TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

Masterarbeit

GPU based Clipmaps

Implementation of Geometry Clipmaps for terrain with non-planar basis

Ausgeführt am Institut für
Computergrafik und Algorithmen
der Technischen Universität Wien

unter Anleitung von
Univ.Prof. Dipl.-Ing. Dr.techn. Werner Purgathofer
und Mitwirkung von
Dipl.-Ing. Dr.techn. Robert F. Tobler M.S.

durch

Anton Frühstück
Hegergasse 18/3
A-1030 Wien

Wien, am 23. April 2008

Abstract

Terrain rendering has a wide range of applications. It is used in cartography and landscape planning as well as in the entertainment sector. Applications that have to render large terrain are facing the challenge of handling a vast amount of source data. The size of such terrain data exceeds the capabilities of current PCs by far.

In this work an improved terrain rendering technique is introduced. It allows the rendering of surfaces with an arbitrary basis like the spherical-shaped earth. Our algorithm extends the *Geometry Clipmaps* algorithm, a technique that allows to render very large terrain data without losing performance. This algorithm was developed by Losasso and Hoppe in 2004. Asirvatham and Hoppe improved this algorithm in 2005 by increasing the utilization of modern graphics hardware. Nevertheless both algorithms lack of the ability to render large curved surfaces.

Our application overcomes this restriction by using a texture holding 3D points instead of a heightmap. This enables our implementation to render terrain that resides on an arbitrary basis. The created mesh is not bound by a regular grid mesh that can only be altered in z-direction. The drawback of this change of the original geometry clipmap algorithm is the introduction of a precision problem that restricts the algorithm to render only a limited area. This problem is handled by patching the whole surface with individual acting, geometry clipmap quads.

Kurzfassung

Das Rendering von Terrain-Daten findet in vielen Anwendungsgebieten Verwendung. Neben kartographischen Anwendungen wird es auch im Unterhaltungssektor eingesetzt. Anwendungen, die ein sehr ausgedehntes Gelände anzeigen sollen, stehen vor dem Verarbeitungsproblem von großen Datenmengen. Es ist in der Regel nicht möglich, diese Datensätze mit herkömmlichen PCs direkt anzuzeigen.

In dieser Arbeit wird eine verbesserte Terrain Rendering Technik vorgestellt. Diese *Level-of-Detail* Technik erlaubt das Anzeigen von auf beliebiger Basis aufgesetztem Terrain. Ein Beispiel dafür ist die Erde, deren Oberfläche auf einer Kugel basiert. Der vorgestellte Algorithmus baut auf dem *Geometry Clipmaps* Algorithmus auf, der das Anzeigen von unbeschränkt großen Terrain-Daten ohne Leistungsabfall erlaubt. Dieser Algorithmus wurde 2004 von Losasso und Hoppe entwickelt und 2005 von Asirvatham und Hoppe verbessert, um herkömmliche Graphik-Hardware besser ausnützen zu können und dadurch die Rendering-Leistung zu steigern. Dennoch können beide Algorithmen nur Gelände auf ebener Basis darstellen.

Unsere Anwendung überwindet diese Einschränkung, indem anstelle einer Höhentextur eine *Floating-Point* Textur mit den 3D-Koordinaten der einzelnen Oberflächenpunkte verwendet wird. Durch diese Änderung ist es unserer Anwendung möglich, Gelände mit beliebiger Form anzuzeigen. Das erzeugte Drahtgitter ist nicht an ein reguläres, nur in z-Richtung veränderbares *Mesh* gebunden. Der Nachteil dieser Änderung des ursprünglichen Geometry Clipmap Algorithmus ist das Auftreten eines Genauigkeitsproblems, das dem Algorithmus das Rendern von nur beschränkt großen Terrains erlaubt. In dieser Arbeit wird dieses Präzisionsproblem durch Teilen der gesamten Oberfläche in kleinere, ohne Probleme verarbeitbaren Teile gelöst.

Credits

I would like to thank my supervisor for the great support during the development of this work, my parents for making my study at the university possible and last but not least my girlfriend who encouraged me and showed understanding for the little spare time I had.

Contents

List of Figures

List of Tables

Chapter 1.

Introduction

Terrain rendering has a wide range of applications. It is used in cartography and landscape planning as well as in the entertainment center (e.g. in games). Applications like Google Earth [1] or NASA World Wind [2] fit in between those areas and enjoy great popularity.

High resolution terrain data is usually captured by satellites. TerraSAR-X, a satellite of the German Aerospace Center and European Aeronautic Defence and Space Company, was launched on the 15. June 2007. The satellite uses a X-band Radar to gain high resolution images [WBB^{+}]. The resolution of the images ranges from 1-2 meters in *spotlight mode* (10 x 10 km) to a resolution of 16 meters for 100 km wide strips. Therefore a full image set of the earth produced by the TerraSAR-X satellite, results in roughly 30.000 million data points.

In the game sector artistic tricks allow the rendering of larger terrain without getting problems with the amount of data. Applications that have to render real terrain, however, face the challenge of handling large amount of source data. The size of the data exceeds the capabilities of current PCs by far.

There are a number of applications that allow the rendering of very large datasets in different ways. Some of those render terrain on a flat basis (compare Figure 1.1 left) whereas others superimpose the data on spherical objects (compare Figure 1.1 right).

During our literature research we were not able to find implementations that render terrain based on differently shaped objects. In some cases it might be desired to not only render

[1]http://earth.google.com
[2]http://worldwind.arc.nasa.gov

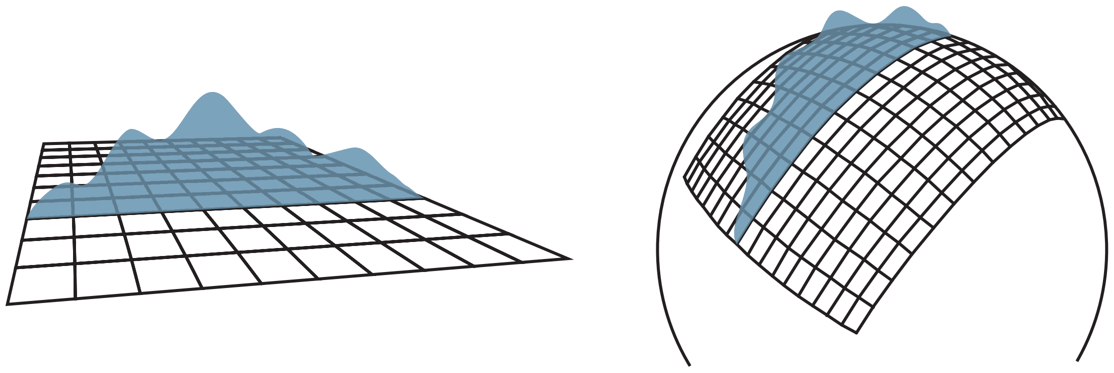


Figure 1.1.: Rendering a height field on a flat basis (left) versus a spherical basis (right).

terrain on a sphere but for example on a cylinder. The user shall not be restricted to a specific basis from which the terrain is elevated.

The first problem can be solved by a variation of *Geometry Clipmaps* which will be discussed later but this introduces another difficulty. By way of example during the rendering of spherical objects with terrain on the GPU numerical problems can arise due to the limited 32 bit floating point textures on current graphics cards. This limitation allows the application to render only a limited area without getting precision problems.

1.1. Thesis Objectives

As the problems of terrain rendering were discussed, the main objectives of this work can be outlined:

1. Create a working geometry clipmap implementation.
2. Extend the implementation to use 3D elevation data instead of a single channel heightmap to be able to render spherical based objects.
3. Overcome the 32 bit floating point restriction to allow the rendering of arbitrary large terrains with arbitrary resolution.

1.2. Structure of this Thesis

We will now provide an overview of this thesis:

- Chapter 2 outlines the state of the art in graphics hardware accelerated landscape engines and the according algorithms.
- Chapter 3 discusses the theory behind our approach as well as the performance enhancing techniques used by our application.
- Chapter 4 describes the implementation details of our application.
- Chapter 5 evaluates the performance of the algorithm, points out its weak points and our solution approaches.
- Chapter 6 briefly recaps the introduced algorithm and outlines the future work.

Chapter 2.

State of the Art

In the following three sections the state of the art in computer hardware accelerated landscape engines and the according algorithms is discussed. Similarly to Brettell [Bre05] we classify the existing techniques into three categories: *hierarchical algorithms*, *triangular irregular meshes* and *GPU-based approaches*.

The algorithms in the first category recursively subdivide the height field using common data structures. Irregular mesh techniques reduce a triangulated terrain into triangles of any shape and size to give the most faithful representation of the terrain. According to Clasen and Hege [CH06] GPU-based terrain rendering algorithms take the advantage of the huge geometry bandwidth of current GPUs. This advantage overcompensates their deficiencies in accuracy.

2.1. Hierarchical Algorithms

Hierarchical rendering algorithms use bisection of the terrain's triangles to gain a better level of detail (LOD). Recursive splitting is done until a desired LOD is reached. According to Brettell [Bre05] the only requirement is that one instance of a shape can be partitioned into *n* smaller copies of the same shape. The result of this process is a hierarchy. The possibility of cracks in the subdivided terrain is a drawback of the splitting process and one challenge for the algorithms.

A good overview of hierarchical algorithms is contained in the following two papers: *Visualization of large terrains made easy* by Lindstrom *et al.* [LP01] and *ROAMing Terrain:*

Real-time Optimally Adapting Meshes by Duchaineau *et al.* [DWS$^+$97]. In the following two subsections those algorithms are shortly reviewed.

2.1.1. Real-Time, Continuous LOD

The paper of Lindstrom *et al.* [LKR$^+$96] about real-time terrain rendering was among the first papers published in the terrain rendering sector. The main goal of the authors is to build a mesh with a smaller number of triangles out of a much larger mesh. This enables them to render an approximated version of the orginial mesh which is too large to be rendered directly. During the rendering task the authors want to remain highly memory and compute efficient. The coarser mesh should be a good approximation of the original mesh for a given view.

The authors [LKR$^+$96] describe a method to simplify and render a fine-grained mesh in two steps. First the mesh is reduced with little computational cost by using a conservative estimate of whether certain groups of vertices can be eliminated block wise. The organization of the blocks is done by a quadtree. This first step saves computational effort that would otherwise arise if the whole mesh was processed only by the following second step.

In the second step the resulting mesh is processed with a computational more expensive method for further, finer-grained simplification and rendered afterwards. The simplification process of the triangles is carried out if the accuracy loss introduced by the reduction of triangles is smaller than a certain constant values.

Lindstrom and Pascucci [LP01, LP02] improved the upper algorithm by changing the bottom-up (fine-to-coarse) simplification technique to a more efficient top-down, out-of-core algorithm. Out-of-core algorithms are such that process data which is too large to fit into the PC's or the graphics card's main memory and therefore has to be streamed into the memory on demand.

In the original version of the algorithm Lindstrom *et al.* [LKR$^+$96] process the data into a quadtree and simply traverse the tree during the rendering process. In the improved version of the algorithm [LP01] an interleaved quadtree is not directly used for rendering but outputs a single long triangle strip in an efficient way. The main advantage of this triangle strip is that current graphics cards are optimized for the input of such triangle strips. As a result the render process is optimized and simplified.

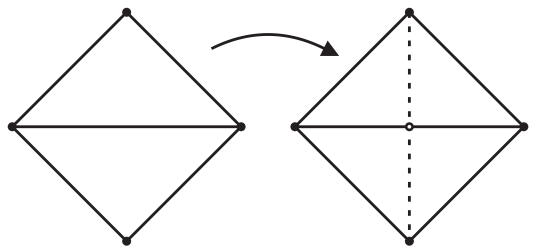


Figure 2.1.: Longest edge bisection of a right-angled triangle-pair.

During a preprocessing step an interleaved quadtree is generated by gradually subdividing the original mesh into different levels. Each vertex of this tree is either labeled white or black, depending if it is created at an odd or an even level of refinement. In the implementation of Lindstrom and Pascucci [LP01] they use a subdivision based on *longest edge bisection* where an isosceles right-angled triangle is refined by dividing its hypotenuse (compare Figure 2.1). The results of this preprocessing step are two quadtrees – a white and a black one. The top row of Figure 2.2 shows the first three levels of the white quadtree whereas the second row views the according black quadtree. It is important to note that during the creation of the black quadtree, vertices outside of the terrain boundaries have to be created. The resulting unused space is filled with the the first two levels of the white quadtree (indicated by the arrows in Figure 2.2).

The quadtrees are implemented as direct acyclic graphs (DAGs) in which an edge is formed by connecting the vertex at the center of the baseline and the opposite vertex. The data is stored within the DAG from coarse to fine levels. Within one level, vertices that are geometrically close are stored close together in the memory to preserve neighborhood properties. This can be achieved by computing the index of the k^{th} child of the parent by using following formula:

$$c(p,k) = 4p + k + m \qquad \text{with } k = 0,1,2,3$$

m is a constant and depending on the index of the root and the index distance between consecutive levels of resolution.

The data is eventually stored as a single linear array. Unwanted holes arise within this array but can be avoided by duplicating vertices. According to Bradley [Bra03] the drawback of this is that it can result into sending twice as many triangles to the graphics card than actually seen on the screen.

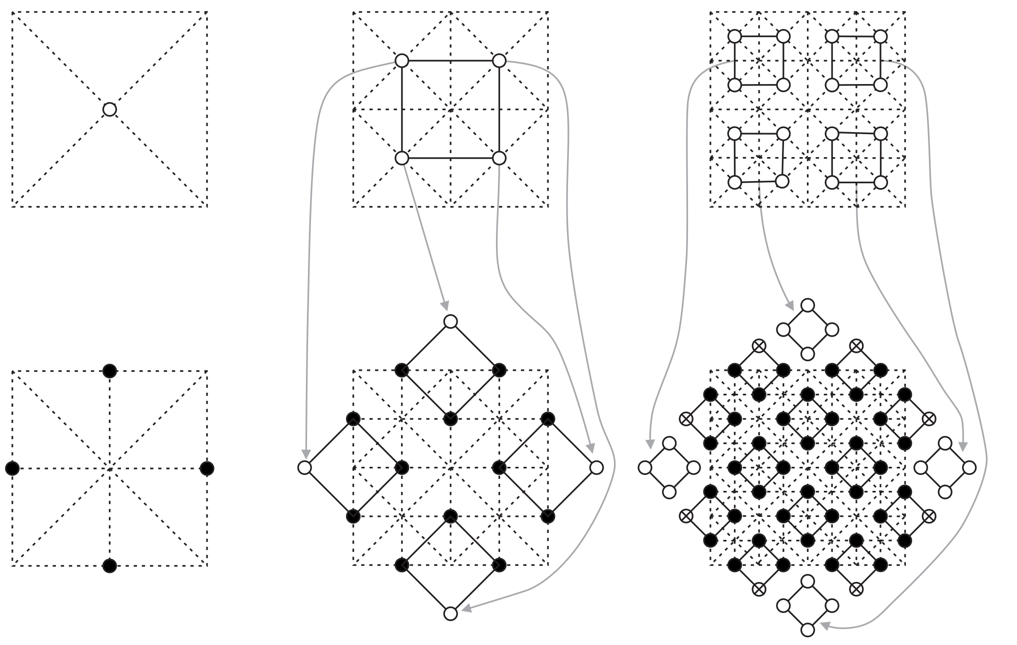


Figure 2.2.: Embedding of the first two levels of the white quadtree structure (first row) into the black quadtree structure (second row). The arrows indicate the new position of the different parts of the white quadtree (following Lindstrom and Pascucci [LP01]).

The triangle strip has to be rebuilt every frame in order to achieve continuous LOD. The algorithm starts from four separate large triangles (the southern, eastern, northern and western triangular regions) and determines if the current node is *active*. A node being active means that its screen space error is greater than a user-defined error tolerance. The screen space error is the projection of the difference between the original terrain and the actually rendered terrain on to the screen. If the node is active, the algorithm recursively refines the left child of the node, adds the apex of the node to the triangle strip and refines the right child of the node.

According to the Lindstrom and Pascucci [LP01, LP02] the improved version of the original algorithm [LKR$^+$96] renders the same scene significantly faster and more efficient but the negative effect mentioned by Bradley [Bra03] and the small exploitation of the graphics hardware makes this algorithm not suitable for our needs.

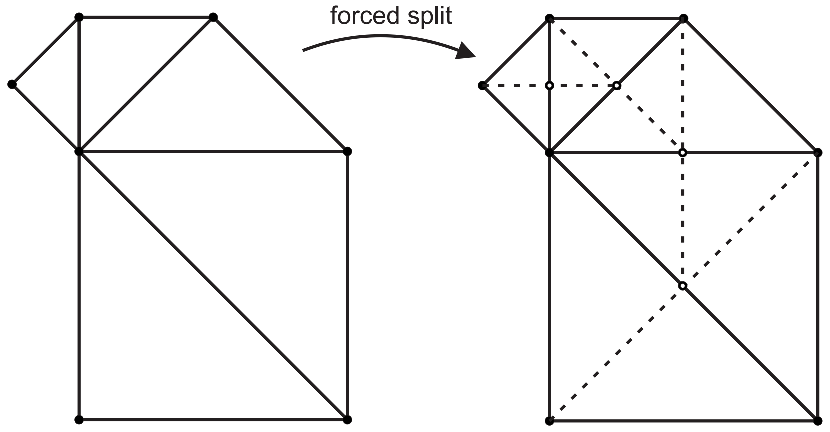


Figure 2.3.: Recursive forced splitting of a triangle with a base neighbor from a coarser level (following Duchaineau *et al.* [DWS+97]).

2.1.2. ROAM

The *Real-time Optimally Adapting Meshes* (ROAM) algorithm was introduced by Duchaineau *et al.* [DWS+97] in 1997. ROAM is similar to the algorithm described in the previous subsection.

