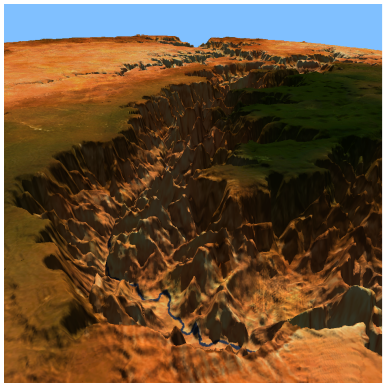
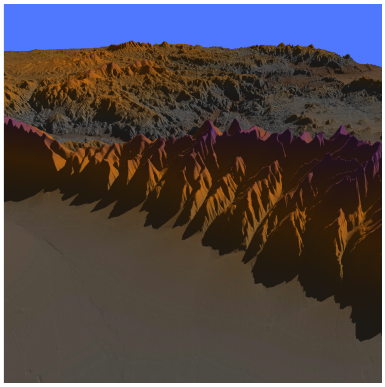
TÉCNICO LISBOA

A Real-Time Terrain Ray-Tracing Engine

André Filipe Domingos Silvestre

Thesis to obtain the Master of Science Degree in

Computer Engineering

Supervisor(s): Prof. João António Madeiras Pereira
Dr. Vasco Alexande da Silva Costa

Examination Committee

Chairperson: Prof. Miguel Nuno Dias Alves Pupo Correia
Supervisor: Prof. João António Madeiras Pereira
Member of the Committee: Prof. Fernando Pedro Reino da Silva Birra

June 2017

Dedicated to my family

Acknowledgments

I would like to thank my mother and my sister for their unconditional support, my friends who kept me motivated to come this far, and my supervisors for the feedback they provided.

Resumo

Ray tracing é um algoritmo de rendering, capaz de produzir imagens realistas com alta qualidade, muito utilizado em aplicações não-interativas com foco exclusivo na qualidade das imagens. Com os avanços tecnológicos que todos os anos vão melhorando a performance das GPU, muitos algoritmos de Ray Tracing têm surgido com o objetivo de desenhar cenas virtuais em tempo real e com melhor qualidade gráfica do que a rasterização, que é mais utilizada em aplicações de tempo real. Além disso, vários trabalhos já demonstraram que o traçamento de raíos primários é, por definição, uma alternativa de alto desempenho à rasterização para visualizar terrenos digitais grandes.

Com base numa análise sobre vários algoritmos de ray tracing para visualização de terrenos, neste trabalho foi implementado um motor de Ray Tracing para terrenos, baseado nas conclusões dessa análise, capaz de produzir imagens em tempo real. Este motor não utiliza polígonos nem converte o terreno numa malha polígonal, aplica um esquema de nível de detalhes baseado na distância tridimensional ao observador para simplificar o terreno e faz também traçamento de raíos secundários para sombreamento, algo que trabalhos anteriores não testaram.

Palavras-chave: Real-Time Ray Tracing, Scalable Terrains, Maximum Mipmaps, Bilinear Patch, Secondary Rays

Abstract

Ray Tracing is a rendering algorithm, capable of generating highly realistic images, typically employed in non-interactive applications that disregard real time performance and focus exclusively on image quality. As technological advances progressively improve GPU performance, many Ray Tracing algorithms have been proposed to render virtual scenes in real time with better quality than rasterization, which is more widely used in real time applications. Moreover, several works have demonstrated that ray casting is, by definition, a high performance alternative to rasterization when it comes to terrain rendering.

Based on an analysis of several terrain ray tracing algorithms, a real time terrain ray tracing engine was implemented leveraging those algorithms. The implemented engine does not require conversion from heightmaps to polygonal meshes, employs a level-of-detail scheme based on the three-dimensional distance from the viewer and is also capable of secondary ray tracing for shadows, which has not been applied in previous works.

Keywords: Real Time Ray Tracing, Scalable Terrains, Maximum Mipmaps, Bilinear Patch, Secondary Rays

Contents

Acknowledgments v
Resumo vii
Abstract ix
List of Tables xv
List of Figures xvii
Glossary xxi

1 Introduction **1**
1.1 Motivation 1
1.2 Topic Overview 2
1.3 Objectives 2
1.4 Contributions 3
1.5 Thesis Outline 3

2 Related Work **5**
2.1 Terrain Representations 5
2.1.1 Heightmaps 5
2.1.2 Volumetric Rendering 6
2.2 Challenges in terrain rendering 7
2.2.1 Limited Memory 7
2.2.2 Level of Detail 7
2.2.3 CPU-GPU Data Transfer 9
2.3 ROAM 9
2.4 Geometry Clipmaps 10
2.5 CDLOD 11
2.6 Terrain Ray Tracing 12
2.6.1 Ray Marching 13
2.7 Displacement Mapping Techniques 14
2.7.1 Shell Maps 14
2.7.2 Cone Step Mapping 15
2.8 Signed Distance Fields 16

2.9 Maximum Mipmap . . . 17
2.10 Hybrid Rendering . . . 18
2.11 Ray Tracing on Geometry Clipmaps . . . 18
2.12 Pros and Cons . . . 19

3 Terrain Ray-Tracer Engine Implementation 21
3.1 Algorithmic Techniques . . . 21
3.2 System architecture . . . 22
3.2.1 Application workflow . . . 23
3.3 Terrain storage and texture formats . . . 24
3.3.1 Heightmap and Maximum Mipmap . . . 24
3.3.2 Normal Map . . . 26
3.3.3 Photo Texture . . . 27
3.4 Handling texture data . . . 28
3.5 Calculating primary rays . . . 29
3.6 Ray Traversal Algorithm . . . 32
3.6.1 Maximum Mipmap Traversal . . . 34
3.6.2 Terrain patch intersection . . . 37
3.6.3 LOD control . . . 38
3.7 Scalable engine differences . . . 40
3.7.1 Texture storage in the scalable engine . . . 41
3.7.2 Scalable engine tile conversion . . . 43
3.8 Shading . . . 45
3.8.1 Photo shading . . . 46
3.8.2 Height Shading . . . 47
3.8.3 Depth Shading . . . 47
3.8.4 Grid Shading . . . 47
3.8.5 LOD Grid Shading . . . 48
3.8.6 Lighting Shading . . . 48
3.8.7 Lighting Grid Shading . . . 48
3.8.8 Normal Map Shading . . . 49
3.8.9 Ray Step Shading . . . 49
3.8.10 Secondary Step Shading . . . 49
3.9 Rendering artifacts . . . 49

4 Evaluation and Results 51
4.1 Frame Rate . . . 54
4.1.1 Small-scale ray tracer . . . 54
4.1.2 Scalable ray tracer . . . 55
4.2 Rays per second . . . 56

4.2.1 Small-scale ray tracer 57
4.2.2 Scalable ray tracer 58
4.3 Traversal steps per ray 59
4.4 Memory Consumption 61
4.5 Result analysis and comparison 62

5 Conclusions and Future Work **67**
5.1 Achievements 67
5.2 Future Work 68

Bibliography **69**

List of Tables

2.1 Brief comparison of related algorithms. 19

3.1 OpenGL texture formats selected for each component of the terrain. 28

4.1 FPS results from both primary and secondary ray tracing for the small-scale engine. . . . 54
4.2 FPS results from both primary and secondary ray tracing for the scalable engine. 55
4.3 Comparison of memory consumed with varying terrain resolution and precision. 61
4.4 Comparison of memory consumed with varying terrain tile and active region resolutions for the scalable engine. 62

List of Figures

2.1 Example low resolution heightmap. 6
2.2 Small terrain rendered with voxels. 6
2.3 Two artifacts: (Left) Crack, (Right) T-junction. 8
2.4 (a) An example ROAM triangle strip appears to retain more detail towards the left side where a higher number of triangles has been split. (b) The result of a split and merge operation in both directions. 10
2.5 Top down overview of a clipmap. From ”Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids” [7]. 11
2.6 CDLOD morphing. From ”Continuous-Distance Level of Detail for Rendering Heightmaps” [6]. 12
2.7 Ray traversal using ray marching on a grid. 14
2.8 A cone being skipped during ray traversal. 15
2.9 SDFs have multiple applications, one of which is to store fonts in memory. Rendering SDF fonts allows us to render text in multiple resolutions using a single bitmap texture, or apply effects such as antialiasing and outlines. 16
2.10 Illustration of a maximum mipmap structure on top of the terrain. From ”Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering” [9]. 17

3.1 Terrain Ray-Tracer engine architecture diagram. 23
3.2 Pixel content comparison between per point and per patch formats. Per point format stores a single height value per texel while per patch format stores four neighbouring height values per texel. 25
3.3 Comparison between an 8-bit heightmap with a 16-bit heightmap. The quantization is very noticeable. 26
3.4 Normal filter calculation. Determine a normal for all four directions and average the resulting normals. 26
3.5 Vertical field-of-view comparison: (a) Has a higher FOVy than (b) and thus captures a larger volume of the scene than the latter. 30
3.6 The image on the left was ray traced with interpolated rays, on the right rays were individually calculated in the Fragment Shader. The final image is exactly the same. 31
3.7 Simple ray tracing vs multisampled ray tracing. 31

3.8 Bounding box intersection diagrams and formulas for calculating ray min and max parameters, as well as deltas, for each axis. 33
3.9 Each patch of the Maximum Mipmap represents a bounding box around the terrain's domain it covers horizontally. When an intersection is found at any particular level, traversal proceeds to the next level, at a finer resolution. 35
3.10 Descending operation example. Entrance point is colored green, exit point is colored magenta and the intersection point has a cross. a) An intersection is found at the coarsest mipmap level so a descending operation is performed. b) There is no intersection found at this patch so traversal moves on to the next. c) Again an intersection is found and another descending operation would be performed. 36
3.11 Ascending operation example: a) No intersection is found at the current level. Ascending is not possible because the next patch has the same parent on the coarser level. b) There is no intersection again, but this time ascending is possible. c) The ascending operation was done and traversal continues at the next level and next patch. MAX represents the coarsest level with only one height value. 37
3.12 LOD zones overlayed on the terrain with color coding. Using 3D distance, the shape of LOD regions depends on the terrain structures and may even be divided. 39
3.13 LOD levels are determined based on distance to the viewer and one distance parameter. This parameter represents the distance that spans the finest LOD. Each subsequent level will span two times the distance of the previous level. 40
3.14 Active tiles stored in memory may not be contiguous within the active region texture. An index array keeps track of their positions in memory so that each can be accessed properly in the ray tracing algorithm. 42
3.15 Illustration of how active tile conversion applied. The tile that should be rendered at the top left corner of the active region, index 6, was stored at index 8 in memory. An underlying conversion array of [2,0] is used to perform the correct access. 44
3.16 Showcase of shading functions. From top to bottom and left to right: Photo Shading, Height Shading, Depth Shading, LOD Shading, Light Grid Shading, Normal Shading, Light Shading, Ray Step Shading, Secondary Ray Step Shading 47
3.17 Shadow artifacts visible at LOD transitions. Due to instant altitude differences secondary rays immediately intersect the terrain. 50

4.1 Four terrain datasets tested with the small-scale ray tracer: Synthetic, Grand Canyon, Puget Sound and Colorado. 52
4.2 Two terrain datasets tested with the small-scale ray tracer: Puget Sound and Colorado. . 52
4.3 Rays Per Second analysis for the small-scale engine datasets. The results are presented in mega rays per second (millions of rays per second). 58
4.4 Rays Per Second analysis for the scalable engine datasets. The results are presented in mega rays per second (millions of rays per second). 59

4.5 Ray step heatmaps in the Colorado dataset at different viewer positions. 60
4.6 Comparison between ray step shading modes and varying light angles. From left to right: primary steps, secondary steps vertical light, secondary steps 45 degrees and secondary steps 75 degrees. The shadow boundaries are notably visible. Note that steps from rays that exited the terrain are not represented and instead the skybox is visible. 60

Glossary

FOV Field of View represents the opening angle of a camera and the extent of the virtual scene that is observable.

FPS Frames Per Second is the number of images that an application generates per second.

GPU Graphics Processing Unit is a programmable logic processor specialized for display functions.

LOD Level-Of-Detail is a rendering technique that controls how much detail is rendered into parts of an object based on the distance from the viewer.

VAO Vertex Array Object is an OpenGL object that stores all of the state needed to supply vertex data to the shader pipeline.

Chapter 1

Introduction

1.1 Motivation

This work focuses on real time terrain ray tracing. Ray tracing is a high quality rendering technique that was proposed in 1979 by Turner Whitted [1], capable of outputting highly realistic digital images through the use of lighting effects like realistic shadows, reflections and refractions, unlike those utilized typically in rasterization which require clever algorithms to grossly simulate those effects.

Behind the ray tracing algorithm lies a physics based rendering model, where light rays are back-tracked from the user's point of view towards a three dimensional scene, one ray per pixel on screen. Finding the intersection with objects in the scene and then measuring the contribution of all light sources towards an intersection point is the basic notion behind ray tracing.

The most critical drawback in the original ray casting work is that the algorithm is said to spend over 90% of its execution time testing intersections with objects in the scene [2, 1], multiplied by the number of rays traced per frame. The most common computer screen resolutions used today are HD formats of 1366x768 and 1920x1080 pixels with aspect ratios of ~16.9 and 16.9 respectively. Other resolutions are utilized as well, lower resolutions with aspect ratios of 4:3, such as 640x480 and 1024x768, and higher resolutions of 2560x1600 for a minority of users. Frames in typical and high resolutions are rendered with an approximate number of 1 to 4 million pixels, the same number of primary rays traced. The number of rays traced will only increase both with higher resolutions and when taking into account secondary rays. As such, ray tracing is very computationally expensive and that is why it primarily targets applications that disregard real time performance and instead focus on improving image quality at the expense of performance - each image may take a long time to be completely generated.

Nevertheless, thanks to the incredible advances in hardware and innovative algorithms proposed since then, ray tracing has become a more feasible option in real time environments.

On the other hand we have terrains. Terrain rendering tends to be performed with rasterization. Large terrain datasets captured from the earth using satellites, or procedurally generated datasets, are rendered in movies, video games, simulations or simply for visualization.

When we think of applications like games or visualization, we think of real time rendering. So how

can we join terrain visualization and real time ray tracing? Well, thanks to the fact that terrains are usually represented as a grid of height samples, the terrain itself already works partially as a ray tracing acceleration structure. Many algorithms take advantage of this property to achieve real time ray tracing of terrains.

That being said, using ray tracing to render terrains appears to be a very attractive choice. The most notable works on terrain ray tracing date back to 2009 [3] however, given the advances in hardware and the many different algorithms presently at our disposal, real-time ray tracing for terrains can be improved.

1.2 Topic Overview

Previous terrain rendering algorithms have some obvious deficiencies. While the performance of rendering algorithms has been highly optimized, the rendering quality achieved is not necessarily great. The fact that rasterization is currently the norm in rendering heavily influences how developers approach ray tracing. Many algorithms used in ray tracing are adapted from rasterization and still follow the typical constraints that apply to rasterization, even if the goal is to utilize ray tracing.

Many engines continue to store terrain elevation as triangle meshes. However, ray tracing is not restricted to rendering only triangles like rasterization does, hence parametric surfaces are another possibility. Linear interpolation and bilinear interpolation of the terrain might result in rendering smoother terrain than with triangles, that this is just one example of using ray tracing specific features. Naturally performance might drop compared to using triangles, but this quality tradeoff is worth considering.

LOD (Level of Detail) is an optimization to rendering, in general, that reduces the complexity of an object in multiple layers based on the distance from the viewer. It comes from the fact that the full complexity hardly makes a difference in the final image because the captured scene must be sampled down to pixels, so it is not possible to view all the details nonetheless. This is a commonly overlooked part of ray tracing and most implementations settle for the same LOD metrics as used in rasterization algorithms such as Geometric Clipmaps. Because ray tracing traces each ray into the scene, it is easy to calculate the 3D distance from the user's position to the intersection points - something difficult to do with rasterization - hinting that ray tracing could offer a unique way to create smoother LOD transitions.

In rasterization the complete geometric mesh must be sent into the pipeline to be rendered. Even before that a simplified terrain mesh must be calculated each frame using LOD metrics to accomodate for the viewer's movement and position. One other fact noted is that for ray tracing the engine does not necessarily have to precompute a terrain mesh. The LOD transitions can be calculated at run time without having to meld different LOD levels together in a clipmap mesh.

1.3 Objectives

With this work I intend to analyze the existing drawbacks and gaps of terrain ray tracing algorithms and implement a ray tracer based on that analysis. The main goal is to find ways to improve terrain ray tracing as well as to explore issues to be analyzed in future work.

Our solution will prioritize ray tracing exclusive properties:

- No triangle meshes. Only the original heightmap is necessary with no further conversions.
- Ray-surface intersection using Bilinear Patch Interpolation.
- Secondary ray tracing for realistic shadows.
- All ray tracing calculations are done on-the-fly, including LOD control, intersections and lighting.

1.4 Contributions

Two versions of a terrain ray-tracing engine were implemented in C++ and OpenGL, one ray-tracer aimed at single heightmaps and the other capable of rendering large-scale terrains divided in tiles. While most of the implementation is based on existing work, the use of secondary rays to generate shadows appears to be a novelty in real-time terrain ray-tracing. This work demonstrates that real-time frame rates can be achieved with secondary ray-tracing and presents some of the issues thereby introduced.

1.5 Thesis Outline

his document is divided into Related Work, Engine Implementation, Evaluation and Results, and Conclusion.

The Related Work section will cover several works in ray tracing and terrain rendering, important details and the most important algorithms for our implementation.

Next, in the Implementation section, I further describe the implementation choices and goals behind the project as well as the target platform and architecture of the application, and the algorithms implemented in the ray tracing pipeline.

Performance Testing presents the evaluation methods utilized to test the engine as well as their corresponding results, both overall and relative to other works.

Finally I will summarize the most important highlights from this work and hint at possible avenues on future work.

Chapter 2

Related Work

In this section we will introduce several concepts and rendering algorithms, including geometry representations used in terrain rendering and previous algorithms developed for rasterization. We will then draw a bridge from terrain rendering to ray tracing and describe the existing challenges in terrain ray tracing as well as an overview of ray marching, algorithms inspired by relief mapping and finally current algorithms made specifically for terrain ray tracing. Ending this section, a short discussion about the pros and cons of each algorithm and how relevant they are for our work.

2.1 Terrain Representations

2.1.1 Heightmaps

Although rendering is generally performed on polygons, or triangles, geometry data can be stored in many different formats either to save memory or simply because they are more appropriate for certain algorithms. Among several formats, the one most often associated with terrains is the heightmap (or heightfields). A heightmap is a special image that stores height values instead of colors, so each texel represents a vertical displacement value with respect to a minimum altitude value, as shown in Figure. 2.1.

Each group of four neighbouring points in a heightmap is called a *patch*. These values are sampled points that must be interpolated in order to generate the missing elevation data. Different heightmap formats can be used depending on the algorithms that will process them, the LOD scheme, etc.

Heightmaps may be interpreted in various ways: each pixel may represent a vertical three-dimensional bar; the heightmap may be converted into a tessellated triangle strip may be generated; or each patch is defined by four control points that can be interpolated with, for instance, Bilinear Patch interpolation. The advantages of using heightmaps include their direct representation as a texture, which can be directly fed to a GPU, and the application of easy compression methods similar to those used in image formats. The major disadvantage of using heightmaps is that each texel maps to one, and one only, altitude value, so we cannot represent terrain such as caverns, arches and overhangs.

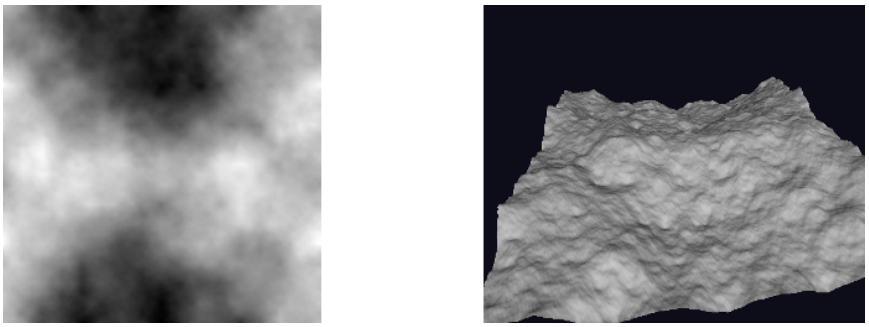


Figure 2.1: Example low resolution heightmap.

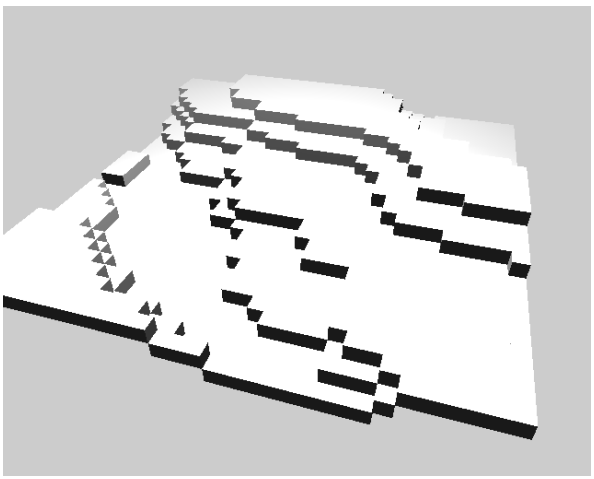


Figure 2.2: Small terrain rendered with voxels.

