UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

Handling Massive Terrains Based On Digital Elevation Data

Trabalho de Graduação

Alex Thomas Almeida Frasson

Santa Maria, RS, Brasil
2015

Handling Massive Terrains Based On Digital Elevation Data

Alex Thomas Almeida Frasson

Trabalho de Graduação apresentado ao Curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de

Bacharel em Ciência da Computação

Orientador: Prof. Dr. Cesar Tadeu Pozzer (UFSM)

Trabalho de Graduação N. 405

Santa Maria, RS, Brasil

2015

Universidade Federal de Santa Maria

Centro de Tecnologia

Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada,

aprova o Trabalho de Graduação

Handling Massive Terrains Based On Digital Elevation Data

elaborado por

Alex Thomas Almeida Frasson

como requisito parcial para obtenção do grau de

Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

[illegible]

Prof. Dr. Cesar Tadeu Pozzer (UFSM)

(Orientador)

[illegible]

Prof. Dr. Benhur de Oliveira Stein (UFSM)

[illegible]

Prof. Dr. Mateus Beck Rutzig (UFSM)

Santa Maria, 09 de Dezembro de 2015.

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

Handling Massive Terrains Based On Digital Elevation Data

Autor: Alex Thomas Almeida Frasson
Orientador: Prof. Dr. Cesar Tadeu Pozzer (UFSM)
Local e data da defesa: Santa Maria, 9 de Dezembro de 2015.

Terrain rendering is the area of computer graphics that deals with the aspects of visualizing landscapes on computers. With the advancement of hardware and precision of data acquisition and current diversity of fields in which it is used, terrain visualization remains an interesting topic of research. However, interactive visualization of very large-scale terrain data brings up a multitude of problems. Therefore, this project proposes the development of an architecture for rendering large-scale terrains at interactive frame rates. This architecture should then be used on a application that is already being developed at LaCA-UFSM(Laboratório de Computação Aplicada).

KEY-WORDS: GIS; large-scale terrains; simulators; real-time rendering.

LISTA DE ABREVIATURAS E SIGLAS

OpenGL	Open Graphics Library
LOD	Level of detail
CDLOD	Continuous Distance-Dependent Level of Detail
TIN	Triangulated Irregular Networks
GIS	Geographic Information System
SSIM	Structural Similarity Index
GLSL	OpenGL Shading Language

1. INTRODUCTION

Terrain rendering is the area of computer graphics that deals with the aspects of visualizing landscapes on computers. With the advances in hardware and precision of data acquisition and current diversity of fields in which it is used, terrain visualization remains an interesting topic of research. As described by [4], it is used in visualization by construction engineers or architects when constructing a visual presentation of their designs as well as by special effects makers for making virtual environments for cinematic films. But perhaps the field that can take more advantage of terrain visualization is the field of simulations. Flight simulations, given that aircrafts can traverse long distances in short periods of time, naturally need vast amounts of landscapes. Moreover, military training simulations, in pursuit of realistically simulate the battlefield, also have great interest on large-scale terrains.

However, interactive visualization of very large-scale terrain data brings up a multitude of problems. The main problem in real-time graphics is rendering efficiency. Scene complexity must be reduced as much as possible without leading to an inferior visual quality. One way to increase efficiency is the use of different levels of detail (LODs) for different areas of the visible scene. Large-scale terrains are usually too large to be displayed as a whole, thus only a fraction of such large data set can be rendered in real-time. The fraction that is being rendered is given by the changes of the view parameters. Therefore, the data structure holding the terrain data must support spatial access.

How large terrain a terrain system should be able to support depends on the application. Tools such as Google Earth and NASA World Wind let us explore every corner of the planet Earth. Being a visualization only tool makes life way easier for the developers of such tools, since the focus is not on ground level detail. On the other hand applications such as simulations, especially the ones that get to the ground level e.g., military armored vehicle simulation, require a more detailed terrain. For instance, the simulation training solution Virtual Battlespace 3 (VBS3) is capable of rendering a 2200 km by 2200 km area. This shows how application dependent the terrain system is.

1.1. Goals

1.1.1. General Goals

This project proposes the development of an architecture for rendering large-scale terrains at interactive frame rates. For that, as describe in [5], the following features should be considered throughout the development of this project: a multiresolution terrain model that supports continuous adaptive LOD triangulation; effective storage model and data compression for reduced memory consumption and simple and direct digital elevation data access for other non-rendering usage scenarios.

1.1.2. Specific Goals

It is defined as specific goals of development:

- Investigate LOD techniques that can be efficient while being able to accurately visualize terrain data.
- Use the C++ programming language together the graphics library OpenGL to implement the chosen LOD techniques without considering the out-of-core scenario.
- Compare and analyze the obtained results with the present literature.
- Address the floating point precision problem.

2. FOUNDATIONS AND LITERATURE REVIEW

In this chapter a description of terrain visualization basic concepts together with a brief introduction to some of the tools used throughout the development of this project are given. Furthermore, all the most significant related works are also going to be analysed and explained in the following sections.

2.1. Data Format

Terrain data come from a wide array of sources. Real-world terrain used in virtual globe and GIS applications is typically created using remote sensing aircraft and satellites. Artificial terrains for games are commonly created by artists, or even procedurally generated in code. Depending on the source and ultimate use, terrain may be represented in different ways.

2.1.1. Heightmap

In terrain visualization a heightmap, which is a raster image, is interpreted as map for the surface elevation. It can be thought of as a grayscale image where the intensity of each pixel represents the height at that position. Typically, black indicates the minimum height and white indicates the maximum height. Because of its simplicity and the large amount of tools available for modifying and generating this format, it is the most widely used terrain representation. Since only one height per xy-position is stored, a single height map cannot represent terrain features such as vertical cliffs, overhangs, caves, tunnels, and arches.

