For every LoD layer, we will determine a rectangular bounding area that covers all grid cells of the circular region. This early grid cell exclusion avoids further processing of unnecessary grid cells, that clearly lie outside of a LoD layer.

3.3.1 Level of Detail

First we need to choose a minimum and maximum LoD level. The maximum represents the highest resolution LoD surrounding the viewer's position, while the minimum represents the most distant, barely visible lowest resolution layer. The values are chosen based on the distance from the viewer to the planet surface, to always have a desired resolution at all visible levels. The size of the LoD levels is based on a fixed, pre-determined radius R_0 at the maximum LoD level. Every radius R_L for a specific LoD level L is two times greater than the radius from the next-higher level. It can therefore always be calculated from R_0 and the current level L as:

$$R_L = R_0 * 2^L$$

3.3.2 LoD Grid Dimension

From the radius R_L, we can infer the grid dimensions D_L to render. First, we need to transform the viewer's position $\vec{v}$ and $\vec{v} + R_L$ to the local space of the grid with the transformation $\phi(x)$. Taking the difference of the resulting local positions gives us half the grid, so we need to double the result to get the grid dimensions that cover the circle of R_L. To get D_L, we add a one-cell-wide margin around the resulting dimensions, to account for the fact that the viewer moves on floating point coordinates and not in cell-sized steps.

$$D_L = ((\phi(\vec{v} + R_L) - \phi(\vec{v})) * 2) + 4$$

3.3.3 Shader Configuration

After computing the LoD grid dimensions, the dimensions and radius for each visible LoD layer can be set as input to the task shader. While the radius can be directly submitted, we need to first calculate how many cells per grid we need to submit. Every LoD layer except the innermost is a ring and not a full rectangle. Hence, for a 6×6 grid with 36 cells and an inner 3×3 grid with 9 cells, we only need to submit 27 cells. The starting position of the grid and its inner empty space can be calculated by centering it around the viewer position. If the starting position falls outside of the grid that covers a cube side, the dimensions of the LoD grid need to be clamped accordingly.

3.4 Level of Detail Classification

On the GPU, the task shader retrieves all the cells of each LoD ring as input and performs the LoD classification. For each LoD ring, we usually launch multiple task shader workgroups and each has 32 threads. To determine a cell to work on in each thread, we can use the task shader's built-in variables *WorkGroupID* and *InvocationID*.

$$\texttt{cell} = \texttt{WorkGroupID} * \texttt{GroupSize} + \texttt{InvocationID}$$

The `GroupSize` is a user-defined configuration for the task shader that tells it how many threads there are per workgroup. As we want to have as many threads as possible to work on cells, we choose to take the maximum of the first generation Turing cards of 32. The `WorkGroupID` indicates the active workgroup for this shader invocation. `ThreadID` is the id [0, 31] for each thread in the workgroup. Each thread per workgroup will work on one cell, so the total numbers of workgroups will be the number of cells submitted, divided by the `GroupSize`.

3.4.1 Cell Offset

For an unbroken grid, the cell to work on can be simply computed from the task shader's built-in parameters: `cell` = `WorkGroupID` ∗ `GroupSize` + `InvocationID`. This works for the highest resolution LoD, because it does not enclose another LoD. However, for every other LoD we render, we need to take the empty space reserved for enclosed, higher-resolution LoDs into account. If a determined cell lies in the reserved area, we need to shift it to the next position outside of this area, as can be seen in Figure 3.4.

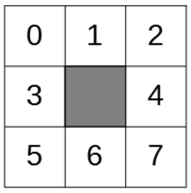


Figure 3.4: Cells are accessed in sequential order starting at zero. If there is a hole because of a higher resolution LoD layer, all cells need to be moved. Counting from left to right, the grey cell in the middle would have the index 4, but this cell will already be covered by a higher-resolution layer. Hence, all cells after cell 4 need to be shifted to the right by one cell. Therefore the 6^{th} cell has the index 4 in the sketch.

3.4.2 Level of Detail Decision

At this point, the remaining cells in every LoD ring form a rectangular grid. To create the circular detail falloff we want to achieve, we need to perform a few tests to determine which cells overlap exactly with the radius of LoD ring L and which fall outside. From the radius of L we can calculate the radius of level $L+1$, which is exactly half the size, as well as the radius for level $L-1$, which is twice the radius of level L. Figure 3.1c shows our cases we need to check. If a cell lies between the red circle (L) and the blue circle ($L+1$), we split the cell in four and start four mesh shader workgroups for this cell. If a cell lies between the red circle (L) and the outer yellow circle ($L-1$), the cell stays as it is and we spawn one mesh shader workgroup for processing. For cells that lie outside of the yellow circle ($L-1$) and inside of the blue circle ($L+1$), we do not start a mesh shader workgroup. One exception to the last case is the LoD level with the highest resolution, because it does not have a next-higher level LoD. We do not reject cells in this case or else we would create an unwanted hole at its center.