2.1.2 Volumetric Rendering

Of course there are other possible representations for terrains, in particular that can model multiple heights. One example is volumetric terrain rendering (voxelized terrain). Voxels are the three-dimensional counterpart of pixels and they represent a cube volume in a 3D grid, and an example can be seen in Figure. 2.2. Voxels can have two states, air or solid, making it possible to represent holes and cavities as opposed to heightmaps. There is a tradeoff between performance and smooth looking terrain, if the voxels' size is large then the terrain will clearly appear cubic and quantized, on the other hand using small sizes will result in terrain being smoother but at a much higher resolution. Since the states must be explicitly represented, *air* and *solid*, volumetric terrain is a highly memory intensive approach.

Partitioning methods are used to load voxels as sets of tiles, such as chunks. Occlusion culling avoids rendering hidden voxels. Other techniques, such as tessellation, can be employed to smoothen the cubic appearance of voxels at the corners.

Terrains can also be represented by a standard 3D mesh, making terrain smoother than voxels while still able to represent overhangs. But for many applications this makes simulations more complex and terrain irregularities lead to more complex storage and LOD schemes as well. Hence our work will focus on techniques based on heightmaps.

2.2 Challenges in terrain rendering

There are several challenges that terrain rendering algorithms (even rendering algorithms in general) attempt to overcome, including ways to optimize GPU memory, to increase performance by simplifying the terrain mesh, minimize and optimize CPU-GPU memory transfers, view-frustum culling, and other issues mentioned in previous works. The following sub-sections provide a more in-depth view of three of the more important challenges and how they are overcome.

2.2.1 Limited Memory

One of the major problems rendering algorithms face is the limited amount of memory that is made available by the GPU. In the case of terrain rendering this translates to rendering terrains that could, in their entirety, easily exceed the GPU's video memory. There are two ways to solve this issue: compress the terrain so that it fits in VRAM and then decode on-demand, or divide terrain in multiple tiles of manageable size that will be dynamically loaded based on their distance to the viewer's location. The latter method is referred to as out-of-core rendering. Since loading large amounts of data from memory into the GPU is known to be a bottleneck, a specialized thread for I/O operations may be needed to prevent stalling and other optimizations.

2.2.2 Level of Detail

Another strategy to increase performance in terrain rendering in real-time scenarios is the use of Level-of-Detail (LOD) control schemes [4]. Detail can be seen as the number of triangles in a mesh - a high LOD contains more triangles and a low LOD fewer triangles. Intuitively, even in the real world, when we gaze over a landscape we can find the highest amount of detail on the objects lying closer than objects farther away. If we were to render a terrain at full detail we would observe an identical situation. By definition lower detail can be seen on the terrain farther away, since all geometry has to be discretized into pixels. But the real intuition behind the use of LOD schemes is that there is no need to render the entire terrain at full detail when most triangles at a distance are going to be projected to very few pixels on screen, or even less than a pixel. In which case each pixel would be a blend of contributions from multiple triangles and not only would the final image contain noisy pixels, rendering would also become more expensive as shading is unnecessarily executed on the same pixels multiple times.

In general LOD schemes partition the terrain so that the closest triangles remain at maximum detail and farther triangles are stored with an ever decreasing level of detail, usually merged into larger triangles. Partitioning is done with respect to an heuristic. For example a heuristic based on screen-space projections, as in [5], may assign the LOD based on the area of the triangle's projection onto the screen. LOD could also be assigned as a logarithmic function, as in [6], based on the distance to the viewer. The result is that different LODs are melded together into a single mesh with varying detail. Since each level contains differences, only consecutive levels may be directly connected - meaning that neighbouring triangles may have a maximum difference of 1 LOD.

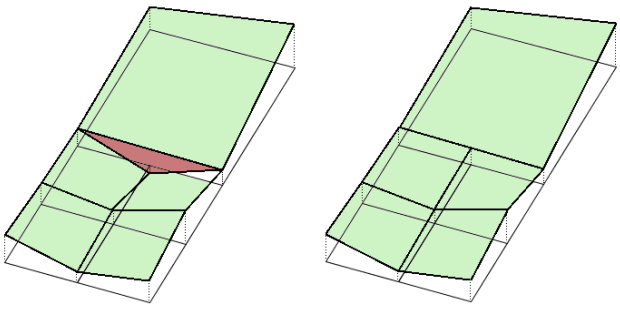


Figure 2.3: Two artifacts: (Left) Crack, (Right) T-junction.

Because each level is inherently different, the LOD scheme must provide a way to connect the simplified mesh partitions. Although LOD schemes simplify the mesh and increase performance, new issues arise in the form of rendering artifacts caused by changes in LOD: popping, cracks and T-junctions. The regions where LODs change are referred to as LOD transition zones. When the viewer moves over the terrain, the LOD mesh must be updated to accommodate that change. This update can be visualized as the transition zone shifting over the mesh and depending on the LOD technique used these shifts are more or less noticeable. This effect is called *popping*. Popping is particularly noticeable on parts of the terrain containing high-frequency detail such as thin spikes.

Neighbouring triangles in LOD transitions belong to different levels. The higher level triangles are smaller and the lower level triangles are larger, meaning that the larger triangle is adjacent to two smaller triangles. The problem is that the small triangles share one vertex that the large triangle does not. When the extra vertex is not aligned with the lower level triangle's edge, lying either below or above the line, we call this artifact a crack because there is a hole in the mesh. If the vertex is aligned with the triangle's edge we call it a T-Junction, because even though both sides are aligned the lower level triangle still does not share that vertex and the mesh is not closed. Figure 2.3 shows the difference between a crack and a T-junction.

While popping and cracks obviously reduce the visual quality of the mesh and affect rendering quality, T-junctions may introduce rounding errors and inconsistencies in the pipeline, which result in very small cracks nonetheless. T-junctions may also affect non-graphical computations applied to the terrain. In addition, shading effects such as lighting or reflections may be subject to visual errors where any of these artifacts occur. Overall LOD schemes aim to increase the overall performance of rendering algorithms but also have to deal with these shortcomings.

LOD algorithms can be discrete (DLOD), creating sharp transitions between levels, or continuous (CLOD), where transitions are extended into transition zones and the mesh is morphed from a high level into a low level. Both LOD types provide different solutions to minimizing artifacts.

2.2.3 CPU-GPU Data Transfer

One of the greatest problems in rendering, in general, has been the bottleneck caused by CPU-GPU transfer of data. While Random Access Memory (RAM) is the primary memory used by the CPU, Video RAM is the primary memory of the GPU. Transfering data from RAM to VRAM over the video bus, is known to be a relatively slow process and most of the data transfer should be done upon the application's initialization. Data transfers should be kept to a minimum each frame. When VRAM is exhausted in the GPU, subsequent data transfers will be loaded into the main RAM instead. When the GPU requires handling data that is currently stored in the main RAM, that data must be swapped into VRAM causing the GPU to stall before executing the next assembly instructions.

Before GPGPU and shaders became the norm, algorithms were mostly implemented on CPU leaving only the rasterization process for the GPU, which further exacerbates this problem. Current GPUs, however, offer more than enough VRAM for most purposes. Nonetheless, rendering extremely large datasets may still require tight control over memory usage.

2.3 ROAM

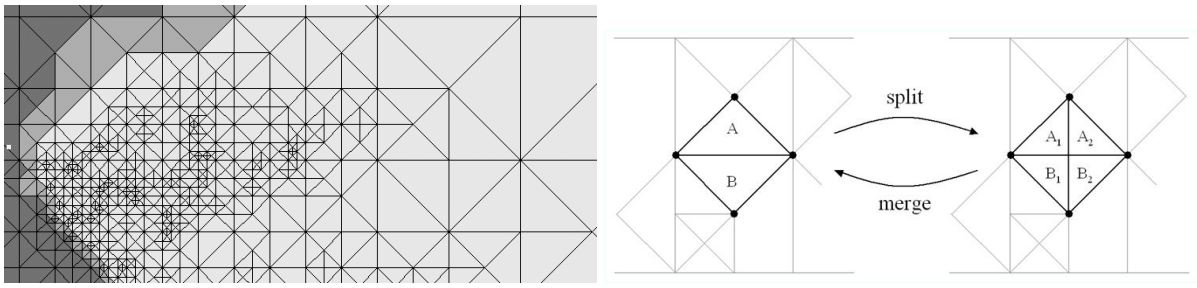
Real time Optimally Adapting Meshes (ROAM) [5] is a terrain rendering LOD technique that converts a heightmap into a triangle strip, built only of right isosceles triangles, using triangle binary trees. The triangle structure can be updated in real-time to take into account the viewer's position and LODs. Relying on frame-to-frame coherence, ROAM executes view-dependent rendering under the assumption that each frame has only very small changes, minimizing updates on the mesh, and exploiting view-frustum culling.

A triangle binary tree is built, bottom-up, in a pre-processing step by dividing the entire terrain in right isosceles triangles. The fact that the strip contains right isosceles triangles allows for seamless transitions in LODs without special care being taken to avoid cracks or T-Junctions. Updating the triangle binary tree involves two operations, split and merge. Each triangle can be split in half, in two children triangles. Two children triangles, resulting from a split operation, can be merged back together if the neighboring triangle in a diamond can also be merged. Splits may cause a chain of recursive splits if a T-Junction is formed during an intermediary step, until no T-Junctions are present.

The algorithm utilizes two priority queues during updates, for split and merge operations respectively, that aid in deciding which triangles should be split or merged. Triangles in the binary tree are tagged with a priority value. Flat geometry should have lower priority for splits and higher frequency geometry should have higher priority. This way the triangle strip maintains higher detail around high frequency data. An illustration of a resulting mesh and the split/merge operations is shown in Figure 2.4.

Updating the mesh occurs as the viewer moves or turns according to four phases. Recursive incremental update to view-frustum culling, update tagged priorities, triangulation update using greedy split and merge operations, updates for the triangle strips affected by the previous steps.

ROAM has since been improved upon by other works. Triangle binary trees create only an approxi-



(a) (b)

Figure 2.4: (a) An example ROAM triangle strip appears to retain more detail towards the left side where a higher number of triangles has been split. (b) The result of a split and merge operation in both directions.

mation of the original terrain, requiring many triangle splits to even come close to that quality. Reliance on geometric distorion on screen has been replaced by more effective error-metrics. The work limited the number of triangles rendered per frame and the time alloted per frame, ending rendering earlier if time was exhausted beforehand. However with new LOD methods and higher throughput of current GPUs, limiting rendering by the number of triangles rendered each frame is not a major concern anymore.

2.4 Geometry Clipmaps

Geometry Clipmaps [7, 8] is a rendering technique widely used in the gaming industry to handle terrain LODs. This technique creates mipmaps of the terrain's heightmap and clips those mipmaps (hence the name clipmap) into nested regular grids centered around the viewer, as seen in Figure 2.5, so that each render different parts of the terrain at different LODs. Each subsequent level has half the detail of the next finer level. Levels are selected based on the distance between viewer and terrain. Unlike with previous techniques, which create a global hierarchy, Geometry Clipmaps create a quadtree-like hierarchy centered about the viewer that may actually be smaller than the entire terrain, making it a scalable approach. In fact one of the terrains tested in [7] had a resolution of 216000x93000, with over 20 billion samples, stored with lossy compression.

There is a distinction between clip region and active region. The active region is the region surrounding the viewer that he should be able to see, and the clip region is the portion of the terrain currently clipped and ready to render. If the user moves too quickly the clip region may fall behind the active region and lower detail terrain will be rendered first, as the clip region attempts to keep up with the viewer's new position. The clipmap structure also allows incremental updates by shifting the clip region when the viewer moves, thus optimizing cache. Every frame the algorithm will determine the active region, update the clipmap, crop the active region and render.

LOD transition using transition zones is a direct way to fix cracks. Zones have a predefined length and a morph function which interpolates the height from the values in the current LOD and the next coarser LOD. T-Junctions are possible, however they're avoided by stitching level boundaries with degenerate triangles[1], which suffer the same rounding errors as the T-junction point and make the transition

[1] Triangles whose points are collinear. Due to rounding errors these points may be displaced.

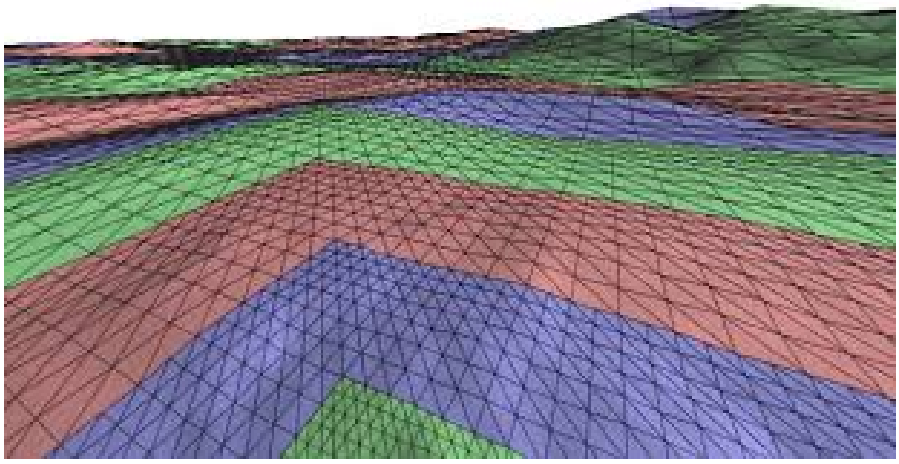


Figure 2.5: Top down overview of a clipmap. From "Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids" [7].

consistent.

Texture mipmap levels can be directly associated with clipmap levels, however custom view-dependant mipmapping is used, instead of hardware mipmapping, to match LOD transitions more accurately. Instead of calculating geometry normals, a normal map is stored with twice the resolution of the terrain geometry for better lighting quality.

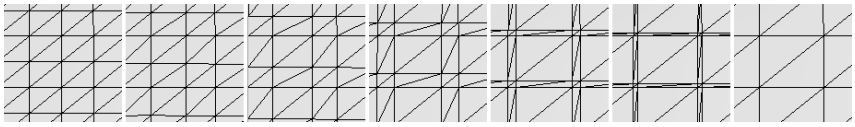
Terrain compression and synthesis are used. A lossy compression method is utilized which applies linear filtering to generate coarser levels and stores residual errors between levels as well as the coarsest level of terrain. From this data the system can upsample coarse levels into fine levels. Since compression is lossy and data is upsampled, a simple form of terrain synthesis is utilized to generate additional detail using fractal noise. Extra detail can be synthesized by calculating noise variance from coarser to finer geometry levels.

Some advantages of this approach are the intuitive division of the terrain in a clipped region, scalability due to clipping only a portion of the visible terrain, distance-based LOD and a method to create seamless transitions. Disadvantages include very noticeable popping artifacts due to the use of linear filtering, use of lossy compression (although different compression schemes may be used) and the employed distance based LOD metric does not take viewer altitude into account.

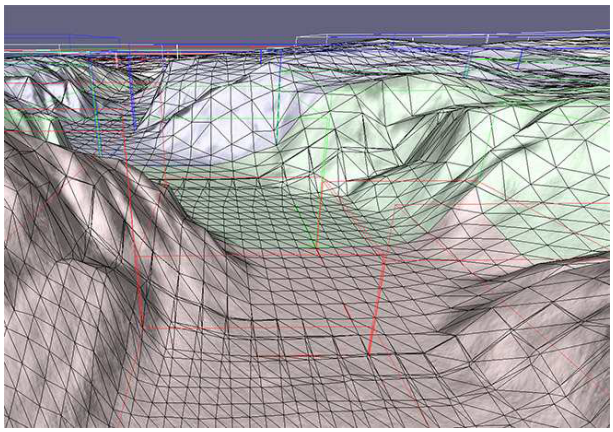
2.5 CDLOD

Continuous-Distance Level of Detail (CDLOD), presented in [6] is a LOD technique that improves rendering quality over previous techniques, namely the fact that Geometry Clipmaps does not consider viewer altitudes, by utilizing the three-dimensional distance from viewer to terrain in the LOD selection, per-vertex, and reducing LOD artifacts. The three-dimensional distance can be approximated, however the distance calculation must be deterministic and consistent so as to generate the same LOD for coincident vertices in different terrain nodes/tiles. As a result of using the three-dimensional distance, the LOD clip regions become circular instead of rectangular.

The terrain is structured into a quadtree where each level of the tree corresponds to one LOD. Nodes contain the minimum and maximum height values present in the region they cover. In the rendering phase nodes are tested for view-frustum culling and LOD selection.



(a) Morphing stages with varying morph rate.



(b) Terrain morphing in LOD transitions.

Figure 2.6: CDLOD morphing. From ”Continuous-Distance Level of Detail for Rendering Heightmaps” [6].

Each LOD contains four times the complexity of the next coarser level. LODs are selected in a logarithmic scale with distance so that all triangles' projections on screen, no matter how far from the viewer, will have similar size. Morph areas deal with LOD transitions from one level to the other using a morph rate value between 0 and 1, so terrain is simplified by expanding one patch out of the 4 patches in the finer level and effectively collapsing the others until the new level is reached. CDLOD morphing is illustrated in Figure 2.6. In order to prevent T-Junctions the collapsed triangles become degenerate triangles.

Despite not being explained too in-depth in the article, one of the accompanying demos utilizes a streamed quadtree which contains only the active terrain. This alternative allows both the use of dynamic terrains and scalable terrains that do not fit in memory, which happens to be the case for some of the tested datasets.

The quality gains from having less popping artifacts can be reflected in a slight performance hit. However no comparisons were made with other algorithms.

2.6 Terrain Ray Tracing

Ray tracing is a physics based model that solves the rendering equation. When employing ray tracing, instead of directly discretizing polygons into pixels on-screen, we cast rays into the scene, for each pixel on screen, searching for the first intersection with an object. From that intersection we determine the final pixel color taking into account lighting, other objects' shadows, reflections, refractions and other

phenomena. Contrary to rasterization many types of surfaces can be directly rendered from flat polygons to spheres, cones, parametric surfaces and even non-euclidean geometry.

The major issue with ray tracing is with the amount of computational power it requires. Most of the processing time is spent testing ray-object intersections. In a naive algorithm we would test intersections with every object in the scene, per ray. Therefore it is not surprising that ray tracing techniques attempt to optimize intersection testing, either employing fast intersection algorithms or spatial partitioning structures, in conjunction with ray traversal techniques, which reduce the number of intersection tests by the process of elimination.

There are three main categories of acceleration structures: Grids, Bounding Volume Hierarchies (BVH) and Binary Space Partitioning (BSP) trees. Grids divide world space into evenly sized blocks, and objects may be contained in multiple cells. A grid is traversed by marching rays through each intersected cell. A BVH is a tree of nodes. Each node represents an encapsulating bounding volume. If a ray intersects the topmost node then the algorithm can proceed to test the children nodes. The actual objects to be intersected lie at the leaves surrounded by a proxy bounding volume that has a less expensive intersection test. BSPs recursively subdivide space in two regions along splitting planes. In terrain ray tracing grids and quadtrees are more commonly used to accelerate ray traversal.

When we combine ray tracing and terrain rendering the challenges from both areas come together. Techniques attempt to make use of an acceleration structure, smart ray traversal or fast intersections along with an LOD scheme, geometry compression algorithm and as few I/O operations as possible in the case of large terrains. An exception is view-frustum culling which is implicitly done in ray tracing thanks to the use of partitioning structures.

We can classify two kinds of ray tracing algorithms, as referred in the paper on Maximum Mipmaps by C.Dick et al [9], those that require offline pre-processing to accelerate real-time processing and those with less rendering performance which do not need pre-processing. Offline algorithms face the problem of impractically high pre-processing times for increasing terrain complexity while real-time algorithms perform relatively worse on small terrains but can be applied to large terrains. In addition terrain ray tracing techniques have three things in common, they utilize some form of ray marching, an acceleration structure and an LOD scheme. Some convert the heightmap into a compressed triangle network while others directly traverse the original heightmap or a similar format.

2.6.1 Ray Marching

By definition, ray marching is a ray traversal algorithm that advances through the ray's path by sampling the ray equation in finite steps. We can think of ray marching as checking if each successive sampled point lies inside an object, in which case an intersection is very likely to have occurred.

In the context of relief mapping and terrain rendering many papers also refer to this technique as Linear Search where we sample the ray in constant linear steps and determine if the ray falls below the terrain's altitude at each sampled point. The problem with this traversal algorithm is glaringly obvious, if we choose large sampling steps the algorithm may skip intersections with thin objects, if we choose very

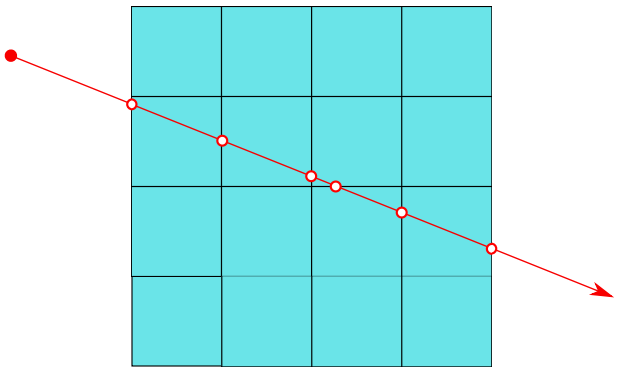


Figure 2.7: Ray traversal using ray marching on a grid.

small steps performance will suffer as more intersection tests have to be performed per ray and there is still no guarantee that intersection points will not be skipped.

Another variant of ray marching is Binary Search. Using Binary Search, instead of advancing through the ray in steps, we mark the start and end points on the ray's path which define a line segment, and then check whether the start point or end points lie inside the geometry. If this condition is satisfied we can subdivide the line segment in two segments and recursively apply the algorithm until we find an intersection point or reach a depth limit. This variant is more appropriate when the geometry, or terrain, is mostly flat or monotone, which is not necessarily the case for real terrains, but in general faces the same sort of limitations as Linear Search.