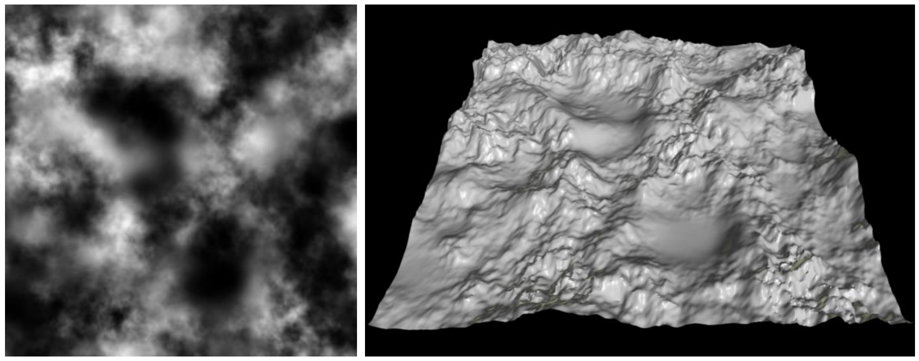


Figure 1.1 - A heightmap (left) and its rendered 3D mesh (right).

2.1.2. Triangulated Irregular Networks

A triangulated irregular network (TIN) is basically a triangle mesh. TINs are formed by triangulating a point cloud to create a watertight mesh. An advantage of using a TIN over a raster is that the points of a TIN are distributed variably based on an algorithm that determines which points are most necessary to an accurate representation of the terrain. That give us a nonuniform sampling: large triangles can cover flat regions and small triangles can represent fine features. Given its triangle-mesh nature, a TIN representation allows for vertical and overhung features.

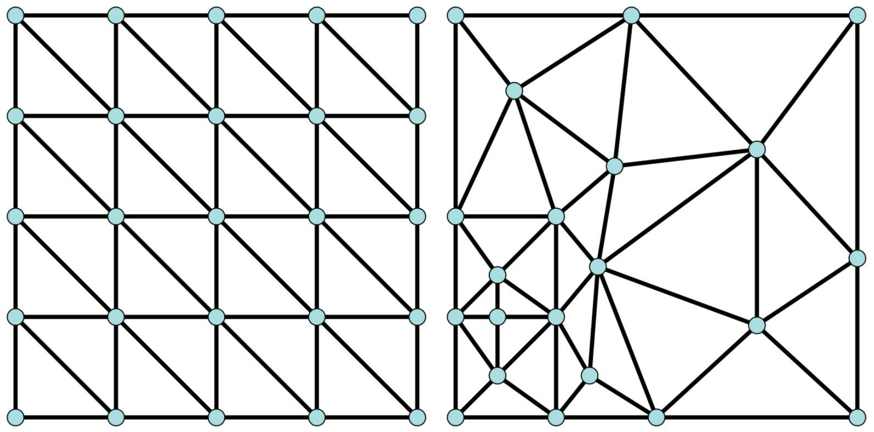


Figure 1.1 - Uniform grid generated from a heightmap (left) and the non-uniform representation of the TIN (right). From the right image we can deduce that the lower-left corner is less flat than the rest of the terrain.

2.2. Out-of-core

Out-of-core or external memory algorithms are algorithms that are designed to process data that is too large to fit into a computer's main memory at one time. Such algorithms must be optimized to efficiently fetch and access data stored in slow bulk memory such as hard drives. Data sets in large applications are often too massive to fit completely inside the computer’s internal memory. The resulting input/output communication (or I/O) between fast internal memory and slower external memory (such as disks) can be a major performance bottleneck.

2.3. Level of detail (LOD)

When an object is far away from the viewer there is no need to render it with its full complexity, i.e. full triangle count. One can decrease the complexity of an object according to some metric e.g, the distance from the viewer, in order to better utilize the processing power available. This way less processing power is needed and also the viewer will not notice any quality change. In computer graphics, this technique is called level of detail (LOD). Several other metrics can utilized such as the distance of the object from the camera, the speed of the camera, the size of the object etc. Usually the LOD levels of an object are precomputed and the appropriate level of detail is selected in run time, based on the metrics we mentioned above.

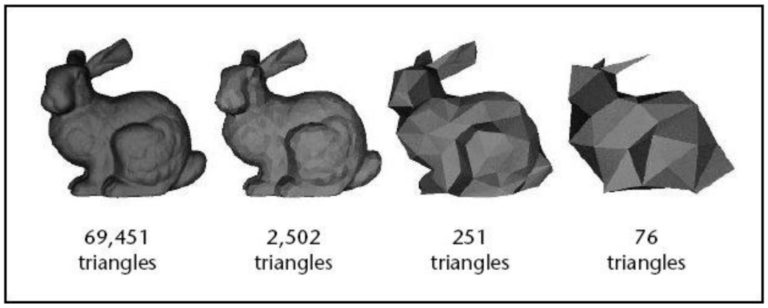


Figure 1.3 - Example of discrete LOD.

2.3.1. Pixel Error

If a LOD level is changed too soon visual errors will become visible. One way of determining if a we should switch between LOD levels without losing visual quality is by using pixel error as a metric. Pixel error is is computed after projecting the geometry to the screen and is given by the difference between the high quality LOD level geometry and the low quality LOD level geometry. In terrain rendering it is usually calculated per vertex or per patch.

2.3.2. Popping

Abruptly switching between LOD levels can generate an undesirable visual effect called popping. While the viewer gets closer to an object, the LOD algorithm will switch between two LOD levels, causing it to pop as it becomes suddenly more detailed. This is a problem that must be addressed while using discrete LOD approaches. On a continuous LOD algorithm it will become less noticeable or even inexistent.

2.3.3. Mipmap

Mipmaps are pre-calculated, optimized sequences of textures, each of which is a progressively lower resolution representation of the same image. The height and width of each image, or level, in the mipmap is a power of two smaller than the previous level. A high-resolution mipmap image is used for high density samples, such as for objects close to the camera. Lower-resolution images are used as the object appears farther away.