3.4.3 Frustum Culling

The lower-level LoD rings of the terrain can get very large and cells may land outside of the view frustum. View frustum culling of the cells can be performed in clip space by transforming each cell's corners with the view-projection matrix. While this implies that we must perform one matrix multiplication for each cell's corner, it helps to avoid unnecessary mesh shader submissions if cells lie outside of the frustum. Note that at this point, we have no information regarding the height data represented by the cell, since it will only be sampled in the mesh shader. To prevent culling too aggressively near the frustum, we therefore add a bias to extend the frustum by a small amount.

3.4.4 Submitting Mesh Shader Workgroups

After the granularity decisions on how many mesh shader workgroups all cells will spawn are made, we need to count their total number to parameterize the mesh shading stage. To this end, we use the built-in `subgroupBallot` and `subgroupBallotBitCount` operations to set and count the task shader invocations that will spawn mesh shader workgroups. Every task shader that spawns a mesh shader will set the bit in an unsigned integer mask ($Vote_1$ and $Vote_4$), which we then use for counting. We need to count task shaders that spawn one ($Count_1$) and task shaders that spawn four ($Count_4$) mesh shader workgroups separately. The count only tells us how many task shader threads want to spawn a mesh shader workgroup, so we need to multiply the result of $Count_4$ by four. Adding the final results of $Count_1$ and $Count_4$ gives us the total number of workgroups ($Count_T$) to spawn. These steps can be seen at the beginning of Algorithm 3.1. To pass the data to the next stage, we must pack it into an array to access it in the mesh shader via the workgroup id. We cannot use the cell index we used in the task shader as output data array index, because cells could have been moved and rejected during the last steps. Therefore, we sort the output data so that all threads that start four mesh shader workgroups write their data at the beginning of the output array and all threads that start only one mesh shader workgroup write their data after that. Algorithm 3.1 shows how the index Idx is calculated for the output array using `subgroup` operations. Figure 3.5 illustrates an example of the Idx determination for `ThreadID` 7, which wants to spawn 4 mesh shader workgroups, by using 12bit values for $Vote_4$ and the bit mask.

```
Vote4:     010011010100
 Mask:   & 000001111111
           ____________
           000001010100  => Idx = 3*4 = 12
```

Figure 3.5: The image shows an example of the Idx determination for 12bit values and the `ThreadID` 7, which wants to spawn 4 mesh shader workgroups. First the $Mask$ for the `ThreadID` 7 is created by shifting 1 to the 7^{th} position and subtracting 1. Using a logical `and` operation on $Vote_4$ and $Mask$ selects the set bits from the 6 lowest. Counting these bits and multiplying by 4 equals the write index to use for `ThreadID` 7.

3.5 Geometry Output

Finally, the mesh shader stage produces the actual triangle data which is forwarded to the rasterizer. As the user-defined input, we use the cell data output from the task shader. Mesh shader workgroups are sequentially numbered from zero to the number of workgroups launched by the task shader. Since we packed the cell data into a continuous array, we can directly use the built-in `WorkGroupID` to find the data to work on.

Algorithm 3.1: The algorithm first calculates how many mesh shader workgroups need to be spawned based on the input. To pass data on to the next stage, an index into a tightly packed array must be determined, as not every task shader will spawn a mesh shader workgroup. The index is calculated by counting set bits before the actual `ThreadID` by using a mask to retrieve them.

```
input  : $One$ to indicate that a single mesh shader workgroup should spawn,
         $Four$ to indicate that four mesh shader workgroups should spawn.
output : The index $Idx$ for the output data array and the total number $Count_T$
         of mesh shader workgroups to spawn.
$Vote_1 \leftarrow subgroupBallot(One)$;
$Count_1 \leftarrow subgroupBallotBitCount(Vote_1)$;
$Vote_4 \leftarrow subgroupBallot(Four)$;
$Count_4 \leftarrow subgroupBallotBitCount(Vote_4)$;
$Count_T \leftarrow 4 * Count_4 + Count_1$;
$Mask \leftarrow (1 << \texttt{ThreadID}) - 1$;
if $One$ then
    $Idx \leftarrow 4 * Count_4 + subgroupBallotBitCount(Vote_1\ \&\ Mask)$;
end
if $Four$ then
    $Idx \leftarrow 4 * subgroupBallotBitCount(Vote_4\ \&\ Mask)$;
end
```

3.5.1 Grid Mesh

Each mesh shader workgroup is by design parallel and split up into multiple threads. The smallest execution unit on NVidia GPU's is called a warp and has 32 threads. To efficiently utilize a warp, we will let each of the 32 threads output one triangle, so the workgroup will output a 4×4 grid in total for each cell that was not rejected by the task shader.