Ray Marching is used to aid traversal through acceleration structures. For instance, when traversing grids, as illustrated by Figure 2.7, we march each ray through every intersected cell based on the parametric equation deltas. We may also use binary search for terrain patch interpolation to pinpoint the ray-terrain intersection point. Since terrain ray tracing often utilizes heightmaps and grid structures, ray marching plays a big role.

2.7 Displacement Mapping Techniques

Some of the more recent techniques used in terrain ray tracing were originally developed for displacement mapping, which emerged in response to the limitations of bump mapping. Bump mapping utilizes a normal map, a texture storing surface normals, to add shading detail at each point on a surface. The end result is that shading has higher frequency detail due to the normal variation, but the surface itself is still flat. Ray tracing based relief mapping techniques attempt to solve this problem by actually modelling the geometry displacement, similar to what can be achieved with tessellation or geometry shaders.

2.7.1 Shell Maps

[10] introduces an interesting concept to displacement mapping techniques. Unlike similar techniques, which model the geometry displacement from heightmaps as depth values inside a surface, the Shell Map models height displacement outwards of the original surface as would be expected of displacement

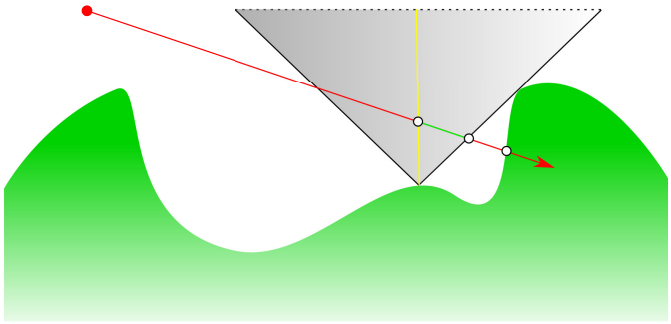


Figure 2.8: A cone being skipped during ray traversal.

mapping. Shell Mapping adds an extra layer of geometry over the original surface, figuratively creating a shell. In the context of this technique, the original surface is called the base surface, the surface at the outer end of this layer is called the offset surface, and the space that lies between these two surfaces is the Shell Space. A Shell Map is thus a mapping between a surface and a three dimensional volume, or in other words from texture space to shell space. Despite being used in relief mapping this algorithm has not been applied in the context of terrain rendering.

2.7.2 Cone Step Mapping

Cone Step Mapping (CSM) [11] was first developed as a ray tracing displacement mapping technique, although subsequent works applied it in terrain rendering. CSM is a technique that aims to improve ray traversal speed over a heightmap using inverted cones of empty space that can be entirely skipped when traversing rays. The empty cones are calculated in an offline pre-processing step and each sample of the heightmap will be assigned one cone. Cones are stored into a texture called a *Cone Map* in the form of cone opening angles.

Cones are assinged to texels based on the following invariant, every cone must be the largest possible cone that does not touch any part of the terrain. So the pre-processing step involves testing a very large combination of heightmap samples in order to find the most appropriate cone, which results in a very costly computation. When rendering we traverse the ray as usual by marching through each texel, but we now fetch the cone data at each texel and step the ray through the empty cone space, effectively skipping multiple texels at once. As a side note, since we only test the cone of the current texel rays only skip one half of the cone and never the entire cone. This traversal technique is illustrated in Figure 2.8. CSM and similar techniques are usually called ”Safety Zone” techniques, as they generate empty zones that can be ”safely” skipped during ray traversal without missing geometry details while skipping.

Relaxed Cone Stepping

Relaxed Cone Stepping (RCS) [12] is an improvement over the basic CSM algorithm. It relaxes the invariant utilized in CSM in the following way, each texel is assigned the largest cone possible such that any ray cast from the viewer will intersect the terrain only once. Combining the strengths of CSM and Binary Search, since each ray intersects the terrain at most once per cone, we can utilize Binary Search

to accelerate traversal and almost guarantee rendering is artifact free. There must, however, be a depth limit to binary search which means some artifacts may still appear. According to comparative tests in subsequent articles [9], Relaxed Cone Stepping can perform better than even some of the more recent algorithms. But this performance gain is not free, since cone data must be generated in a very slow pre-processing computation and placed into a texture. The expected pre-processing times are impractically high for terrain resolutions higher than 2048x2048. So the cost of using a cone map is the storage of an additional texture and long pre-processing times, which limits the use of this algorithm to low resolution heightmaps.

2.8 Signed Distance Fields

A Signed Distance Function is a function that calculates the signed distance from one point to the closest point in a certain set of interest, such as a surface or a boundary. In computer graphics the signed distance function may be calculated for points in a texture. This kind of texture is called a Signed Distance Field (SDF), typically a single-channel texture where each texel contains a signed distance value. Texels with distance 0 belong to the boundary while other texels will contain the signed distance to the closest point in the boundary. Points inside the boundary have a negative value and points outside have a positive value. Similarly to heightfields, SDFs do not represent an actual 3D object but a 3D surface, which in turn can be used to describe 3D geometry. 3D SDFs have been used for that effect which is not in the scope of this research.

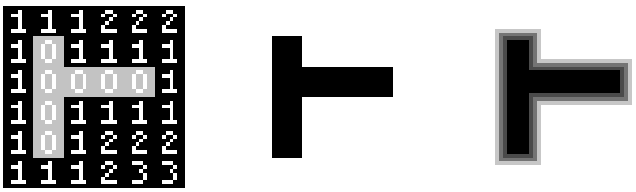


Figure 2.9: SDFs have multiple applications, one of which is to store fonts in memory. Rendering SDF fonts allows us to render text in multiple resolutions using a single bitmap texture, or apply effects such as antialiasing and outlines.

This technique has been applied in mesh rendering to model meshes with properties similar to heightfields. Given a boundary with a certain shape, we can determine the relative altitude of each point to the boundary shape. It has also been used in font rendering. Fonts often need to be pre-rendered into a texture at a specific size before being used, and if multiple sizes are needed then memory costs increase. The signed distance in a distance field could instead represent an alpha value used to smoother the characters' borders. Linear filtering can be applied to a signed distance field to smoothen the distance values, therefore a single distance field texture may be utilized for multiple font sizes. Figure 2.9 shows how SDFs can be used to render fonts.

Ray tracing/marching has also been employed on signed distance fields [13]. Since points outside the surface have positive values, we can march a ray over positive values until we reach a point with value less than or equal to zero. A quadtree or grid structure can be utilized to accelerate traversal and

skip multiple points outside of the surface.

2.9 Maximum Mipmap

The Maximum Mipmaps technique has proved to be a highly scalable solution [9]. Although Maximum Mipmap structures were already used in previous works, Christian Dick et al. greatly improve the ray traversal method for such structures by making full use of the quadree structure of mipmaps.

When we apply mipmap generation to textures, we will obtain successively smaller textures that are half the size of the previous textures up to a texture of size 1x1. Most usually the filters applied to the generation process are the nearest neighbour, linear interpolation, trilinear interpolation or others. Since terrains are stored as heightmaps we can apply those same operations. For the Maximum Mipmap structure we apply a maximum filter, where the texels in each texture level contain the maximum height value of the corresponding 4 texels in the next finer level. Thus this structure can be used as a quadtree, as illustrated in Figure 2.10.

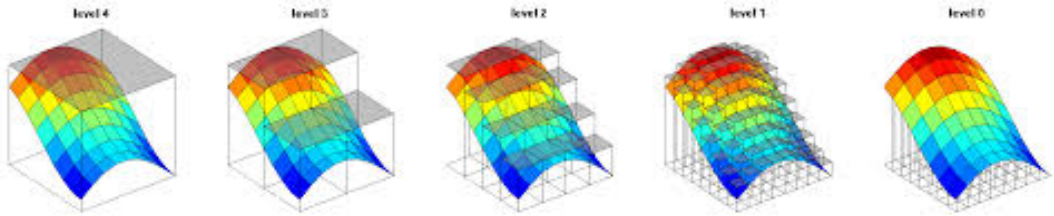


Figure 2.10: Illustration of a maximum mipmap structure on top of the terrain. From ”Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering” [9].

Ray traversal can be processed in the same way as usual for heightfields. However we can now use the Maximum Mipmap as a BVH. If a ray does not intersect the coarser levels of the Maximum Mipmap structure then it will certainly not intersect with the terrain. When it does intersect one of the mipmap levels, then we move the intersection testing to the next finer level of the mipmap. Once we reach the finest level, where the maximum values in each texel correspond the to the maximum height value of the corresponding terrain patch, we can attempt to calculate the real ray-patch intersection if there is one, otherwise we keep testing intersections until we find one or we reach the terrain bounds. This entire process has the potential to skip intersection tests with an enormous number of terrain patches, because skipping one texel in any level of the Maximum Mipmap structure is equivalent to skipping two texels in the next finer level, four in the second level and so on.

One way to improve this algorithm is to ascend one level in the mipmap structure when an intersection test fails, so that we don’t force the ray to take small steps each time if some of the intersection tests were satisfied but no intersection actually occurs.

Maximum Mipmaps is compared to linear search, binary search and relaxed cone stepping, performing better than linear and binary search but worse than relaxed cone stepping. While relaxed cone stepping requires too large pre-computation times for large terrains, maximum mipmaps can be employed in rendering large terrains as the mipmap structure can be generated on the fly with almost

negligible times. For this reason maximum mipmaps can also be used to render dynamic terrains. A limitation of this algorithm is that it can specifically be used with heightmaps in power-of-two resolutions.

The algorithm describes the intuition behind the structure but also leaves room for other additions. In particular LOD can be done by limiting traversal to descend at most to a certain level in the mipmaps base on the traversed distance, but Maximum Mipmaps can also be combined with different LOD schemes.

2.10 Hybrid Rendering

The hybrid approach presented in [14] is incorporated into a tile based terrain viewer which divides the terrain into multiple tiles. Instead of being employed in the entire frame, rasterization and ray-casting are selected in a per-tile basis. So for each renderable tile an *oracle*, or estimation function, decides which rendering method to use based on the number of triangles in the tile, how many pixels the tile projects to and a combination of constants determined for the specific GPU being used, which can be directly obtained from the specification or learned in a training phase. Ray-casting performs best on fewer pixels covered and rasteriation of few triangles,

There is a tradeoff to be considered, either keep in memory both structures for rasterization and ray casting increasing memory consumption or recreate the other structure on demand which increases CPU-GPU bus transfer slightly. Tests showed that the resulting CPU-GPU transfer overhead is relatively small.

Performance tests show that hybrid rendering may perform 2 to 4x better than previous algorithms.

While hybrid rendering combines rasterization and ray casting to get the most performance out of both methods, it also limits the visual effects that can be applied to terrain. Lighting can be calculated the same way for rasterization as for ray casting. But if we intend to render additional visual effects we would have to implement them in the exact same way for rasterization and ray casting. However that is impossible because rasterization cannot reproduce visual effects with the same quality as secondary ray tracing.

2.11 Ray Tracing on Geometry Clipmaps

The work presented in [15] employs ray tracing in a geometry clipmap structure. The technique is heavily inspired by [9] making use of maximum mipmaps together with a clipmap structure. The terrain dataset is divided in tiles both to make terrain data more manageable for IO operations and to simplify the clipmap structure. Every frame, when the clip center moves, the algorithm determines, for each clipmap level, which tiles need to enter the clipmap structure and updates the clipmap cache.

Ray casting can be terminated early using some metrics based on the available clipmap levels. It is possible that not all levels of the clipmap are available in memory (perhaps they were not fully loaded) and it is not possible to even test ray intersections with that level. This level of detail is called the low LOD. On the other hand, a minimum LOD is also selected based on the size of the current tile's projection

on-screen, called optimal LOD. Given the minimum value of both LODs, early ray termination happens if ray traversal is at the obtained level.

Summing up, this work combines the strengths from both clipmaps and maximum mipmaps. However it is not clear how intersections are handled for early ray termination and no solution is provided for avoiding LOD transition artifacts.

2.12 Pros and Cons

In this subsection a discussion is done about the researched algorithms with special emphasis on Geometry Clipmaps, Maximum Mipmaps and CDLOD. Table 2.1 summarizes the most important differences between the algorithms previously presented in this chapter. The more relevant techniques researched, in the context of this work, were Geometry Clipmaps, Maximum Mipmaps and CDLOD.

Algorithm	Scalable?	Pre-Processing?	Performance	Quality
ROAM	Yes	Adaptive Mesh	High	Low
Geo. Clipmap	Yes	No	High	Decent
CDLOD	Yes	Geomorph	Good	High
RCS	No	Cone Map	Very High	Very High
SDF	?	No	High	N/a
Max. Mipmap	Yes	No	High	LOD-dependent

Table 2.1: Brief comparison of related algorithms.

Geometry Clipmaps are still currently used in real time terrain rendering, namely in video games. Despite being originally developed with rasterization in mind, work has been made in adapting this technique for ray tracing. The LOD scheme employed by clipmaps is very flexible, allowing different LOD selection heuristics, and scalable. By employing ray tracing it becomes viable to traverse the heightfield without generating a corresponding triangle mesh. However, in that case, transition zones are not as straightforward to implement for ray tracing and another alternative may be needed.

Maximum Mipmaps is very simple to implement, very intuitive, and also specifically made for ray tracing. As demonstrated in [3], this acceleration structure allows ray traversal in very high resolution heightmaps and does not require extremely costly pre-processing like other safety zone techniques. In fact, the cost of generating a Maximum Mipmap structure is comparatively small and up to 4K resolutions it can be generated every frame without hitting performance. That is useful for dynamic terrains or high resolution terrains that do not fit in a single texture, in which case the maximum mipmap structure won't fit as well. A simple LOD scheme is suggested in [15] where rays can be early terminated based on the distance they've been traversed into the mipmap, but it is not clear how intersections are calculated in that case and how LOD transitions are handled. This technique can therefore be combined with other techniques that provide a well established LOD scheme.

CDLOD makes improvements on LOD selection and LOD transitions by reducing popping artifacts, fixing cracks and t-junctions. This is achieved while using the real three-dimensional distance from the

viewer to the terrain, contrary to the Geometry Clipmaps which utilize the horizontal distance to the terrain. Having less popping artifacts also improves shading, i.e lighting and shadows. More so in ray tracing since a lower resolution version of the terrain may show drastically different results for lighting.

Other techniques will not be considered for this work. ROAM focuses on generating a fast triangle strip which requires converting the heightmap into a triangle stip. While a triangle binary tree can be combined with ray traversal, traversal can also be done directly on the heightmap with no additional conversions. Shell Maps are not typically used in terrain rendering but rather to add detail to common objects. Relaxed Cone Stepping, although very efficient at run time, requires very long pre-processing times which are impractical for large terrains. Signed Distance Fields were a possible solution as well, yet, as far as my research goes, I found no implementation applied to terrain rendering. A Hybrid technique, aside from employing rasterization, would greatly increase the development cost as two algorithms would have to be implemented along with the heuristics that determine which rendering method to use.

Chapter 3

Terrain Ray-Tracer Engine Implementation

The research exposed in Chapter 2 served as a base for understanding the advantages and disadvantages of state-of-the-art algorithms, and older algorithm, in terrain ray tracing as well as to recognize how to leverage their qualities in a ray tracing engine.

In this chapter I will explain my implementation of a terrain ray tracing engine, starting with the implementation choices, overall structure of the engine as per the implementation's workflow, towards describing how data is stored and accessed in GPU and the algorithms implemented for ray tracing.

3.1 Algorithmic Techniques

Based on the research done, three algorithms seemed particularly interesting to implement in a terrain ray tracer: Maximum Mipmaps, Geometry Clipmaps and CDLOD.

- Maximum Mipmaps is the fastest scalable approach to heightmap ray tracing. With this algorithm it is possible to instantly visualize any heightmap with no need for pre-processing. The Maximum Mipmap structure resembles a quadtree and is built directly from the heightmap in its original image format, requiring no further conversions for traversal. From the fact that the structure resembles a quadtree, integrating an LOD control scheme is almost trivial. While Cone Step mapping achieves the highest frame rates for small terrains, the pre-computation times are simply impractical for large terrains, not to mention how complicated it would be to generate coherent Cone Maps across multiple terrain tiles.

- Geometry Clipmaps synergize very well with Maximum Mipmaps. The intuition behind Geometry Clipmaps lies in melding mesh from multiple resolutions of the heightmap, and Maximum Mipmaps already provide part of the structure required for that, each maximum mipmap level represents a bounding box over the equivalent resolution level of the heightmap, which enables intersection testing with lower resolutions of the terrain.

- CDLOD implements an LOD control scheme that utilizes the 3D distance as the distance parameter. This metric is more natural to ray tracing than rasterizaton because the 3D distance is implicitly determined for every ray traced. Unique LOD schemes may be possible with ray tracing.

So the basic idea behind the implementation is to read a terrain dataset, generate the Maximum Mipmap structure and low resolution mipmaps and perform ray traversal on the Maximum Mipmap while limiting the traversable levels with the 3D ray distance traversed to render the clipmapped terrain. However, as will be explained further in this Chapter, CDLOD was not implemented due to certain issues identified along the way.

Because ray tracing can naturally render non-polygonal surfaces, bilinear patch interpolation will be used instead of triangles to render smoother terrains.

In order to match [9] performance tests, multiple rendering modes should be supported to visualize different information at run time, including traversal steps heatmaps, LOD regions, normal maps, and even secondary ray tracing for shadows.

3.2 System architecture

Before delving into the details about the rendering pipeline of the system I will first describe the implementation choices in terms of programming languages and libraries. With this decision I wanted to enforce cross-platform support, so that the final ray tracer can be easily adapted to run in different platforms, and utilize common graphics software on a programming language that I was comfortable programming in.

Thus the project was developed in C++, OpenGL 4.5, GLSL, with the GLAD[1], SDL 2.0 and DevIL libraries.

- SDL2.0 is the layer that manages the window properties and the OpenGL context. It is a very commonly utilized library for cross-platform development of video games and other multimedia applications.

- GLAD is an OpenGL Loader Generator, a tool that generates an OpenGL loader - a file that defines OpenGL function pointers and values, similarly to more well known extensions like GLEW. This tool allows the user to customize their own OpenGL loader file, selecting only the necessary features and target OpenGL version and extensions. It requires no additional statically linked DLLs accompanying the executable file. The only requirement is to include two headers and one source file to compile the project. Lastly, SDL offers a function pointer that GLAD will utilize to find and load all the OpenGL functions.

- OpenGL/GLSL was selected for its widespread availability across GPUs and its cross-platform support. The version utilized during development was OpenGL 4.5, as it is the latest version and supports most extensions available. This simplifies development should a new extension become

[1] GLAD: Multi-Language GL/GLES/EGL/GLX/WGL Loader-Generator based on the official specs, github.com/Dav1dde/glad, glad.dav1d.de/

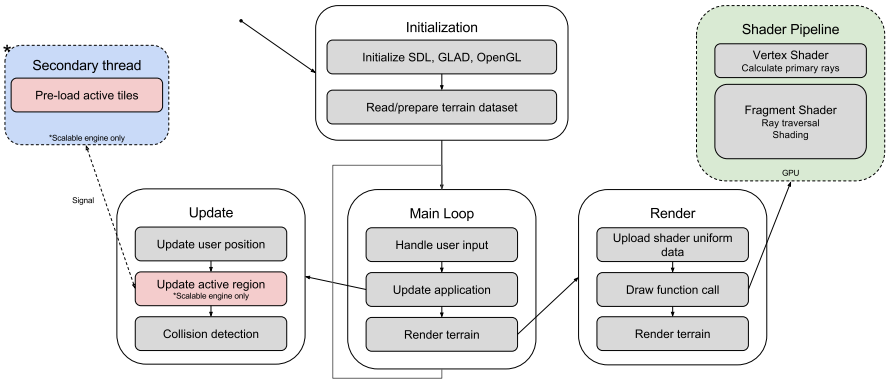


Figure 3.1: Terrain Ray-Tracer engine architecture diagram.

necessary for the project. Despite being compiled with OpenGL 4.5, the final implementation can be adapted to run with OpenGL 3.1, or later, since the most advanced feature it uses is GLSL Subroutines, a feature present only in OpenGL 4.0 or higher.

- DevIL, the Developer Image Library, is the library we selected to handle image loading, a crucial element of heightfield rendering.

The selected libraries are very commonly utilized by the programmers' community showing how accessible ray tracing can be. It is important to note also that ray tracing is entirely performed in the GPU through the OpenGL shader pipeline.

As a matter of fact, not one but two ray tracing engines were implemented. They are very similar but the first engine, a small-scale ray tracer, can render a single heightfield tile, while the second engine, a scalable ray tracer, can render large terrains split up into multiple tiles. The major difference between both engines is that the scalable ray tracer must manage an active region of nine tiles (3x3) and that rendering must take into account how the tiles are loaded in memory. In contrast, the small-scale ray tracer only needs to manage a single tile which, in a way, facilitates the addition and testing of new rendering features compared to the scalable engine where multiple tiles must be taken into account in the code.

3.2.1 Application workflow

Multimedia applications usually follow an event driven approach after intialization. There is a main loop, where event handling, application updates and rendering take place.

The architecture diagram of the engine's implementation is represented in Figure 3.1. Both engines have a similar workflow except for the scalable-exclusive stages.

- First the engines initialize the libraries and setup the terrain dataset.
- During the update stage, the user's position is updated and collision detection is performed.
- Lastly the terrain is rendered. For rendering we employ two shaders, the Vertex Shader and the

Fragment Shader. Other shader stages, such as the Geometry and Tessellation Shaders were deemed unnecessary as we will see in the following sections.