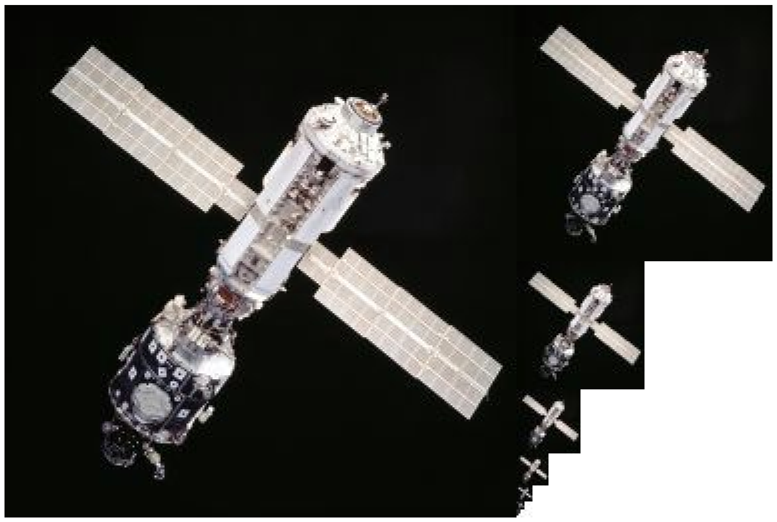


Figure 1.4 - Example of mipmap.

2.4. Culling

In many scenes, only a small portion of the triangles composing the scene are actually visible. The rest are invisible, either because they are hidden behind other triangles or because they are outside of the field of view. Culling reduces the amount of detail that needs to be rendered by eliminating these triangles that don't contribute to the scene.

2.4.1. View-Frustum culling

The view frustum is the volume that contains everything that is potentially visible on the screen. This volume is defined according to the camera's settings and, when using a perspective projection, takes the shape of a truncated pyramid. Everything that will potentially be visible on the screen is inside, at least partially, the truncated pyramid. There is no need to render what is outside the frustum since

With that in mind, we can test the object against the six planes of the view frustum before rendering it. If the object is inside or intersects, it is rendered; otherwise, it is discarded. Since the test happens on the CPU, the object is not processed at all by the GPU if it is not visible.

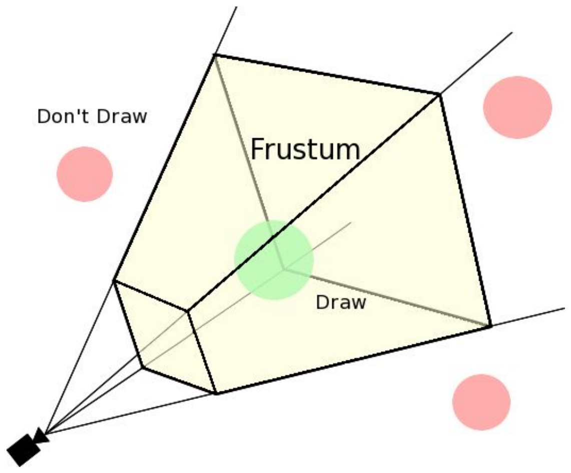


Figure 1.5 - Example of view-frustum culling. The three red spheres don't need to be rendered.

2.5. Structural Similarity Index

The structural similarity (SSIM) index is an method for predicting the perceived quality of digital pictures as well as videos. It is used for measuring the similarity between two images where the measurement or prediction of image quality is based on an initial uncompressed or distortion-free image as reference. The resultant SSIM index is a decimal value between -1 and 1, where value 1 is only reachable in the case of two identical sets of data i.e., the closer to 1, the higher the quality. It can be calculated for an entire image as well as for a single pixel, which allows to create an index map which, when converted to an image, shows where are the differences between the reference and the tested picture. It is used on this project in order to compare the visual quality of the implemented LOD techniques.

2.6. OpenGL Pipeline

The API OpenGL was used throughout the development of this project to handle the project's computer graphics needs. OpenGL is graphics API for rendering 2D and 3D graphics. OpenGL exposes features of the graphics hardware as a set of functions which may be called by the client program, alongside a set of named integer constants. Together with GLSL (OpenGL Shading Language) it allows the graphics hardware to be programed by exposing several stages of its pipeline to the programmer.

2.6.1. Pipeline

Its pipeline is constituted of 5 programmable stages. The first is the vertex shader where each vertex is processed by it and turned into an output vertex. After it there are 3 optional stages called tessellation control shader (TCS), tessellation evaluation shader (TES) and the geometry shader. The TCS and TES stage expose to the programmer the tessellation capabilities of the graphics hardware. The tessellation uses hardware acceleration to tessellate a primitive into more primitives at runtime, allowing to render more primitives than the number of previously specified primitives. The geometry shader has a similar objective, since it also generates more primitives for each input primitive, but it used for different scenarios when compared to the tessellation stage.