As pointed out by the authors [DWS+97] ROAM consists of a preprocessing component and several runtime components. During the preprocessing step a dynamic binary triangle tree is created as a mesh representation including nested, view-independent error bounds for the triangle tree. The space of continuous binary triangle tree meshes is the same as the one used by Lindstrom and Pascucci [LP01] in their algorithm. The root triangle is a right-isosceles triangle at the coarsest level of subdivision. The children of this root triangle are created by splitting the parent triangle along an edge from its apex vertex to the midpoint of its base edge. Continuing this process recursively creates the whole binary triangle tree.

The main difference of the ROAM algorithm to the algorithm by Lindstrom and Pascucci [LP01] is the splitting an merging technique. A triangle can not only be split into two children, but two neighboring triangles can also be merged into a larger one. Two mergeable triangles are referred to as a *mergeable diamond*. If a triangle has a base neighbor from a coarser level and has to be split, the base neighbor has to be split in advance. This can result into a recursive forced split-process as visualized in Figure 2.3.

The authors [DWS$^+$97] annotate that any triangulation can be obtained from any other triangulation by a sequence of splits and merges. These operations can be animated during the update process by vertex morphing. Instead of immediately moving a vertex to its new position, the vertex's position is linearly interpolated over time from the old to the new position. Using the split and merge operations, fine-grained updates can be made to an existing triangulation.

The authors [DWS$^+$97] present a greedy algorithm that drives the split and merge process. This algorithm consists of two priority queues. The first queue starts with the base triangulation and recursively splits the triangles with the highest priority. The priority of an triangle is typically an error bound and is discussed later. The only requirement for the split queue is that a child's priority is not larger than its parent's. During the merge queue, mergeable diamonds are handled, starting from a previous optimal triangulation when the priorities have changed. Therefore advantage of frame-to-frame coherence is taken.

Duchaineau *et al.* [DWS$^+$97] describe several error metrics and bounds which are used for computing the queue priorities. For heightmap triangulations they introduce *wedgies* that can be generated in a preprocessing stage. Wedgies are pie-shaped volumes that determine the accuracy of each triangular section of the landscape. A wedgie is produced by parallel shifting of the three triangle points along the y-axis. The length of the shift is defined by the local error value. Nested wedgie bounds are built bottom-up (compare Figure 2.4). This ensures that the children of a high-level node describe the same volume more accurate [Hou04].

The actual tasks that are carried out per frame are following. After view-frustum culling (that is described in next paragraph), the priorities of the nodes that are potentially split or merged is updated. With the use of these priorities the greedy algorithm updates the triangulation. The new triangulation is then ready to be built into a triangle strip. In the last step the terrain is rendered using this triangle strip.

In the paper about the ROAM algorithm four different performance enhancement techniques are described. View frustum culling tests every triangulation whether it is within the current view frustum. Each triangle in the binary tree is given an IN, OUT, ALL-IN and DONT-KNOW flag depending on the triangles position and the position of the according wedgie relative to the six half spaces. Those flags are updated every frame by efficient recursive

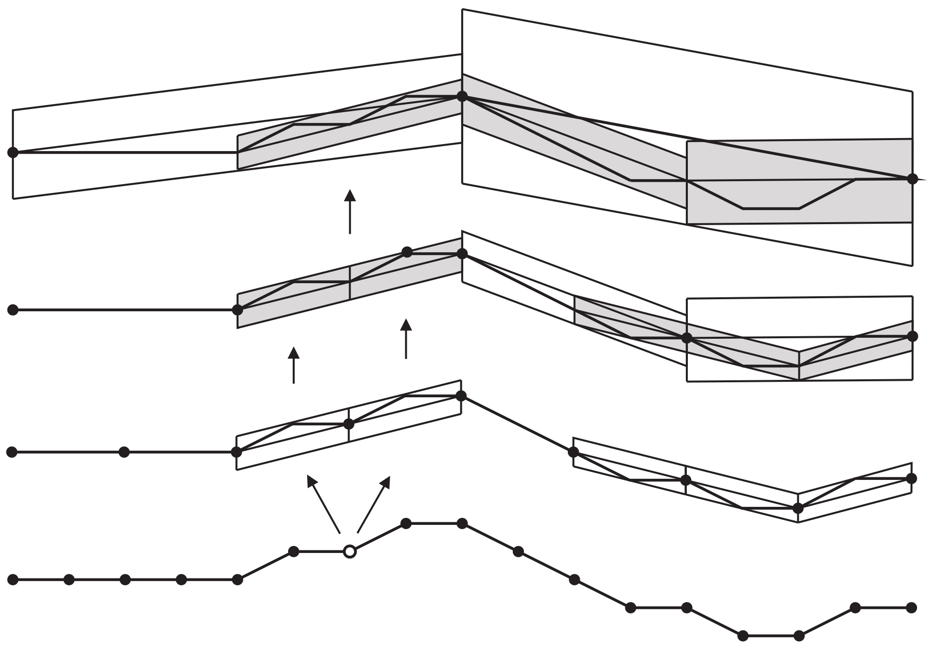


Figure 2.4.: Nested wedgies for the 1D domain (following Duchaineau *et al.* [DWS+97]).

binary tree traversal. Depending on the flags the rendering engine knows whether a triangle has to be rendered or not.

The second performance optimization is incremental T-stripping. Duchaineau *et al.* [DWS+97] use a simple, sub-optimal, incremental approach to organize triangles into strips. This yields into strip lengths of about four to five triangles and allows to gain advantage of the graphics cards optimized ability to handle such strips. During frame update only minimal reorganization of the strips has to be performed.

Because recalculating error bounds of all triangles for every frame is too costly, the authors suggest renewing the priorities only when they potentially affect a split or merge decision. Recomputation of a triangle can safely be delayed until its priority bound exceeds the crossover priority (defined as the maximum split-queue priority).

The fourth optimization described in the paper [DWS+97] is titled as *Progressive Optimization*. Triangulation optimization should stop when the time runs out for the current frame update. This results into non-optimal triangulations. However the split and merge steps are performed in decreasing order of importance so that the partial work is as good as possible as time permits.

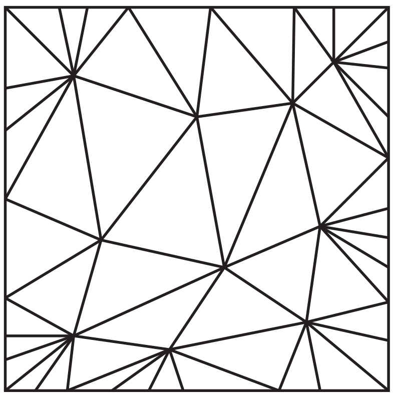


Figure 2.5.: Schematic representation of a triangular irregular mesh (following Hoppe [Hop98]).

The ROAM algorithm is similar to the Real-Time, Continuous LOD algorithm by Lindstrom and Pascucci [LP01, LP02]. It meets our requirements neither due to the same problems described in the previous subsection.

2.2. Triangular Irregular Meshes

Triangular Irregular Meshes (in the following abbreviated by TIN) represent a terrain as a number of triangles of different shape and size [Bre05]. Figure 2.5 shows a schematic representation of a TIN. Such meshes allow a optimal approximation for a terrain for a given number of triangles. An example of a TIN algorithm that refines the triangulation as the viewport changes is the *view-dependent progressive mesh* (VDPM) framework by Hoppe [Hop98]. In the following subsection this algorithm will be reviewed.

2.2.1. Smooth View-Dependent Level-of-Detail Control

A technique called *progressive meshes* was developed by Hoppe [Hop96]. Progressive meshes allow an efficient, lossless, continuous-resolution representation of highly detailed geometric models. Smooth geomorphing, level-of-detail approximations, progressive transmission, mesh compression and selective refinement is possible.

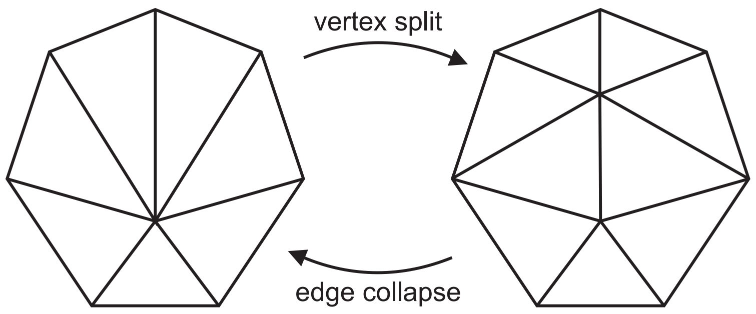


Figure 2.6.: Vertex split refinement and the inverse operation, edge collapse.

This technique was later extended into the VDPM (*view-dependent progressive mesh*) framework. According to the author [Hop97] the framework allows to retrieve accurate approximating meshes from a hierarchy of geometrically optimized refined meshes. Hoppe again improved the framework to provide temporal coherence through the runtime creation of *geomorphs*, which smoothly transition surface geometry over several frames. This is necessary for eliminating the popping effect (a visible sudden change of the mesh from one frame to the next). In his work the author also describes how to specialize the VDPM framework for terrain rendering because Hoppe did not mainly develop progressive meshes for the rendering of height fields.

According to Hoppe a progressive mesh representation is an arbitrary triangle mesh with a sequence of refinement transformations to progressively recover detail. As refinement transformations Hoppe uses *vertex splits* (compare Figure 2.6). During a vertex split process, a vertex is replaced by two new vertices with all the edges that are needed for a correct triangulation. The inverse vertex split transformation is called *edge collapse*. Two vertices as well as the effected edges are merged.

To achieve view-dependent rendering of progressive meshes certain properties of every vertex are checked per frame as the viewing parameters change in the VDPM framework. A vertex must intersect the view frustum, its screen projected deviation from the base mesh must exceed a user-defined tolerance and the surface surrounding the vertex must not be strictly oriented away. If those requirements are fulfilled, a split refinement or an edge collapse transformation is opposed on each vertex of the active front before rendering. As a result the mesh is adapted to the new viewing parameters and an optimal image of the scene is created.

For being able to apply the idea of view-dependent progressive meshes upon terrain, Hoppe decided to create a hierarchical progressive mesh construction [Hop98]. In a preprocessing step the surface is reduced into a number of approximations with different coarseness. For larger models those pre-simplified meshes may still be too large to fit into the memory. Therefore the author suggests to partition the refinement database into blocks.

During the render process, the blocks that are going to be needed, are pre-fetched into the memory. When the refinement criteria indicate the need for a vertex split or an edge collapse, the transformation is not performed immediately. Instead it is carried out over a series of frames as a geomorph if and only if the region of the affected surface is visible. Regions outside the view frustum are refined instantaneously.

Next to the position other attributes have to be interpolated as well, like the normal, color and texture coordinates. Interpolation is mostly done linearly, only the normals are interpolated over the unit sphere.

For our approach view-dependent progressive meshes are not sufficient. The CPU-load is too great to efficiently render very large terrains.

2.3. Geometry Clipmaps

The idea of clipmaps is based on mipmaps defined by Williams [Wil83]. Mipmapping is a technique for avoiding aliasing during texture rendering. A mipmap consists of multiple correlated images with reduced resolutions. The *level 0* of this mipmap pyramid holds the finest and therefore largest image. Lower levels are created by skipping every second texel in both image directions of the previous image. This results in an image with 1/4 of the size of the finer image.

Tanner *et al.* [TMJ98] state that during the rendering with a mipmap pyramid the engine chooses a texel from a mipmap level where the display's pixel-to-texel mapping is closest to a 1:1 mapping. For very large mipmaps the majority of the mipmap pyramid is not used during the render process of a single image – no matter what geometry is rendered. Therefore they define a clipmap as an updateable, minimal subset of a mipmap (compare Figure 2.7).

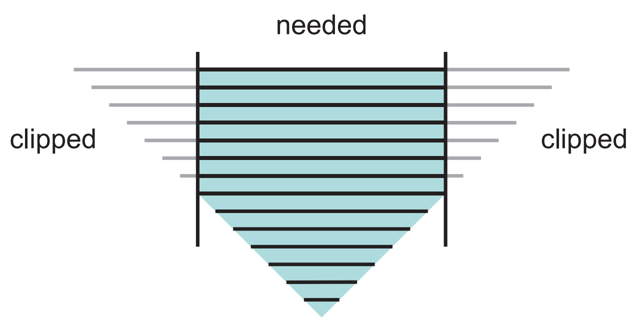


Figure 2.7.: Clipmap region within a mipmap (from Tanner *et al.* [TMJ98]).

Each level is clipped to a specific maximum size. As a result a clipmap is a dynamic texture representation that enables real-time rendering of arbitrary large textures using a finite amount of physical memory.

The idea of clipmaps can be used for rendering arbitrary large terrains by elevating a planar grid. This technique is called geometry clipmapping and is described in the first subsection. Using the original version of geometry clipmapping it is not possible to render spherical shaped terrain like the earth. Terrain rendering using spherical clipmapping makes this possible and is discussed in the second subsection.

2.3.1. Planar Clipmaps

Rendering a large terrain without any level-of-detail (LOD) control can lead to aliasing artifacts, just like texturing without mipmaps. To overcome such problems, Losasso and Hoppe introduced *Geometry Clipmaps* [LH04]. In their implementation terrain is created by elevating a basic flat mesh. This mesh is stored as a traditional vertex buffer. As a result the CPU is heavily used in updating and rendering the terrain. Asirvatham and Hoppe improved the original geometry clipmap algorithm by using vertex textures [AH05] and therefore took advantage of the great processing capabilities of current graphics cards to decrease the CPU-load.

In the improved version of the geometry clipmap algorithm the geometry of the terrain is split into two parts. The x and y coordinates are stored as constant vertex data while the accordingly z coordinate is stored in a single channel 2D texture. This elevation map is used in the vertex shader to elevate the flat vertex data.

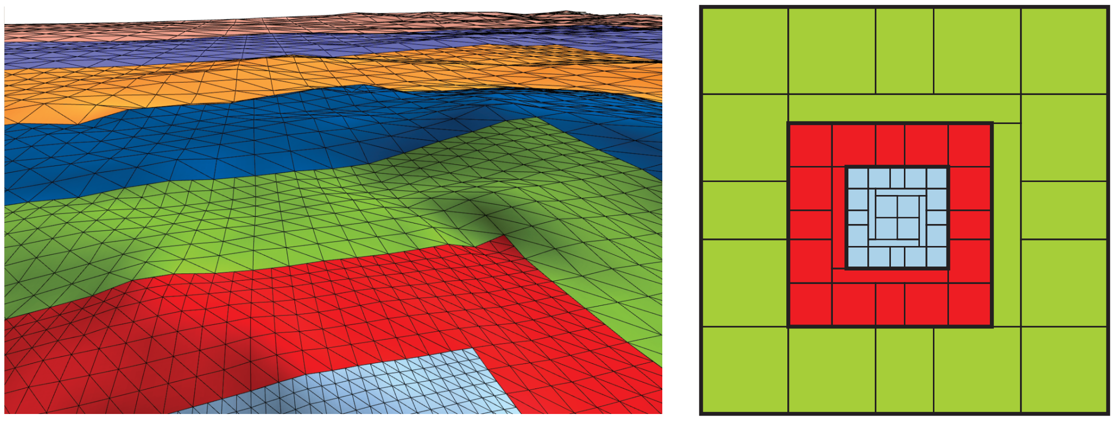


Figure 2.8.: Left: Rendering of clipmap rings (from Asirvatham and Hoppe [AH05]). Right: the structure of the three finest clipmap rings in top-down view.

The vertex data is not stored as one big mesh but is instead composed by different smaller constant parts (2D footprints). During the rendering process those parts are scaled and translated to form regular grid rings which are centered about the viewer (compare Figure 2.8). Figure 2.9 shows the different footprints and how the clipmap rings are composed.

The grid size of a whole clipmap ring has to be odd to ensure that each level lies on the grid of the next coarser level. The size of the elevation texture is chosen to be a power of two because the hardware is often optimized for this size. As a result the clipmap ring size is $n = 2^k - 1$ (with any desired k).

The 2D elevation map that is used for the terrain creation is prefiltered into a mipmap pyramid with L level. Figure 2.10 shows a schematic clipmap pyramid. In this figure the uppermost (= coarsest) level of the clipmap pyramid on the left corresponds to the outermost region of the top view on the right. Each clipmap level is used to elevate a different regular grid ring. Coarser levels elevate the rings further away. The finest clipmap level *L-1* elevates not only the closest ring but also a grid square that fills the hole of the nearest ring. The elevation is done by sampling the elevation map as a vertex texture. This is possible since DirectX 9 Shader Model 3.0 on NVIDIA Geforce 6 GPUs and better [GFG04].

For large terrains the whole clipmap pyramid does not fit into the memory. To overcome this restriction only squared parts of the clipmap levels are stored in the memory. The size of the stored clipmap parts corresponds to the number of vertices on one side of the rendered clipmap rings.

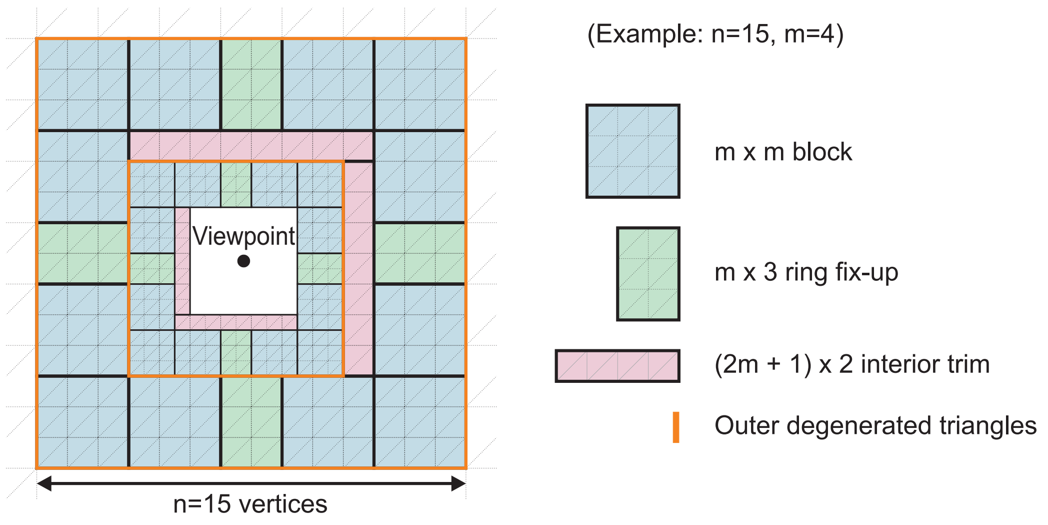


Figure 2.9.: Partitioning of a clipmap ring into different 2D footprints (from Asirvatham and Hoppe [AH05]).