The difference between the two engines is how the active region is handled by the scalable engine. The scalable engine spawns a secondary thread to read and pre-process tile components in parallel to the main application as to prevent stalling. When the active region is shifted, the engine signals the secondary thread to start pre-loading the new tiles. One of OpenGL's limitations is that CPU multi-threading is not supported, so only the main thread can perform OpenGL operations. Therefore once a new tile is fully loaded, the main thread will be signaled to upload it to the GPU. How long the upload operation takes depends on the tile resolution, which is why tiles should have a relatively small resolution. Tile resolutions are also tied to the number of tiles present in the active region.

3.3 Terrain storage and texture formats

Before we explain the ray traversal algorithm we must define how the terrain data is stored in the GPU. This is the first step to be taken before implementing the system, because the Fragment Shader code will handle that data.

Terrain datasets are loaded in the initialization phase. The engine reads a custom configuration file which indicates the directories where the terrain elevation map and photo texture are stored, camera setup and other terrain properties.

3.3.1 Heightmap and Maximum Mipmap

There are two different storage formats for heightfields when stored as 2D images.

Per point format

This is the most commonly utilized heightmap format in 2D images. Per point means that each texel represents a single altitude value in the terrain. As only one value is stored per pixel the resulting image can be stored as a single channel format texture, or grayscale, to avoid wasting unused memory on colored formats. The reason why this format is preferred is that it is the more natural and efficient way to represent a terrain, requiring the least amount of memory. Images that store elevation data per point usually have POT+1 dimensions (257x257, 1025x1025, ...).

Per patch format

This format is less commonly used due to its higher memory requirements, but it is necessary for certain algorithms. Per patch stores one terrain patch per texel or, in other words, four neighbouring height values in the terrain. The resulting image can be stored as a four channel txture. Stored values are redundant since neighbouring texels will contain duplicate height values on their frontiers. The per patch format is expected to require about four times more memory than the per point format, however certain

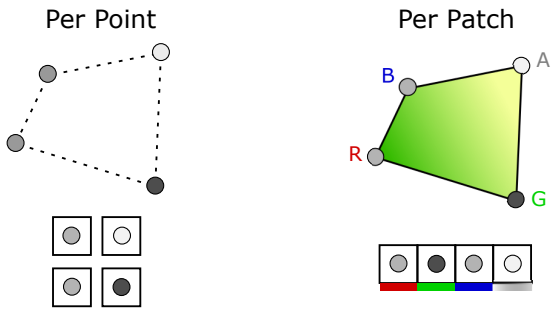


Figure 3.2: Pixel content comparison between per point and per patch formats. Per point format stores a single height value per texel while per patch format stores four neighbouring height values per texel.

algorithms require a pre-processing conversion to the per patch format because handling patches may be more convenient. Images that store elevation data per patch are normally converted from their per point counterparts to have POT dimensions (256x256, 1024x1024, ...).

This is a very important distinction to make. GPUs optimize processing for POT textures and non power of two dimensions should be handled differently or padded. Odd-sized images require special care. Not only heightfields, but other components of the terrain can be expressed in either format as well. A comparison between both formats can be seen in Figure 3.2.

Despite the drawback in memory consumed, accessing all the four height values in a terrain patch requires a single texture access compared to four accesses in per point format. For instance, the implementation in [9] employed the per patch format to reduce the number of texture accesses in Maximum Mipmap traversal to increase rendering performance, and our implementation also employs this optimization.

One step of the engine initialization is to convert the elevation map from per point to per patch format. This format is more convenient to build the Maximum Mipmap structure because each texel already represents a patch, and each mipmap level must have a POT resolution to effectively create a quadtree.

The resulting heightmap is also mipmapped into lower resolution versions of the that the engine can use to render the terrain with LODs.

The engine was implemented to read both 8 bit and 16 bit elevation maps in the PNG file format, though other image formats are also supported. Terrains can be stored in different formats, but PNG is a common file format and already contains elevation data in the desired format for our engine.

8-bit elevation maps can store up to 256 altitude values while 16-bit maps can store up to 65536 values. The range of values in 8-bit heightmaps is very small and causes real terrains to have clustered samples with repeated height values due to quantization. The result is that there is a sharp difference between consecutive altitude values, which can be visualized as if the terrain was made of stairs. 16-bit heightmaps, on the other hand, have a large enough range of values to render smooth terrains.

The difference between 8-bit and 16-bit heightmaps can be seen in Figure 3.3, showcasing the Colorado dataset in 8-bit and 16-bit. The 8-bit version appears to be quantized while the 16-bit version is much smoother.

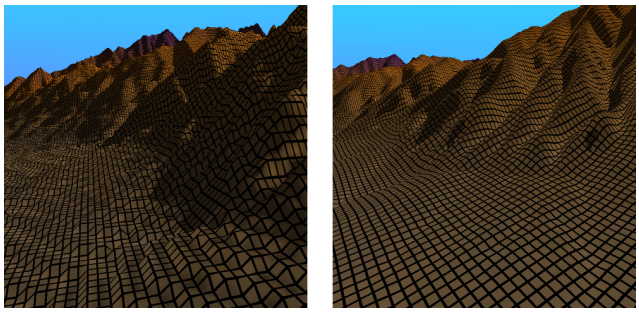


Figure 3.3: Comparison between an 8-bit heightmap with a 16-bit heightmap. The quantization is very noticeable.

3.3.2 Normal Map

Bilinear Patch Interpolation allows us to determine the normal at the intersection point by interpolating the patch corner normals. However these normals are not sufficient as they do not take neighbouring patches into account, causing lighting to appear flat. For this reason the engine pre-computes a normal map to achieve smooth lighting.

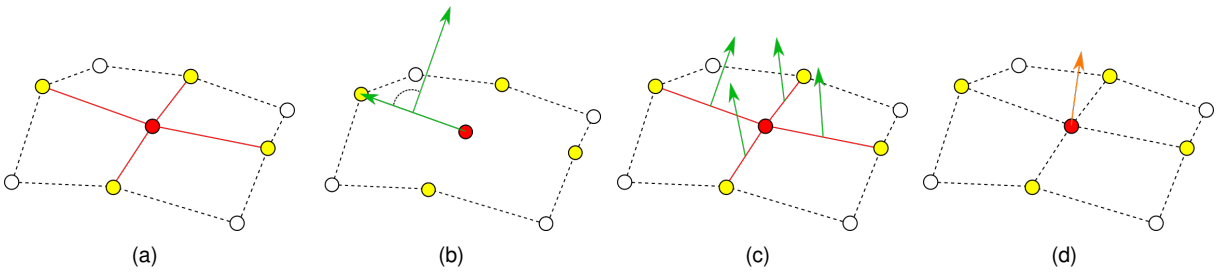


Figure 3.4: Normal filter calculation. Determine a normal for all four directions and average the resulting normals.

Normal maps are pre-computed using a simple filter illustrated in Figure 3.4. This filter calculates a normal for each point by averaging normals calculated for four directions: north, south, east and west. Taking the height difference between the center point and any point in one of the four directions we obtain:

$$directionVector = targetPoint - centerPoint$$

Where the target point is any neighbouring point in one of the four directions. This direction vector can then be rotated, 90 degrees, into a normal vector:

$$normal = normalize(rotate(direction))$$

After calculating all 4 directional normals, we determine their average and assign the final normal to point [x,y]. If the center point is on the terrain border and does not have a neighbour in any direction, then the filter will utilize a default vertical normal. For large terrains, the pre-processing should take tile borders into account.

$$N[x, y] = normalize(north + south + east + west)$$

The quality of normals obtained with this filter is decent, but higher quality normals can be obtained, for example, by applying a Sobel filter instead.

3.3.3 Photo Texture

Loading heightfields takes one last step, reading the photo texture. The photo texture is an RGB image whose resolution may be different from the elevation map, usually even higher to add extra detail.

Common resolutions for single heightfields vary from 1024 x 1024 to 4096 x 4096, but larger heightfields are divided into small tiles that can be managed more easily in memory. And typical tile sizes range from 512 x 512 to 2048 x 2048, higher than that may be too expensive to read on demand, lower may require reloading tiles too often. When photo textures have a higher resolution than the elevation map, then these images as a whole take up an enormous amount of space in disk as well as in memory when loaded into an engine, which is another reason why terrain tiles should be small.

Therefore the greatest concern when handling photo textures is to minimize the required space to store them, with compression techniques. Because these images have 3 color components, a photo tile may require over 6 times more memory space to be stored than an elevation map, depending on the resolution.

One common compression method for terrain imagery is S3 Texture Compression (S3TC) or DXTC. Five different types of compression are specified and they are named from DXT1 to DXT5. While other DXT formats are made to compress alpha values, DXT1 is the only DXT format made especially for opaque images, like photo textures, although it also offers the option to compress 1-bit alpha values. DXT1 can reduce up to 6x the memory required to store a 24 bit image, but it is also worth noting that it is a lossy compression format and the image quality will be reduced.

For this reason, our engine supports images in the DXT1 compression format using the OpenGL extension $GL_COMPRESSED_RGB_S3TC_DXT1_EXT$. When loading non-compressed images the engine will automatically compress them, although this feature may be turned off if the original texture is not too large to fit in memory.

One last note on textures dimensions. Since maximum mipmaps function as a quadtree, especially in the Fragment Shader code accessing parts of it, textures must be square and have power of two dimensions. If a heightmap does not meet these requirements the engine will automatically pad the texture. This padding is applied only to the heightmap and maximum mipmaps, the photo texture and normal map resolutions are independent. The scalable-engine does not support this feature.

The padded region can be skipped during traversal if entering an out-of-bounds patch. This has two downsides, one is consuming more memory than necessary for the original terrain and secondly render performance is reduced by introducing one extra condition. An alternative approach would be to utilize similar management techniques to those employed in the scalable engine, which will be discussed in the scalable engine sections.

Component	Data Type	Internal Format
Heightmap 8b	GL_UNSIGNED_BYTE	GL_RGBA
Heightmap 16b	GL_UNSIGNED_SHORT	GL_RGBA16
Maximum Mipmap 8b	GL_UNSIGNED_BYTE	GL_R8
Maximum Mipmap 16b	GL_UNSIGNED_SHORT	GL_R16
Photo Texture	GL_UNSIGNED_BYTE	GL_RGB
Photo (Compressed)	–	GL_COMPRESSED_RGB_S3TC_DXT1_EXT
Normal Map	GL_UNSIGNED_BYTE	GL_RGBA

Table 3.1: OpenGL texture formats selected for each component of the terrain.

3.4 Handling texture data

Some implementations store terrain as a geometry mesh, others an adaptive geometry mesh, some use clipmaps, or maximum mipmaps, and each may require slightly different intersection tests and traversal algorithms. Since our implementation utilizes Maximum Mipmaps and no particular polygonal mesh, the terrain components will be stored directly as 2D textures.

As mentioned before, in the section about texture formats, the engine must convert the elevation map into the per patch format and generate lower resolution mipmaps, build the maximum mipmaps from the highest resolution level of the elevation map, read the compressed photo texture and generate the normal map. Each of these textures must be stored with the appropriate format for their respective data. Table 3.1 matches each texture to the appropriate OpenGL format.

Although the engine can render terrains with 8 bits or 16 bits, the heightmap will always have 4 channels in per patch format but the underlying data type per channel is a single byte (unsigned byte) or two bytes (unsigned short) respectively. The Maximum Mipmaps are stored as a grayscale image as they require a single color channel to store the maximum altitude values, which have the same data type as the elevation map. The normal map is stored as an RGB colored image in order to obtain good quality normals. If memory space is an important constraint then the normal map can be stored in a single channel texture at the cost of lighting quality. Lastly, the photo texture is loaded as either an RGB image or a DXT1 compressed RGB image.

At this point, however, the ray tracer is still unable to process the elevation data. The reason is that the per patch format stores four height values, representing the lower left, lower right, upper left and upper right corners of the patch, in an arbitrary order defined by the programmer. Our implementation utilizes the following order:

```
Red = lower left altitude
Green = lower right altitude
Blue = upper left altitude
Alpha = upper right altitude
```

This order, seen before in Figure 3.2, must be consistently enforced throughout the entire program because different CPUs may represent words in different sequential orders (little-endian or big-endian) possibly causing bytes to be uploaded into the GPU in a a different order than intended. OpenGL offers a range of parameters that define byte orderings in addition to the data type, allowing the user to, for

instance, reverse or maintain byte order when uploading a texture. Care must be taken when reading, processing and uploading images. Once this order is estabilished both the initialization and ray tracing code can handle the elevation data properly.

3.5 Calculating primary rays

Calculating primary rays is no different from other ray tracing engines. All the necessary parameters for the ray calculation formula are also used in rasterization camera setups. We utilize the camera's position, referential, field-of-view and the screen resolution to generate a ray for each pixel.

$$\begin{aligned}
AspectRatio &= \frac{ScreenWidth}{ScreenHeight} \\
Screen_x &= \frac{pixel_x}{ScreenWidth} \\
Screen_y &= \frac{pixel_y}{ScreenHeight} \\
Cam_x &= (2 \times Screen_x - 1) \times tan(\frac{FOV}{2.0}) \times AspectRatio \\
Cam_y &= (2 \times Screen_y - 1) \times tan(\frac{FOV}{2.0}) \\
RayOrigin &= CameraPosition \\
CamDirection &= vec3(Cam_x, Cam_y, -1.0) \\
RayDirection &= normalize(CamReferential \times CamDirection)
\end{aligned} \tag{3.1}$$

All the rays start at the same point, the camera's current position. Each ray's direction depends on the field of view, screen resolution and respective pixel. The field of view works the same way as for rasterization, it controls the opening angle of the camera view and the greater the value the larger the rendering volume.

For rasterization the field of view controls the size of the rendering volume and in a way the number of polygons rendered, because a larger volume will capture more polygons in the scene and thus increase the rendering complexity per frame. For ray tracing that is not necessarily the case since the same number of primary rays is traced regardless of the field of view. The only possible issue ray tracing faces is that overall the rays may be slightly more spread apart and each traverse increasingly different paths in the heightfield traversal algorithm. Some ray tracing engines [16] trace batches of neighbouring rays simultaneously to optimize cache coherence, because neighbouring rays are more likely to traverse the same patches of terrain. However if the rays are spread farther apart due to an increasing field of view value then their paths may be less coherent. On the other hand ray tracing complexity is mostly dependent on the density of objects standing in the path of each ray causing the algorithm to test more, or less, intersections per ray.

In short, if we were to assume that a hypothetic scene exists where all rays, no matter their origin and direction, have the same cost in rendering performance, then modifying the field of view would not

reduce the ray tracing performance. The least that could happen would be a possible loss in performance on ray batch optimizations.

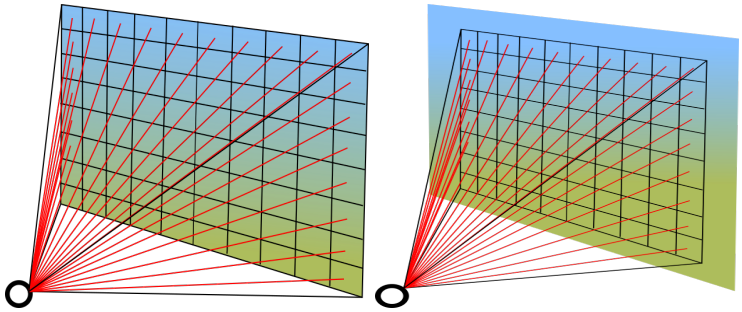


Figure 3.5: Vertical field-of-view comparison: (a) Has a higher FOVy than (b) and thus captures a larger volume of the scene than the latter.

The screen resolution dictates the number of pixels the scene is rendered to. In terms of ray tracing, the number of target pixels is the same number of primary rays traced into the virtual scene. Rays are directed towards the center of their associated pixel to obtain a centered image. To determine that center, the screen resolution, aspect ratio and the pixel coordinates are utilized.

Rays can be provided to the rendering engine individually or in batches, depending on optimizations and other implementation details.

Primary rays in our implementation

As mentioned before, this implementation employs the typical Vertex and Fragment Shaders from the shader pipeline. But the primary ray calculation is entirely done at the Vertex Shader stage.

Ray tracing, in the implemented engine, starts by rendering a screen sized quad. This quad is arranged into a VAO containing basic vertex and texture coordinates. The Vertex Shader is executed once per vertex in a rendered object, and thus four times for our quad. For each corner vertex the Shader calculates a primary ray from the formula above, using the texture coordinates and uniform variables to determine the ray's direction. The ray direction is then sent to the Fragment Shader and marked for interpolation. The Shader pipeline automatically interpolates variables sent to the Fragment Shader using the barycentric coordinates of the underyling primitive triangles. Basically this results in a linear interpolation of the rays that were calculated for the four corners. Once in the Fragment Shader, the rays must be normalized before being utilized in ray-heightmap traversal.

A naive approach would calculate each ray individually for each pixel, which can be achieved by calculating rays based on texture coordinates in the Fragment Shader. But in theory, ray directions are calculated based on a virtual quad, that represents the screen, and pass through the center of a pixel in that quad. Since ray directions are symmetric and equally distanced, the normalized corner rays can be interpolated to determine directions for all the other pixels.

In Figure 3.6 is a side-by-side comparison of the terrain ray traced using ray interpolation and texture coordinate interpolation. There is no apparent difference in the image at all. The only difference lies in

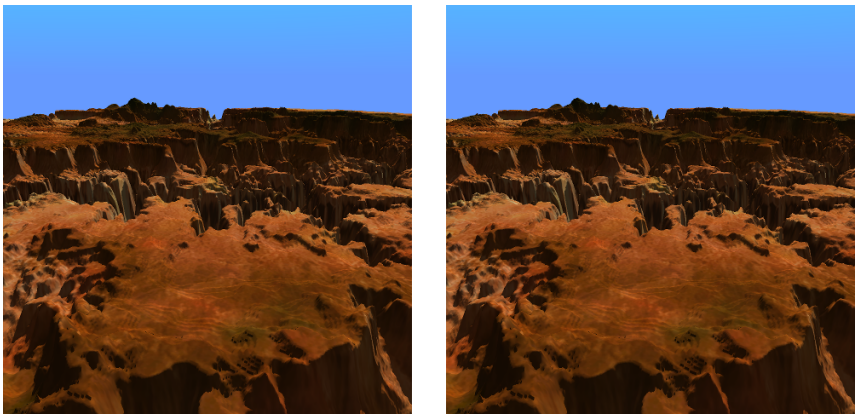


Figure 3.6: The image on the left was ray traced with interpolated rays, on the right rays were individually calculated in the Fragment Shader. The final image is exactly the same.

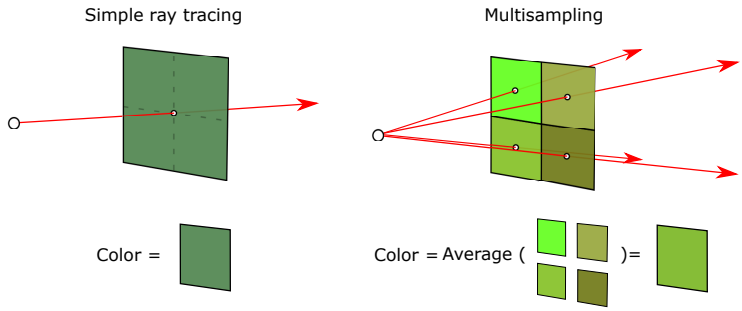


Figure 3.7: Simple ray tracing vs multisampled ray tracing.

performance. Both approaches were tested and the result is that interpolated rays are faster to compute, resulting in a small increase to frame rate.

Overall, rendering a screen sized quad might appear somewhat unusual. Perhaps rendering some sort of three dimensional bounding box like a cube would make more sense and would even allow the Fragment Shader to skip ray traversal on pixels outside of this bounding box. This is a very solid alternative to our rendering approach used even in [9] for relief mapping. However that would be considered hybrid rendering, where rasterization is used to render the terrain's bounding box on certain pixels and then ray tracing would operate only over those pixels. This approach, despite obviously utilizing rasterization to render a two-dimensional quad, attempts to make the rendering process as close as possible to pure ray tracing.

The fact that only a quad is sent through the rendering pipeline also means that there is no need to utilize the Geometry nor the Tesselation shader stages.

The greatest disadvantage to using this approach is that the screen-sized quad sets all depth pixels in the Z-Buffer to the same value, so OpenGL will not automatically generate a correct depth buffer in case we wanted to render other objects over the terrain. Fortunately there is a way to overcome this shortcoming and we will come back to that in the shading section.

OpenGL has a feature called multisampling that can be enabled to create an antialiasing effect. Once activated, each pixel is rendered as the averaged color of several sub-pixels. Because our engine lets OpenGL interpolate the primary rays automatically, enabling multisampling would also automatically

trace multiple rays per pixel to perform antialiasing with no additional changes to the code.

3.6 Ray Traversal Algorithm

This stage is entirely implemented in the GPU using OpenGL's Fragment Shader. The primary rays previously calculated in the Vertex Shader are automatically interpolated before being passed into the Fragment Shader, so when the latter is executed it must normalize the interpolated ray before further use.

The ray traversal algorithm is, naturally, the most important part of ray tracing. This stage encompasses several steps such as testing intersection with the terrain's bounding box, traversing the Maximum Mipmap structure, LOD control using the mipmapped versions of the original terrain (effectively a clipmap) and intersection testing with Bilinear Patch Interpolation.

```
bounding box intersection;
determine entrance/exit point;
initialize deltas;
while do Traversal Loop
    check stop conditions;
    determine limitLOD;
    if currentLOD < limitLOD then
        ascend mipmap level;
        recalculate deltas;
    end
    test intersection with maximum patch;
    if intersected then
        if currentLOD > limitLOD then
            descend mipmap level;
            recalculate deltas;
        else if currentLOD = limitLOD then
            bilinear patch interpolation at currentLOD;
    else if ascending is possible then
        ascend one mipmap level;
        recalculate deltas;
end
advance to next patch;
```

Algorithm 1: Ray traversal pseudo code.