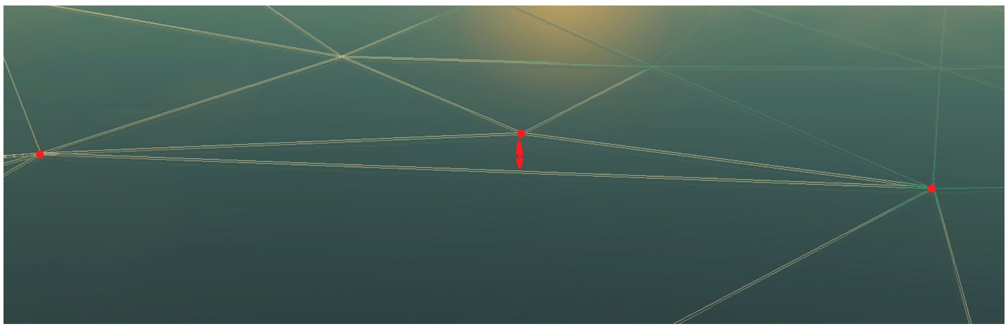
At program start-up we upload a 4×4 grid mesh to the GPU. This mesh can be re-used for every cell, by scaling and moving it to the final position on the cube. To transform the cube to a sphere, the same transformation as in Proland is used. This is the basis for cube maps where a point $\vec{p}$ of the cube is normalized to a vector $\hat{v}$ on the surface of a unit sphere [Gre86]. The unit sphere is then scaled by the radius of the planet to its final size. The height for every vertex is sampled from a heightmap texture in GPU memory. After this, every vertex is transformed to clip-space.

3.5.2 Resolving T-Junctions

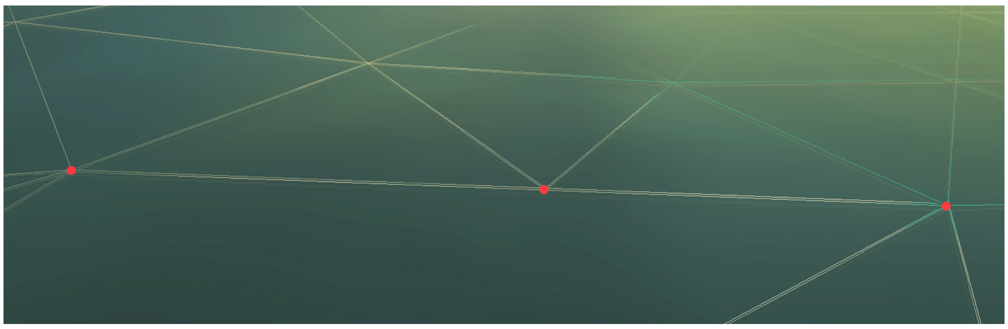
At the border between two LoD levels, holes in the terrain can occur as seen in Figure 3.6a. This is due to one level having double the resolution of the other and a height difference between two vertices from the lower resolution layer. We correct this by adjusting the height of the higher-resolution layer to the height of the other, see Figure 3.6b. This is done just at the border between the layers, so we still have a higher-sampled terrain in the rest of the layer. For this, we calculate a blend factor b for each vertex, where we use the distance $\vec{d}$ between the vertex and the viewer.

$$b = 1.0 - clamp((\vec{d} - R_L)/R_L, 0.0, 1.0)$$

b will be between zero and one, where zero means the vertex is at the border and everything else signifies space in between viewer and the border. Using b, we then adjust the height to obtain a smooth transition towards the border of two neighboring LoDs.



(a) The red arrow indicates a height difference between the red vertex in the middle and the edge of the two vertices on the left and right.



(b) The height of the red vertex in the middle is adjusted to match the height of the line between the two vertices on the left and right.

Figure 3.6: At the border between two LoD layers, holes in the terrain can occur due to one layer having higher resolution than the other. In Figure 3.6a the red arrow indicates such case, where a vertex is of different height. To close the gap, the height of the vertex is adjusted to the height of the lower-resolution LoD layer in Figure 3.6b.

CHAPTER 4

Results

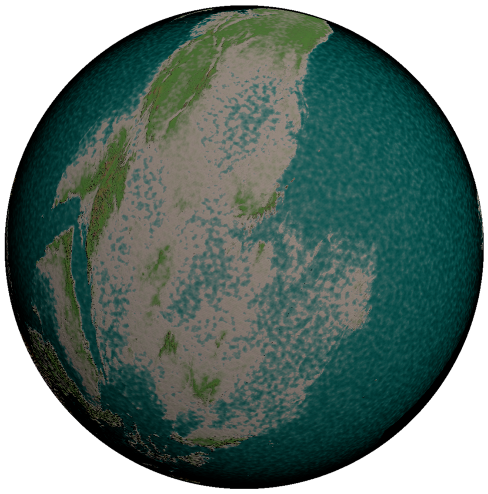
This chapter shows and discusses the preformance and competitiveness of our terrain rendering method. The proposed method, as well as a quadtree implementation similar to proland [Pro], was implemented in an existing planetary renderer in C++ and OpenGL. First we present and interpret the visual output of our implementation. For this, we render a planet with a radius of $6.357km$, similar to the Earth, with appropriately selected parameters for LoD levels and their radius. We then look at the generated output and the quality of the circular LoD regions. Finally, we select comparable configurations with respect to resulting geometry load for both approaches and make a performance comparision. Table 4.1 shows the hardware configuration that was used to render the visuals and create all results on a personal desktop machine running Windows 10.