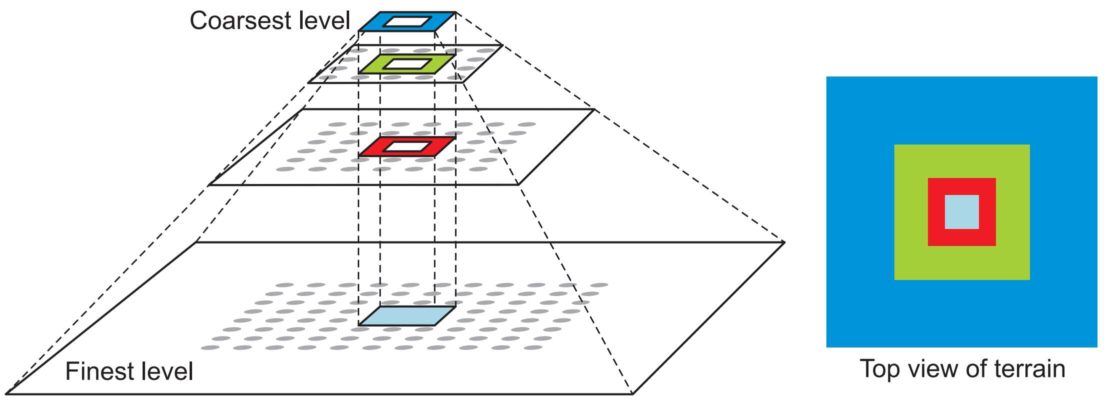


Figure 2.10.: The clipmap pyramid (from Asirvatham and Hoppe [AH05]).

If the position of the viewer moves, the missing parts of the clipmap levels in the memory are updated. Those parts are generally L-shaped. 2D wraparound addressing of the clipmap level textures makes efficient incremental update possible. Only the required parts are updated and therefore translating existing data in the memory is avoided. Using this technique the upper left corner of the actual elevation image does not correspond to the upper left corner of the texture but moves within the texture. During the rendering process the graphics card automatically wraps the texture.

Finer levels are never exactly centered within the next coarser levels. According to Asirvatham and Hoppe [AH05] the advantage of this is that a requirement of the algorithm is fulfilled – finer levels are allowed to shift within the next coarser levels while those stay fixed.

During the rendering process not all the levels are necessarily rendered. If the grid extent of a level is greater than 2.5 times the height of the viewer, this level is skipped and the arisen hole is filled by the next coarser level.

View frustum culling is easily possible. The x and z coordinates remain always the same during the rendering process. Only the z value is affected by the vertex elevation. During view frustum culling the footprints of the levels are elevated and intersected with the view frustum in 3D. According to Asirvatham and Hoppe [AH05] the rendering load is reduced with this view frustum culling technique by a factor of 2 to 3 for a 90-degree field-of-view.

The authors state that the main bottlenecks in their implementation are the vertex texture lookups used for the vertex elevation. Removing the lookups increases the rendering rate by a factor of about 40% in their rendering scenario [AH05].

Losasso and Hoppe use a GPU based technique to overcompensate the fact that the rendered mesh is not optimal in terms of terrain approximation accuracy. The biggest issue for us is the fact that only planar terrain can be rendered, although this terrain is allowed to be infinite large due to the out-of-core technique of the algorithm.

2.3.2. Spherical Clipmaps

The geometry clipmap implementation by Losasso and Hoppe [LH04] does not allow to render spherical objects. Rendering such objects using rectangular support geometry results in

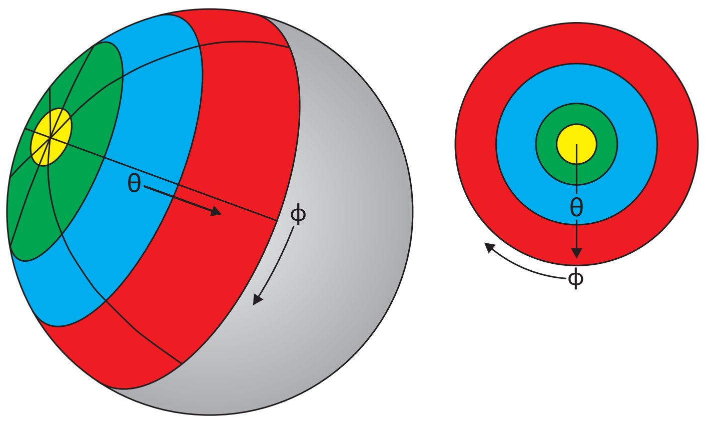


Figure 2.11.: Concentric ring support geometry for spherical clipmapping (from Clasen and Hege [CH06]).

the grid becoming infinitely thin towards the poles and stopping there. Spherical coordinates do not wrap around in θ direction. For being able to visualize spherical terrains (e.g. planets) on many scales, *Spherical Clipmaps* were introduced by Clasen and Hege [CH06].

In the implementation by Clasen and Hege [CH06] the underlying geometry is changed from a rectangular grid to concentric rings. The shape of the support geometry of spherical clipmaps is shown in Figure 2.11. Changing the support geometry involves a change of the parameterization. An equally simple parameterization of the plane (x,y) are spherical coordinates. Any point of a sphere can be addressed using the angle θ to the z-axis and the angle ϕ of the projected point on the x,y-plane to the x-axis (compare Figure 2.11). The value of θ ranges from 0 to π and the value ϕ from 0 to 2π.

Clasen and Hege implemented spherical clipmaps using two different spaces [CH06]. In the view space the viewer is located at the north pole of the spherical object which allows using static geometry that can be precomputed and transferred to the GPU only once. The hemisphere around the viewer is parameterized by $(\hat{\phi},\hat{\theta})$. The view space is rotated to absolutely orient the spherical terrain. The resulting space is called world space and parameterized by (ϕ,θ).

The mapping between those two spaces is defined as

$$f(\theta_v,\hat{\phi},\hat{\theta}) \rightarrow (\phi,\theta)$$

The viewer is assumed to be located exactly above the 0-meridian at $(0,\theta_v)$. Any deviation of the viewer in ϕ_v direction results in a simple ϕ-offset in the heightmap.

A point $\hat{p}$ with local spherical coordinates $(\hat{\phi},\hat{\theta})$ on the hemisphere has the coordinates

$$\hat{p} = \begin{pmatrix} \cos\hat{\phi}\cdot\sin\hat{\theta} \\ \sin\hat{\phi}\cdot\sin\hat{\theta} \\ \cos\hat{\theta} \end{pmatrix}$$

The world coordinates of this point are:

$$p = \begin{pmatrix} \cos\theta_v\cdot\hat{p}_x - \sin\theta_v\cdot\hat{p}_z \\ p_y \\ -\sin\theta_v\cdot\hat{p}_x + \cos\theta_v\cdot\hat{p}_z \end{pmatrix}$$

The conversion back into spherical coordinates can be done using the following equation:

$$\begin{pmatrix} \phi \\ \theta \end{pmatrix} = \begin{pmatrix} \tan^{-1}\frac{p_y}{p_x} \\ \cos^{-1}(p_z) - \theta_v \end{pmatrix}$$

Points on the view hemisphere are transformed into the world space using the upper transformations. The rectangular heightmap is parameterized by $(s,t) = (\phi,\theta)$ and can therefore be sampled with the parameters of world space. These steps are pictured in Figure 2.12.

Clasen and Hege [CH06] create the grid of the hemisphere by dividing $\hat{\phi}$ into n fixed steps. $\hat{\theta}$ is divided into concentric rings which exponentially grow in increasing $\hat{\theta}$ direction and correspond to the grid rings of the levels in the original implementation by Losasso and Hoppe [LH04]. Each concentric ring is subdivided into m fixed rings and the resulting discrete elements partitioned into two triangles.

As a result of this partitioning scheme the 1:1 correspondence of vertices and height samples is lost. In contrast to the original geometry clipmaps no T-intersections at level boundaries are created.

The visibility of the different concentric rings is dependent of the camera's height. The lower bound is determined by the earth curvature. Parts behind the horizon (from the camera's point of view) are not visible. Same as in the original geometry clipmap implementation finer clipmap levels are skipped if the triangles cover less than one pixel. Therefore the

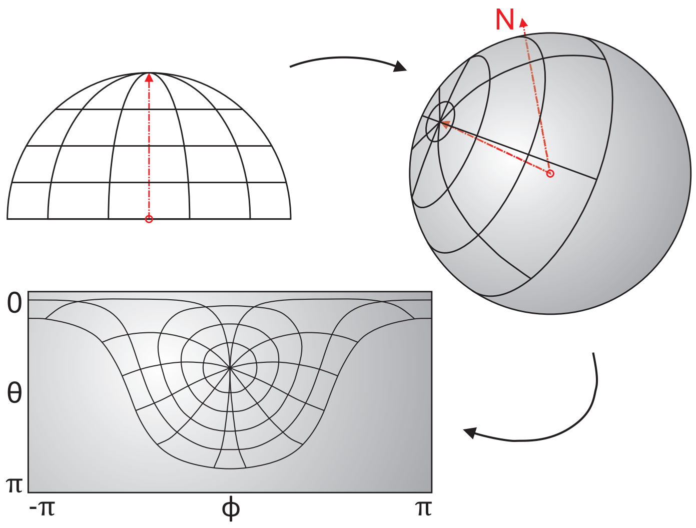


Figure 2.12.: Transformation from spherical coordinates to sample coordinates (from Clasen and Hege [CH06]).

upper bound is defined by the height of the camera above the surface. It is important to note that no matter how far the camera is relative to a planet only one hemisphere of the spherical object is visible.

Like in the improved GPU geometry clipmap implementation by Asirvatham and Hoppe [AH05], the vertex texture fetch is the bottleneck of the implementation by Clasen and Hege.

Clasen and Hege solve the issue of the implementation by Losasso and Hoppe [LH04] that does not to allow the rendering of spherical objects. As a drawback of this, they run into the precision problem discussed in Chapter 1. Therefore using this technique allows only rendering a limited range of terrain and is not suitable for our needs.

Chapter 3.

Theory

The main requirement for our application is to render arbitrary large terrain in real-time. As reviewed in Chapter 2 there exist a number of algorithms that accomplish that but none of them fully meets our demands. The GPU-based geometry clipmap algorithm by Asirvatham and Hoppe [AH05] attracted us the most but unfortunately it lacks of the ability to render large arbitrary curved surfaces. For instance it is not possible to display large spherical shaped objects like the earth. Their implementation only allows rendering terrain that resides on a flat basis. Therefore we present an algorithm that overcomes this planar basis restriction. In this chapter we describe the theory behind our GPU-based geometry clipmap implementation.

3.1. Overview

The main problem of all the terrain rendering algorithms is the huge amount of data that current CPUs and graphics cards are not able to handle. Level-of-detail techniques have to be used which process the data so that the engine is able to render the terrain in real-time without losing visual information.

The GPU-based geometry clipmap algorithm creates a terrain with decreasing sample density when moving away from the viewer. Asirvatham and Hoppe [AH05] state that by this way triangles are rendered that are uniformly sized in screen space and therefore fulfill the level-of-detail demand.

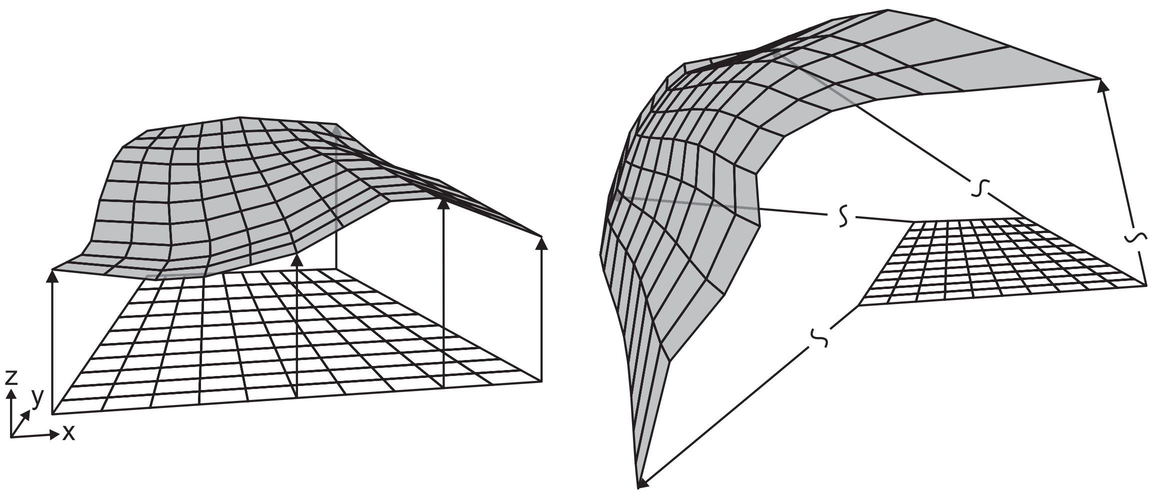


Figure 3.1.: Simple heightmap elevation (left) compared with the use of a 3D vertex texture (right).

Same as Asirvatham and Hoppe [AH05] we store the terrain in vertex textures. In the original implementation the authors use 2D elevation images, prefiltered into a mipmap pyramid of L levels. Their application caches square windows of $n \times n$ samples for every level. Those windows are used during the rendering to elevate regular grid rings centered about the viewer and thus create the terrain. Every sample of the elevation image stores the z coordinate of the according terrain point explicitly in its value and the x and y coordinate implicitly by its position within the image. Figure 3.1 left shows a simple grid that is elevated using a heightmap texture.

Instead of using 2D elevation images like described above, our implementation stores all three coordinates of the terrain samples in the RGB channels of a RGBA[1] floating point texture. The information about the sample's position within the whole texture is only used to connect the points to triangles. This technique is the main difference between the algorithm presented by Asirvatham and Hoppe [AH05] and our implementation and enables it to render terrain that resides on arbitrary basis because the created mesh is not bound by a regular grid mesh that can only be altered in z-direction (compare Figure 3.1 right).

[1]**R**ed, **G**reen, **B**lue, **A**lpha: the different channels of a four channel texture.

3.2. Preprocessing

In a preprocessing step a data structure is generated that helps the render engine to reduce per-frame calculations as much as possible. In the following the terrain data is referred to as *terrain image* that is indexed by x and y and holds on position $P_{x,y}$ the actual (x_a, y_a, z_a) coordinates of the terrain sample.

The preprocessing is done by a separate part of our implementation and consists of following steps:

1. Tile the terrain data into large patches if the data's extent is too great.
2. Prefilter the terrain data patches into mipmap pyramids of L levels.
3. Tile each level of a mipmap pyramid into tiles of fixed sized.
4. Calculate a bounding box for each tile.
5. Save the tiled levels and the according bounding boxes into separate binary files.

In the first step of the preprocessing procedure the application checks the raw terrain data on its extent. If the extent of the terrain is too large to be rendered by one instance of the algorithm, i.e. the algorithm runs into a precision problem, the terrain is tiled into large patches (compare Section 3.4).

In the next step the terrain patches are prefiltered into mipmap pyramids by taking the terrain image patches and removing every second texel in x and y direction. L terrain image levels for every terrain patch are produced by repeating this step recursively until a minimum mipmap level size of $(n+1+s_t) \times (n+1+s_t)$ is reached.

The terrain image levels usually do not fit into the memory as a whole. Therefore they have to be stream on demand. To enable this, the individual levels are separated into tiles. The size s_t of those quads is calculated by dividing the clipmap size n (compare Section 3.3) by a user determined factor c_t (that has to divide the clipmap size whole-numbered):

$$s_t = \frac{n+1}{c_t}$$

A larger c_t value produces smaller tiles which will result in streaming more often less amount of data into the memory.

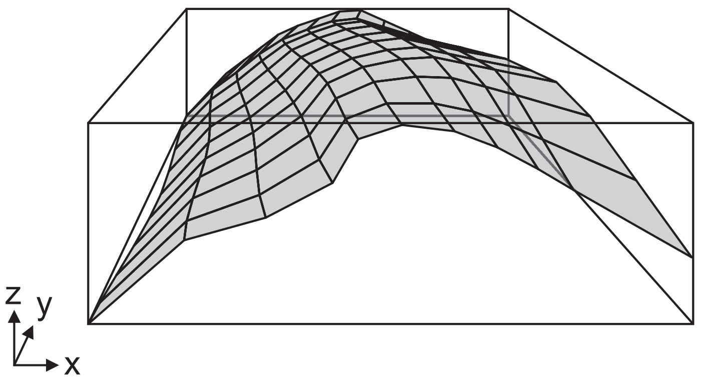


Figure 3.2.: An axis-aligned bounding box for the terrain image points of a tile.

To enable view frustum culling an axis aligned bounding box is calculated for every created tile (compare Figure 3.2). This bounding box describes the minimum and maximum extend in x_a, y_a and z_a direction of the terrain points saved in the according tile. During the rendering process it is used for a quick check whether the tile's terrain points are visible or not (compare Section 3.5.4).

After having calculated the level tiles and the according bounding boxes, all the data is saved to disk. Starting with the upper left tile of the terrain image all the tiles are traversed line-by-line and converted to binary streams. All the streams of one terrain image level are merged into a large one and saved to disk. Same is done with the bounding boxes. As a result every level creates two files on the hard disk. Disk space consumption and therefore the amount of data that has to be read from disk can be reduced by compressing those streams. The drawback of the compression is the heavier CPU load during the rendering process due to the decompression.