The algorithm starts off by testing the intersection between the primary ray, traced from the current fragment[2], and the terrain's bounding box, basically a ray-cube intersection. To achieve this we utilize an adaptation of the Fast Grid Traversal algorithm [17]. The grid algorithm calculates the ray equation parameters for the minimum and maximum planes, in all 3 axis, that define the terrain's bounding box, as shown in Figure 3.8.

These parameters are then reordered, if the ray direction in any axis is negative, and validated, in case the ray does not actually intersect the bounding box even when some of the planes are intersected. When this validation fails, ray traversal is entirely skipped for that fragment. If validation was successful

[2]The Fragment Shader is executed once per fragment. Generally one fragment corresponds to a pixel, but if multisampling is activated a fragment may be smaller than a pixel.

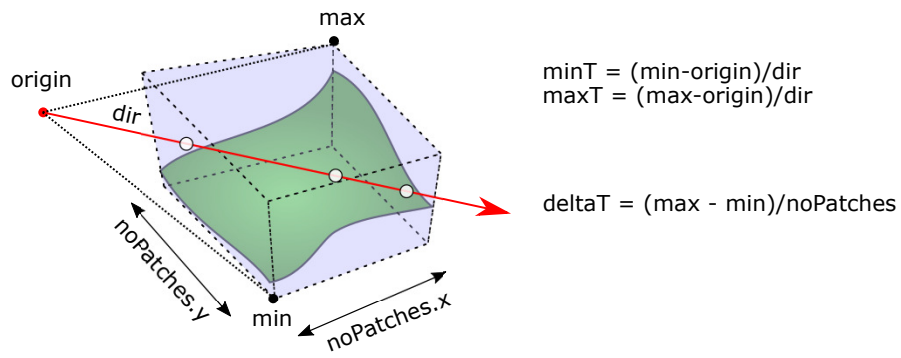


Figure 3.8: Bounding box intersection diagrams and formulas for calculating ray min and max parameters, as well as deltas, for each axis.

the next step is to initialize several variables for the traversal loop:

- We must determine the two points where the ray enters and exits the bounding box. If the ray's origin is already inside the grid then that point is considered the entrance point, otherwise we take the greatest parameter for the minimum bounding planes and calculate the entrance point from the ray equation. The end point is calculated the same way, applying the lowest parameter for the maximum bounding planes to the ray equation.

- The ray parameter deltas represent the distance that must be traversed in the ray to completely span a grid cell in one axis. These deltas are calculated as the difference between the maximum and minimum planes' ray parameters, calculated previously, and divided by the number of cells in the grid. These values allow the traversal algorithm to move one cell at a time.

- The next ray parameter values represent the starting point of the next grid cell to be tested. The algorithm tests if the ray intersects one cell and in case it does not then in the next iteration the next parameter becomes the current parameter. Effectively these values allow the algorithm to keep track of the correct sequence of cells to evaluate in traversal.

Regarding our implementation, there is a difference between the original grid algorithm and our adaptation. The maximum and minimum plane parameters are still necessary to test the ray intersection with the bounding box, but the Z axis deltas are not necessary because the heightfield is not a three dimensional grid. One other implementation choice regarding the bounding box intersection is to skip ray traversal if a ray going upwards intersects the bounding box from below. Because the maximum mipmap structure optimizes ray traversal from above the terrain, rays would be forced to traverse the highest resolution level and test intersections unnecessarily, causing a major performance hit in rendering. Instead, the terrain's boundaries are rendered.

Traversal loop summary

Ray traversal starts at the coarsest mipmap level which we will call *LastLOD*. This value is not zero, in OpenGL level 0 represents the original texture level. Rendering works usually refer to LOD or mipmap

levels as fine or corase levels, for high resolution and lower resolution levels respectively. Therefore the finest LOD level in OpenGL would be level 0 and, again, the coarsest level would be LastLOD.

The traversal loop must keep track of the current LOD, how many patches at the current mipmap level (eg. 1x1, 2x2, 4x4, etc) and patch width at each level (eg. 1024, 512, etc). This data will be necessary in the traversal loop to determine deltas, descend and ascend mipmap levels and test intersections with the terrain. We set these variables at initialization and update them when ascending or descending mipmap levels by dividing or multiplying by two, rather than on demand because they may require an expensive square root or an exponential operation.

After this initial setup we enter the traversal loop. As seen in Algorithm 1, the first step is to check the stop conditions. Ray traversal terminates under two circumstances, if the patch to be tested in the current iteration is out of bounds or if the distance traversed exceeds the far distance value, with the same meaning as the z-far value used in rasterization. For the purposes of visualization this value is a shader uniform variable that can be modified during execution.

Maximum Mipmaps, LOD control and intersection testing, will be further explained in the following sections.

3.6.1 Maximum Mipmap Traversal

Although the LOD control step comes before testing the ray against the Maximum Mipmap structure, both stages are tied to one another. However, underestanding how Maximum Mipmap traversal works on Maximum Mipmaps will make it easier to understand how LOD control is integrated into the algorithm.

The Maximum Mipmap structure is a mipmapped texture that contains at each texel the maximum altitude value found in the corresponding heightmap patch. For coarse levels, the maximum values represent the maximum value found in the horizontal domain that patches span at that level. Because this structure functions as a quadtree, we can say that one patch at a given level is the parent of 4 patches at next finer level.

Any ray that intersects the finest level of the Maximum Mipmap must also intersect at least one patch in all of the coarser levels. Therefore, our goal when employing the Maximum Mipmap technique is to determine if the ray intersects with all levels starting at the coarsest level. Ray traversal then proceeds to the finer levels for more specific intersection tests as the ray comes closer to intersecting the terrain. Upon intersecting the finest level one more intersection is performed with the terrain itself.

There are three operations that the ray tracing algorithm can perform on the Maximum Mipmap: perform ray-mipmap intersection, descend a mipmap level and ascend a mipmap level.

Ray-mipmap intersection

At any given iteration, the traversal loop keeps track of the entrance and exit ray parameter values of the current mipmap patch being traversed, which is done using the initial ray parameter values and ray deltas. In order to determine if a ray falls bellow the maximum patch's altitude we must calculate the altitudes at which the ray enters and leaves the domain of that patch

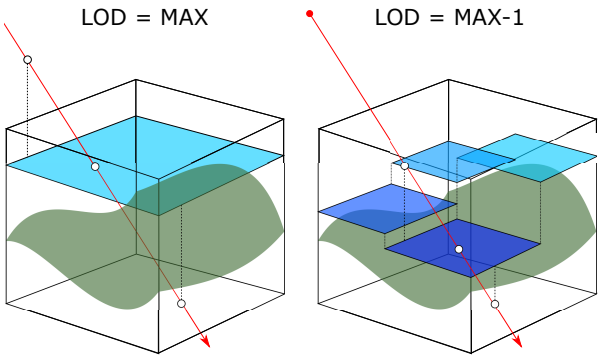


Figure 3.9: Each patch of the Maximum Mipmap represents a bounding box around the terrain's domain it covers horizontally. When an intersection is found at any particular level, traversal proceeds to the next level, at a finer resolution.

In Figure 3.9 the intersection between a ray and a maximum mipmap patch is shown. Rather than finding the exact intersection point it is sufficient to test the altitudes at the points where the ray enters and exits the patch's domain. If one value is above and the other below the maximum altitude at that patch, then there must be an intersection.

The case where both entrance and exiting points lie below the maximum patch altitude occurs when the ray is traced from outside the terrain's bounding box or when the ray originates from inside the terrain. In that situation, the traversal algorithm may descend to the finest mipmap level and test intersection with every patch until one is found or the ray exits the terrain, resulting in a large performance drop. An additional check may be necessary to prevent either case from causing performance drops.

Descending a mipmap level

When an intersection is found, ray traversal descends into the next finer level to pinpoint the real intersection more accurately.

Descending one mipmap level requires recalculating the indices of the first intersected patch at the next finer level, the entrance and exit ray parameters on that patch and the ray deltas for that resolution level.

Next level patch indices are determined based on the relative position of the entrance, or intersection point, between the ray and the parent patch and the center point of the parent patch. The center point is necessary because the entrance point usually lies in the boundaries of two patches, and converting its coordinates to integer indices may return incorrect values due to floating point errors, causing severe problems in the traversal loop.

```
indices.xy = indices.xy * 2;
if entrancePoint.x > centerPoint.x then
    indices.x = indices.x + 1;
end
if entrancePoint.y > centerPoint.y then
    indices.y = indices.y + 1;
end
```

Algorithm 2: Recalculating indices for the descending operation.

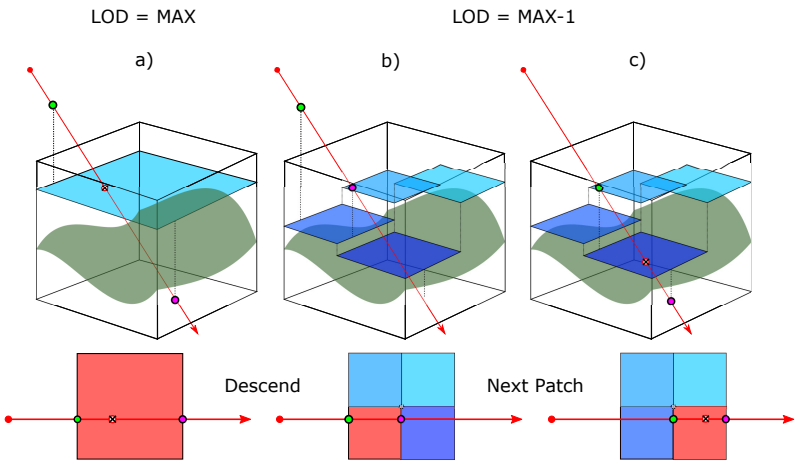


Figure 3.10: Descending operation example. Entrance point is colored green, exit point is colored magenta and the intersection point has a cross. a) An intersection is found at the coarsest mipmap level so a descending operation is performed. b) There is no intersection found at this patch so traversal moves on to the next. c) Again an intersection is found and another descending operation would be performed.

Algorithm 2 reflects how the indices are recalculated in a descending operation. Observing Figure 3.10, we can see that the neighbouring finer level has twice as many patches. From this property alone, multiplying the current indices by 2 yields the indices of the child patch in the lower left corner of the current patch. But the next patch to be traversed is one of the four children patches, the one that is traversed first from the entrance point. We find this patch by comparing the entrance point to the center point. If the entrance point lies to the right of the center point so does the patch. If the entrance point lies ahead of the center point so does the next cell. Crossing these two conditions yields the correct indices for the next level.

An optimization can be made that utilizes the ray-patch intersection point, instead of the entrance point, to determine next level patch. Because the maximum patch spans a bounding box over the terrain, skipping to the intersection point may result in skipping up to two traversal iterations. Back to the example in Figure 3.10, this optimization would translate to descending immediately to the lower right cell, effectively skipping one iteration. However, this optimization only applies to rays traversing downwards. Upwards rays may not skip any patch. In addition, care must be taken in the case that both entrance and exit points already lie below the terrain which may also occur in LOD transitions.

Deltas are calculated the same way as in the initialization stage of the ray tracing algorithm, take the ray parameters calculated for the minimum and maximum bounding planes of the terrain and divide the differences by the number of cells in the next level. Although the new delta values should be half of the previous level deltas, we opt to calculate these values from scratch so as to minimize floating point errors.

Ascending a mipmap level

Ascending a mipmap level, illustrated in Figure 3.11, is almost the opposite of the descending operation. It is so because ascending happens when intersection fails with the maximum mipmap but, unlike

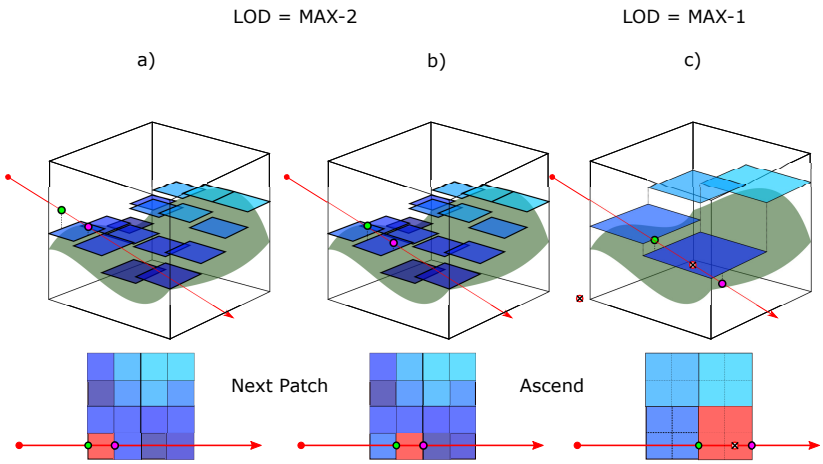


Figure 3.11: Ascending operation example: a) No intersection is found at the current level. Ascending is not possible because the next patch has the same parent on the coarser level. b) There is no intersection again, but this time ascending is possible. c) The ascending operation was done and traversal continues at the next level and next patch. MAX represents the coarsest level with only one height value.

descending, we want traversal to continue from the ray exit point from the current cell instead of the entrance or intersection point. For this reason there is an additional condition for ascending one level, it is that the ray exit point lies on the border of the parent patch in the coarser level. If it does not, then the algorithm would ascend to possibly repeat an intersection test with the coarser level and then descend again.

What this means is that the traversal algorithm should test against all the traversed children patches before ascending one level, so that the parent patch is never intersected twice. We can determine if ascending is possible by checking that the next patch has a different parent from the current patch. As an example, in Figure 3.11 (a) the next patch to be intersected has the same parent as the current patch, while in (b) the next patch has a different parent which allows the ascending operation to be executed, and (c) shows the result of the ascending operation.

The indices of the coarser patch are calculated by dividing the current indices by 2 and then adding the index delta at the coarse level to move on to the next patch.

The remaining parameters, deltas and exit parameters, are calculated from scratch as in the descending operation.

3.6.2 Terrain patch intersection

One of the advantages of using ray tracing is that the heightmap data is, to an extent, open to interpretation. In the per patch format, each texel contains the altitude values of all 4 corners of the underlying terrain patch. Meanwhile the intersection surface is left unspecified, which means that the implementation is free to perform an intersection test with any kind of surface. In addition it would be possible to implement different intersection surfaces and swap them at runtime - we did not, however, implement this functionality.

We implemented Bilinear Patch Interpolation [18]. While this option is also more expensive to compute than triangles, the quality of the output image is noticeably superior.

Once traversal reaches the lowest level of the Maximum Mipmap and an intersection is found at that level, the algorithm can perform an intersection test with the real terrain patch. An intersection is not guaranteed to exist and two different branches can occur:

- When intersection with the bilinear patch occurs, the last step of the algorithm is to prepare the hit data to be returned. The bilinear interpolation algorithm returns the intersection point from which we can deduce the corresponding texture coordinates in the heightmap and obtain the interpolated normal.
- No intersection happens when the ray falls below the maximum altitude of the terrain patch but only grazes the terrain. In that case, the algorithm will apply the ray deltas and continue traversal.

3.6.3 LOD control

LOD control is one of the least exploited features in terrain ray tracers. While previous works do utilize LOD metrics and transitions in rendering [3, 15], their approaches resemble typical rasterization engines, using square clipmaps and smooth transitions with triangles which require a pre-computed mesh. One reason for this behaviour is that real-time ray tracing works typically focus on performance over quality and settle for common LOD techniques. However, ray tracing has the potential to offer unique LOD control schemes.

Our original goal was to determine LODs on a per ray basis, based on the 3D distance from the primary ray's origin and the intersection point as it would be a specific metric to a ray tracing engine together with CDLOD. However several problems were identified with this approach. We settled for an approximation of the 3D distance but were unable to implement CDLOD. Three issues were identified:

- While previous algorithms that utilized simpler metrics would start LOD transitions on the boundaries of terrain patches, utilizing the real 3D distance in ray tracing means that LOD transitions could be possible in the middle of a terrain patch.
- The real 3D distance between the primary ray's origin and the intersection point can only be calculated once the final intersection point is encountered. An alternative metric to the real 3D distance must be employed.
- LOD regions are not shaped as rectangles or even circles. Due to using the 3D distance, each level may be defined by a unique shape, or multiple, as shown in Figure 3.12, depending on the terrain's relief. A specialized algorithm, or filter, would be required to meld two mipmap levels together.

As mentioned in Section 2, LOD control does not have to utilize the real 3D distance. An approximation is sufficient. But the LOD function should be coherent for every patch, meaning that the same patch should be assigned the same LOD regardless of the 3D distance at which different rays intersect.

Two alternative LOD schemes were considered to determine the 3D distance. One is to utilize the distance that has been traversed thus far in the Maximum Mipmap. The downside of this approach is that

Figure 3.12: LOD zones overlayed on the terrain with color coding. Using 3D distance, the shape of LOD regions depends on the terrain structures and may even be divided.

the LOD transitions can still occur in the middle of patches. The other alternative would be to determine LODs based on the center point of the Maximum Mipmap patches at each level. But this is not feasible because at coarse levels the center points between each patch may be too far and skip multiple LODs simultaneously, causing the algorithm to be unable to descend into the next level.

Combining both approaches produces a coherent LOD function. LODs can be determined based on both the center of the current Maximum Mipmap patch and the distance traversed. Descending operations are limited to the LOD determined by the distance traversed, while the center of the patch would determine the LOD level at which the intersections are performed with the patch. With this approach LODs are never skipped and each patch is always rendered with the same LOD.

One very interesting property of Maximum Mipmaps is how straightforward it can be to integrate with LOD control. Each mipmap level has an equivalent level in the heightmap and each value is the maximum altitude value of the corresponding texel (or patch) in the heightmap. Thus, whereas the highest resolution mipmap is a simplified bounding box over the original terrain, each maximum mipmap level is also a simplified bounding box over the same level elevation mipmap.

An LOD control scheme can take advantage of these properties to impose artificial limits on the traversal algorithm, so that descending operations cannot be performed past a certain level. If the user is far enough then intersections should be tested on a lower resolution version of the heightmap, instead of the original resolution. In the traversal loop, LOD control takes place before testing the maximum mipmap intersection. The formula we utilize for the 3D distance is:

$$limitLOD = floor(log2((distance + lodParameter)/lodParameter) \tag{3.2}$$

Similar to previous works, the LOD regions are determined based on a logarithmic scale with distance from the user. Every subsequent level covers twice the radial extent of the previous level. The *lodParameter* represents the radial extent of the first level. For example, if *lodParameter*=256 then the finest LOD covers a distance of 256, the second level has a radius of 768 but covers a radial extent of 512 because the previous level covers the remaining 256, etc. Figure 3.13 illustrates the use of this distance parameter.

For visualization purposes the *lodParameter* is a uniform shader variable that can be modified at run

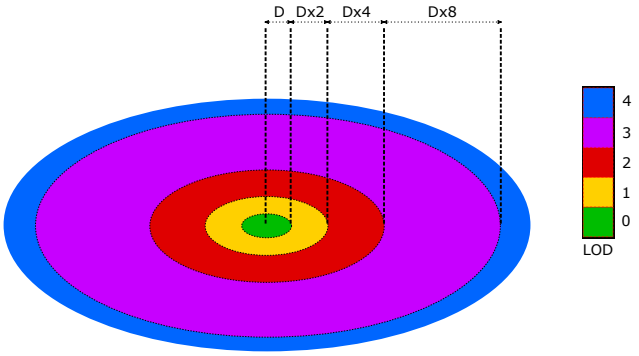


Figure 3.13: LOD levels are determined based on distance to the viewer and one distance parameter. This parameter represents the distance that spans the finest LOD. Each subsequent level will span two times the distance of the previous level.

time to change how the LODs are determined.

LOD control requires one more step. Each iteration keeps track of the current LOD where traversal is taking place. Whenever a new LOD limit is calculated, the algorithm must check if traversal was being executed beyond that limit. This situation may occur after an intersection fails or an ascending operation is executed, especially at coarse levels where ray steps cover larger distances causing traversal to skip directly into a new LOD. If that is the case, we must immediately perform one or multiple ascending operations until the limit LOD is reached. This ascending operation, however, at the risk of repeating an intersection, must not advance traversal to the next cell otherwise the algorithm may skip possible intersections.

```
limitLOD = lodControlFunction( rayParameter );
while currentLOD < limitLOD do
    ascend one level;
end
if intersected with maximum mipmap then
    if currentLOD > limitLOD then
        descend;
    else if currentLOD = limitLOD then
        intersect bilinear patch at limitLOD;
    end
end
...
```

Algorithm 3: LOD control integrated in the maximum mipmap traversal.

Integrating LOD into the already implemented Maximum Mipmap traversal is simple. After calculating the LOD limit we must test if an ascending operation is necessary and, when the maximum mipmap is intersected, prevent traversal from descending below that limit. This change is also shown in Algorithm 3.

3.7 Scalable engine differences

Despite following the same rendering pipeline, the fact that the scalable engine has to manage several heightfield tiles results in a few differences in how this data is stored and accessed during ray traversal.

As in the small-scale engine, the terrain dataset is specified in a configuration file. Tiles are named with a base name and a suffix of their respective indices. The configuration file specifies the directory, base name and file extensions, and the engine will concatenate that data to obtain the correct file name for each tile. In addition, two new parameters were added to the configuration file: visibility far and loading far. The visibility far (radius) is equivalent to the far rendering plane and represents the maximum extent of the scene that can be rendered, while the loading far (radius) determines the minimum distance the user must be from the active region limits to trigger an active region shift. The loading far should be larger than the rendering far so that when the active region shifts the new tiles are loaded before the user can reach and observe them mid-way. A good value for the loading far would be between half the size of one tile and $\frac{1}{4}$ of the active region.