Hardware	Type
CPU	Intel Core i7-9700K 3.60GHz
GPU	Nvidia RTX 2060 6GB VRAM
RAM	32GB DDR4 3200 MHz CL16

Table 4.1: Hardware configuration used to create the results for our evaluation.

4.1 Visual Output

For both images in Figure 4.1, we chose a maximum of 11 LoD levels and therefore have a grid size of 2^{11} as the finest resolution on LoD level 11. As a base radius for LoD level 11 we used $512km$. The screen capture in Figure 4.1b was recorded near the surface where the entire terrain up until the highest mountain near the left side is rendered with LoD level 11. In Figure 4.1a, we provide a distant shot of the same location from space to show the whole planet. The finest LoD level in this shot is 8 with a radius of $512 * 2^{11}/2^8 = 4096km$.


(a) A whole planet with a visible continent rendered with a maximum LoD level of 8.

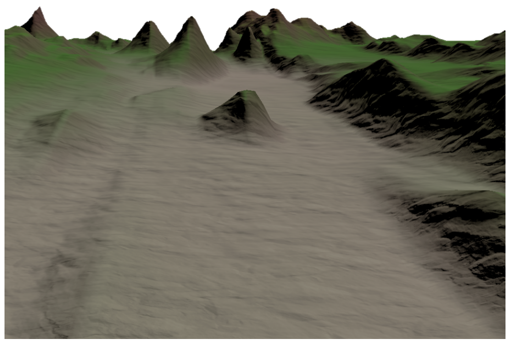
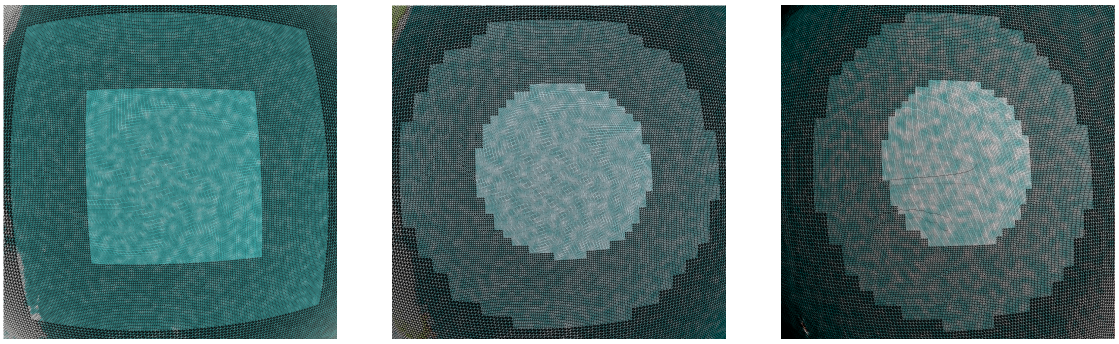

(b) A close-up of the planet from Figure 4.1a with a maximum LoD level of 11.

Figure 4.1: The result of rendering a planet with our method. Figure 4.1a shows a shot from space of the whole planet and Figure 4.1b a shot from the surface.

In Figure 4.2b two nested, circular LoD regions in the middle of one of the six cube side terrains are visible. That means the viewer observes uniform resolution into all directions, as opposed to rectangular regions produced by clipmaps or quadtrees (Figure 4.2a)s. In Figure 4.2c the view position has been moved towards an edge between two cube sides. The more the viewer moves toward an edge, the more squished the circular regions appear. That means for the viewer the resolution is not entirely uniform into all directions anymore. This is due to the cube-to-sphere projection we use via simple normalization of a point $\vec{p}$ on the cube to a vector $\hat{v}$ on the surface of the unit sphere, which is not area-preserving. Hence, equally-sized grid cells of a cube face are not guaranteed to be mapped to equally-sized areas on the sphere. A solution to this would be to use a better projection as discussed by Zucker *et al.* [ZH18, CO75, OL76]. However, during our evaluation, we have found this deformation to be only marginal, with little influence on the perceived quality for the spectator.

4.2 Performance Comparison

To evaluate our method for real-time performance, we tested our method under various conditions and compared it to the quadtree method. We measure the average frame time to render one view at multiple locations of the planet with both our and the quadtree rendering method. For every test case, the measurement is done once on the ground and once at higher altitudes to evaluate performance for high- and low-detail terrain generation, respectively (see Figures 4.3–4.7). Since the quadtree also has to manage a tree structure on the CPU, we report its total memory consumption.



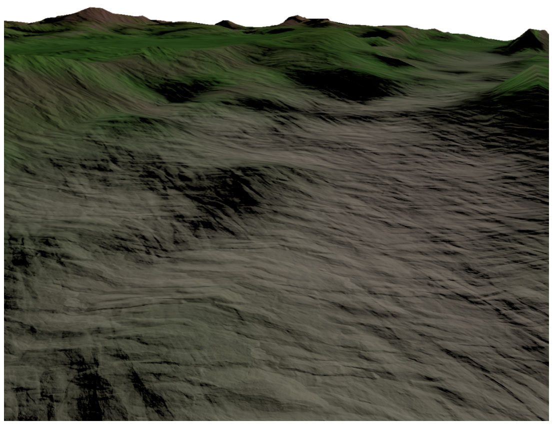
(a) Two visible, rectangular LoD regions on a cube side with the viewer at the center.