For the shading of the surface during the rendering process the normal of every terrain sample is needed. In our application we implemented two different ways of creating the normals. Because all the terrain information is available in the vertex shader the normals can be computed on the fly during the rendering process. On the other hand it is possible to precompute the normals into a normal map in a preprocessing step. Similar to the handling of the terrain images the normal map is loaded on demand during the rendering into the GPU's memory and accessed by the vertex shader using vertex texture lookups.

The advantage of the preprocessing approach is the reduced computational effort during the vertex shader execution while the drawback is double the amount of data that has to be streamed from the hard disk to the graphics card. Both the increased vertex texture lookups that are needed for the computation of the normals on the GPU as well as the doubled streaming data are bottlenecks for fluid rendering. It turned out that the greater bottleneck for our implementation is the amount of data to be streamed. Therefore we choose to compute the normals on the GPU in our final implementation. Nevertheless the creation of the normal maps in a preprocessing step is described in the following.

Every sample point $P_{x,y}$ is in x and y direction surrounded by 4 sample points:

$$\begin{array}{ccc} \dots & P_{x,y-1} & \dots \\ P_{x-1,y} & P_{x,y} & P_{x+1,y} \\ \dots & P_{x,y+1} & \dots \end{array}$$

We calculate the normal of $P_{x,y}$ by the cross product of vector $(P_{x-1,y} - P_{x+1,y})$ and vector $(P_{x,y+1} - P_{x,y-1})$. Using the exact normal for rendering the image produces a very harsh image. To reduce this harshness the normals are altered by a smoothing coefficient and filtered afterwards with a *Gauss kernel*. By storing each normal with the same x and y index like its according image point a normal map is created.

Same as the terrain image, the normal map is filtered into a mipmap pyramid with L levels. After tiling all the mipmap levels into quads of the same size like the clipmaps, each normal map level is saved to disk into one binary stream.

3.3. Clipmap Structure

The original algorithm of geometry clipmaps by Asirvatham and Hoppe [AH05] has to be adjusted only little to fit our new requirements. Same as the authors we cache a $n \times n$ square window of samples for every level. These windows produce nested grid rings centered about the viewer. As we give up the x and y correspondence between the looked-up samples and the grid used for the lookup in our implementation, the resulting grid usually won't be a regular grid anymore. Nevertheless the finer-level windows have smaller spatial extent than the coarser ones. Like in the original implementation this ensures that the triangles of the nested grid windows are uniformly sized in screen space.

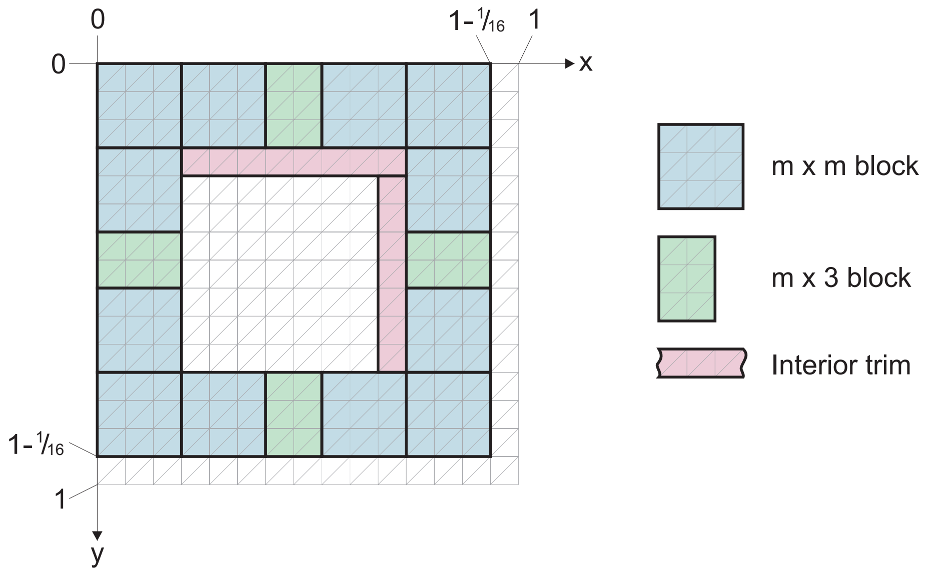


Figure 3.3.: Spatial extent of a clipmap ring ($n = 15$).

The grid size of a whole clipmap ring has to be odd to ensure that each level lies on the grid of the next coarser level. The size of the elevation texture is chosen to be a power of two because the hardware is often optimized for this size. As a result the clipmap ring size is $n = 2^k - 1$ (with any desired k).

The grid that encodes the *(x,y)* geometry and is used for the texture lookup is not created by a single vertex and index buffer but rather composed by different footprints. Those small sets of constant buffers are reused for every level. As Asirvatham and Hoppe state in their work [AH05] the use of multiple vertex and index buffers both reduce memory costs and enables view frustum culling.

All the clipmap levels are rendered as rings which are centered about the viewer. Only the finest level is rendered as a complete grid square. Each clipmap ring is generated by four different footprint pieces (compare Figure 2.9 and Figure 3.3). The first footprint of size $m \times m$ with $m = (n+1)/4$ is used 12 times and covers most of a clipmap ring. A smaller footprint ($m \times 3$) is used to fill the gaps between the $m \times m$ pieces. When the next finer level is fit in the hole of a level, a L-shaped hole arises that is filled by *interior trims* (blocks of size $(2m+1) \times 2$ and $2m \times 2$). Finally a set of degenerated triangles is used on the outer perimeter of the level to avoid mesh T-junctions and fill small holes that possibly arise between the two

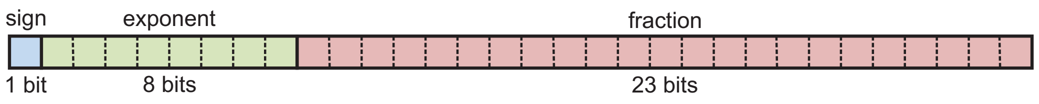


Figure 3.4.: Structure of the IEEE standard for binary floating-point arithmetic.

levels.

During the rendering of the footprints, the vertex and index buffer are constant – only affine transformations (translation, rotation, scaling) are applied. In the original implementation by Asirvatham and Hoppe [AH05] the location of the footprints differs from level to level because the x and y coordinate does not change during the elevation process. This is different in our implementation. For every level in our implementation, the footprints form a ring (or a square for the finest level) with a spatial extent of 0 to $(1 - \frac{1}{n+1})$ in x and y direction (compare Figure 3.3). This allows the vertex shader to use the x and y coordinates as u and v values for the vertex texture lookup and therefore those values have to be the same for every level. The shift of the vertices to their desired position within the terrain is done by the vertex shader by overwriting their coordinates with the values stored in the vertex texture.

The footprints are encoded in stets of constant vertex and index buffers. Following Asirvatham and Hoppe [AH05] we use 16-bit indices for the index buffer, resulting in a maximum block size of $m = 256$ and therefore a maximum clipmap size of $n = 1023$. The length of the indexed triangles encoded by the index buffer enables optimal vertex caching.

3.4. Precision Problem and Solution

The drawback of changing the original geometry clipmap algorithm from the use of heightmaps to textures holding 3D information is the introduction of a precision problem. Like stated in the introduction of this work numerical problems can arise due to the limited 32 bit floating point textures on current graphics cards.

The IEEE standard for binary floating-point arithmetic stores in the first bit of the 32 bits the sign, in the succeeding 8 bits the exponent and in the last 23 bits the fraction (compare Figure 3.4). The fraction is used for storing the actual number, the exponent "only" scales this number. If the 23 bits are used for storing a texture, the value can range from -2^{23} to 2^{23} (= 16.777.216) arbitrary units. Assuming a terrain with a distance between two samples of one meter results in a maximum extent of around 16.800 km (maximum surface size of ~280 million km^2) without getting numerical errors for the x and y coordinates. Therefore the earth with a surface size of around 510 million km^2 sampled every one meter does not fit into a 32 bit floating point texture, not to think about higher sampling resolutions.

Our solution to overcome this restriction is to split up the surface into multiple individual patches. Each one of these patches can have the maximum extent of a floating point texture. Often the patching has to be done anyway. Spherical objects for instance have to be partitioned for being able to be stored in rectangular quads.

During the rendering process our implementation checks every frame whether the border of one terrain patch is reached. In case the inquiry returns true the algorithm is instantiated a second time. During the first run the algorithm continues to render the terrain like in the previous frames (with a slightly different camera position). Because the border of the terrain part is reached not all of the actually visible terrain can be rendered. During the second run the adjacent terrain patch is loaded and the missing parts of the terrain are rendered. Naturally the different parts of the terrain rendered during the two runs have to fit seamlessly together. To archive this in the second run the camera has to be moved from the local coordinate system of the first terrain patch the appropriate position within the local coordinate system of the second patch.

Figure 3.5 schematically shows the rendering of such patch-overlapping clipmaps in 2D. During the first rendering pass the camera is set to its position (visualized by the orange line) within the coordinate system of the first patch (X_0, Y_0) and the green parts are rendered. During the second pass the camera is moved to its position (yellow line) within the second patch's coordinate system (X_1, Y_1) and the missing parts (in blue) are rendered.

If the camera entirely moves out of the first patch the rendering of the first pass as well as the first patch can be dismissed and the second pass handles all the terrain. At most four patches and therefore four algorithm instances have to be loaded if the camera moves to the corner of one patch (compare Figure 3.6).

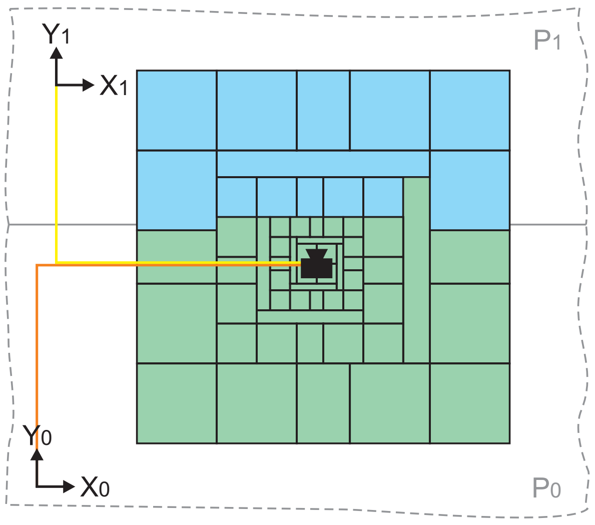


Figure 3.5.: Rendering of clipmaps which overlap two terrain patches (P_0 and P_1).

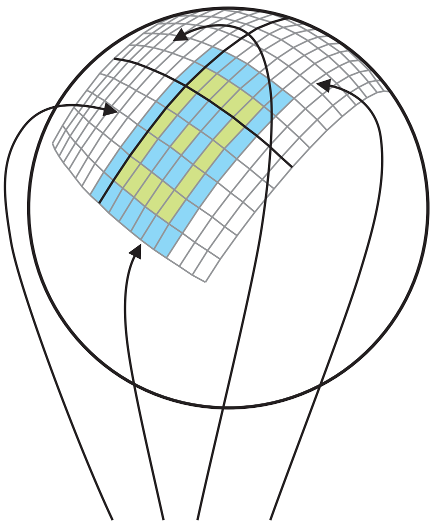


Figure 3.6.: A maximum of four algorithm instances at the corner of one patch.

3.5. Rendering Process

The rendering process of our implementation can be summarized by following steps:

1. Load the needed parts of the terrain image levels, (the normal maps,) and the bounding boxes into the system's and GPU's memory.
2. Upon camera movement detect if missing data has to be reloaded from the hard disk into the main memory.
3. Update the textures on the GPU
4. From fine to coarse, render the different clipmap levels with the footprints. Detect if the rendering of individual footprints can be neglected by performing view frustum culling.
5. Repeat step 2. to 4. until the user request termination. Reinstantiate the rendering process with a different viewpoint and clipmap level textures, if the viewer moves to the edge of the terrain patch.

The upper steps are described in more detail in the following subsections.

3.5.1. Disk-to-Memory Handling

With the start of the application a camera position is defined within the terrain's area. Using this information the application calculates which of the terrain image tiles cover the clipmap levels around the camera. Like stated in Section 3.2 the size of one tile is $s_t = (n+1)/c_t$, whereas n is the clipmap size and c_t a user determined factor. In general a clipmap level is covered by ${c_t}^2$ tiles (compare Figure 3.7 left). However if the level shifts, it can happen that an extra band of tiles is needed on both x and y direction to cover the whole level.

In our implementation the number of tiles which are loaded does not change. At any time $(c_t+1)^2$ tiles are kept in the main memory. The advantage of the greater number of loaded tiles is the lesser necessity to update the terrain image information. Analog to the loading of the terrain image tiles the application loads the according bounding boxes and the tiles for the normal maps (if normal map usage is enabled).

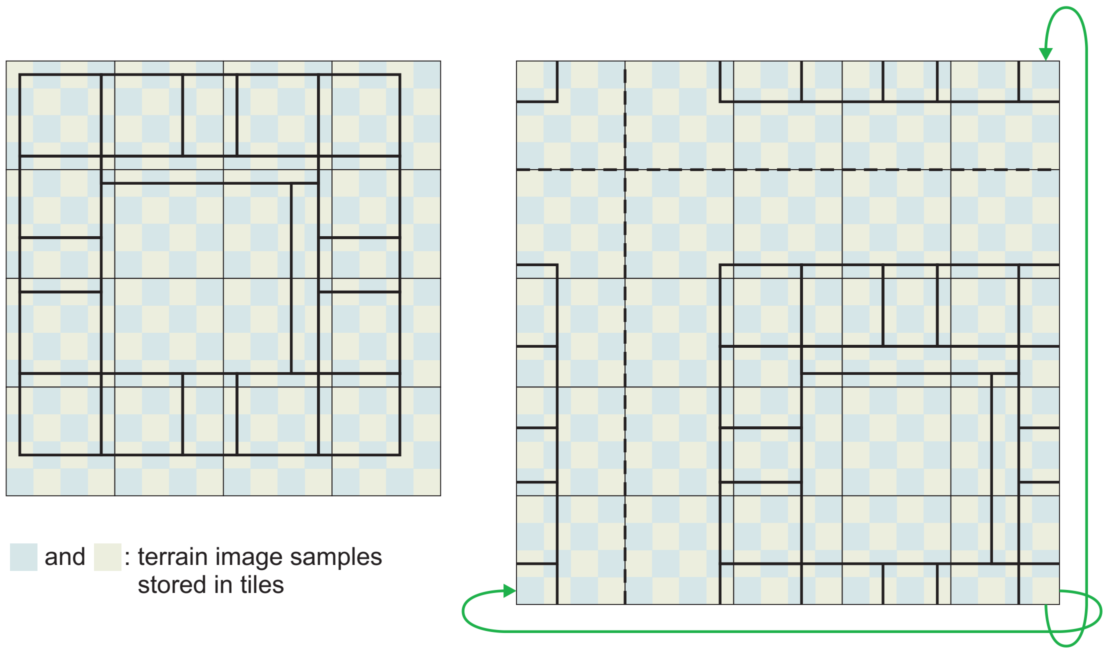


Figure 3.7.: A clipmap ring ($n = 15$) referenced within terrain image tiles ($c_t = 4$). Left: minimum covering. Right: optimal covering with wrap-around addressing.

If the camera is requested to move the application checks whether the new position of the clipmap within the tiles exceeds the area covered by the ones current loaded and gets the tiles that are missing. In the main memory the individual tiles are stored in a 2D-array and referenced by wrap-around addressing (compare Figure 3.7 right). If an array index exceeds the arrays dimension boundaries in any direction, the index is wrapped around the boundaries by subtracting the length of the according dimension. This technique avoids updating all the indices of the tiles if new ones are loaded.

3.5.2. Memory-to-GPU Handling

After loading the terrain information into the main memory, the application creates floating point textures on the GPU. For each level two RGBA floating point textures are allocated: one for storing the terrain information and the second one for the normal map. Both textures leave the alpha channel unused. If the application calculates the normals on the fly only one 128 bit floating point texture is needed.

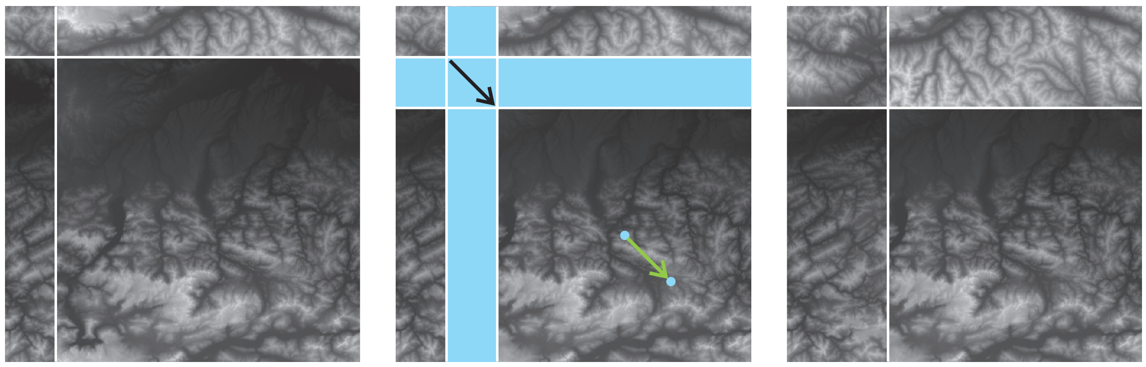


Figure 3.8.: Processing of a toroidal update (following Asirvatham and Hoppe [AH05]). Left: texture before update. Middle: viewer motion in green and update region in light-blue. Right: texture after update.

Similar to the management of the tiles, the application takes advantage of the graphics cards built in 2D wraparound addressing feature. If uv values exceede the range $[0,1]$ they are automatically remapped into this range. Therefore the upper left corner of the actual terrain image does not correspond to the upper left corner of the texture but moves within it.

If the position of the viewer moves and the system's main memory is already updated, the missing parts of the clipmap levels in the GPU's memory are updated as well. Those parts are generally L-shaped. 2D wraparound addressing of the clipmap level textures makes efficient incremental update possible and avoids copying existing data from the main memory to the GPU's memory (compare Figure 3.8). Only the required parts are updated. In general the L-shaped update region becomes a cross shaped region on the GPU-texture. The update is done by overwriting one to four quads within the texture.