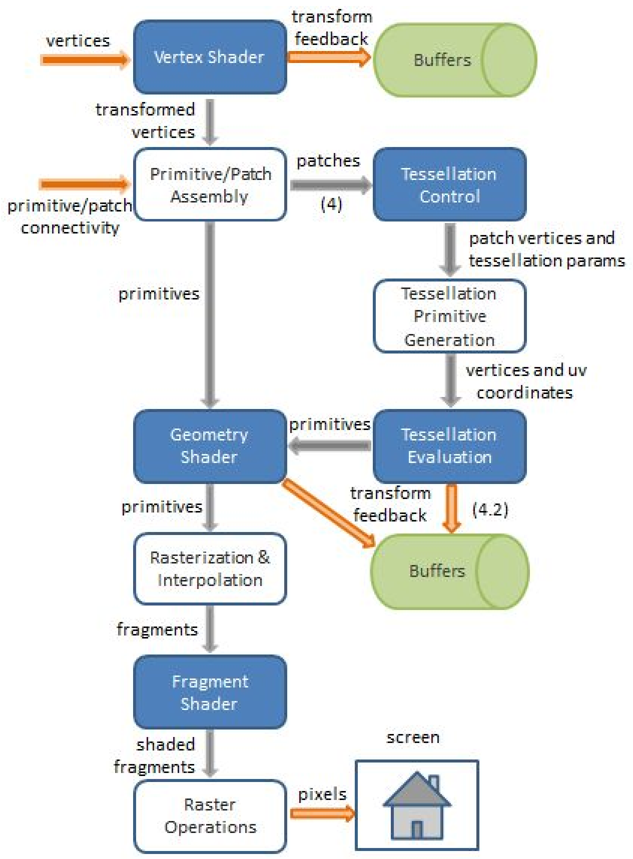


Figure 1.6 - OpenGL 4.2 pipeline.

After the graphics hardware rasterized all primitives into fragments. These fragments are then processed one by one by the last stage in the pipeline, the fragment stage. It is generally used for the shading process.

2.7. LITERATURE REVIEW

2.7.1. Level of Detail

In this chapter will be presented concepts and related work to level of detail (LOD).

2.7.1.1. Chunked LOD

Chunked LOD was proposed by Urich in siggraph 2002 [8]. It is a hierarchical LOD system that breaks an entire terrain into a quadtree of tiles, called chunks. The root chunk is a low-detail representation of the entire world. The four child chunks of the root evenly divide the world into four equal-sized areas and provide a higher-detail representation. Each of those chunks then has four child chunks itself, which further divide the world. Each node in the quadtree is generated in a preprocessing step by simplifying a subset of the overall terrain mesh to achieve a specific level of geometric error.

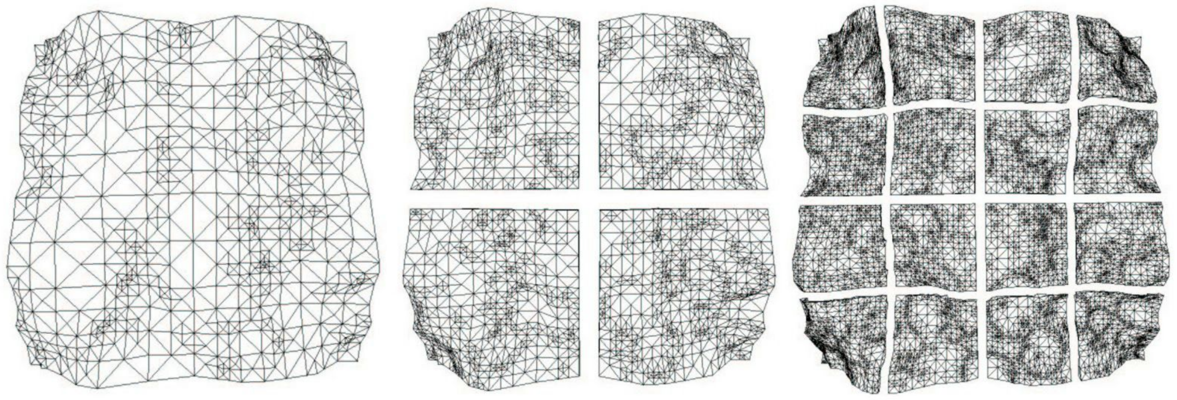


Figure 2.1 - Example showing different levels of the quadtree used in Chunked LOD.

At runtime, chunks are selected for rendering by projecting their geometric error to screen space to compute the screen-space error, in pixels, of the chunk. If the error is too high, its children are visited instead. When two chunks of different LODs are adjacent to each other, their edge vertices will not necessarily coincide, leading to cracks between the chunks. A particular concern is how to fill these cracks so that the mesh appears seamless.

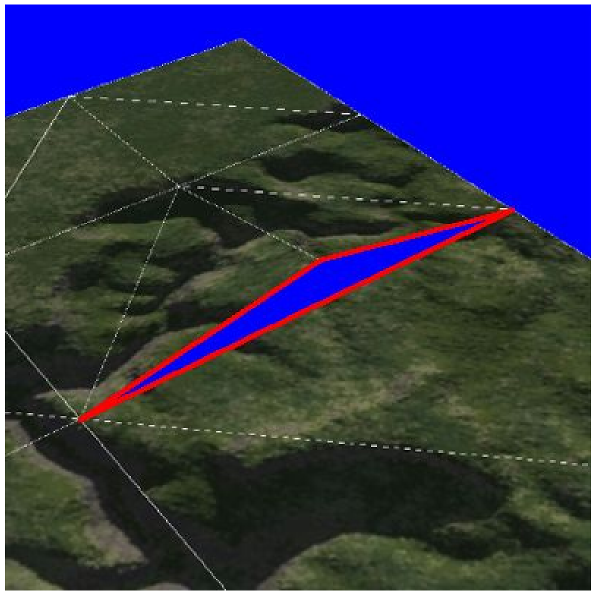
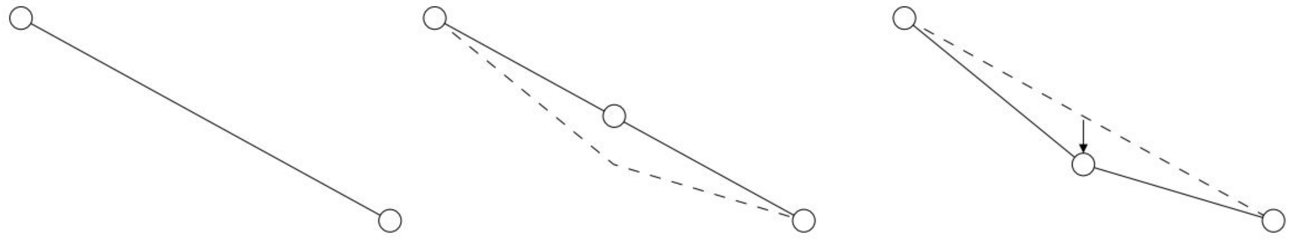


Figure 2.2 - Visible terrain crack generated from the difference of detail between two chunks.