(b) Two visible, circular LoD regions on the same cube side surface.

(c) Two visible LoD regions when approaching the border between two cube sides.

Figure 4.2: Figure 4.2a shows two rectangular LoD regions on the sphere, produced by the quadtree approach of Proland. The regions appear not perfectly rectangular due to the cube-to-sphere projection. In Figure 4.2b there are two circular LoD regions on the sphere, produced by our method. The circular regions give the viewer at the center uniform terrain resolution in all directions. As the viewer moves towards the edge to another cube side, as shown in Figure 4.2c, the circular regions appear squished because the used cube-to-sphere projection is not area-preserving.

Both methods are parameterized by a maximum subdivision level on the ground where no further subdivision happens, but the size of each LOD layer is different: The quadtree is subdivided when the viewer moves towards the bounding box of the cells. A parameter k can be used to influence how soon this subdivision happens and therefore also influences the size of each LoD layer. In contrast, our method uses a base radius R_0 to control the LoD layer size of the highest subdivision LoD_{max}. The radius R_L for all coarser levels L is $R_0 \times 2^L$. Thus, subdivision for our method is simply based on the height of the viewer above the terrain. We set these parameters for each test case separately, so that LoD layers have roughly the same perceived size and detail as in the quadtree method. All tests are executed with a resolution of 1718×1368 and frustum culling enabled. Frustum culling differs in that the quadtree method performs it on the CPU, while our method does it on the GPU as part of the task shading stage. Table 4.2 shows the average frame times for both methods and where they were taken. The column LoD_{max} shows the maximum subdivision level and therefore resolution of the LoD layer beneath the viewer at the position while taking the measurements. While the gap between the frame times of our method and the quadtree is not that big for a lower resolution like in Figure 4.3a or Figure 4.3b, it starts to increase in favor of our method the more details are added and the more cells are visible. This is as expected, as we only submit one draw call per LoD level, while the quadtree method has to execute one draw call for every leaf of the tree. With increasing detail level, we also noticed a lag of up to 2 seconds, when testing the quadtree method, at program startup. This is due to the initial, recursive quadtree generation and visibility testing.


(a) A hilly landscape with $LoD_{max} = 8$, $k = 2$ and $R_0 = 1024$.

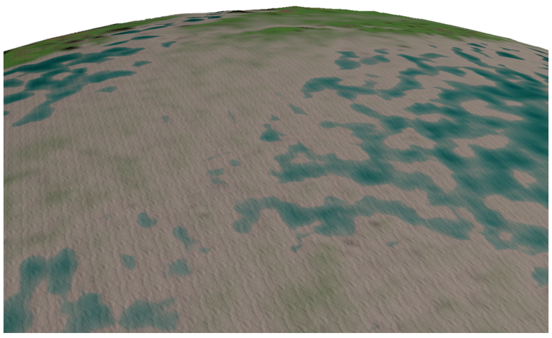
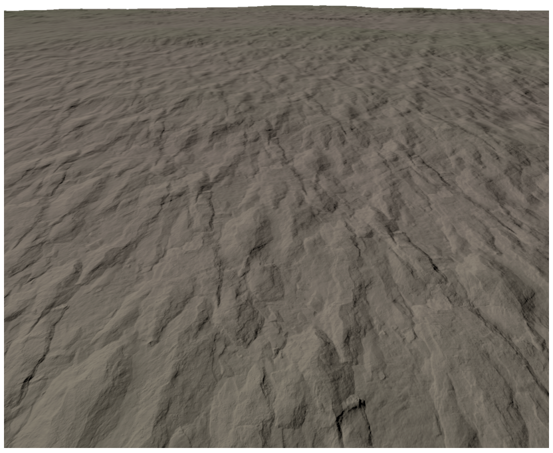

(b) A flat landscape with the small hills from 4.3a on the horizon, rendered with $LoD_{max} = 7$, $k = 2$ and $R_0 = 1024$.

Figure 4.3: Figure 4.3a shows a hilly terrain that can be seen on the horizon in Figure 4.3b.


(a) A flat landscape with $LoD_{max} = 10$, $k = 8$ and $R_0 = 512$.