The following subsections will cover further differences between both engines in terms of terrain data storage and access in the shaders.

3.7.1 Texture storage in the scalable engine

The scalable engine must load 9 tiles to initialize the active region on startup and 3 to 5 tiles when the user moves close to the limits of the active region. Each tile includes an elevation map, maximum mipmaps, photo texture and normal map. To facilitate indexing in the Fragment Shader we allocate one texture, for each terrain component, large enough to contain 9 tiles. The GPU we tested supports texture dimensions up to 32k, but a texture array could be utilized as well for GPUs that do not support large texture sizes. Loading a heightfield tile into the engine means reading all the three tile components, generating the Maximum Mipmaps structure and uploading each to their assigned slot in the active region.

When terrains do not have POT resolutions in the small-scale engine they are padded into the nearest POT resolution so that the Maximum Mipmaps can be generated. But for the scalable engine it is mandatory that tiles have square power-of-two dimensions, because padding terrains in parallel to the main rendering thread would increase tile loading times and decrease performance unnecessarily. Thus the resolution of the dataset as a whole must be a multiple of the tile's.

Every update cycle the engine must determine if the user is near the borders of the active region and decide if the region must be shifted. Active region updates involve removing inactive tiles and reading the new active tiles from disk.

While in the small-scale engine the terrain components are guaranteed to be contiguous in memory, this assumption cannot be made for the scalable ray tracer. Maintaining the terrain components contiguous in memory would involve shifting all tiles to their contiguous slot in the active region. This would be a very expensive operation to perform.

Instead of keeping the data contiguous, the engine loads new active tiles into the available slots left by the removed tiles and manages an array of indices that indicate which tile is loaded into which slot of the active region.

Each index corresponds to one of the nine slots in the active region textures and contains one unique

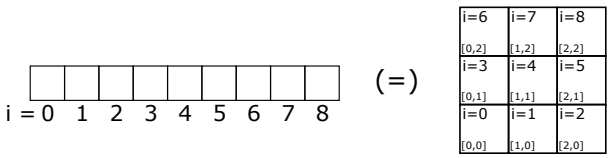


Figure 3.14: Active tiles stored in memory may not be contiguous within the active region texture. An index array keeps track of their positions in memory so that each can be accessed properly in the ray tracing algorithm.

value ranging from 0 to 8. Both indices and respective values represent a linearization of the bidimensional coordinates of each slot in the active region, as shown in Figure 3.14. The indices represent the contiguous tile slot and the values at the index represent the slot where that tile is stored. The engine performs an index conversion during ray traversal to access the tiles in a contiguous. In the next section we will revisit how this conversion is processed.

In case the region is shifted, the new tile indices that must be loaded are placed into a stack. A secondary thread is then signaled to load each tile in parallel so as to avoid stalling the main program and rendering. This thread will read the elevation map, convert into per patch format if necessary, build the maximum mipmaps, read the photo texture and generate, or read, the normal map. The next time the main thread enters an update cycle all the pre-processed tile data is uploaded into the GPU, at most one tile per frame to minimize hiccups in the main application.

One other possibility considered was the use of Sparse Textures, a relatively recent OpenGL extension that allows the use of partially resident textures. Partially resident textures may be partially loaded into memory on demand which is especially useful for large textures for which only a portion is required at any given time so the rest of the image does not have to be loaded into memory at the same time. OpenGL manages the underlying operations on this texture while allowing the programmer to control which portions of the texture are committed into memory or uncomitted. Sparse textures require setting up a page size with a certain resolution, for instance 256 x 256. Pages are the minimum sized texture data units that OpenGL commits into VRAM, and the sparse texture dimensions must be multiples of the page size.

Several issues arised from exploring this option. First is that the maximum mipmaps structure must be entirely present in memory so that ray traversal can be fully executed. Mipmap levels with at least one dimension smaller than the page size are not supported for sparse textures, because they cannot be committed as pages. Also, combined use of compressed textures and sparse textures is not guaranteed by the implementation, therefore loading photographic images as sparse textures is not guaranteed to work.

In the end, we scratched the idea of utilizing sparse textures and opted to manually manage the active region using multiple textures, one per component, that can store all tiles as a 3x3 tile matrix.

3.7.2 Scalable engine tile conversion

In the previous section we explained how tiles are stored in memory. When the active region is shifted, tiles that fall outside are removed and new tiles are loaded into the resulting empty slots of the active region. For this reason tiles may be stored in any order, which is maintained by an array of indices indicating the correct ordering. To enable traversal, the Fragment Shader must have access to these indices to perform an index conversion into the correct tiles.

The active region textures are mipmapped up to $\log_2(TileSize)$ where the coarsest mipmap will have dimensions 3x3, one texel per tile. Mipmapping any further would be difficult, as it would require an additional effort to handle mipmaps across tiles, and perhaps unnecessary.

Our ray tracing implementation utilizes both world and texture coordinates. Ray traversal and intersection data, such as the ray origin, ray direction, intersection points and patch corners are expressed in world coordinates. Integer indices are used for Maximum Mipmap and heightmap texture accesses.

The following formulas are used to determine the index conversion vector:

```
LinearIndex = X + Y * 3
IndexArray[ LinearIndex ] = RealIndex
RealX = RealIndex mod 3
RealY = RealIndex / 3
Conversion Vector = [ RealX-X, RealY-Y]
```

The index conversion vector enables the conversion from contiguous tile indices to the real indices of the active region slot where that tile was stored. The contiguous tile indices are the indices where a certain tile would be contiguous in the active region. The real indices indicate where that tile was actually stored in memory.

An example can be seen in Figure 3.15. The left side is shows the active tiles' layout in memory, in the center is the state of the indices array, and on the right what the active region should look like after index conversion.

First we need the linear index, whose meaning was illustrated in Figure 3.14. It ranges from 0 to 8 and is obtained as a linearization of the contiguous indices. At position *IndexArray[linearIndex]* is the value *realIndex*. This value is the linearization of the real indices. Finally, the index conversion vector is obtained as the difference between the real and the contiguous indices.

Here is how the example from Figure 3.15 would be determined:

```
1.  X=0, Y=2
2. LinearIndex = 0 + 2 * 3 = 6
3. RealIndex = IndexArray[ 6 ] = 8
4. RealX = 8 \% 3 = 2
5. RealY = 8 / 3 = 2
6. ConversionVector = [2-0,2-2] = [2,0]
```

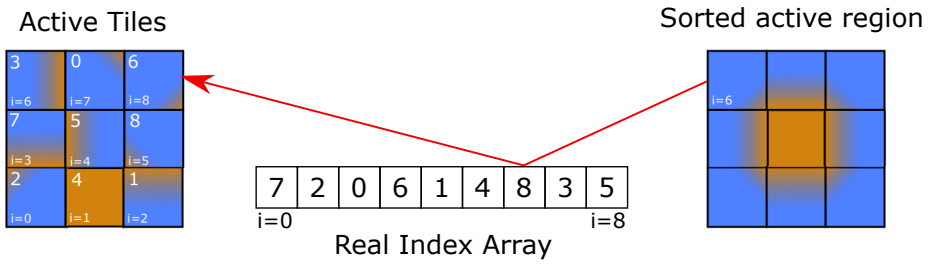


Figure 3.15: Illustration of how active tile conversion applied. The tile that should be rendered at the top left corner of the active region, index 6, was stored at index 8 in memory. An underlying conversion array of [2,0] is used to perform the correct access.

Texture indices are expressed as contiguous indices in the ray tracing algorithm. Every iteration traversal is calculated as if the active tiles were contiguous. Only when performing texture accesses do these indices need to be converted to the real indices. The meaning behind the conversion vector is the number of tiles that the contiguous indices must be shifted to point to the real tile. In the example above, the contiguous indices [0,2] would be converted to [2,2], which matches the conversion done in the figure.

Thus, the conversion vector represents a distance in terms of active tiles. An additional operation must be done depending on the current LOD being traversed. The conversion vector must be multiplied by the number of patches the tile contains at that LOD, in any axis. The coarsest level only has one patch per tile.

As an example, assuming the original tile has a resolution of 1024 patches and traversal is occurring at LOD = 2, the final conversion vector would look like this:

```
1. FinalVector[LOD=X] = ConversionVector * NoPatches[LOD=X]
2. Resolution[LOD=0] = 1024
3. Resolution[LOD=2] = 256
4. FinalVector[LOD=2] = [2,0] * 256 = [512,0]
```

In terms of the algorithm's code, three modifications are required:

- Before the traversal loop, the current active tile must be determined.
- Due to the ray parameter advances done every iteration, a new condition must be added to determine when a new macro tile is reached or the active region is exited.
- All texture accesses must be done with the conversion vector.

The code listing in Algorithm 4 shows how the starting active tile is determined. Traversal starts at the coarsest level, which is divided in 3 patches per axis. The active tile indices, which we refer to as macro indices, are based on the ray's direction in each axis, x and y, and the difference between the starting traversal point of the ray and the lower left corner of the active region.

```
int currentLOD = uMaxLOD;
float divLOD = 3;
ivec2 macroIndex;
macroIndex.x = int( clamp( ( startPoint.x - uHeightmapLL.x ) / uTileDimensions.x, 0, 2 ) );
macroIndex.y = int( clamp( ( startPoint.y - uHeightmapLL.y ) / uTileDimensions.y, 0, 2 ) );
ivec2 conversionVector = ivec2( uTileIndices[ macroIndex.x + macroIndex.y * 3 ] % 3,
 uTileIndices[ macroIndex.x + macroIndex.y * 3 ] / 3 ) - macroIndex;
```

Algorithm 4: Determining the starting active tile in the scalable engine.

```
vec2 tileCenter = vec2( (float(indices.x)+0.5)*patchLODSizeX, (float(indices.y)+0.5)
 *patchLODSizeY );
if !isInMacroTile( macroIndices, tileCenter + uHeightmapLL.xy ) then
    if !isInActiveRegion( indices, currentLOD ) then
        return result;
    end
    macroIndex.x = int(indCenter.x / uTileDimensions.x);
    macroIndex.y = int(indCenter.y / uTileDimensions.y);
    int lindex = uTileIndices[ macroIndex.x + macroIndex.y * 3 ];
    conversionVector = ivec2( lindex % 3, lindex / 3 ) - macroIndex;
end
```

Algorithm 5: Determining if traversal has reached a new active tile.

In the traversal loop, the algorithm must also determine when traversal enters a different active tile. Because the coarsest level has an odd number of patches, the Maximum Mipmap algorithm must handle that level differently.

Every iteration in the traversal loop the ray tracing algorithm must determine if the current patch to be intersected belongs to a different tile than in the last iteration, or is outside of the active region. This condition can be evaluated before LOD control because the ascending operation caused by LOD differences does not advance the patch, and the macro tile indices do not change. Algorithm 5 illustrates this check.

When entering the traversal loop for the first time, we take the assumption that the macro indices are valid and traversal occurs inside an active tile. A patch can only be outside of the active region if it belongs to a different tile. so as an optimization the active region check is done within the macro tile branch.

The rest of the algorithm remains the same, with the exception that texture accesses require an index conversion.

3.8 Shading

Shading is the last stage of the ray tracing algorithm. Once traversal is complete, a structure is returned containing data on whether an intersection occurred, the intersection point in world and texture coordinates, the interpolated normal, the ray equation parameter for the intersection point and the intersection altitude. Other data not related to the intersection itself includes the LOD where the intersection was found and the number of traversal iterations (also called steps) the ray has performed.

Starting in OpenGL 4.0 a new feature was introduced to shaders, Shader Subroutines. Subroutines

allow the programmmer to define a shader function signature as a variable type, which can be utilized as a data type for uniform variables which works similarly to a pointer to a function. Subsequently, the programmer can write shader functions, that adhere and refer to the subroutine's signature, and modify the function pointer from the CPU, through the appropriate OpenGL functions, by passing the function's name to the pointer. This feature effectively can replace code that in previous versions of GLSL would be implemented as if-else chains.

```
subroutine vec3 ShadingType( RayHit hit );
subroutine uniform ShadingType ShadingFunc;
...
subroutine (ShadingType)
vec3 photoShading( RayHit hit )
begin
    return texture( uPhotoTexture, hit.texCoords );
end
```

Algorithm 6: Shader subroutine sample code.

Our implementation utilizes this feature to define shading subroutines. The intersection data returned by the ray traversal algorithm is passed to a uniform shading subroutine, as seen in Algorithm 6, based on the current rendering mode. Here follows a list of the rendering modes implemented:

- Photo Shading
- Height Shading
- Depth Shading
- Grid Shading
- LOD Grid Shading
- Lighting Shading
- Lighting Grid Shading
- Normal Map Shading
- Ray Step Shading
- Secondary Ray Steps Shading

Any of these subroutines can be freely swapped during execution as well as edited and recompiled to provide different rendering results in real time. They facilitate the visualization of data. The last two shading modes will be further explained in Chapter 4.

3.8.1 Photo shading

Taking the texture coordinates from the intersection structure, the photo texture is sampled to color the heightfield. This is the default shading mode and is intended to render terrain with the same quality as expected from other terrain renderers.

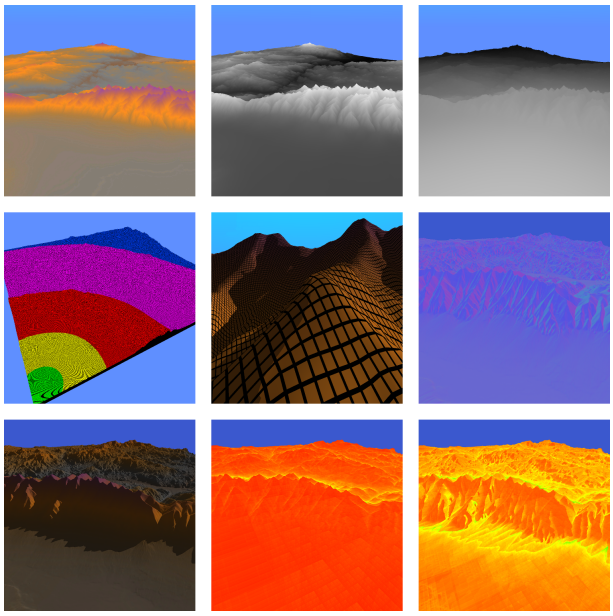


Figure 3.16: Showcase of shading functions. From top to bottom and left to right: Photo Shading, Height Shading, Depth Shading, LOD Shading, Light Grid Shading, Normal Shading, Light Shading, Ray Step Shading, Secondary Ray Step Shading

3.8.2 Height Shading

- Instead of sampling the photo texture, the heightfield is colored based on the intersection altitude effectively rendering the heightfield with the same color as the original grayscale elevation map. With this mode one can visualize the relative heights between parts of the terrain.

3.8.3 Depth Shading

From the distance value returned in the intersection data, it is possible to render the scene as a depth map where closer intersection points are colored towards white and farther intersections are color towards black. This type of shading has been hinted at in the section on Calculating Primary Rays. Due to the way our implementation calculates primary rays, OpenGL does not automatically generate a correct depth map. If the developer were interested in rendering other objects besides the terrain, that could be achieved by rendering Depth Shading into a depth texture bound to a secondary Fragment Shader output. The resulting depth texture would then be utilized in a second rendering pass to draw other objects. Another, more expensive, alternative would be a second pass with the Geometry or Tessellation shaders to generate real geometry from the depth map.

3.8.4 Grid Shading

This rendering mode overlays a grid on top of the terrain while helping the user to visualize patch limits. Although the grid is rendered taking LODs into account, the transition between grid LODs is difficult to spot.

3.8.5 LOD Grid Shading

In addition to grid shading each LOD is shaded with a keycolor. The highest level is shaded with green, followed by yellow, red, purple and blue. Any subsequent levels are shaded with the same color because any levels beyond 4 are not normally rendered due to the long distance. Different terrain parameters, such as the patch size, would make it possible to render more LODs.

3.8.6 Lighting Shading

The most important shading subroutine implemented. Light Shading is the subroutine that best showcases the rendering quality of ray tracing. Secondary rays are cast from the intersection point towards a directional light source using the same ray traversal function as the primary ray. If an intersection is found with the terrain then the light source is obstructed from the primary intersection point and is in shadow. Otherwise, the intersection point is shaded using a combination of diffuse and ambient lighting.

As with the original ray tracing algorithms, secondary rays must be traced with a small offset from the intersection point, otherwise they will sometimes collide with the intersection point itself causing unwanted shadow artifacts. However, both the patch interpolated normal and the normal map normals are used. The interpolation normal is used to apply an offset to the intersected point while the normal map normals are to calculate lighting. Effectively, this allows us to avoid generating mipmaps for the normal maps, which remain in a per point format, and use the original normal map as a bump map to add detail on the coarser resolutions of the heightmap.

It is important to note that shadow rays are subjected to the same LOD quality control mechanisms as primary rays. To keep the rendered scene consistent, shadow rays must utilize the 3D distance based on the origin of the primary ray instead of their own origin. Otherwise each individual shadow ray traverses with their own LOD control data, causing severe shadow artifacts on coarser levels of the terrain.

Adding ray traced lighting to LODs also opens the door for a new array of rendering artifacts: shadow popping and shadow cracks. Shadow popping occurs when visualizing LOD transitions on the terrain and are much more noticeable than terrain popping. Shadow cracks result from cracks in the terrain when the LOD transition is not smooth even if it is continuous. More detail will be provided on these artifacts in the last section of this chapter.

Casting secondary rays greatly reduces the performance of the rendering algorithm as we will see in the Performance Tests section. Even if shadow rays begin above the terrain and travel upwards, they force traversal to descend into the highest resolution levels of the heightmap pyramid.

3.8.7 Lighting Grid Shading

- A debug mode resulting from a combination of Grid Shading and Lighting Shading, which makes easier the visualization of lighting near patch limits.

3.8.8 Normal Map Shading

- Another debug mode that overlays the normal map on top of the heightmap. This shading mode allows us to check if the normal map is consistent with the geometry of the terrain.

3.8.9 Ray Step Shading

- In order to help evaluate performance comparatively, we implemented this rendering mode which was mentioned in [9]. Basically each pixel in the final image represents the number of steps that the corresponding primary ray was subjected to in the ray traversal algorithm. Lower values can be observed for lower resolution LODs and for flat terrain. Higher values can be observed near higher frequency terrain and higher resolution LODs.

3.8.10 Secondary Step Shading

- Based upon the previous shading mode, this mode simulates the Lighting Shading mode. Taking the intersection result from both the primary and shadow rays, the number of steps from both is added and used to color the image. The final result is an image using the same color coding as Ray Step Shading and following the same nuances. In addition, shadow rays cast from intersection points near a shadow border have a higher number of steps than those in the middle of shadow. Shadow rays that graze the terrain also show a higher number of steps.

3.9 Rendering artifacts

Because the implemented LOD control method does not create smooth transitions, gaps can occur in LOD transitions both in primary tracing and secondary tracing. For the lack of a solid LOD control approach, a workaround was implemented to fill gaps in the terrain.

Bilinear patch intersection between the ray and terrain may fail. The typical reason why this happens is that the ray does not fall below the patch which results in traversal moving on to the next terrain patch. But this situation may also occur if the ray has just transitioned to another LOD, in which case the next level patch may not be continuous anymore causing the ray to go underneath the bilinear patch. As a side-effect, the ray would continue to intersect with the finest level until it reached the end of the terrain, causing severe performance hits when too many rays behaved this way.

This issue was fixed checking if the ray enters a terrain patch already below the entry altitude value. In which case an intersection point is returned anyway to fill in the gaps.

This workaround completely solves the problem of gaps for primary tracing but the problem persists for secondary tracing. The reason is that these filler points actually lie below the terrain creating a small vertical wall and, when traversal begins, there is a high chance that the shadow ray will immediately intersect with the terrain very close to the primary intersection point leaving that point in shadow when it should not be. These gaps, shown in Figure 3.17, are visible as a circle of shadow artifacts near LOD

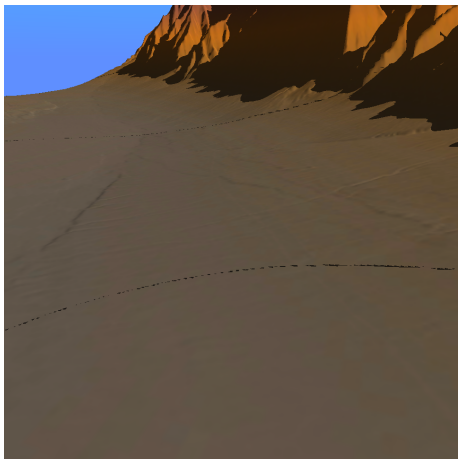


Figure 3.17: Shadow artifacts visible at LOD transitions. Due to instant altitude differences secondary rays immediately intersect the terrain.

transition points. Although they may become less visible by increasing the LOD distance parameter, a better control scheme is necessary for secondary tracing.

Popping artifacts are also hardly noticeable with primary tracing. But with secondary tracing shadow popping is very noticeable, more so due to the visible gap shadows.

Chapter 4

Evaluation and Results

Three kinds of tests were performed on both engines: frame rate, number of rays per second and number of traversal steps per ray. In addition to those three tests we also provide an analysis to the GPU memory consumed during execution.