3.5.3. Vertex and Pixel Shader

During the execution of the rendering pipeline the vertex shader creates the actual terrain. This shader takes the footprint pieces which remain in their local coordinate system and shifts them to their world position using the given world matrix. This matrix consists of simple translation, rotation and scaling transformations. After the world transformation the z coordinate of the vertices is 0 and the values of the x and y coordinates range from 0 to $(1-\frac{1}{n-1})$.

In the next step the shader uses the values of the x and y coordinate as u and v values for the vertex texture lookup in the terrain image texture. The resulting RGBA values are interpreted as the new *(x,y,z)* coordinates of the vertex used for the lookup. As mentioned before the A channel of the RGBA floating point texture remains unused in our case.

When using a normal map, the vertex shader uses the previous u and v values to get the according precomputed normal from the normal map. In our final implementation we do not use a normal map but instead compute the normal on the GPU. This requires four more vertex texture lookups for every processed vertex in the vertex shader. In u and v direction the circumjacent vertices are looked-up by in- and decreasing the u and v value by one step $t = 1/(n+1)$. Analog to the offline creation of the normal map the normal of a point $P_{u,v}$ is computed by the cross product of vector $(P_{u-t,v} - P_{u+t,v})$ and vector $(P_{u,v+t} - P_{u,v-t})$ (compare Section 3.2). On the borders of the texture one of the four adjacent points is not available. Instead of this missing fourth point we use $P_{u,v}$ itself. Usually the difference between two adjacent points is not very high and therefore no visually noticeable distortions occur. The resulting normals can be altered by a smoothing factor to reduce their harshness.

At last the vertex shader applies the view-projection matrix to the vertices. Apart from the vertex position and the normal, the light direction, the viewer direction and the original uv values are also packed together as input parameters for the pixel shader.

We chose to render our scene using the Phong illumination model with directional per pixel lighting [Wik07, Kon07]. This technique interpolates the surface normals across the rasterized polygons. As a result the lightning calculation is carried out for every fragment. The pixel shader calculates the needed parameters ($I_{texture}$, c_{shadow}, $c_{diffuse}$, $I_{specular}$) and combines those with predefined values (I_{debug}, $I_{ambient}$, $I_{diffuse}$, $c_{specular}$) using following formular:

$$I_{out} = I_{debug} \cdot I_{texture} \cdot (I_{ambient} + c_{shadow} \cdot (c_{diffuse} \cdot I_{diffuse} + c_{specular} \cdot I_{specular}))$$

I_{debug} is used to color the different footprints of the clipmap levels on user request in different colors. The output of the pixel shader is the color I_{out} of the according fragment.

In contrast to the implementation of Asirvatham and Hoppe [AH05] we do not perform geometric blending between the different clipmap levels. The overlapping vertices on the edges of two levels are the same as a result of our mipmap creation technique. The hole that arises because of the coarser resolution of the follow-up clipmap level is filled using the degenerated triangles.

3.5.4. Performance Optimization

Performance optimization is critical for the most rendering applications to avoid unnecessary work that decreases the frame rate. Naturally it is important to use any hardware specific optimization. In this section we will discuss various performance optimization techniques we took advantage of in our application.

Depth Test

Because of the nature of the geometry clipmap algorithm coarser levels are more far away from the camera than the finer ones. Therefore we render the levels from fine to coarse. As a result triangles more closer to the camera are rendered earlier. Secondly we render the larger footprints (of size $m \times m$) before the smaller ones which might be occluded by the bigger parts. All of this allows the graphics cards to use its depth test feature (Z-buffering) in an optimal way. Rewriting a fragment by the rasterizer is costly and therefore the rendering speed is improved if a close triangle creates fragments that omit the rewriting because of the depth test.

View Frustum Culling

Another performance optimization that greatly reduces rendering load is view frustum culling. In contrast to the implementation by Asirvatham and Hoppe [AH05] we cannot directly extrude the footprints in z direction and test them on visibility by intersecting the extruded volume with the view frustum in 3D. In our implementation we do not know the boundaries of the footprints because the information is foremost extracted in the vertex shader from the vertex texture. Therefore we had to come up with another idea which we assume to be at least equal good performing than the view frustum culling technique by Asirvatham and Hoppe.

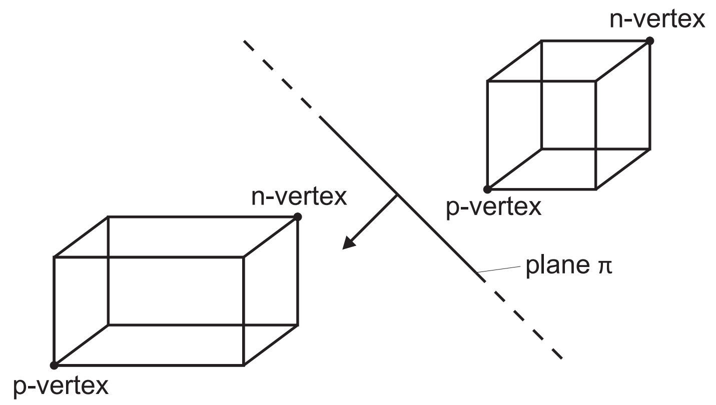


Figure 3.9.: The n-vertex and p-vertex of bounding boxes corresponding to plane π and its normal (from Assarsson and Möller [AM00]).

As described in the Section 3.2 during the preprocessing our application creates for every tile a bounding box by calculating the maximum extend of the tile's terrain in 3D. During rendering, apart from the tiles, the according bounding boxes are loaded into the main memory. Usually the footprints are not covered by a single tile, depending on the type of footprint and the factor c_t that indirectly specifies the side length of the tiles. Therefore multiple bounding boxes have to be combined to create a larger bounding box that fully covers the footprint's terrain. If a footprint is to be tested for visibility relating to the view frustum, our application first calculates the tiles that cover the footprint. With the use of that information, the according bounding boxes can be combined. If a bounding box is defined by (x_{min}, y_{min}, z_{min}, x_{max}, y_{max}, z_{max}) the combined bounding box can simply be created be taking the minimum values of x_{min}, y_{min} and z_{min} and the maximum values of x_{max}, y_{max} and z_{max}. The resulting larger bounding box is then tested against the view frustum with a simple outside test following an algorithm presented by Assarsson and Möller [AM00].

Assarsson and Möller suggest to test only two vertices of an axis aligned bounding box (AABB) with the view frustum instead of all vertices. They call those vertices the n-vertex (negative far point) and the p-vertex (positive far point). Figure 3.9 shows the correspondence between the two points and a plane. For the determination of the n- and p-vertex a look-up table can be used (compare Table 3.1). For every view frustum plane the signs of the normal's components give the indices (n_x, n_y, n_z) for the look-up table.

The basic collision test by Assarsson and Möller uses only two dot products and determines

n_x	n_y	n_z	p-vertex	n-vertex
$+$	$+$	$+$	$[x_{max}, y_{max}, z_{max}]$	$[x_{min}, y_{min}, z_{min}]$
$+$	$+$	$-$	$[x_{max}, y_{max}, z_{max}]$	$[x_{min}, y_{min}, z_{min}]$
$+$	$-$	$+$	$[x_{max}, y_{max}, z_{max}]$	$[x_{min}, y_{min}, z_{min}]$
$+$	$-$	$-$	$[x_{max}, y_{max}, z_{max}]$	$[x_{min}, y_{min}, z_{min}]$
$-$	$+$	$+$	$[x_{max}, y_{max}, z_{max}]$	$[x_{min}, y_{min}, z_{min}]$
$-$	$+$	$-$	$[x_{max}, y_{max}, z_{max}]$	$[x_{min}, y_{min}, z_{min}]$
$-$	$-$	$+$	$[x_{max}, y_{max}, z_{max}]$	$[x_{min}, y_{min}, z_{min}]$
$-$	$-$	$-$	$[x_{max}, y_{max}, z_{max}]$	$[x_{min}, y_{min}, z_{min}]$

Table 3.1.: Look-up table for determining the n- and p-vertices.

one of the AABB's states: outside, inside or intersect. The first test detects if the n-vertex is outside the half-space defined by the view frustum plane used for the test. If the first test fails the second test is run for the p-vertex. If this vertex is outside, the AABB intersects the view frustum plane. If both vertices are inside the plane, the whole AABB is inside the half-space.

Following pseudo code implements the two tests using the n- and p-vertex (`nVertex`, `pVertex`), the plane's normal (`nPlane`) and the plane's offset (`dPlane`) (following Assarsson and Möller [AM00]):

```
a = nVertex · nPlane + dPlane;
b = pVertex · nPlane + dPlane;

if(a > 0) {
  return OUTSIDE;
} else if(b > 0) {
  return INTERSECT;
} else {
  return INSIDE;
}
```

After testing the n- and p-vertex of an AABB with all the view frustum planes the position of the bounding box corresponding to the whole view frustum is known. In our case the

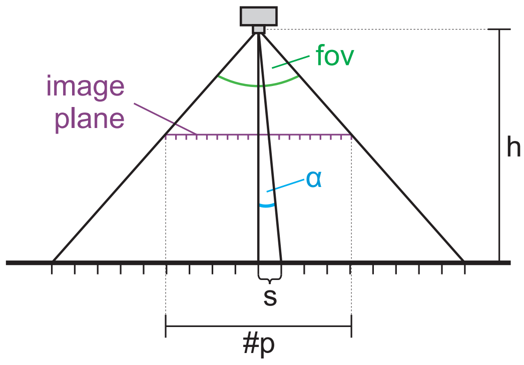


Figure 3.10.: The size of one screenpixel s calculated by the viewer's height h and the angle α.

intersection test can be omitted – if a terrain tile is partly visible we have to render it anyway. This slightly reduces computational effort of the view frustum culling because only the n-vertex has to be tested.

Level Deactivation

If the camera is far enough from the underlying terrain the finest clipmap levels might be unnecessarily dense. Asirvatham and Hoppe [AH05] state that apart from the fact that too much information is rendered without getting a visually better result, even aliasing artifacts might occur. Therefore during the rendering process not all of the L allocated clipmap levels are used for rendering. Based on the height of the user over the underlying terrain a set of active levels 0 to $L-1$ is defined.

Following Clasen and Hege [CH06] the size of one screen pixel mapped onto the surface can be calculated with the height h of the user, the field of view angle fov and the number of pixels $\#p$ per scanline (compare Figure 3.10):

$$s = h \cdot tan\alpha = h \cdot tan\frac{fov}{\#p}$$

After having calculated s, it can be compared with the average size of a triangle of the finest active level. This size on the other hand is calculated by dividing the level extent by the number of samples in one direction. If the average triangle size is smaller than twice the screen pixel size (i.e. a triangle is rendered by less than two pixels), the level is omitted in our application. If a level is shifted, both the height of the user from the ground and the levels extent are updated using the samples stored in the system's main memory.

If the user moves fast over the surface the rendering engine might slow down due to the massive amount of data that has to be streamed into both the systems and GPUs memory. As a result the frame rate drops to an unwanted low level. If the application notices that the loading of the terrain is too slow it deactivates finer levels to ensure a certain frame rate. Our application loads the terrain information from coarse to fine level. That ensures that coarse levels are updated sooner and enough terrain is available for rendering. This technique does not have a visually bad impact on the rendered image because of the viewer's higher motion fine details are not noticeable anyway.

3.5.5. Camera Movement

The connection between the camera movement and the level update is slightly more complicated than in the original geometry clipmap implementation. Differently from Asirvatham and Hoppes [AH05] implementation a shift of the camera in x and y direction does not necessarily has to result in a shift of the levels in the same direction because the terrain is allowed to reside on a curved surface. Generally the shift of the camera has to be projected onto the basis of the terrain. If the projection is greater or equal than the predefined unit for shifting, a level shift has to be requested. The problem is that the basis is not directly known by the implementation and even if the the basis were known by the application it can happen that there might exist multiple possible projections for an arbitrary basis.

Our solution is to use the corner points of the finest level (or the current area saved by the tiles of the finest level) to create a quad (compare Figure 3.11). In the next step the camera's position is normal projected onto this quad. Because the number of samples hold by the tiles in one direction is known, an average step size can be calculated for the two directions of the quad. The application only needs to divide the side lengths by the number of samples. Now the application can check every frame if the projected camera position exceeds one of the two averaged step sizes and if so, issue a level shift. Upon a level shift the application has to recreate the quad and recalculate the averaged step sizes.

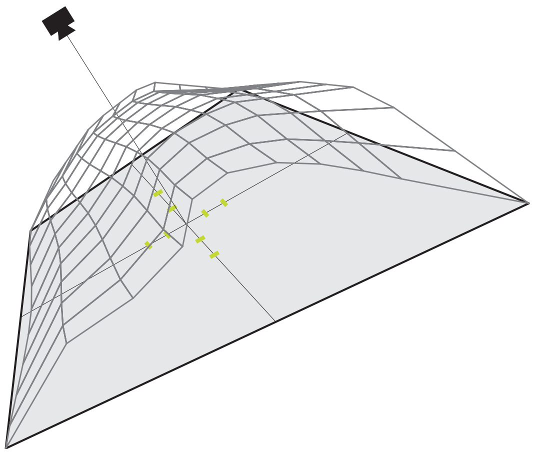


Figure 3.11.: Controlling of the camera movement by a quad defined by the finest level with the averaged step sizes (small green lines).

3.6. Issues

In this chapter we raise several issues that arise due to changing the original geometry clipmap algorithm from the use of heightmaps to textures holding 3D information. Those issues cannot be controlled or corrected by our implementation but have be considered by the user during the terrain preprocessing.

3.6.1. Grid Size

In their original geometry clipmap implementation Asirvatham and Hoppe [AH05] hope use regular grids as a basis for the heightmap elevation. During the elevation process this basis does not change in x and y direction and therefore the distances between adjacent samples are very similar, differing only by a factor introduced due to the z-displacement.

In contrast to that our implementation has no direct control over the distance between the samples. On the average coarser levels will have double the distance than the next finer levels but the charge of the actual position of the samples lies at the user who controls the preprocessing of the terrain. If the user chooses a tessellation of the terrain that creates very unequal sample distances in x and y direction, our algorithm cannot guarantee anymore that the triangles of the nested grid windows are uniformly sized in screen space. The terrain created by the user in the preprocessing step does not have to reside on a regular grid basis but as a general guideline the user should try to approximate such a regular basis as much as possible.

3.6.2. Terrain Shape

Our implementation is able to visualize arbitrary shaped objects but the main purpose is to render large terrains. Such terrains do not have to reside on a flat basis and can be convex as well as concave. Theoretically our application is able to render not only spherical objects from the outside but also from the inside. But it has to be considered that our application renders only a finite amount of levels and hence a limited range. If the user wants to render closed concave objects like a sphere from the inside and looks at the ceiling there might not be any terrain visible because the distance measured on the ground is too large.

Another issue that has to be kept in mind is that the slope of the object's virtual basis should not be too steep (in other word: the virtual radius of the object is too small). The result would be that the coarser level might lap into each other and produce distortions. The user has to make sure that at any point on the surface the coarsest level has to fully fit on the surface without overlapping.

3.6.3. Patching of the Surface

Our algorithm has been developed for very large datasets that exceed the floating point precision as discussed in Section 3.4. It is not a problem for our algorithm to display such datasets but they have to be partitioned into large tiles to be able to be input into the application. In order to be able to tile the whole dataset an appropriate partitioning scheme has to be found. This can often be a difficult task but is crucial for the usability of the dataset for our application.

An example for a partioning scheme for a sphere was presented by K. M. Górski *et al.* [GHB+05]. They developed a *Hierarchical Equal Area isoLatitude Pixelization* (HEALPix) which can be used for partitioning a sphere in four square regions (compare Figure 3.12).

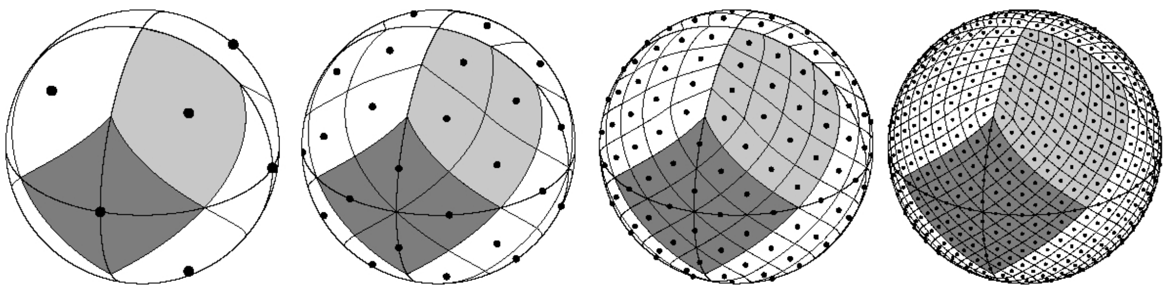


Figure 3.12.: Orthographic view of the HEALPix partition of the sphere (from K. M. Górski *et al.* [GHB$^+$05]).

Chapter 4.

Implementation

In this chapter implementation details of our application are discussed. The Unified Modeling Language (UML) [OMG07] is used for visualizing relations or processes within our application.

4.1. Development Environment

Our application is implemented using the C# programming language[1] with the use of the DirectX API[2] for accessing the graphics hardware. The vertex and pixel shader are written in the HLSL language. We chose C# for development because in our opinion this programming language made writing the application easier than other languages that are used for 3D application development like C++. C# manages the objects and provides a garbage collector which can result in slower execution speed than comparable code in an unmanaged language. Fortunately C# allows excluding certain parts of the code from being managed and therefore possibly run with increased speed. We did this for the parts in our code which we assumed to be runtime sensitive and therefore the overhead of the object management provided by C# is not too great compared with non-managed programming languages like C++.