When moving in toward a chunk, there comes a moment where the error estimate for the chunk gets too high, and at that moment, the chunk will subdivide into its children. This is likely to create an obvious and objectionable popping artifact. Chunked LOD addresses this problem by gradually morphing between the levels of detail.



Morphing of a coarser chunk (left) into a more detailed one (right) used in order to avoid popping.

Its drawbacks are the lengthy pre-processing step involved, inability to modify terrain data in real time, and a somewhat inflexible and less correct LOD system since it uses an LOD function that is the same over the whole chunk, providing only approximate three-dimensional distance-based LOD. Another drawback of the techniques of Chunked LOD is that the discontinuities between LOD levels require additional work to remove gaps and provide smooth transitions. The solution for the cracks proposed by Ulrich [8] is to use a skirt that starts at the perimeter of the chunk and extends down below it.

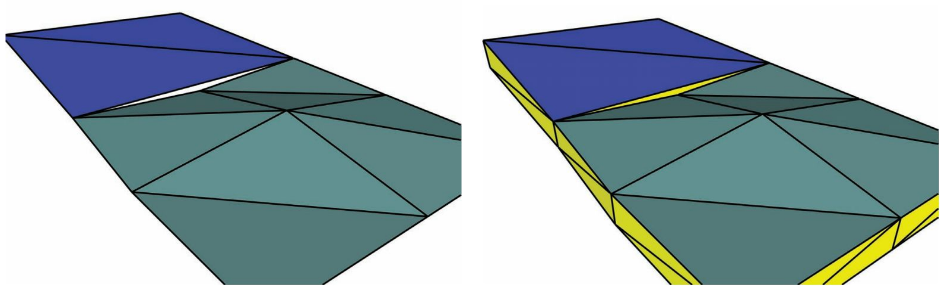


Figure 2.3 - Visible terrain crack (left) and the skirts filling it (right).

2.7.1.2. Geometry Clipmapping

Geometry clipmapping is a terrain LOD technique in which a series of nested, regular grids of terrain geometry are cached on the GPU. Each of the grids is centered around the viewer and is incrementally updated with new data as the viewer moves. The grid levels form concentric squares.

Geometry clipmapping renders rasterized elevation data in the form of a height map. Prior to rendering, the height map is prefiltered into a mipmap pyramid. Each concentric square, called a clipmap level, corresponds to a level in the mipmap. Clipmap levels each have a texture in GPU memory that holds the subset of heights in the corresponding mip level that is closest to the viewer.

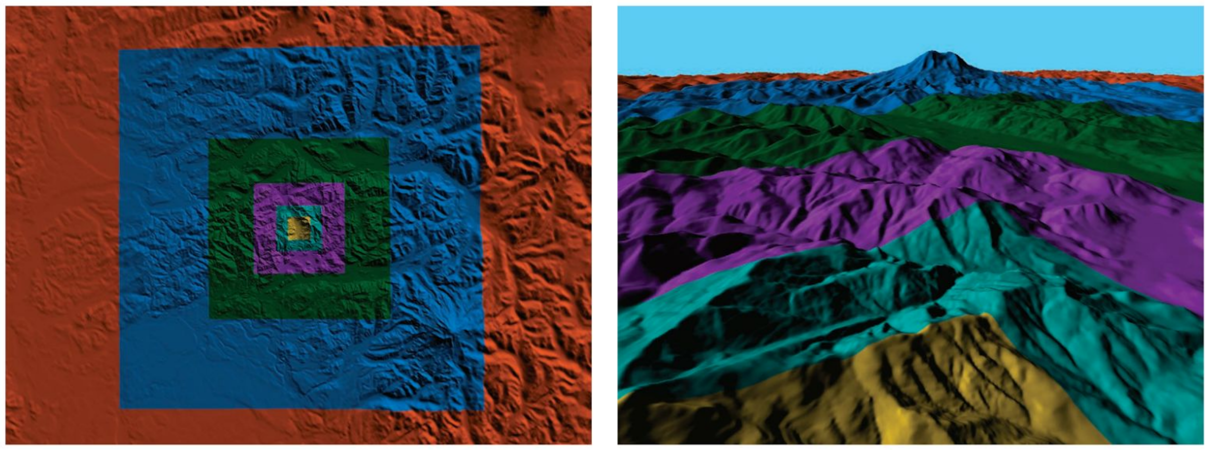


Figure 2.4 - Two images showing clipmap levels. Notice how each successive clipmap level covers four times more area than the previous level.

The innermost clipmap level closest to the viewer corresponds to the most detailed mip level of the height map, and successive levels far away from the viewer are increasingly less-detailed mip levels. Each clipmap level is square, and all levels have exactly the same number of heights, while covering four times the area of the next-finer level. The aim is to maintain triangles that are uniformly sized in screen space. As the viewer moves, the clipmap levels are updated such that the clipmap pyramid remains centered on the viewer.