Figures 4.1 and 4.2 show the terrain datasets utilized in the performance tests. All tests were performed in the same machine with the following hardware specifications:

- OS: Windows 10 x64
- CPU: Intel Core i7-6700HQ @ 2.60GHz - 3.50GHz
- RAM: 16GB DDR4 2133MHz
- GPU: NVIDIA GeForce GTX 1060 6GB (Mobile)
- HDD: 1TB 7200rpm SATA3

Our terrain base includes terrains procedurally generated with perlin noise and real terrains, with 8-bit and 16-bit heightmaps, in varying resolutions:

- Perlin generated (terrain in 1k, 2k, 4k, 8k, 16k, 32k)
- Puget Sound (terrain in 2k,4k,16k)
- Grand Canyon (terrain in 2k x 4k)
- Colorado (terrain in 2k, 3.6k, 4k, 16k)

The Synthesized terrain was generated with perlin noise. The Puget Sound and Grand Canyon datasets were obtained from the Georgia Tech website[1]. The Colorado dataset was obtained from the USGS national website[2] and pre-processed into a desirable format.

[1] http://www.cc.gatech.edu/projects/large_models
[2] https://viewer.nationalmap.gov/basic/

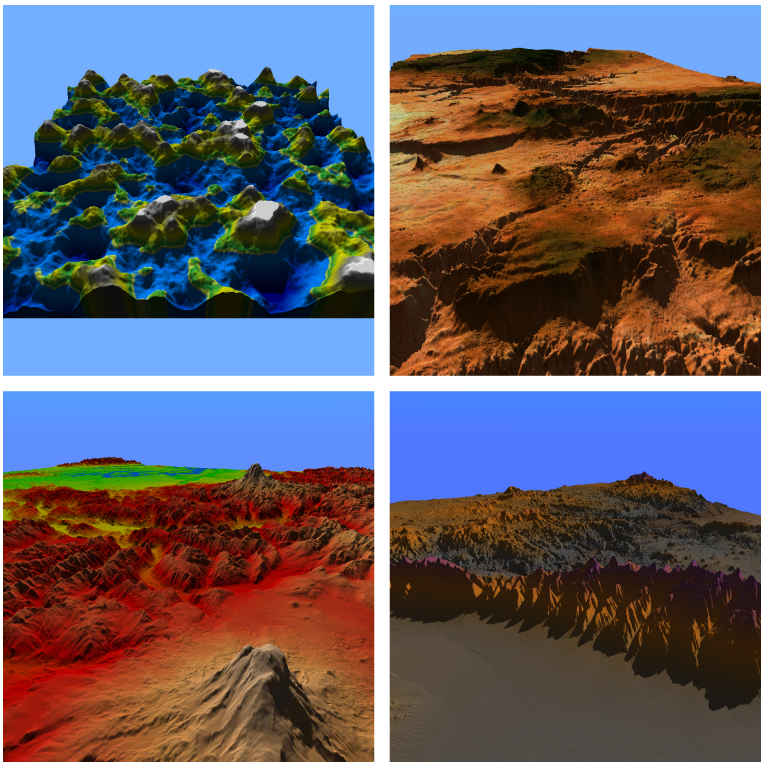


Figure 4.1: Four terrain datasets tested with the small-scale ray tracer: Synthetic, Grand Canyon, Puget Sound and Colorado.

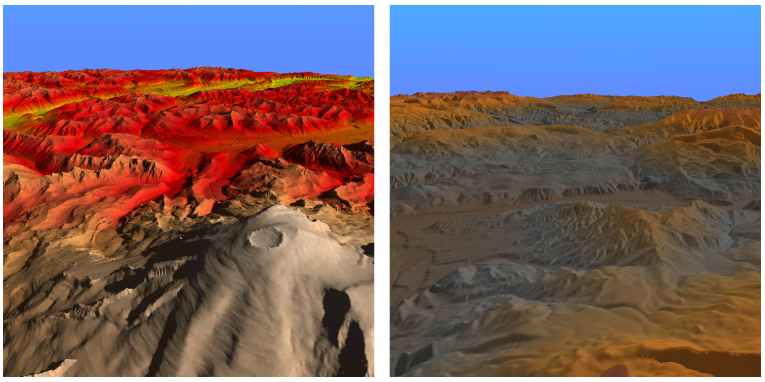


Figure 4.2: Two terrain datasets tested with the small-scale ray tracer: Puget Sound and Colorado.

Testing environment

The small-scale engine was tested on four data sets: a synthesized terrain with 1024x1024 resolution, a sub-region of the Puget Sound data set with 4096x4096 resolution, Grand Canyon with a 2048x4096 resolution and a data set of Colorado with 4096x4096. All data sets have 8-bit heightfields except the Colorado data set which has a 16-bit heightfield.

The scalable engine was tested with two data sets: an 8-bit Puget Sound terrain with a total resolution of 16384x16384 samples divided in 4x4 tiles of 4096x4096 resolution making up an active region of 12288x12288 samples, and a large version of the 16-bit Colorado dataset with a total resolution of 16384x16384 samples, divided into 8x8 tiles of 2048x2048 resolution making up an active region of 6144x6144 samples. These tile dimensions are *not recommended* but were selected to test the limits of the engine and also because the photo textures have the same resolution as the heighfield.

Tests were performed in both engines following the same set of rendering parameters across different screen resolutions and using primary ray tracing and secondary ray tracing. Patch size was set to 1, the LOD distance parameter was set to 256, FOV was set to 60 degrees and the far-z value was set to a very large value so that the entire terrain is rendered at all times. The range of altitudes for each terrain was also the same. For each dataset a camera path was recorded so as to allow subsequent executions to perform the same camera course. In short, results at different resolutions were recorded using the same camera course and rendering parameters to keep the results consistent.

As mentioned in the list of terrains used, each terrain has multiple datasets in varying resolutions. Do note that the different resolutions do not necessarily represent the same data. For instance, the small-scale Puget Sound dataset (4k) is a subset of the large dataset, but the small Colorado dataset (4k) is not a subset of its large counterpart. Thus, the camera paths tested in both engines are different and their results, while obtained under the same conditions, have no direct connection either.

The large far value, unitary patch size and large tile resolution values are deliberately selected to test extreme conditions. Using these values the entire terrain and active region can be rendered at any point. Thus, the results presented in the following sections happen in a ”worst case” scenario where rays can traverse through the entire extent of the terrain or active region. In normal conditions, terrains should be rendered with an appropriate patch size (eg. each patch covers a distance of 10 meters in the real world), a smaller far-z and a more manageable active region, resulting in higher performance statistics than shown in our results.

As a last note, to keep the results coherent with the other terrains, the Colorado dataset was tested both with the original 16-bit heightfield format and with a downsampled 8-bit format. The test results in both precision formats were very similar with insignificant differences of about 1-3 frames across all resolutions and both rendering modes. Therefore, we can conclude that between 8-bit and 16-bit the most impactful difference lies not in runtime performance but in the better rendering quality, greater amount of memory consumed and higher image loading times. Having said that, the performance results presented for Colorado are the ones obtained from the 16-bit versions.

4.1 Frame Rate

Frame rate, also referred to as Frames per Second (FPS), is the most common performance test for rendering engines and it measures the number of images the system can generate per second.

The frame rate alone tells us how interactive an application is. A high frame rate means that a response is returned to user input more quickly than a low frame rate. But that in itself is not a sufficient measure of the renderer's performance. Frame rate varies with each scene, viewing direction and number of objects in sight.

A more accurate measure in terms of frame rate would provide insight on the worst case scenario, best case scenario and the number of frames we can expect the engine to achieve overall. Since the implementation already manages frame rate in the main loop, a test was created that records the number of FPS rendered over a period of time. Once recording completes the minimum, maximum and average frame rates are automatically determined.

These tests were performed several times on 3 terrain datasets: Puget Sound (8-bit), Grand Canyon (8-bit) and Colorado (8 and 16-bit). Note that the Grand Canyon dataset is padded into a 4096x4096 resolution.

4.1.1 Small-scale ray tracer

		Photo Shading			Lighting Shading		
Dataset	Screen Res.	Min FPS	Avg FPS	Max FPS	Min FPS	Avg FPS	Max FPS
Synthetic	1024x768	173	218	327	70	101	171
	1280x1024	119	150	223	47	68	118
	1920x1080	79	109	213	33	48	101
Grand Canyon	1024x768	183	244	329	82	116	181
	1280x1024	123	167	220	54	77	121
	1920x1080	89	119	162	37	53	88
Puget Sound	1024x768	167	255	392	67	121	249
	1280x1024	113	179	267	44	82	180
	1920x1080	76	125	193	31	56	126
Colorado	1024x768	161	243	384	74	112	246
	1280x1024	109	171	313	49	75	153
	1920x1080	81	121	201	33	51	100

Table 4.1: FPS results from both primary and secondary ray tracing for the small-scale engine.

Despite the lower resolution the synthetic terrain demonstrates similar frame rates to the other, higher resolution terrains. In fact, it presents mostly lower values. The synthetic terrain contains only high frequency structures with fast variations from the minimum to the maximum altitude values while real terrains tend to have more constant altitude values rather than such fast variations. This difference appears to makes traversal far more expensive.

On average, taking a look at Table 4.1, under the default photo shading mode, the engine is capable of rendering all terrains well above 60 FPS. Considering only averages values, then frame rates are above 100 FPS. Frame rates for Puget Sound and Colorado are quite similar, while the Grand Canyon presents higher minimums. The Grand Canyon contains high frequency structures and most altitude

values are already very high, meaning that these high frequency structures tend to obstruct each other, causing rays to terminate intersection earlier than otherwise. The Puget Sound and Colorado, on the other hand, have mostly low altitude values. One half of the Puget Sound dataset is very flat and the other half contains a mountainous region. Capturing a large portion of the terrain near the high frequency structures will cause traversal to descend quickly into the finer levels and then traverse a long distance over the flat terrain. The Colorado dataset is quite similar to the Puget Sound in regards to the terrain layout, except that it contains more variation overall, thus presenting the lowest results out of all the real datasets.

The maximum frame rates, of 300 FPS and above, are obtained when rendering a flat part of the terrain or when parts of the sky are rendered from outside the terrain's bounding box. Rays cast from outside the box are not subjected to the ray traversal algorithm and are exited early, causing unusually high FPS if most rays behave this way in a frame. Facing only the terrain does not necessarily minimize the FPS as vertical downwards rays are quite fast to compute.

When secondary rays are activated in lighting mode, the overall frame rate drops to slightly less than 50% of the default rendering mode - hinting that shadow rays may be slightly more costly to trace than primary rays.

The terrain resolution affects frame rate logarithmically. Each consecutive power of two size, for the terrain resolution, adds one new layer to the maximum mipmap pyramid and causes all rays, on average, to perform one additional descent/ascent operation. As a result, the difference in frame rate between two consecutive terrain resolutions appears to be related to the ratio between the number of LODs. The fact that the synthetic terrain presents similar results to the higher resolution terrains, indicates that the large variation in altitude values may influence frame rate more heavily than 2 additional mipmap levels.

4.1.2 Scalable ray tracer

Frame rate results from the scalable ray tracer are presented in Table 4.2. Note that the resolution of the active region is 3 times the tile resolution, 12288x12288 samples for Puget Sound and 6144x6144 samples for Colorado.

			Photo Shading			Lighting Shading		
Dataset	Tile Res	Screen Res.	Min FPS	Avg FPS	Max FPS	Min FPS	Avg FPS	Max FPS
Puget Sound	4096	1024x768	92	212	285	48	95	141
		1280x1024	78	147	197	40	63	91
		1920x1080	49	101	134	26	42	60
Colorado	2048	1024x768	131	192	288	53	81	126
		1280x1024	91	134	216	38	54	85
		1920x1080	66	92	138	25	36	56

Table 4.2: FPS results from both primary and secondary ray tracing for the scalable engine.

The results are quite similar to the small-scale ray tracer despite the resolution difference. Overall the average FPS are only slightly lower and the scalable engine is also capable of reaching over 60 FPS for primary tracing and over 30 FPS for secondary tracing on the highest resolution.

For the Puget Sound dataset, the minimum values are especially low due to the large tile resolution.

Despite taking relatively long load each tile, around 0.4s, the real cause of this performance drop is when the main thread is signaled to upload the new tiles into the GPU, stalling the main application. There is no way to accelerate this data transfer, but limiting the amount of data sent to the GPU per frame, starting with coarser LOD levels, would be a plausible solution. Nevertheless, tile resolutions are usually smaller precisely to avoid the problem of GPU data transfer times, which is slightly less noticeable in the Colorado dataset.

On the other hand, these tests have active region resolutions of 12288x12288 and 6144x6144 respectively. It is not surprising that there is a decrease in frame rate compared to the small-scale tests. However, it is more surprising that the difference is not too accentuated as well.

The decrease in FPS also owes to an extra conditional branch compared to the small-scale ray tracer, as the scalable ray tracer must check when a new active tile is traversed and perform index conversions.

In the scalable engine the difference between 8-bit and 16-bit terrains is more noticeable than in the small-scale ray tracing. Although not shown in the tables, the quantity of data to be loaded per tile is smaller for 8-bit precision and, therefore, the time it takes to upload one tile to the GPU is also shorter. Though the average frame rate remains the same, the minimum frame rate tends to be higher when handling an 8-bit terrain.

4.2 Rays per second

In ray tracing, in addition to the number of frames per second, the number of rays traced per second is also used to measure performance. While FPS are a very common way to test rendering performance, the number does not tell us how powerful the rendering system actually is. The number of rays per second serves as a more accurate estimate of the engine's brute force performance. More rays per second always means that the system is capable of rendering more FPS, but more FPS does not necessarily mean more rays per second - other factors weigh in like screen resolution, early traversal exits, secondary rays and hardware.

The OpenGL Shader Pipeline is quite difficult to debug. Simple debug options typically include writing if branches to visualize values as colors. Fortunately a new shader feature was introduced in OpenGL 4.2, the Atomic Counter[3], which allows the fragment shader to execute operations on a "global" variable whose data persists across multiple shader executions. Three operations can be applied to an atomic counter variable, read, increment and decrement. Since multiple instances of the Fragment Shader run in parallel, access to this variable is automatically synchronized by OpenGL, hence the name atomic.

In terms of our implementation, such a variable would allows us to calculate the number of rays traced per frame more efficiently than any other method involving reading back data from the Fragment Shader into a buffer. An atomic counter was integrated into our shader implementation as in the following code sample:

Based on the code sample in Algorithm 7, we increment the number of rays traced under the condition that the current ray entered the traversal loop. A ray does not enter the traversal loop when it fails to

[3]More information available at khronos.org/opengl/wiki/Atomic_Counter

```
layout(binding=0, offset=0) uniform atomic_uint totalRaysTraced;
...
void main()
begin
    ...
    intersection = rayTraceTerrain( rayOrigin, rayDirection );
    if intersection.traversed then
        atomicCounterIncrement( totalRaysTraced );
    end
    ...
end
```

Algorithm 7: Fragment Shader using an atomic counter.

intersect the terrain's bounding box. All other rays are taken into account no matter how costly they were to compute, including rays that were traced through the terrain for which no intersection was found.

The application side of OpenGL can later access this value through the appropriate functions. Besides the frame rate, the performance test we mentioned in the previous section on frame rate also records the number of rays traced per second.

4.2.1 Small-scale ray tracer

Figure 4.3 summarizes the results obtained from running performance tests on each terrain dataset. We can see that the number of rays traced without secondary ray tracing is about 30% higher for all terrains.

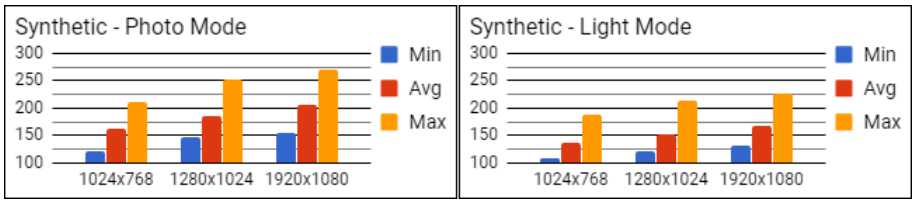
Once again, the Synthetic terrain, with its very high frequency structures, presents the lowest average values out of all terrains.

Despite containing multiple valleys and other high frequency structures, the Grand Canyon dataset demonstrates a higher overall number of rays traced per second. This is likely due to the fact that most altitude values are close to the maximum elevation value and that the mountains obstruct large portions of the terrain. For this reason most rays need fewer traversal steps because the viewer is both closer to the ground and the upper limit of the terrain's bounding box, resulting in more early traversal exits compared to the Puget Sound or Colorado datasets. In fact the Puget Sound dataset, despite showing higher average values, contains very low altitude values and so a greater portion of the sky is captured, which is cheaper to ray trace.

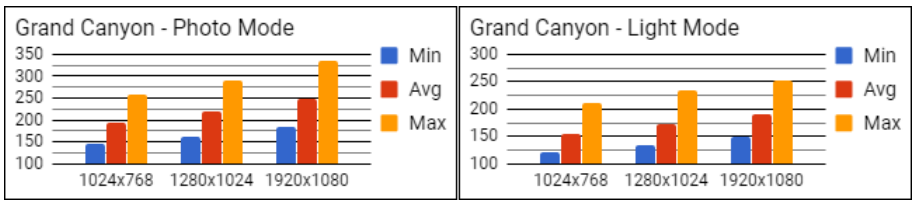
The fact that the Colorado dataset has a 16-bit heightmap, as mentioned before, does not seem to influence both FPS nor rays per second, having very comparable statistics to the other terrains. Results for this dataset still demonstrate the lowest values for the same reasons mentioned in the frame rate section.

The very high maximum RPS values are outliers that tend to occur when a large portion of the sky is captured in the frame. This causes primary rays to exit traversal quite which were much cheaper to compute than a ray that does intersect the terrainearly, resulting in an unusually high number of FPS and rays per second. Looking downwards while close to the terrain causes a similar effect but not as extreme. This is the reason why maximum values are not a reliable estimation of rendering performance.

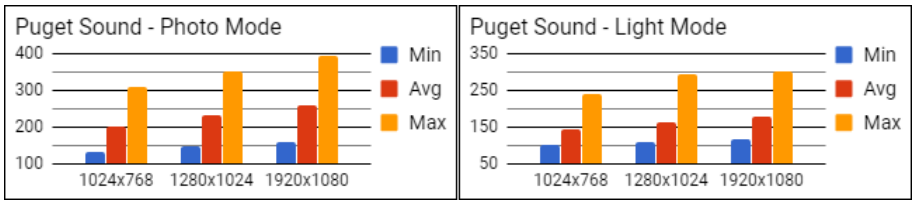
Contrary to the expectation of a constant number of rays traced per second, overall, this number varies slightly with resolution. Since more frames are generated at lower resolutions, the application



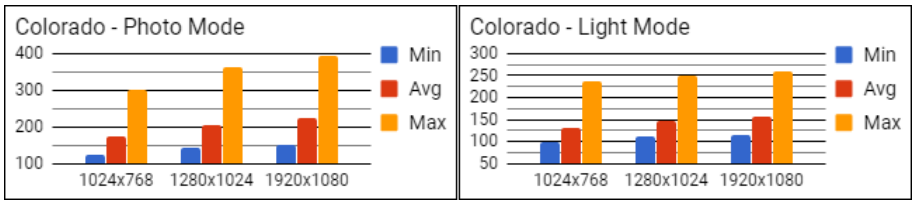
(a) RPS results for the Synthetic terrain.



(b) RPS results for the Grand Canyon terrain.



(c) RPS results for the Puget Sound terrain.



(d) RPS results for Colorado terrain.

Figure 4.3: Rays Per Second analysis for the small-scale engine datasets. The results are presented in mega rays per second (millions of rays per second).

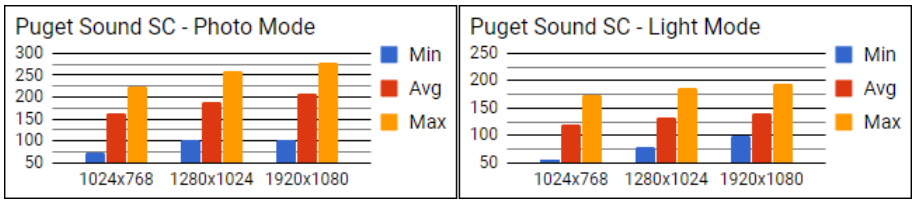
main loop also executes more often and spends extra time handling events and updates. This results in a slightly lower number of rays traced at lower resolutions than in higher resolutions. A different main loop, or a more efficient one, would likely see smaller differences across resolutions.

4.2.2 Scalable ray tracer

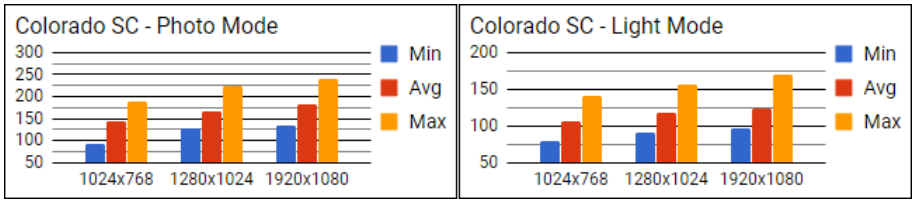
Figure 4.4 summarizes the results obtained for the scalable engine. In terms of rays per second, the scalable engine appears to trace less rays on average than the small-scale engine. At the highest resolution tested the difference is smaller. Very low minimum values occur due to GPU data transfer times, as in the Frame Rate results, and are especially noticeable for the Puget Sound terrain.

While the number of rays in primary tracing would easily reach average values around 200 million rays and 150 million in secondary tracing, for the small-scale engine, in the scalable engine these values are now slightly lower. Secondary rays are definitely more costly to compute across both engines and resolutions.

The same phenomenon as in the small-scalable engine is seen as the number of rays traced slightly



(a) RPS results for the scalable Puget Sound dataset.