[1]`http://msdn.microsoft.com/vcsharp`
[2]`http://www.microsoft.com/directx`

4.2. System Overview

In order to increase system flexibility and readability our application is distributed into different modules (compare Figure 4.1). The `GUI` module provides classes for interaction with the user. If the application is started `SettingsDialog` is called to retrieve various graphical adapter specific settings from the user like the resolution and full screen rendering. Upon occurrence of an unmanageable error during the execution the application's `ErrorDialog` displays the error message before the application is terminated.

The `Rendering` module provides the classes which form the basic rendering engine. The `GeoClipmapsMain` class is the first class of the application to be accessed after the startup. The class `RenderEngine` initializes the DirectX environment with the information gained from the `SettingsDialog` inside the `GUI` module. This class also handles all the peripheral input, both the mouse's as well as the keyboard's. The input of the keyboard is used for moving the camera while the mouse controls the viewing direction which is also used as the primary moving direction. All the camera specific calculations are performed within the `Camera` class.

The module `GeoClipmaps` contains all the classes that handle the loading, rendering and updating of the geometry clipmaps. While the `GeoClipmaps` class does the overall handling of the geometry clipmap algorithm, the loading and updating of the level's data is done by the `LevelManager`. View frustum culling is encapsulated within the `ViewFrustum` class.

The static classes `TimeStat` and `MsgHandler` inside the `Helper` module provide methods of general purpose that are used by most of the other classes. `TimeStat` is used for retrieving the accurate elapsed time at any time as well as other usefull time based values like the the current and averaged frame rate. The static class `MsgHandler` allows the application to output messages into a log file.

4.3. Design Motivation

The main design motivation for our application was to separation the rendering engine from the actual geometry clipmap algorithm. This is accomplished by the two modules

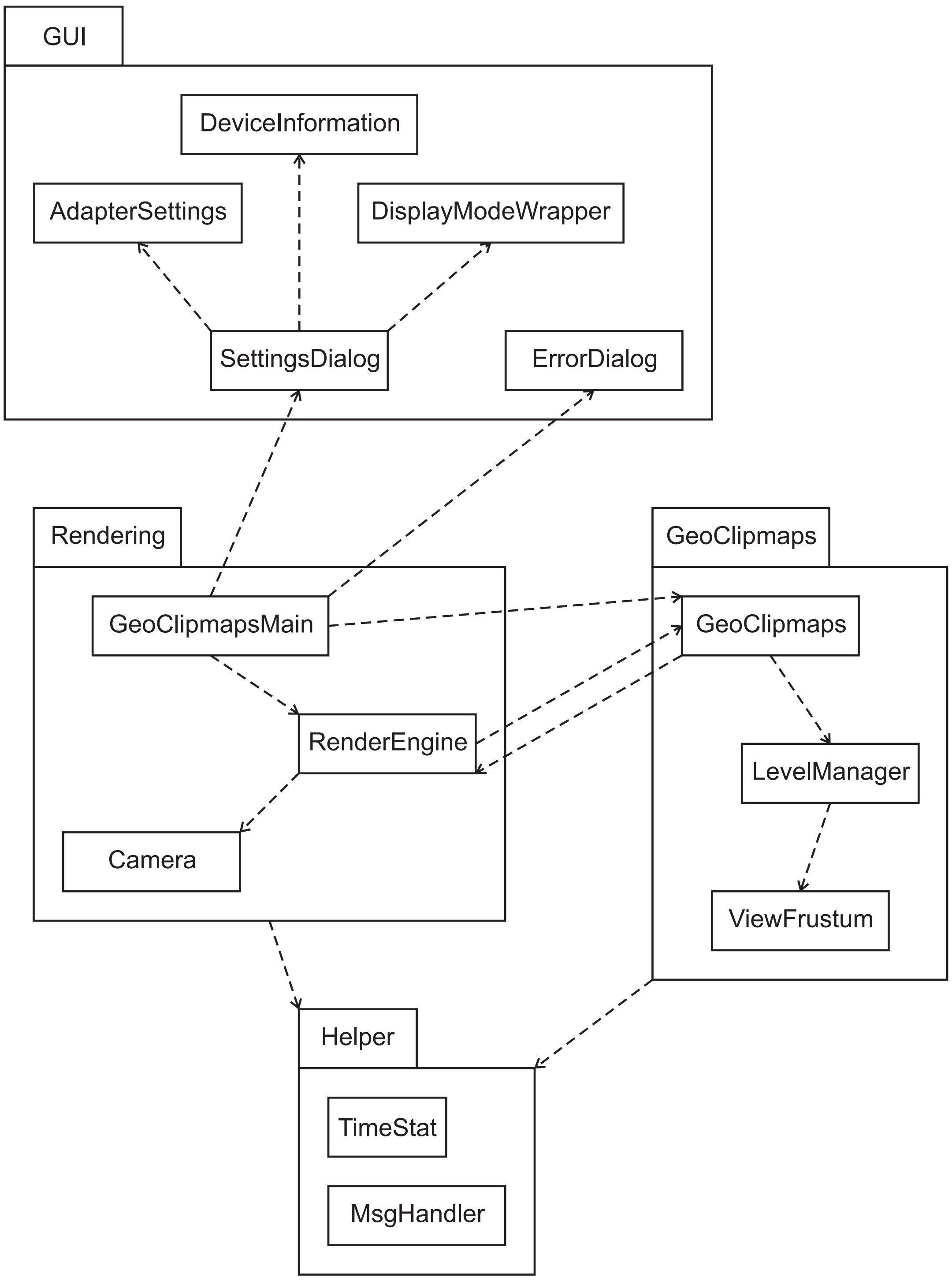


Figure 4.1.: UML class/package diagram of the GeoClipmap application.

`Rendering` and `GeoClipmaps` shown in Figure 4.1. With the use of *delegates*[3] in the C# programming language the rendering done by the `GeoClipmaps` part is integrated into the actual rendering engine. By this modular technique other rendering tasks can simply be plugged into the rendering engine without having to change neither the rendering engine nor the `GeoClipmaps` part. Such tasks can be for example an extended user interface or a head-up display.

The `Helper` classes `TimeStat` and `MsgHandler` are static classes and therefore do not have to be initialized. Those classes can be accessed by all other classes.

4.4. Rendering Process

Figure 4.2 shows a schematic representation of the rendering process. After the start of the application the user is asked to choose the display settings. One of the available adapters can be selected as well as hardware or software rendering. The user is allowed to run the application windowed or in fullscreen mode in an available adapter format and resolution. With this information the application initializes the graphics device. *Z-buffering* and *backface culling* is enabled. This allows the application to use those built-in performance optimizing functions of the graphics card. According to Konerow [Kon07] Z-buffering stores the depth value for every written fragment in a depth buffer and discards every fragment with a larger value than the one currently saved in the depth buffer. This ensures correct rendering of overlapping triangles. Backface culling discards all triangles whose normal face away from the camera (i.e. the backface of the triangle would be visible). In common rendering scenarios such triangles are not visible anyway, therefore quite a lot of rendering load can be quickly omitted.

After the setup of the render device the actual rendering loop can be started. The `RenderEngine` checks for peripheral input in the first step. Next to keyboard events like `F1` for calling an on-screen help menu, W/A/S/D key events are captured and used for moving the camera. As an alternative to the W, A, S and D key the arrow keys can be used. On a mouse movement event the x and y coordinates of the mouse within the rendering window are translated into a displacement vector between the mouse's old and new position. The keyboard events and the displacement vector of the mouse are passed to the `Camera` class.

[3]`http://msdn2.microsoft.com/en-us/library/ms173171.aspx`

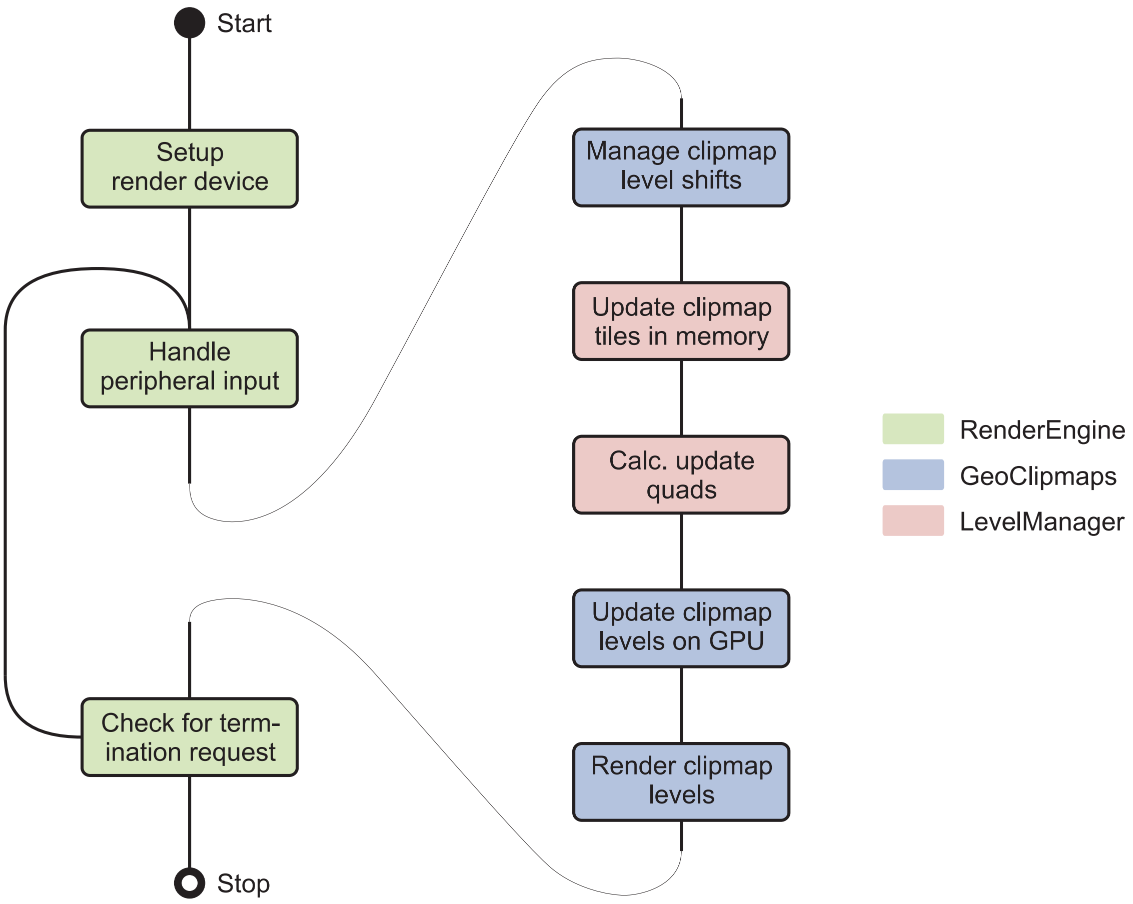


Figure 4.2.: Schematic representation of the rendering process.

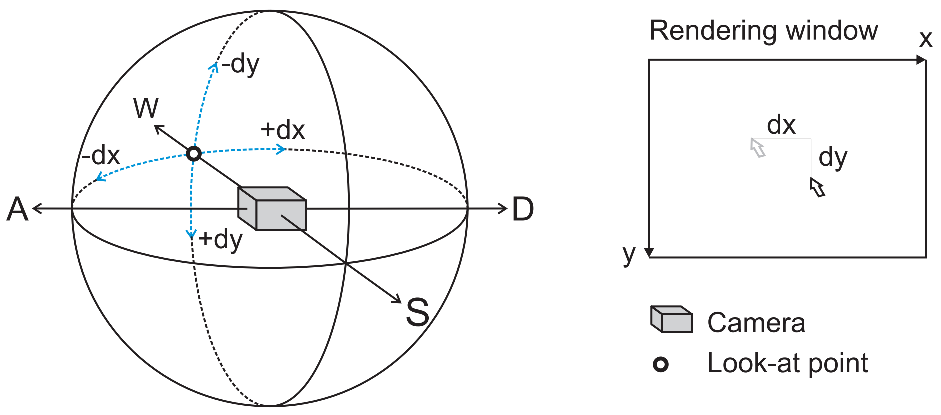


Figure 4.3.: Effect of mouse and keyboard events on the camera's position and the look-at point.

In this class the displacement vector is used to rotate the camera's look-at point around the camera position. The *y* component of the displacement vector is used for tilting the camera up or down while the *x* component affects the panning of the camera. If the user presses the W (or S) key the camera is moved towards (or away of) the look-at point. The A and D key trigger a horizontal sideways movement, right-angled to the vector between the point-of-interest and the camera's position. Figure 4.3 shows the effect of the various events on the camera's position and the look-at point.

In the next step of the rendering process the `RenderEngine` calls the rendering method of the `GeoClipmaps` class passed as a *delegate*. This step will be discussed in the next paragraph. In the last step the `RenderEngine` checks whether the user requested the application to be stopped and either restarts the rendering loop or terminates the execution.

All the geometry clipmap work is separated from the `RenderEngine` class and encapsulated in the `GeoClipmaps` and `LevelManager` class. During the first time the `GeoClipmaps` class is called, all the tiles which hold the clipmap levels are initialized and loaded from the hard disk. Also the geometry that is later used for rendering the different level parts is created. Next the actual per-frame-work is started. The `GeoClipmaps` class checks if the camera has moved and calculates the count of steps that the finest level has to shift to remain centered below the camera. The resulting displacement values are passed to the `LevelManager` which handles the recursive shift of all the levels and the updating of the level tiles (compare Section 3.5.1). Normally only a subset of the level tiles has to be updated from the disk. Only if the shift is larger than the side length of a whole clipmap level quad in the memory, all tiles have to updated.

When the `LevelManager` is finished with the update of the clipmap levels the `GeoClipmap` class calculates the texture regions that have to be updated on the GPU for every level (compare Section 3.5.2). The count of those regions range from one quad up to a maximum of four. For every region the `LevelManager` is requested for the data. It packs this data into a float array and returns it to the `GeoClipmap` class. Now the actual update of the GPU textures can be done.

In the last step of the application's geometry clipmaps part the actual rendering of the clipmap levels is done. To achieve optimal utilization of the graphics cards Z-buffering feature, the levels are rendered front to back. Every footprint is tested with the according bounding box against the view frustum and only rendered if its bounding box is visible (compare Section 3.5.4).

Chapter 5.

Evaluation

In this chapter the performance of our implementation is evaluated. In the first chapter the test scenario is described. Next to the system we used, the performed tests as well as the assembly of those are pointed out. Section 5.2 shows the results of the various test runs. The gathered data is visualized in different graphs and interpreted in Section 5.3. Section 5.4 provides possible solution approaches for the appeared problems.

5.1. Test Environment and Scenario

As the name suggests the primary goal of every real-time algorithm is to run at least at a certain minimum frame rate. Therefore we evaluated the real-time performance of our implementation by testing it under various conditions. All tests were performed on a system with following configuration:

OS	Windows Vista 32bit SP1
CPU	AMD Athlon64 X2 3800+
RAM	2048 MB
GPU	NVidia Geforce 8800GT

The application was tested in fullscreen mode with a screen resolution of 1920×1200 pixels. The vertical synchronization (VSync) was disabled to be able to measure frame rates higher than the monitor's refresh rate. As test scenarios we created four terrains with different level sizes. As the basis for the terrains we used a gray scale image as a height map, resized it to fit the level sizes and converted it to the 3D texture required by our application.

level size	level count	number of triangles
1024	3	6.795.278
512	3	1.693.710
256	3	420.878
128	3	103.950

Table 5.1.: Structure of the test-terrains.

Table 5.1 shows the structure of the different terrains used for the performance tests. Every terrain was rendered using three levels and each of the test-terrains was used for performing three different tests. During the first two tests, data was acquired without moving the camera, e.g. no level shifts occurred and therefore no data was streamed neither from the hard disk to the memory nor from the memory to the GPU. The camera's position and viewing direction remained the same for both tests. The first test was run with view frustum culling enabled while culling was disabled during the second test. With those two tests the rendering speed of the application was profiled. The third test was performed with a moving camera and view frustum culling enabled. Thus the performance impact of the streaming could be investigated.

All the tests were performed for 10 seconds and the gathered data averaged over this time period to get a representative value. Following values were calculated during one test run:

- Average frame rate
- Average shifts per frame
- Triangles rendered
- Triangles culled
- Average time needed for streaming data from the hard disk to the memory (HDD to memory).
- Average time needed for processing the data (Memory processing).
- Average time needed for streaming the data to the GPU's memory (Memory to GPU).
- Average time consumed by the rendering.

In Appendix A four different screenshots show different test terrains used for the profiling. The first two screenshots show the terrain with level size=512 whereas the last two screenshots are taken from the terrain with a level size of 1024 samples. The second and third screenshot show the density of the image samples in wire frame mode. Additionally this image visualizes the different footprints by rendering them in different colors. The $m \times m$ blocks are rendered in green, the $m \times 3$ footprints in blue and the L-shaped interior trims in red (compare Section 3.3).

5.2. Results

In this section we will present and discuss the values gathered during the performance tests. Table 5.2 shows the results of the different test runs. *Test 1* was run with fixed camera position and view frustum culling enabled. The settings of *Test 2* differed from *Test 1* by disabling view frustum culling. *Test 3* was performed during the movement of the camera with constant velocity for all test scenarios.

5.2.1. Test 1 and 2

The first two performance tests performed like we expected. With increasing level size the overall triangle count increases as well like shown in Table 5.1. The result of the increasing triangle count is heavier work during the rendering and as a result the frame rate drops exponentially like shown in Figure 5.2. View frustum culling reduces the amount of data that has to be rendered and therefore *Test 1* performed better for all test terrains than *Test 2*. This can be seen in Figure 5.1: the green timber that visualizes *Test 1* is always larger than the red one (*Test 2*).