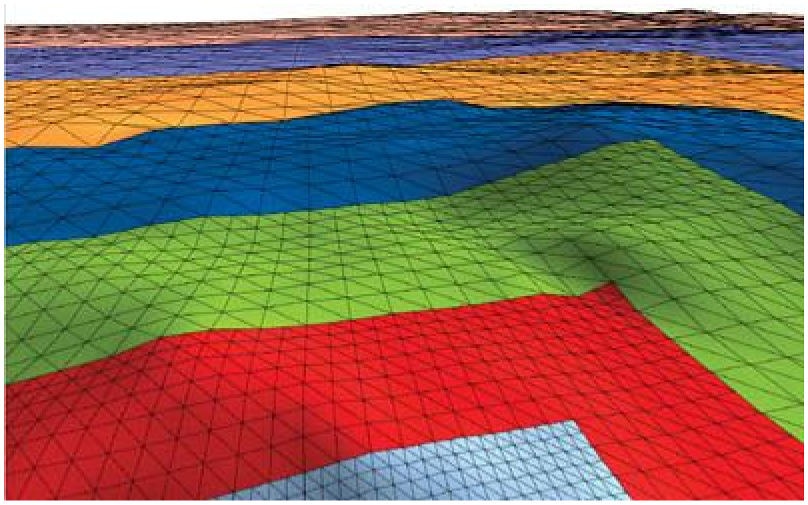


Figure 2.5 - Each successive clipmap level covers four times more area than the previous level but the number of triangles remain the same.

One of geometry clipmap drawbacks is that it uses more triangles to attain a similar visual quality than do other terrain algorithms, such as the chunked LOD. The error in screen space is a function of the terrain data itself and cannot be directly controlled by the algorithm because it aims to producing triangles that are approximately the same size in screen space at each level, however, if the terrain has steep slopes, the triangles can be stretched vertically to arbitrary size.

However, geometry clipmapping has several strong features such as transitions between levels of detail being seamless and easily implemented in shader programs; the frame rate remains relatively steady since it produces triangles that are approximately the same size in screen space; no preprocessing needed, the terrain is rendered from a simple mipmapped heightmap.

2.7.1.3. CDLOD

Continuous Distance-Dependent Level of Detail for Rendering Heightmaps (CDLOD) was proposed by Filip Strugar in 2010 [10]. It is a technique for GPU-based rendering of heightmap terrains that is similar to the clipmap approach, since it draws the terrain directly from the source heightmap data, but instead of using a set of regular nested grids, it is structured around a quadtree of regular grids, more similar to Chunked LOD approach.

One improvement brought by Strugar approach is that its level of detail algorithm uses all three dimensions (x, y and z) of the observer position as a metric while Lasso's [9] approach is based only on the two-dimensional components (x and y). Moreover, Ulrich's [8] LOD function is the same over the whole chunk, giving only an approximate three-dimensional distance based LOD.

Ulrich's [8] and Lasso's [9] techniques suffer from another drawback: both approaches require additional work to remove gaps and provide smooth transitions between LOD levels. The CDLOD technique avoids these problems because the algorithm used to transition between LOD levels transforms the mesh of a higher level into the lower detailed one before the actual swap occurs. This ensures a perfectly smooth transition.

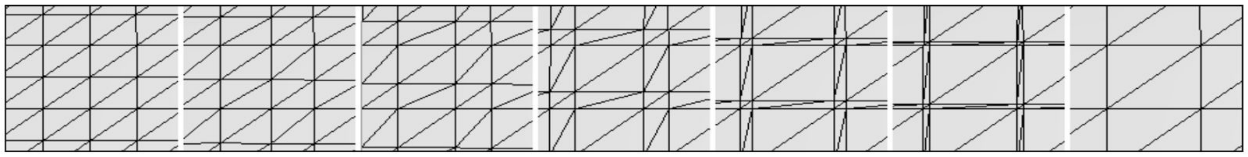


Figure 2.6 - Smooth transition between two LOD levels supported by CDLOD.

One of its drawbacks is that the settings used to generate the quadtree and run the algorithm need to be carefully chosen to match terrain dataset characteristics in order to provide the best performance. Another limitation of the algorithm is that a single quadtree node can only support transition between two LOD layers.

3. DEVELOPMENT

This chapter details the necessary steps to achieve this project's objectives. Being this an experimental research and highly focused on practical tests, three different demos were implemented in order to better understand the difficulties met by those who wish to work with terrain visualization. Many LOD techniques such as Chunked LOD are heavily CPU bound, what makes them unsuited for applications that are not only used for terrain visualization but additionally for other complex tasks that are also CPU bound, e.g. video games and simulations. On the other hand, GPU based techniques, e.g. geometry clipmapping and CDLOD, not only leave the CPU free for other computations, but also take advantage of the current highly parallel and massive GPU processing power. Therefore, most of the techniques chosen to be used in this project were GPU based.

3.1. GPU Based Mesh Displacement

It was natural to start by implementing the most basic approach, being it a naïve GPU based mesh displacement. A flat regular grid mesh is sent to GPU along with elevation data, which is in the format of a simple texture, and each mesh's vertex is individually displaced on the GPU by sampling the elevation texture in order to retrieve the elevation data for that particular position in the world space.

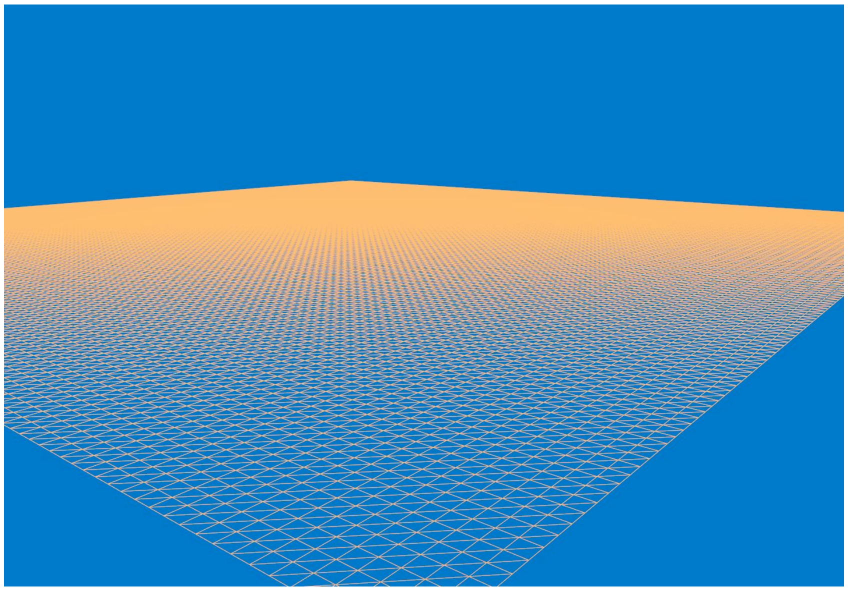


Figure 3.1 - Flat regular grid mesh without displacement.