(b) RPS results for the scalable Colorado dataset.

Figure 4.4: Rays Per Second analysis for the scalable engine datasets. The results are presented in mega rays per second (millions of rays per second).

increases with resolution. While the number of rays should be independent of resolution, the fact that more frames are rendered also implies that the main loop in the CPU is executed more times. At higher resolutions (lower frame rates), the engine should spend more time performing ray tracing than re-iterating the main loop.

4.3 Traversal steps per ray

A very particular method for measuring performance in ray traversal, which has been used, for example, in [9], is the number of traversal steps per ray. This value represents the number of times the ray traversal algorithm attempts to verify if a ray intersects with the terrain. The meaning behind this number is more easily explained with ray marching - in linear ray marching the ray is traversed in constant length steps, in binary ray marching the ray is traversed in a divide and conquer fashion, etc. Maximum Mipmaps work similarly to linear ray marching, in that at the coarsest mipmap levels ray are stepped for large distances while at finer levels rays are stepped for smaller distances. This implies that the number of ray steps in Maximum Mipmaps is the sum of all iterations in the traversal loop at any given LOD, including ascending and descending mipmap levels. A good traversal algorithm should achieve the lowest number of steps per ray as possible.

In the shading section we referred two rendering modes that were added to visualize the number of steps, and they were especially implemented to perform this test.

Unlike the two previous performance tests, the number of ray steps is not an absolute but a relative measurement of the engine's overall performance. The number of ray steps largely depend on the specific path each ray takes. By definition, ray traversal in maximum mipmaps is slowed down if a ray grazes the terrain. Such rays may require nearly twice as many steps as an average ray would. The purpose of this test is to compare the overall performance of the ray traversal algorithm against previous works.

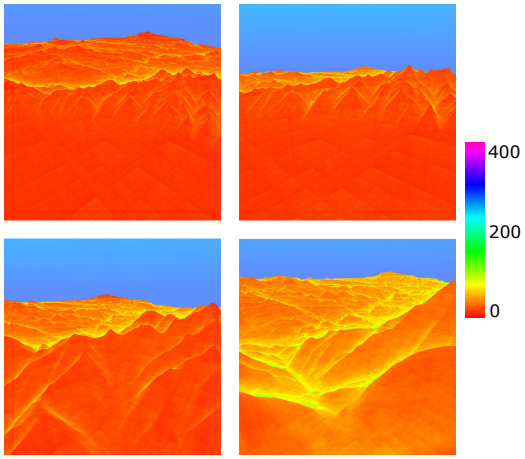


Figure 4.5: Ray step heatmaps in the Colorado dataset at different viewer positions.

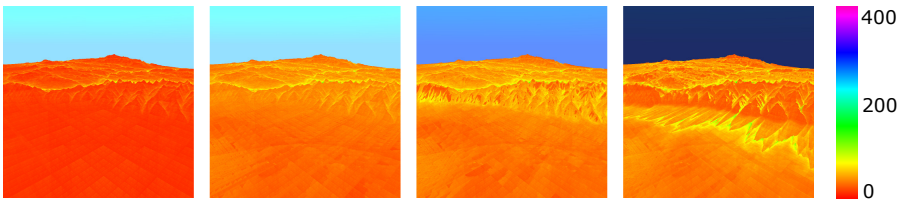


Figure 4.6: Comparison between ray step shading modes and varying light angles. From left to right: primary steps, secondary steps vertical light, secondary steps 45 degrees and secondary steps 75 degrees. The shadow boundaries are notably visible. Note that steps from rays that exited the terrain are not represented and instead the skybox is visible.

Calculating the average number of rays is not very practical and instead heatmaps are used to visualize and compare traversal steps.

We utilized the same heatmap color coding as in [9]. Figure 4.5 shows several step heatmaps in the same terrain but at different locations. From left to right and top to bottom, the viewer is moving forward. We can see that, as the viewer approaches the mountain, frames tend to become more expensive. The Maximum Mipmap structure is partially visible showing that the number of steps may vary ever so slightly based on the specific path of each ray takes through the mipmap pyramid.

Flying far above the terrain results in most rays being traced downwards at a steeper angle. Rays become less likely to graze parts of the terrain and, instead, will directly intersect the terrain with less ascending operations, which is show in the top left image.

When the viewer is standing near a mountain, the ray origins' will stay below multiple Maximum Mipmap altitude values, forcing the traversal algorithm to descend rapidly into finer levels. Descending into finer levels causes ray steps to be shorter, so more steps will be performed overall.

In addition to the number of ray steps we also tested the total number of ray steps including secondary rays traced for shadows as shown in Figure 4.6. For each ray that intersected with the terrain a second ray is traced. From simply activating secondary rays, we can tell by looking at the second heightmap that the entire terrain becomes more expensive to render. Shadow rays are less expensive when the light direction is vertically oriented, more expensive otherwise.

The last image from Figure 4.6 clearly shows distinct boundaries from the projection of the moun-

Terrain resolution	8-bit heightmap	16-bit heightmap
256 x 256	652 KB	1.05 MB
512 x 512	2.54 MB	4.21 MB
1024 x 1024	10.17 MB	16.84 MB
2048 x 2048	40.78 MB	67.35 MB
4096 x 4096	162.7 MB	269.4 MB

Table 4.3: Comparison of memory consumed with varying terrain resolution and precision.

tainous region over the flat part the terrain. It is very interesting to note, albeit not surprising, that it is quite easy to visualize shadow boundaries based on the step count - rays that graze the terrain require more traversal steps from testing false positive intersections.

4.4 Memory Consumption

Previous works typically perform an analysis of the memory consumed by their renderers during the execution. It is especially important for implementations that utilize an adaptive geometry structure that is modified based on LOD control and the viewer's position. Determining the highest and average memory consumption values aid in determining the minimum memory requirements so that the same program may be safely executed in different machines.

Our implementation, however, does not require this kind of analysis because its memory consumption is constant. Once memory has been allocated, on initialization, for the elevation, maximum mipmaps, photo and normal textures, this memory remains unmodified for the entire execution.

Even in the scalable engine these textures are only modified to accomodate for active region shifts. These changes refer to overwrites in memory but never to memory allocations or deallocations.

Despite that fact, our implementation still consumes a certain amount of memory. We deduced two formulas, that apply to either 8-bit and 16-bit heightmaps respectively, based on the textures' specifications: resolution, bytes per pixel and whether it has mipmaps[4] or not. These formulas assume that the photo texture has the same resolution as the heightmap, in per patch format, which is the case for our testing datasets.

$$Mem8 = w \times h \times (4 \times \frac{4}{3} + 1 \times \frac{4}{3} + \frac{3}{6}) + (w+1) \times (h+1) \times 3 \tag{4.1}$$

$$Mem16 = w \times h \times (8 \times \frac{4}{3} + 2 \times \frac{4}{3} + \frac{3}{6}) + (w+1) \times (h+1) \times 3 \tag{4.2}$$

Table. 4.3 compares the amount of memory consumed by the application based on the previous formulas with terrains ranging from 256 x 256 to 4096 x 4096 resolutions. Memory used increases about 4 times when terrain resolution doubles, which is an expectable increase.

While the original elevation maps are stored as PNG images, no kind of compression is applied to the elevation maps during execution. The engine utilizes the raw elevation image, in per patch format,

[4]Generating mipmaps for a texture incurs an additional 33% of memory consumed.

Tile resolution	Active Region	8-bit heightmap	16-bit heightmap
512 x 512	1536 x 1536	22.9 MB	37.9 MB
1024 x 1024	3072 x 3072	91.5 MB	151.5 MB
2048 x 2048	6144 x 6144	366 MB	606 MB

Table 4.4: Comparison of memory consumed with varying terrain tile and active region resolutions for the scalable engine.

which results in the highest possible amount of memory consumed. The maximum mipmaps require a single channel to store a single elevation value, thus consuming 4 times less memory than the original elevation map. On the other hand the photo textures are compressed into DXT1 format and require about 6 times less memory than the raw 24-bit format. The normal maps, which are represented in per point format, are stored as an uncompressed 24-bit image.

While our implementation focused on real time rendering more than memory efficiency, the end result is that the amount of memory consumed is quite high compared to previous implementations, where memory consumption was more heavily constrained. High memory consumption is especially evident in the scalable engine that must store multiple terrain tiles simultaneously.

Reducing the amount of memory consumed can be still be achieved with:

- Compressing the normal map to use less than 1 channel per component. This option may also reduce the quality of normals.
- Lower resolution photo texture is an obvious choice. The DXT1 format already reduces image quality significantly and resolution will further reduce memory consumed and quality.
- Compressing the heightmap may be possible, though using this option was not hinted in previous works. Since the elevation map is stored as an image, common image compression techniques may be usable at the cost of performing decompression on demand. On the other hand it must be compatible with the maximum mipmap generation, which are built from the original heightmap.

One last implementation note is that for 8-bit terrains we encountered one possible driver bug. When storing the maximum mipmaps texture as a single-channel image the last 2 mipmap levels were found to contain corrupted data. Changing the storage type to more than 1 channel, with no further changes to the code, was enough to avoid this problem. For this reason our implementation consumes an additional percentage of memory than the formulas indicate. This issue did not affect 16-bit maximum mipmaps which are stored as a single channel image with two bytes per pixel.

4.5 Result analysis and comparison

The results obtained in the previous sections indicate that primary ray tracing of terrains is a feasible option for real time rendering, with the engine achieving average frame rates above 60 FPS at a screen resolution of 1920x1080 pixels. With secondary ray tracing the average frame rate is above 30FPS with higher frame loads reaching no less than 20 FPS, even in the scalable engine unless tiles are uploaded while viewing an expensive set of frames.

Scalability, as noted in [9, 3, 15] is not an issue for Maximum Mipmaps. Our test results also demonstrate that terrains of almost arbitrary size can be rendered at interactive frame rates as long as the active region dimensions are manageable by the GPU, with the only issue being the amount of memory consumed and texture sizes supported.

Some optimizations were applied over time to the ray tracing algorithm code. However, no major optimizations were implemented that would greatly impact the rendering pipeline. Therefore, how well our implementation performs, compared to previous works, is also largely related to advances in hardware.

Frame rate

Tests from [9] indicate that at a resolution of 1024 x 768 pixels, the ray tracer would produce frame rates between 22-43 using Maximum Mipmaps with Bilinear Interpolation, with varying terrain resolutions. Our performance tests show that with current hardware it is possible to obtain a similar number of frame rates at nearly twice the resolution, 1920x1080, with secondary ray tracing for shadows. Without secondary ray tracing the engine can achieve frame rates well above 60 FPS.

Comparing with [7], they also performed tests on a 1920x1080 resolution. They developed a similar solution to our implementation, in that it also employs Maximum Mipmaps and Clipmaps, although they implemented Hermite Bicubic Patches to further improve the reconstruction of the heightmap. Despite that difference they still managed to achieve interactive average frame rates, about half of our performance tests, but very low minimum frame rates of less than 5 FPS. My engine achieves more stable frame rates while rendering the entire heightmap or active region.

MRays per second

In terms of rays per second there is not a reliable base of comparison for our ray tracer since previous similar works did not measure performance using this metric. Our performance tests could, however, serve as a reference for future work, showing that it is possible to achieve around 100-200 MRays per second with this approach. Highly optimized ray tracers and, perhaps, implementations in more specialized software may obtain even higher results.

Because most rays are taken into account, regardless of traversal cost, the number of rays does not necessarily reflect the real performance of the engine. But there is a correlation between the number of rays and the number of frames. Analyzing the number of rays per second alongside the frame rate can help understand the results more accurately. A high frame rate means that either fewer rays are traced or most had early exits from the traversal loop, whereas in the lower spectrum of frame rates the number of rays traced is also lower but they were more expensive to compute overall.

Traversal steps per ray

Ray step heatmaps demonstrate similar values to previous works' heatmaps, in [9] and [3], with less than 50 steps for most rays and up to 100 steps for more expensive rays. Compared to linear marching, traversal in Maximum Mipmaps is many times more efficient. Linear marching would, in the worst case,

require as many steps as terrain patches in original terrain resolution. Maximum Mipmap complexity scales nearly logarithmically with terrain size. The heatmaps in [14] are not directly comparable because traversal happens only at the intersected tile, and thus the number of steps is based on traversal over a single tile.

Adding second rays to the equation shows that the sum of primary and secondary rays almost double the number of steps compared to primary steps alone. Primary rays that had a high traversal cost do not necessarily have high cost secondary ray counterparts. Nearly horizontal shadow rays are the most costly to compute as they graze more parts of the terrain. The worst case scenario happens when both primary ray and shadow ray graze the terrain, which can be observed when looking away from the light source towards the shadow boundaries.

Memory Consumption

Memory consumption was not a major concern for our implementation. Even previous works that utilized Maximum Mipmaps did not refer the use of any heightmap compression methods, specialized for Maximum Mipmaps or otherwise. In fact[9] hints at heightmap compression methods compatible with Maximum mipmaps for future work. Older works like [5, 7] achieve much more efficient memory usage than our engine through the use of compression methods and adaptive geometric meshes.

Image compression could be applied to the heightmap but it is not necessarily a good approach since the Maximum Mipmap structure must be generated from the heightmap - lossy compression may not be ideal. Lossless compression may require an additional decompression step in the shaders and, therefore, a slight loss in rendering performance. Further memory savings may be achieved with normal map compression or downscaling the photo texture resolution.

As hinted in [9] storing the Maximum Mipmap structure as a 3D texture may even improve cache coherence in texture accesses at an additional cost of memory. Different texture storage methods may provide different results.

Overall

There is a notorious difficulty in comparing this work with previous works due to hardware differences. Despite that, we show that is possible to ray trace terrains in the GPU with current hardware well over 60 FPS, or even above 30 FPS using secondary ray tracing for shadows at a screen resolution of 1920x1080. Lower resolutions achieve even higher values.

Although the tests were performed in current hardware, the implementation was done in OpenGL to support multiple platforms and also older hardware. The final implementation can be adapted to run in at least OpenGL 3.1, or later, due to using very few exclusive features from recent versions. Lower versions may not support certain profiling features.

Both engines demonstrate that real time frame rates are possible with ray tracing. Scalability is easily overcome with the use of Maximum Mipmaps and affects rendering performance very slightly compared to a small-scale renderer.

Memory consumption is manageable for terrain resolutions lower than 2048x2048, but very high otherwise. Lower resolutions of 512x512 and 1024x1024 are recommended for tile sizes in scalable ray tracing. Even then the active region must contain the entire 3x3 active tiles, or 5x5 depending on implementation choice, so reducing memory requirements for the active region would be an important step.

Chapter 5

Conclusions and Future Work

In this work an analysis was done concerning previous ray tracing algorithms, how relevant they are in terrain rendering and how their qualities can be leveraged into a ray tracer implementation.

Based on that analysis two ray tracing engines were implemented in C++ with OpenGL: a small-scale engine that renders single heightmap terrains, and a scalable engine capable of rendering large terrain datasets divided into tiles. Both engines employ the Maximum Mipmaps algorithm for ray traversal, Geometry Clipmaps to simplify the terrain rendered and 3D distance to determine level-of-detail regions, and Bilinear Patch interpolation to improve rendering quality with smooth terrains. The ray tracing algorithm is fully implemented in the GPU to maximize graphical performance and is executed in a single OpenGL Shader pass.

5.1 Achievements

Results show that scalability is easily overcome with the use of Maximum Mipmaps. Although the scalable ray tracer has to execute one additional conditional branch compared to the small-scale ray tracer and, consequently, its frame rates are slightly lower, the difference is not too significant. The traversal loop is executed millions of times per second, therefore even small additions or modifcations have an immediate impact in performance.

Scalability is a typical concern of terrain rendering and, as a result, two ray tracing engines were developed. A small-scale renderer for single heightmaps and a scalable renderer for large terrains split into tiles. Performance is very similar on both engines, however the scalable engine requires an extra conditional branch required to perform traversal between active tiles which slightly slows down rendering. Because the traversal loop is executed millions of times per second, every single optimization to ray traversal visibly impacts performance.

Despite that difference, both engines can achieve frame rates of over 60FPS for primary ray tracing and over 30FPS for secondary ray tracing at a resolution of 1920x1080, which demonstrates that current hardware is very capable of real time ray tracing in commonly used resolutions.

5.2 Future Work

While implementing our solution we have faced multiple issues that could be further explored in future work.

The system's memory footprint is acceptable for terrains measuring up to around 4096x4096 resolution. Larger terrains may not only be unsupported by graphics library or GPU but, according to our memory usage formulas, may even exceed 1GB of VRAM. This fact is most problematic for scalable rendering, especially when photo textures have a much higher resolution than the heightmaps. For these reasons, active regions should be smaller than 4096x4096 resolution which can be achieved using terrain tiles with manageable resolutions of 512x512 or 1024x1024 samples. Despite that, in the future, compression schemes for heightmaps compatible to use with Maximum Mipmaps would further reduce memory requirements.

The largest gap in this work's implementation is related to the LOD control methods available for ray tracing. Previous works on terrain ray tracing seem to avoid details on their LOD control methods. Results in this work shows that LOD control does not need to be continuous for the rendering of primary rays and gaps can be solved by adjusting intersection parameters so that LOD transitions are closed. For secondary ray tracing, however, shadows require special attention in terms of LOD control. Because intersection points can be found at varying LODs, shadow rays should utilize the same LOD control parameters as the primary rays to maintain consistency. Even then, shadow popping is much more noticeable than geometry popping. In the future, a new LOD control method specialized for ray tracing would be desireable to improve lighting and shadow quality, which could be achieved perhaps utilizing an adaptation of the dissolve filter for images or a morphing algorithm for an engine that may pre-compute ray-traceable meshes.

It is very common to render buildings, forests, people, animals and all sorts of objects over the terrain. However, most works that focus on terrain ray tracing, including this work, typically focus terrain ray tracing too much. The resulting algorithms optimize ray tracing over terrains but are not necessarily compatible with other kinds of geometry. Therefore, it would be interesting to find a middle-ground approach that can both efficiently ray trace terrains as well as other kinds of objects.

Lastly, for the scalable engine, a smarter tile management system could be implemented. The active tiles do not necessarily require all LOD levels to be present in memory. At the very least, all levels up to the finest LOD rendered for each tile must be in memory. This fact would allow us to render an adaptive active region able to pre-load tiles beyond the current capabilities with only coarser LODs, allowing rendering to cover greater distance without storing the highest resolution levels for tiles that do not require them. On the other hand, each LOD is computed based on the closest finer level all the way down to the original heightmap. A simple approach might require storing all mipmap levels in disk. Heightmap upsampling from lower resolution levels has been utilized in other works [5, 7], and could be worth exploring for this purpose.

Bibliography

[1] T. Whitted. An improved illumination model for shaded display. *Proceedings of the 6th annual conference on Computer graphics and interactive techniques*, 1979.

[2] A. Appel. Some techniques for shading machine renderings of solids. *SJCC*, pages 27–45, 1968.

[3] C. Dick, J. Schneider, and R. Westermann. Gpu ray-casting for scalable terrain rendering. *Eurogrpahics*, 2009.

[4] D. Luebke, M. Reddy, J. Cohen, A. Varshney, B. Watson, and R. Huebner. Level of detail for 3d graphics. *San Fransisco, Morgan Kaufmann Publishers*, ISBN 1-55860-838-9, 2003.

[5] M. Duchaineau, M. Wolinsky, D. Sigeti, M. Miller, C. Aldrich, and M. Mineev-Weinstein. Roaming terrain: real-time optimally adapting meshes. *In Visualization '97 Proceedings, IEEE*, page 81–88, 1997.

[6] F. Strugar. Continuous distance-dependent level of detail for rendering heightmaps (cdlod). *Journal of Graphics, GPU, and Game Tools*, 14(4):57–74, 2009.

[7] F. Losasso and H. Hoppe. Geometry clipmaps: Terrain rendering using nested regular grids. *In Proceedings of ACM SIGGRAPH*, pages 769–776, 2004.

[8] A. Asirvatham and H. Hoppe. Terrain rendering using gpu-based geometry clipmaps. *GPU Gems 2, chap. Addison-Wesley Longman*, 2005.

[9] A. Tevs, I. Ihrke, and H.-P. Seidel. Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering. *In Proceedings of ACM Symposium on Interactive 3D Graphics and Games*, pages 183–190, 2008.

[10] S. Porumbescu, B. Budge, L. Feng, and K. Joy. Shell maps. *ACM SIGGRAPH 2005 Papers*, July 31 - August 04 2005. Los Angeles, California.

[11] J. Dummer. Cone step mapping: An iterative ray-heightfield intersection algorithm, 2006. `http://www.lonesock.net/files/ConeStepMapping.pdf`.

[12] F. Policarpo and M. Oliveira. Relaxed cone stepping for relief mapping. *GPU Gems 3*, Ch. 18, 2007. Addison-Wesley Professional.

[13] O. Jamriska. Interactive ray tracing of distance fields. *The 14th Central European Seminar on Computer Graphics*, 2010.

[14] C. Dick, J. Schneider, and R. Westermann. Gpu-aware hybrid terrain rendering. *In Proceedings of IADIS Computer Graphics, Visualization, Computer Vision and Image Processing*, pages 3–10, 2010.

[15] D. Feldmann and K. Hinrichs. Gpu based single-pass ray casting of large heightfields using clipmaps. *In Proceedings of Computer Graphics International*, 2012.

[16] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.

[17] J. Amantides and A. Woo. A fast voxel traversal algorithm for ray tracing. *Eurographics*, 87:3–10, 1987.

[18] S. Ramsey, K. Potter, and C. Hansen. Bilinear patch intersections. *Journal of Graphics Tools*, 9(3): 41–47, 2004.