During the tests the averaged time consumption by the different tasks during the rendering of one frame was profiled. As expected during the first two tests very little time was wasted in the first three tasks of the rendering. Because of the steady camera no level shift occurred and thus no data had to be streamed from the hard disk to the GPU's memory. As a result the time consumed by the *HDD to memory* task, *Memory processing* task and *Memory to GPU* task took less than 1.5% of the frame's total processing time. The reason for the slightly higher percentage values of those tasks during the test runs with smaller level sizes is the

	Level size	**1024**	**512**	**256**	**128**
Test 1	**Average frame rate**	72	222	540	1081
	Rendered triangles	4.706.310	1.173.510	291.594	71.887
	HDD to memory	0,7%	0,7%	0,9%	1,4%
	Memory processing	0,4%	0,5%	0,7%	1,1%
	Memory to GPU	0,0%	0,0%	0,0%	0,0%
	Rendering	98,84%	98,77%	98,38%	97,49%
Test 2	**Average frame rate**	51	169	451	1036
	Rendered triangles	6.795.278	1.693.710	420.878	103.950
	HDD to memory	0,9%	0,9%	1,1%	1,3%
	Memory processing	0,6%	0,6%	1,0%	1,0%
	Memory to GPU	0,0%	0,0%	0,0%	0,0%
	Rendering	98,50%	98,49%	97,92%	97,70%
Test 3	**Average frame rate**	8	20	462	637
	Average shifts per frame	16,9	6,9	0,3	0,2
	HDD to memory	36,7%	24,9%	30,3%	55,3%
	Memory processing	61,7%	33,0%	24,0%	9,3%
	Memory to GPU	1,4%	41,4%	27,5%	8,5%
	Rendering	0,3%	0,6%	18,3%	26,9%

Table 5.2.: Results of the performance tests.

fewer rendering load. The work in the first three tasks remains nearly constant for different level sizes and therefore they have a slightly greater impact on the percentage values for smaller level sizes.

5.2.2. Test 3

Test 3 profiled the application with a moving camera. Table 5.2 shows that during *Test 3* the different frame rates between the different test terrains result in different average shifts per frame. If the frame rate is higher, the shifts are not as much agglomerated as with a lower frame rate. Nevertheless about the same amount of shifts per time unit are performed independent from the test terrain. This can be quickly checked by taking the average frame rate into account.

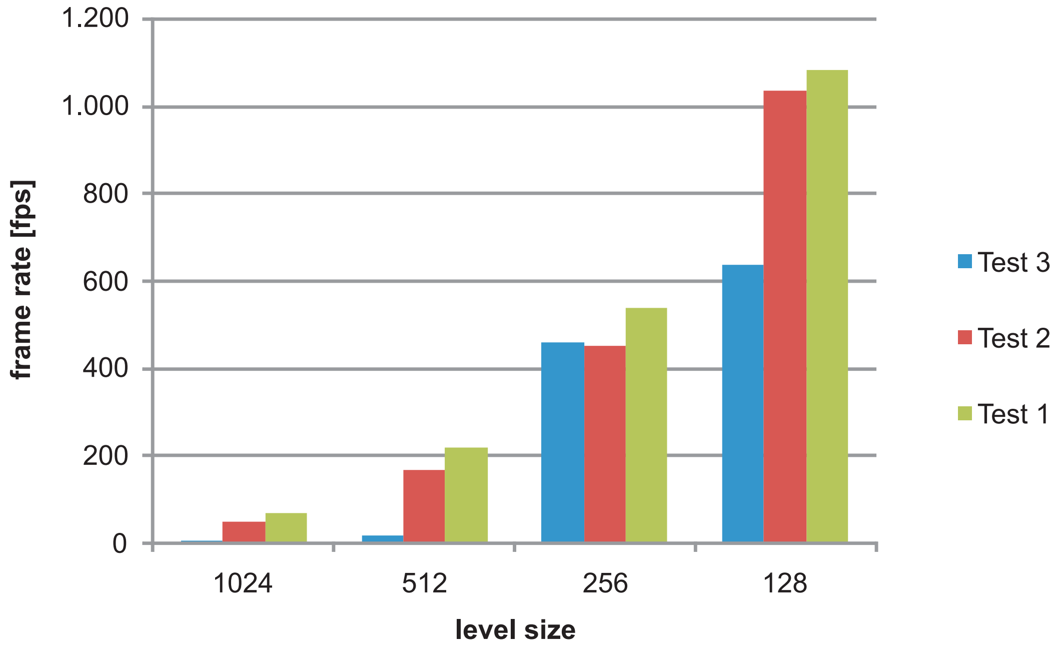


Figure 5.1.: Average frame rate during the test runs.

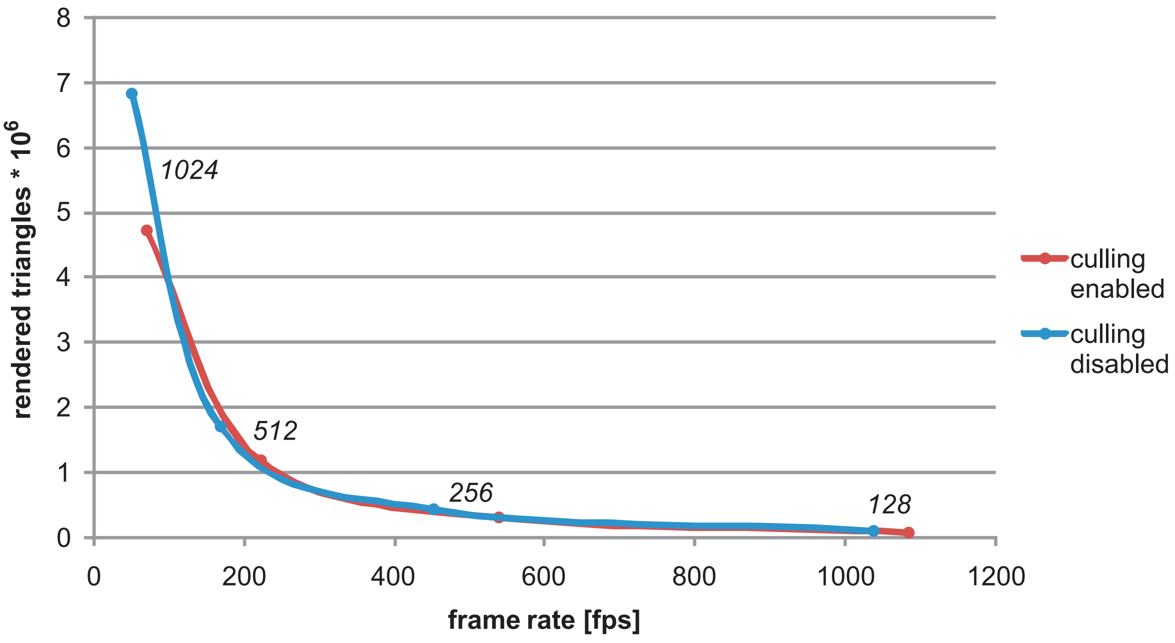


Figure 5.2.: *Test 1* and *Test 2*: number of rendered triangles compared with the resulting frame rate for the different test terrains (level sizes: 1024, 512, 256, 128).

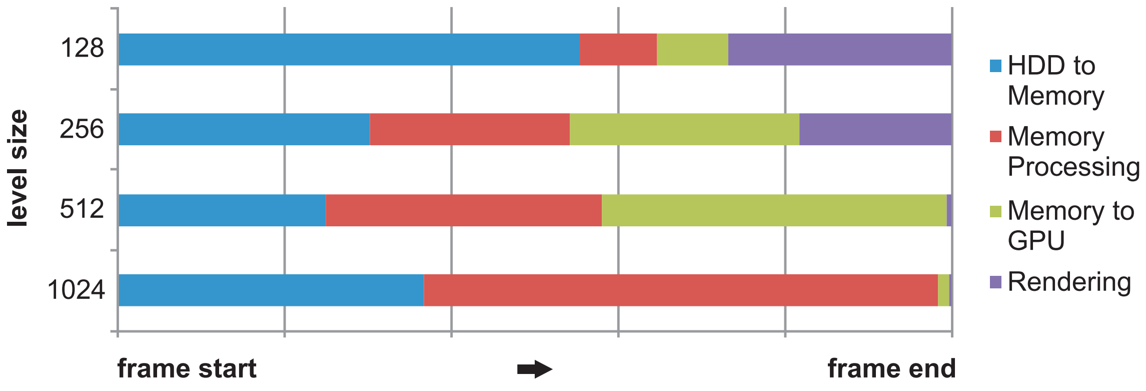


Figure 5.3.: *Test 3:* average time consumption by the different tasks during rendering.

Test 3 revealed the great weak-point of our geometry clipmap implementation. It performs not very well if the terrain data needs to be synchronized between the hard disk and the GPU's memory. As visible in Figure 5.1 the frame rate drops much, especially for larger terrains. Figure 5.3 shows the average time consumption by the rendering task during the rendering. It is visible that with increasing amount of data (e.g. increasing level sizes), the synchronization tasks consume more and more of the rendering time. Particular with a level size of 512 samples that unfortunately turned out to be an optimal level size, the uploading of the terrain data takes too long.

5.3. Analysis

The rendering of the test terrains during *Test 1* and *2* performed as expected. Very steady frame rates of over 200 fps for rendering about 1.7 million triangles is in our opinion an acceptable result and reveals the powerful capabilities of today's hardware.

However *Test 3* did not perform as good as the first two tests. Two problems were detected by us that lead to the frame rate drop during the third test run. The first problem of our implementation is the massive amount of data that has to be synchronized. Our implementation does not support a separate thread for the data upload therefore the rendering thread has to wait for the terrain data streaming to finish. If too much data has to be streamed the render process is suspended too long and the frame rate drops. This also explains why the frame rate is not steadily low but rapidly skips between higher and lower frame rates.

The second problem of our implementation is the costly memory processing of the data just before uploading to the GPU. The synchronization between the hard disk and the memory is handled by a few larger blocks that cover an area larger than the clipmap level's area. Therefore not much time is wasted by the calculations needed for the HDD to memory synchronization. In contrast to that for every level shift the exact region that is affected by the shift is updated on the GPU's memory. To get this data from the memory costly calculations have to be performed that take a considerable amount of time (compare Figure 5.3).

5.4. Problem Solution

The greatest problem of our application is the frame rate drop during level shifts because of the amount of data that has to be streamed. Currently our application does not support textures for the terrain. This feature will soon be added to the application. The resolution of such a texture is normally larger than the underlying terrain information and therefore the data that has to be synchronized between the hard disk and the GPU's memory increases more than 100%. This will have an even greater impact on the synchronization problem we are currently facing in our application.

We would like to shortly introduce three basic approaches for solving the upper problem. The following three sections briefly outline those solution ideas which will be tested and implemented in our application in the future.

5.4.1. Memory Processing Reduction

The first enhancement reduces the memory processing time by using the same technique for the uploading of the data between the system's memory and the GPU like the one used for the synchronization between the HDD and the memory. Instead of uploading the precise region for every shift, the texture on the GPU is tiled by the user defined factor c_t (compare Section 3.5.1). Upon a shift the application checks whether the level exceeds the larger texture on the GPU and if the inquiry returns true whole tiles are updated. The advantage of this approach is that the caching of the terrain data in the system's main memory can be dismissed and therefore the heavy *Memory Processing* task becomes unnecessary.

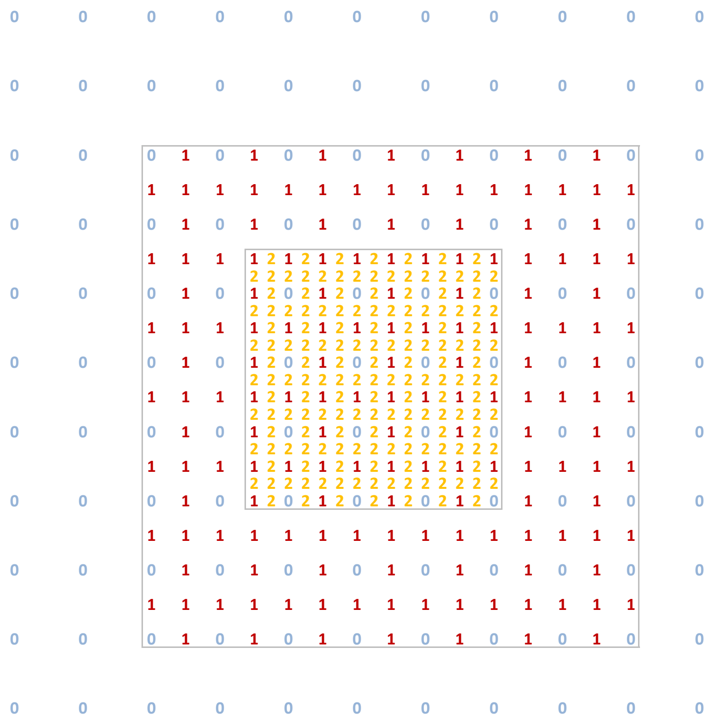


Figure 5.4.: Three levels with the affiliation of the sample points encoded by their value/color.

One disadvantage of the upper approach is an increase of the GPU's memory consumption by a factor of $(c_t+1)^2/{c_t}^2$ when c_t is the user defined tiling factor (compare Section 3.5.1). For a tiling factor of 8 the memory consumption increases by 26.6%. Another negative side effect is the loss of the optimized power-of-two side length of the GPU's textures. A way to avoid the resizing of the GPU's vertex textures is by making the level size dependent on the texture size. As a result the texture size remains the same but the level size is resized depending on the texture size and the tiling factor.

5.4.2. Data Reduction

As the amount of data is a problem in our current implementation the following enhancement technique tries to reduce it. Decreasing the amount of streaming data directly reduces the work load and increases the frame rate.

We came up with two possibilities for data-reduction. The first way is to reduce redundant

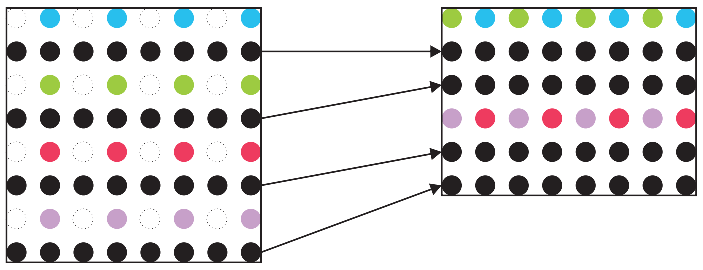


Figure 5.5.: Repacking of the samples after removing the redundant samples.

data. Figure 5.4 shows three levels positioned within each other. Each number represents an image sample and the value of the number corresponds with the level to which it belongs. The image shows that within finer levels certain image samples can be retrieved from coarser levels. Generally for every level that has a coarser level, only three quarter of the samples within the $n \times n$ level window have to be stored and transferred to the GPU. The missing sample points can be retrieved in the vertex shader by sampling from the coarser levels. The drawback of this data reduction is a far more complex vertex shader because the direct correspondence of the relative position between the image samples and the structure of the geometry is lost as the image has to be repacked in order to take advantage of the removal of the redundant terrain image samples (compare Figure 5.5).

The second way for reducing the uploading data is to compress the terrain images. As stated by Wang *et al.* [WWS^{+}07] commodity graphics hardware today does not provide any native compression for HDR textures. In their work they present a method that uses a pair of 8-bit DXT textures for storing a floating point texture. The DXT format, also called *S3 Texture Compression* (S3TC), is a group of related image compression algorithms that are supported on modern graphics hardware. The compression rate achieved by the algorithm of Wang *et al.* [WWS^{+}07] is 3:1 for 16-bit floating point inputs. This method can be used to compress the terrain image tiles into a set of DXT textures and therefore benefit from the reduced data size during the uploading. The drawback of this technique is the lossy characteristic of the DXT texture compression. For HDR images it is often accepted that the data slightly changes because of the lossy texture compression – it is to be worked out whether lossy compression can be acceptable for terrain rendering too.

One condition of both reduction methods presented above is that the terrain samples are loaded in tiles directly from the hard disk to the GPU's memory. Both the removal of redundant data as well as the compression does not allow intermediate processing of the tiles in the system's main memory.

5.4.3. Multi-threading

Often not all of the data can be streamed in one frame without losing an acceptable frame rate. Therefore a multi-threaded out-of-core rendering technique has to be used to overcome this problem. This performance enhancing technique is probably the most import solution for the frame rate dropping problem because even if the two other approaches fulfill what they are promising, the uploading of data can be hindered by effects like multiple concurrent disk accesses by other applications. Using such a multi-threaded technique the rendering process is separated from the uploading task and therefore the rendering thread is prevented from being slowed down by the uploading task. Upon detection of the need to shift levels, the uploading thread is asked to the work. The render engine continues drawing the scene without using the levels that are affected by the shift. The uploading task in the meanwhile uploads the data from coarse to fine to ensure that coarser level are sooner updated and therefore the whole terrain can be rendered even though in poorer quality.

The uploading of the data can not totally be separated from the rendering task. The synchronization from the system's and the GPU's memory has to be phased with the rendering because only single threaded access to the graphics hardware is allowed.

Chapter 6.

Conclusion and Outlook

In the first section of this chapter we will summarize the purpose and the novelties of this thesis. Afterwards an outlook on possible future work is provided.

6.1. Conclusion

The purpose of this work is to present a technique for rendering terrain that resides on an arbitrary basis. This technique was implemented to be able to display large curved surfaces like the earth. Our first task was to search for existing technology. In Chapter 2 we presented the results of our literature search. Next to describing the individual technologies we outlined the advantages and disadvantages. The algorithm that attracted us the most is the GPU based geometry clipmap algorithm by Asirvatham and Hoppe [AH05]. Unfortunately it lacks of the ability to render large arbitrary curved surfaces. Their implementation only allows to render terrain that resides on a flat basis. We extended the algorithm of Asirvatham and Hoppe to overcome this planar basis restriction.

In Chapter 3 we have presented the theory behind our novel geometry clipmap implementation. Unlike the original GPU based implementation by Asirvatham and Hoppe [AH05] our implementation supports the rendering of large curved surfaces like the earth.