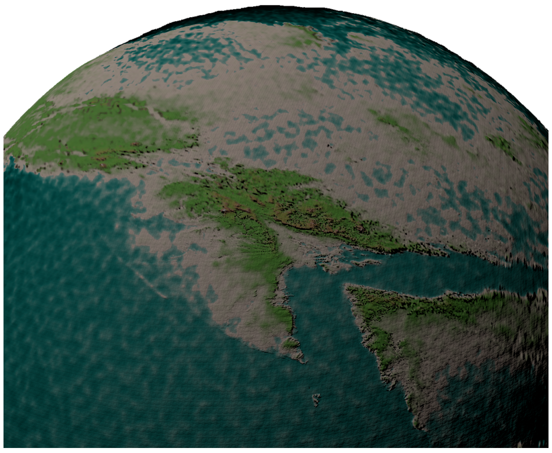
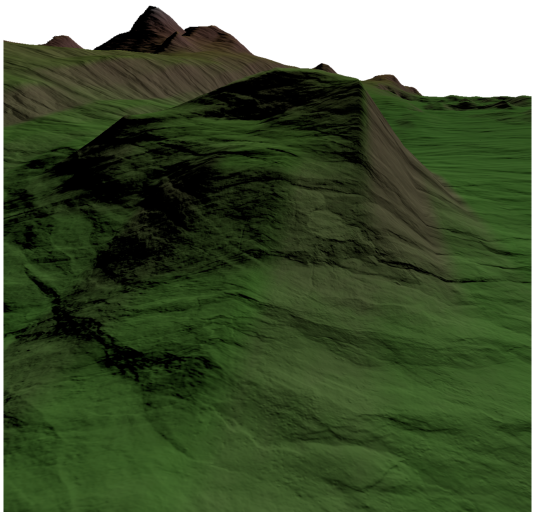

(b) The flat landscape from 4.4a and its surroundings, rendered with $LoD_{max} = 8$, $k = 8$ and $R_0 = 3200$.

Figure 4.4: Figure 4.4a shows a flat terrain that is part of a continent, seen in Figure 4.4b.


(a) A grassy terrain with a few mountains in the background, rendered with $LoD_{max} = 12$, $k = 8$ and $R_0 = 220$.

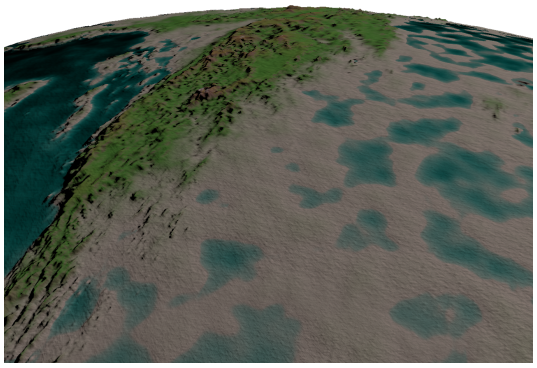
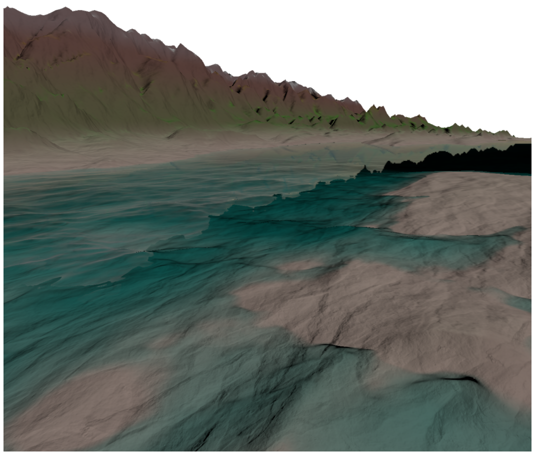

(b) The grassy terrain from 4.5a and its surrounding mountains, rendered with $LoD_{max} = 10$, $k = 8$ and $R_0 = 1024$.

Figure 4.5: Figure 4.5a shows part of a mountain range, surrounded by grass that can be seen from above in Figure 4.5b.


(a) A mountain range with steep, high mountains and water in the front, rendered with $LoD_{max} = 12$, $k = 8$ and $R_0 = 220$.

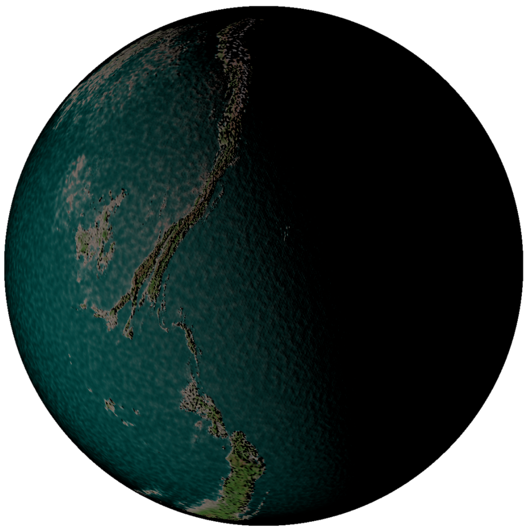
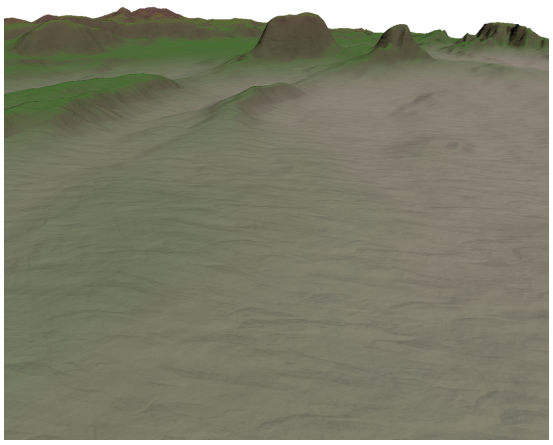

(b) One hemisphere from space, that is covered by the mountain range from 4.6a, rendered with $LoD_{max} = 9$, $k = 10$ and $R_0 = 3200$.