Since this approach makes no use of any form of level of detail, it simply is a regular grid mesh that does not change its triangle count by no means, it is not advisable to use it when a high resolution terrain or when a large view distance is required. However, it can handle low resolution terrains fairly well and is straightforward to implement.

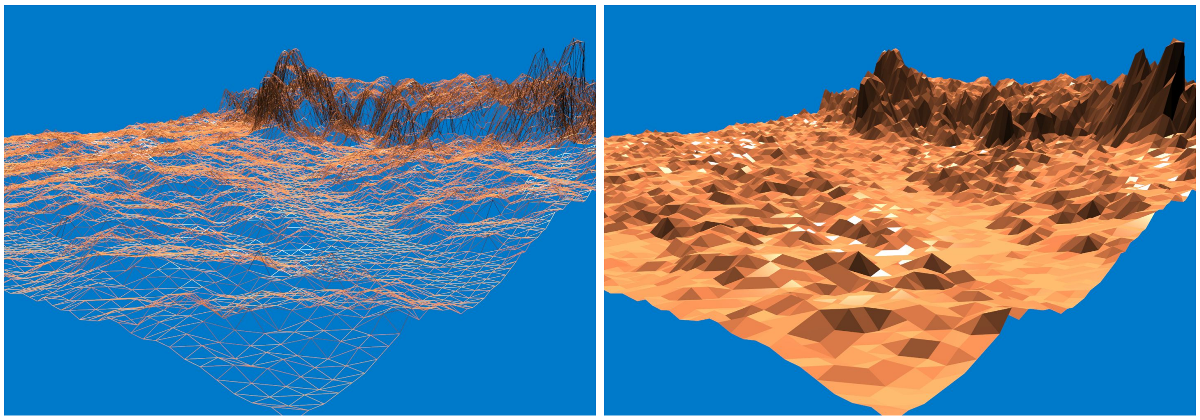


Figure 3.2 - Regular grid mesh displaced on the GPU through vertex texture fetch. The normals were generated at OpenGL's Geometry Shader pipeline stage.

One of the issues found while working with GPU based mesh displacement is the mesh normals. In computer graphics is common to define along with each vertex its normal vector, which will be used later for shading the primitives that contain the vertex. However, while using this technique, it is not possible to define the vertices' normals beforehand since the mesh is displaced at rendering time. To overcome this issue, it is possible to use OpenGL's Geometry Shader pipeline stage to calculate the normals at rendering time after the mesh have been displaced but still before the shading process. Another possible approach would be to use normal maps, i.e. textures where each pixel stores a normal vector value. Thus, it would be necessary to generate a normal map for each heightmap and then sample it in the same way as the elevation texture to retrieve the normal vector for the corresponding vertex position.

3.2. Geometry Clipmapping

Using the same concept from GPU based mesh displacement, a geometry clipmapping was implemented by adding several nested regular grids as can be seen in figure below. In order to achieve that, the regular grid from GPU based mesh displacement was repeatedly drawn and scaled according to the clipmap level, where each level consists of 4x4 patches that slightly overlap each other. Each successive level would then cover four times the area of the last level while keeping the same triangle count. Consequently, it was possible to achieve a much higher view distance along with an also higher terrain resolution close to the viewer while keeping the same frame rate when compared to GPU based mesh displacement.

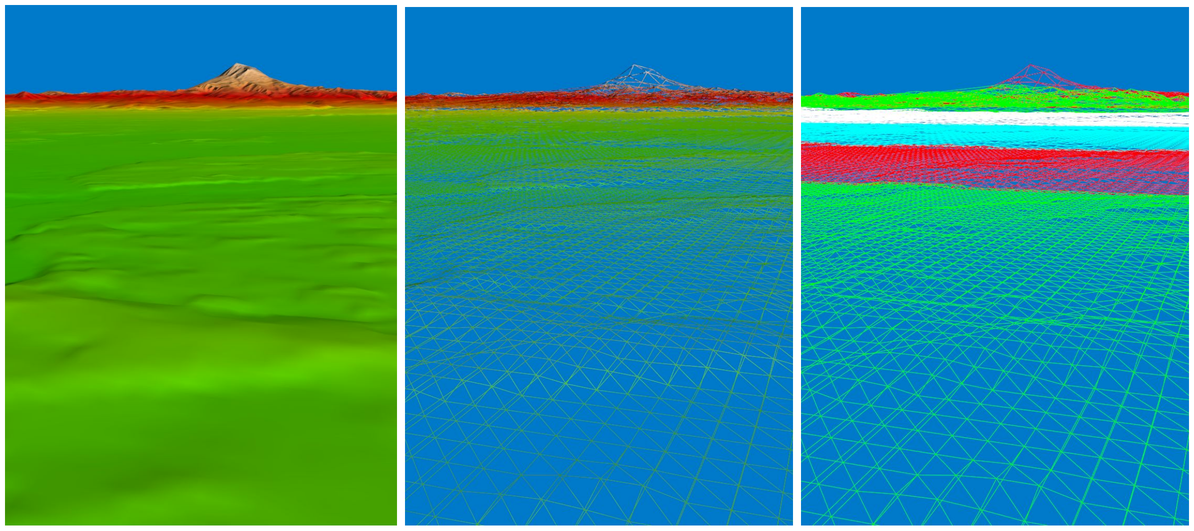


Figure 3.3 - Puget sound dataset rendered using geometry clipmapping. On the rightmost image each color represents a different clipmap level. By looking at the mountain in the background, is easy to notice how triangles become larger with distance.