Same as Asirvatham and Hoppe [AH05] we store the terrain in vertex textures. In the original implementation the authors use 2D elevation images, prefiltered into a mipmap pyramid of L levels. Their application caches square windows of $n \times n$ samples for every level. Those windows are used during the rendering to elevate regular grid rings centered about the viewer and thus create the terrain. Instead of using 2D elevation images like described above, our implementation stores during a preprocessing step all three coordinates of the terrain samples in three of the four channels of a RGBA floating point texture. The information about the sample's position within the whole texture is only used to connect the points to triangles. This technique is the main difference to the algorithm presented by Asirvatham and Hoppe [AH05] and enables our implementation to render terrain that resides on arbitrary basis because the created mesh is not bound by a regular grid mesh that can only be altered in z-direction.

The drawback of changing the original geometry clipmap algorithm from the use of heightmaps to textures holding 3D information is the introduction of a precision problem that restricts the algorithm to render only limited amount of area. Our solution to overcome this restriction is to split up the surface into multiple individual patches. Each one of this patches can have the maximum extent of a floating point texture. During the rendering process the algorithm checks every frame whether the border of one terrain patch is reached. In case the inquiry returns true the algorithm is instantiated a second time with a camera position referenced within the coordinate system of the new patch. Now the algorithm is able to rendering the missing parts of the scene.

6.2. Outlook

Our implementation proofed the concept of the new geometry clipmap algorithm. We are now able to render large curved surfaces. Unfortunately the current implementation has problems during the update of the terrain data. The current single threaded nature of our applications is not able to handle the vast amount of data. In Section 5.4 we already pointed out solutions for this problem. Those solution approaches need to be worked out in more detail to be able to be implemented in future versions of our geometry clipmap implementation.

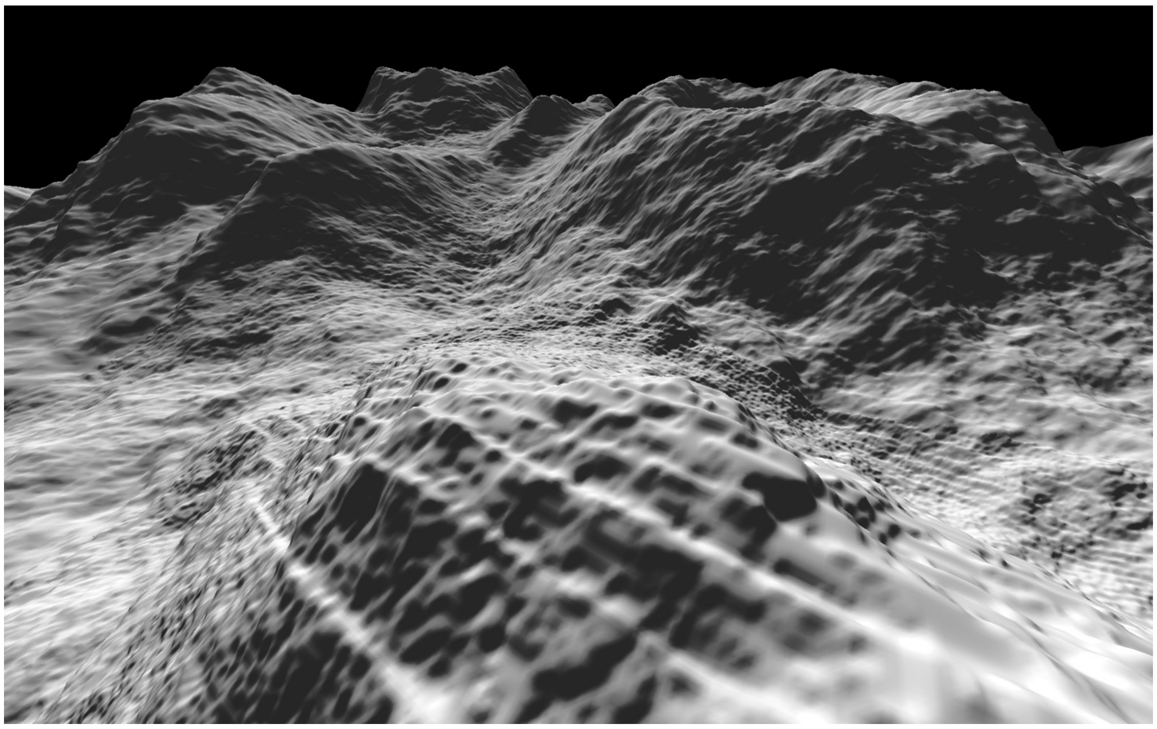
Currently our implementation does not support textures. Usually combined with raw terrain data, textures are available. With the use of such textures the rendered surface can be greatly valorized. For simplifing the development of the current implementation we resigned textures but in future implementations texture support is to be included.

One purpose of our development is to be able render the earth at any scale. Commonly points on the earth are not identified by x, y and z coordinates but embedded in a geodesic coordinate system. Therefore in future implementations we would like to reference the user's position on the surface of the rendered earth with geodesic coordinates.

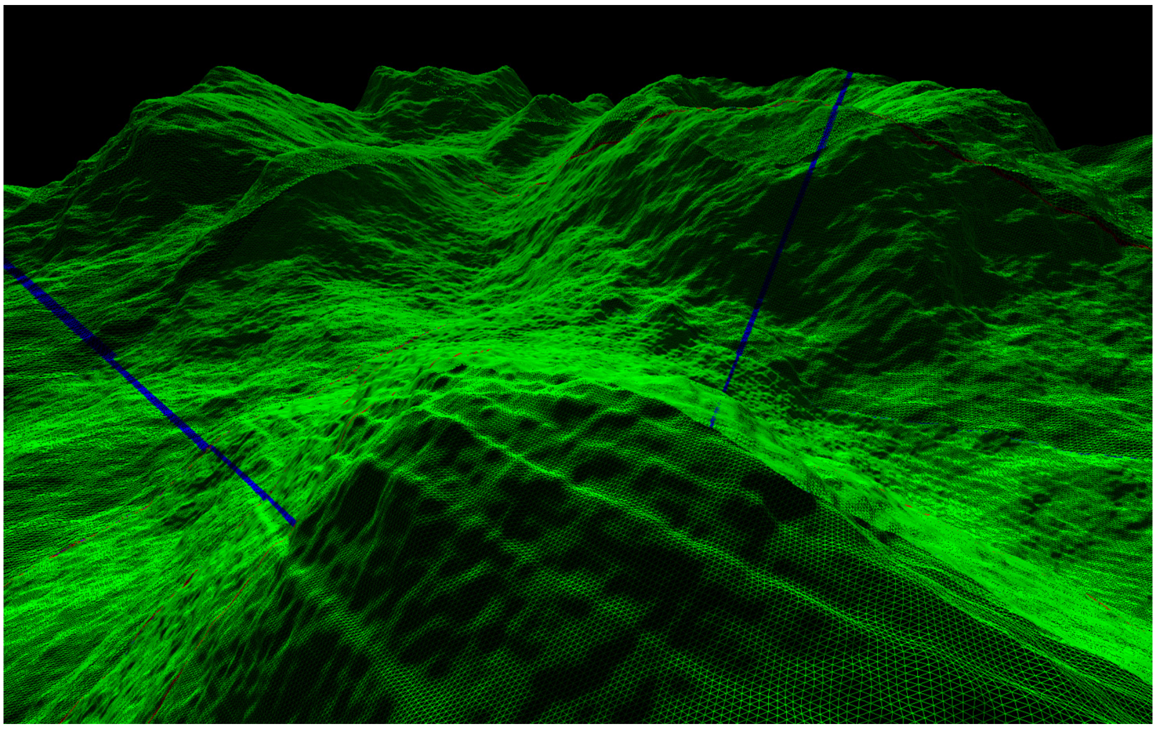
Appendix A.

Screenshots

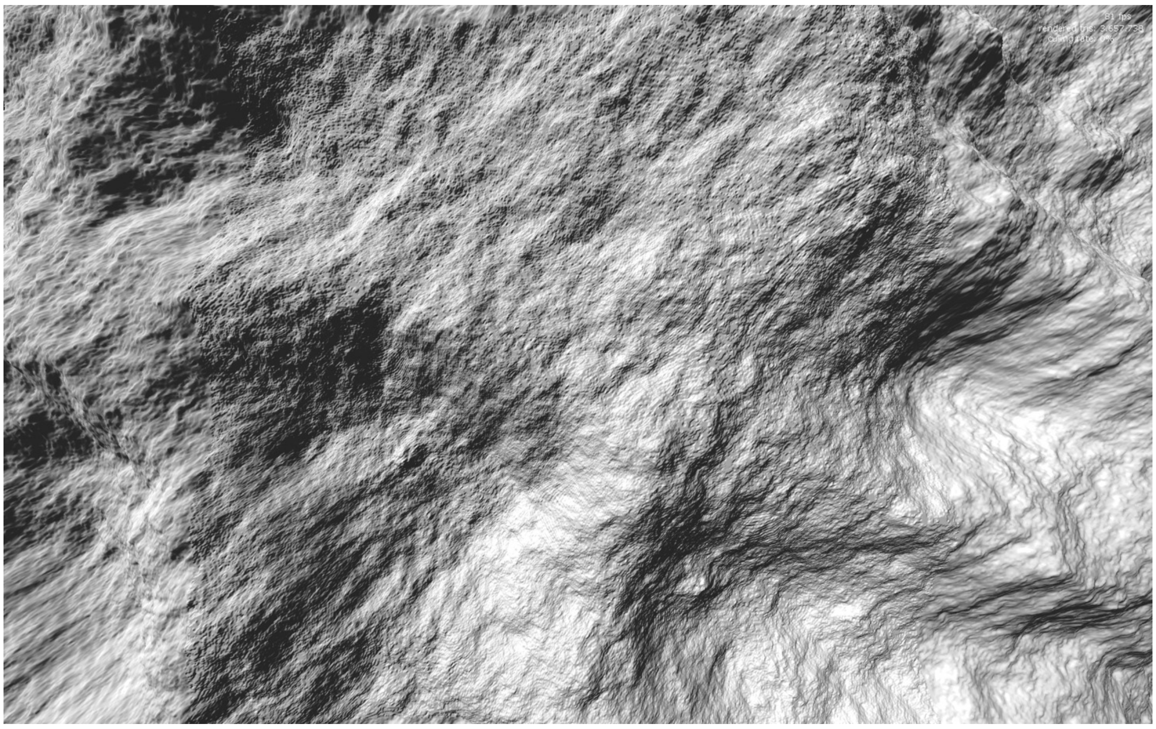
Screenshot of a test terrain, rendered in solid mode.



Screenshot of a test terrain, rendered in block-colored wire frame mode.

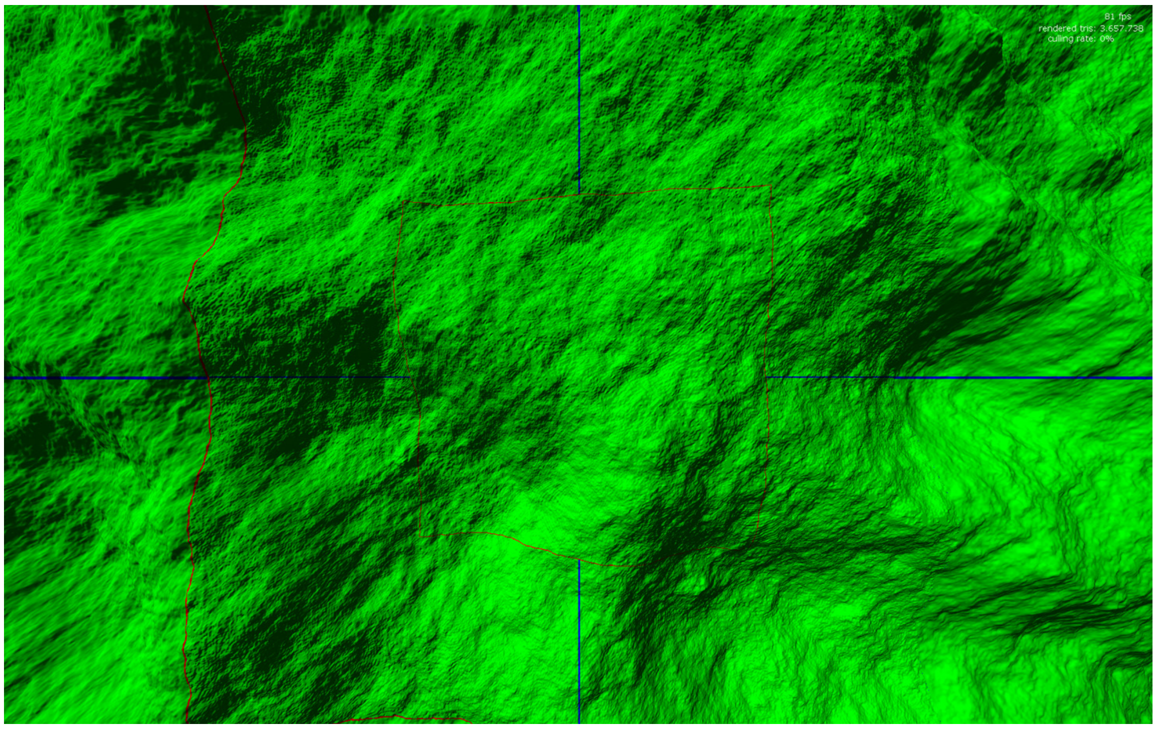


Screenshot of a test terrain in top down view, rendered in solid mode.



81 fps
rendered tris: 3,657,738
culling rate: 0%

Screenshot of a test terrain in top down view, rendered in block-colored mode.



```
81 fps
rendered tris: 3.657.738
culling rate: 0%
```

Appendix B.

Acronyms

AABB	Axis aligned bounding box
CPU	Central processing unit
DAG	Direct acyclic graph
FPS	Frames per second
GPU	Graphics processing unit
HDD	Hard disk
HDR	High dynamic range
IEEE	Institute of Electrical and Electronics Engineers
LOD	Level of detail
OS	Operating system
RAM	Random access memory
ROAM	Real-time optimally adapting meshes
TIN	Triangular irregular meshes
VDPM	View-dependent progressive mesh
VSync	Vertical synchronization

Appendix C.

Bibliography

[AH05] Arul Asirvatham and Hugues Hoppe. Terrain rendering using GPU-based geometry clipmaps. In M. Pharr and R. Fernando, editors, *GPU Gems 2*, chapter 2, pages 27–46. Addison Wesley, March 2005.

[AM00] Ulf Assarsson and Tomas Möller. Optimized view frustum culling algorithms for bounding boxes. *J. Graph. Tools*, 5(1):9–22, 2000.

[Bra03] Derek Bradley. Evaluation of real-time continuous terrain level of detail algorithms. Technical report, Carleton University, 8 August 2003.

[Bre05] Nick Brettell. Terrain rendering using geometry clipmaps. Technical report, University of Canterbury, Department of Computer Science, 2005.

[CH06] Malte Clasen and Hans-Christian Hege. Terrain rendering using spherical clipmaps. In B.S. Santos, T. Ertl, and K. Joy, editors, *EUROVIS-Eurographics/IEEE VGTC Symposium on Visualization*, pages 91–98, Lisbon, Portugal, 2006. Eurographics Association.

[DWS$^+$97] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. ROAMing Terrain: Real-time Optimally Adapting Meshes. In *Proceedings of the conference on Visualization '97*, pages 81–88. ACM Press, 1997.

[GFG04] G. Gerasimov, F. Fernando, and S. Green. Shader model 3.0 using vertex textures. *White Paper*, 2004.

[GHB+05] K. M. Górski, E. Hivon, A. J. Banday, B. D. Wandelt, F. K. Hansen, M. Reinecke, and M. Bartelmann. Healpix: A framework for high-resolution discretization and fast analysis of data distributed on the sphere. *The Astrophysical Journal*, 622(2):759–771, April 2005.

[Hop96] Hugues Hoppe. Progressive meshes. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 99–108, New York, NY, USA, 1996. ACM.

[Hop97] Hugues Hoppe. View-dependent refinement of progressive meshes. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 189–198, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[Hop98] Hugues Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 35–42, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[Hou04] Christopher Hoult. Real-time optimally adapting meshes - an overview, 10 May 2004.

[Kon07] Jens Konerow. *Managed DirectX und C#. Einstieg und professioneller Einsatz.* entwickler.press, 2007.

[LH04] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 769–776, New York, NY, USA, 2004. ACM.

[LKR+96] Peter Lindstrom, David Koller, William Ribarsky, Larry F. Hughes, Nick Faust, and Gregory Turner. Real-time, continuous level of detail rendering of height fields. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 109–118, New Orleans, Louisiana, August 1996. ACM SIGGRAPH / Addison Wesley. ISBN 0-201-94800-1.

[LP01] Peter Lindstrom and Valerio Pascucci. Visualization of large terrains made easy. In *IEEE Conference on Visualization*, pages 363–370, 2001.

[LP02] Peter Lindstrom and Valerio Pascucci. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):239–254, 2002.

[OMG07] OMG. *Unified Modeling Language, Superstructure, V2.1.2*. Object Modeling Group, November 2007.

[TMJ98] Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. The clipmap: a virtual mipmap. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 151–158, New York, NY, USA, 1998. ACM.

[WBB$^+$] Rolf Werninghaus, Wolfgang Balzer, Stefan Buckreuss, Josef Mittermayer, Peter Mühlbauer, and Wolfgang Pitz. The TerraSAR-X Mission. Technical report, Deutsches Zentrum für Luft- und Raumfahrt (DLR) / EADS Astrium GmbH. http://wwwserv2.go.t-systems-sfr.com/tsx/documentation/EUSAR-TX-Mission.pdf.

[Wik07] Wikipedia. Phong-Beleuchtungsmodell – Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 4. April 2008] http://de.wikipedia.org/wiki/Phong-Beleuchtungsmodell.

[Wil83] Lance Williams. Pyramidal parametrics. In *SIGGRAPH '83: Proceedings of the 10th annual conference on Computer graphics and interactive techniques*, pages 1–11, New York, NY, USA, 1983. ACM.

[WWS$^+$07] Lvdi Wang, Xi Wang, Peter-Pike Sloan, Li-Yi Wei, Xin Tong, and Baining Guo. Rendering from compressed high dynamic range textures on programmable graphics hardware. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 17–24, New York, NY, USA, 2007. ACM.