Figure 4.6: Figure 4.6a shows a large, steep mountain range, that covers the entire length of a hemisphere in Figure 4.6b.


(a) A flat terrain starts to get mountainous towards the horizon with a few very steep mountains. $LoD_{max} = 14$, $k = 8$ and $R_0 = 64$.

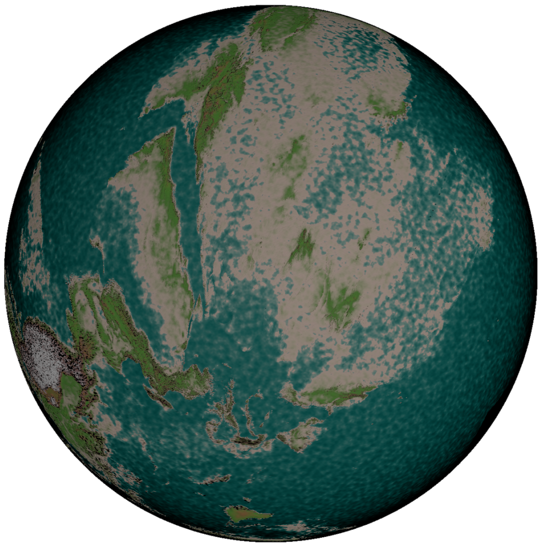

(b) A whole hemisphere from space with the terrain from 4.7a and its surrounding continent. $LoD_{max} = 9$, $k = 11$ and $R_0 = 3200$.

Figure 4.7: Figure 4.7a shows a landscape with various terrain features. The landscape is part of the hemisphere in Figure 4.7b.

Where	Figure	LoD_{max}	Quadtree [kB]	Ours [kB]	Quadtree [ms]	Ours [ms]
ground	4.3a	8	333	0.697	0.52	0.47
space	4.3b	7	219	0.697	0.52	0.47
ground	4.4a	10	373	0.697	0.74	0.64
space	4.4b	8	1258	0.697	2.47	1.53
ground	4.5a	12	1301	0.697	2.69	1.91
space	4.5b	10	1134	0.697	2.11	1.88
ground	4.6a	12	1028	0.697	1.93	1.48
space	4.6b	9	2203	0.697	3.14	1.53
ground	4.7a	14	1674	0.697	3.72	2.53
space	4.7b	9	2537	0.697	3.47	1.62

Table 4.2: The table shows the memory usage and average frame times of our method and the quadtree method. The finest LoD shows the finest level-of-detail at the position seen in the figures listed above.

The memory consumption of the quadtree also increases in general with higher level-of-detail. However, as invisible children are removed from the tree, there are exceptions like in Figure 4.7a. The scene was rendered from the ground with more details than the same place from space in Figure 4.7b. However, the high-detail view actually requires less memory because most of the planet is occluded.

In summary, the results look very promising for our method to become a viable option for future usage in real-time applications. The reduced CPU load and CPU-GPU communication overhead has a clear impact on performance as details are added and more cells are visible. Increasing LoD_{max} near the ground shows that our method gets more efficient with every LoD level added. With increasing altitude, the performance difference widens even more with every LoD level added. This is because more and more cells are visible, which results in longer tree traversals and increased CPU-GPU communication for the quadtree method.

CHAPTER 5

Conclusion and Future Work

The aim of this thesis was to conceive, describe and evaluate a new, high-performance method for terrain rendering using new GPU hardware features from NVidia's Turing architecture [nvi]. To support a range of models, including cheaper consumer-grade ones, the solution needed to be scalable. Additionally, we aimed to make our method robust and free from popping or swimming artifacts, which commonly plague similar approaches. We showed that our introduced method can achieve better performance than the state-of-the-art method used e.g. in Proland [Pro], where planetary rendering is done with a quadtree. The quadtree needs to maintain its contents continuously, and performs recursive subdivision, as well as visibility determination on the CPU. Drawcalls are made for each visible tile, resulting in tremendous CPU-GPU communication for higher subdivision levels. In contrast, our method only performs some bounding for visible LoD layers on the CPU and subsequently launches NVidia's new task shaders, which do most of the work on the GPU while exploiting parallelism. During testing, we saw how the quadtree method got slower the more details were added, and the performance gap between the runtime of the quadtree and our method increased in favor of our method. Another advantage of our method is that we do not need to maintain a data structure on the CPU and therefore do not need extra memory for each visible cell.

Regarding limitations, we were unable to guarantee uniform resolution in all directions around the viewer at all times. Introducing circular level-of-detail regions instead of rectangular ones worked around the center of each cube side. However, the circular regions got more and more oval as the viewer moves towards the edges of the cube. This is due to the cube-to-sphere projection we used and we would like to address this in the future by using a better projection, like tangential projection, shown by Zucker *et al.* [ZH18].

Regarding further performance optimization options, our CPU bounding submits cells to the task shader for the actual LoD decision to create a circular region. The viewer does not always see every part of this region, so a possible speed-up could be a lightweight visibility determination on the CPU, to reduce the cell count before the parallel frustum culling in the task shader. For the LoD layers, we were able to generate visually pleasing LoD levels without visible artifacts and close-to-uniform geometry detail over the viewport. However, the corresponding parameters for subdivision had to be determined in advance rather than automatically. For future work, we would like to introduce a proper height function that decides the different LoD resolutions based on the altitude of the viewer with respect to the terrain by taking the projected triangle size in screen space into account.

In summary, we found that exploiting the Turing architecture enables a new and faster method for rendering terrains at a planetary scale. The obtained results suggest that our method could be a strong candidate for future usage in real-time planetary rendering applications.

List of Figures

List of Tables

List of Algorithms

Bibliography

[CH06] Malte Clasen and Hans-Christian Hege. Terrain rendering using spherical clipmaps. In *EuroVis*, 2006.

[CO75] FK Chan and EM O'Neill. Feasibility study of a quadrilateralized spherical cube earth data base, computer sciences corporation. Technical report, Tech. Report 2-75, Monterey, California, 1975.

[CR11] Patrick Cozzi and Kevin Ring. *3D Engine Design for Virtual Globes*. A. K. Peters, Ltd., USA, 1st edition, 2011.

[Eli] Elite Dangerous terrain quadtree. `https://80.lv/articles/generating-the-universe-in-elite-dangerous`. Accessed: 2020-02-16.

[Gre86] N. Greene. Environment mapping and other applications of world projections. *IEEE Computer Graphics and Applications*, 6(11):21–29, Nov 1986.

[KB14] Goanghun Kim and Nakhoon Baek. A height-map based terrain rendering with tessellation hardware. *2014 International Conference on IT Convergence and Security (ICITCS)*, pages 1–4, 2014.

[KLJ$^+$09] R. Kooima, J. Leigh, A. Johnson, D. Roberts, M. SubbaRao, and T. A. DeFanti. Planetary-scale terrain composition. *IEEE Transactions on Visualization and Computer Graphics*, 15(5):719–733, Sep. 2009.

[KLR$^+$95] David Koller, Peter Lindstrom, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory Turner. Virtual gis: A real-time 3d geographic information system. In *Proceedings of the 6th Conference on Visualization '95*, VIS '95, page 94, USA, 1995. IEEE Computer Society.

[Ksp] Kerbal Space Program terrain quadtree. `kerbalspace.tumblr.com/post/9056986834/on-quadtrees-and-why-they-are-awesome#disqus_thread`. Accessed: 2020-02-14.

[LH04] Frank Losasso and Hugues Hoppe. Geometry clipmaps: Terrain rendering using nested regular grids. *ACM Trans. Graph.*, 23(3):769–776, August 2004.

[LKR$^+$96] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hodges, Nick Faust, and Gregory A. Turner. Real-time, continuous level of detail rendering of height fields. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, page 109–118, New York, NY, USA, 1996. Association for Computing Machinery.

[Mah10] Joseph D. Mahsman. Projective grid mapping for planetary terrain. 2010. Master Thesis.

[nvi] Nvidia introduction to turing mesh shaders. `https://devblogs.nvidia.com/introduction-turing-mesh-shaders/`. Accessed: 2020-02-10.

[OL76] EM O'Neill and RE Laubscher. Extended studies of a quadrilateralized spherical cube earth data base. Technical report, 1976.

[Paj98a] Renato Pajarola. Access to large scale terrain and image databases in geoinformation systems. 1998. Dissertation.

[Paj98b] Renato Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In *Proceedings of the Conference on Visualization '98*, VIS '98, page 19–26, Washington, DC, USA, 1998. IEEE Computer Society Press.

[PF05] Matt Pharr and Randima Fernando. Terrain rendering using gpu-based geometry clipmaps. In *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*, chapter 2. Addison-Wesley Professional, 2005.

[Pro] Proland terrain quadtree. `https://proland.inrialpes.fr/doc/proland-4.0/core/html/index.html`. Accessed: 2020-02-10.

[VHB87] Brian Von Herzen and Alan H. Barr. Accurate triangulations of deformed, intersecting surfaces. *SIGGRAPH Comput. Graph.*, 21(4):103–110, August 1987.

[ZH18] Matt Zucker and Yosuke Higashi. Cube-to-sphere projections for procedural texturing and beyond. *Journal of Computer Graphics Techniques (JCGT)*, 7(2):1–22, June 2018.