In order to increase the performance even more, a view frustum culling was then utilized so that only visible patches would be drawn. To achieve that, a bounding box is generated for each patch and then transformed into clipping space by multiplying each bounding box's corner by the projection matrix. Then, a test is made to check if all bounding box's corners lie on the same side of the view frustum and, if they do, the patch won't intersect the view frustum and does not need to be drawn.

3.3. Dynamically Tessellated Terrain

Even though a regular grid mesh is definitely not advisable for use with high resolution terrain rendering, for its resolution is fixed, one can use a feature available on modern GPUs called hardware tessellation in order to increase the triangle count on demand. Thus, it is possible to have a low resolution regular grid mesh sent to the GPU to have its triangle count increased. Several algorithms can then be used to adjust the amount of subdivisions for each grid cell.

One of the difficulties found while working with tessellation was what is the best way to determine how much a primitive should be tessellated. Two widely used metrics were used in this project. The first one is the camera distance approach which, as the name suggests, uses the distance from the camera as a metric to choose how much an edge should be subdivided. The distance was calculated from the midpoint of each primitive's edge to camera position in world space. After getting that information, it was necessary to transform that distance value into a tessellation factor. One conversion algorithm tested was a simple discrete subdivision of the range from 0 to a D value, where D is user defined and represents the maximum distance at which the tessellation should be applied, by N, an user defined value for the number of discrete intervals, where each interval had an assigned tessellation factor. Those tessellation factors should be carefully tuned for each dataset in order to meet the visual quality requirements since rougher terrains required more triangles. Another variable that also affects tessellation factors is the primitive size. Larger primitives demanded, naturally, more subdivisions to keep up with the visual quality requirements. Furthermore, a continuous subdivision algorithm of the range 0 to D was also tested. The calculated distance from the camera was normalized in the range 0 to D and then subtracted by 1, so that the resulting value would be on the interval [0…1], where 1 means the edge is closer to the camera and 0 that the edge is at the maximum distance D. The resulting value was then multiplied by another user defined variable, MaxSubDiv, which defines how much an edge should be tessellated when close to the camera.

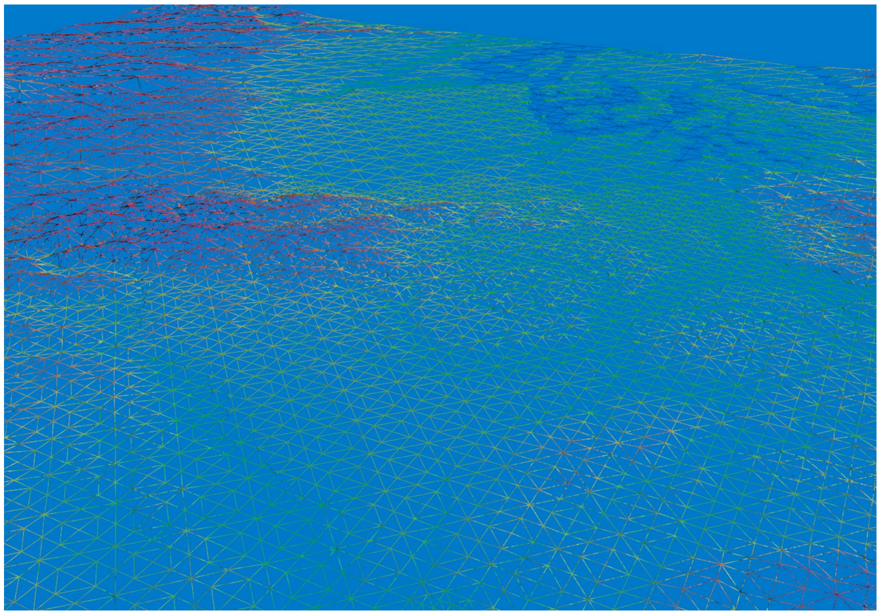


Figure 3.4 - Puget sound dataset rendered by GPU tessellation using the camera distance metric with discrete intervals. It is possible to notice how this approach resembles geometry clipmapping.

The second metric used was the sphere diameter in clip space, which is an approximation of an edge's screen size. With this metric, the user can define a target triangle screen size given in pixels. The goal of this approach is make all triangles be as close to the same size as possible. To achieve this, a bounding sphere is calculated for each edge and then projected into the screen. Then, the sphere's diameter is calculated, which gives the sphere's size in pixels. This value can then be used as a metric to achieve evenly sized triangles across the screen. This approach gives great results when the goal is visual quality while at the same time making it easy to control the expected amount of triangles on the screen.

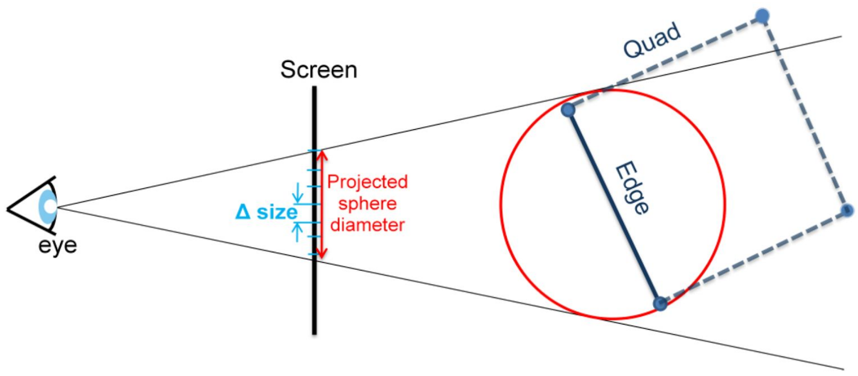


Figure 3.5 - Sphere diameter in clip space metric.

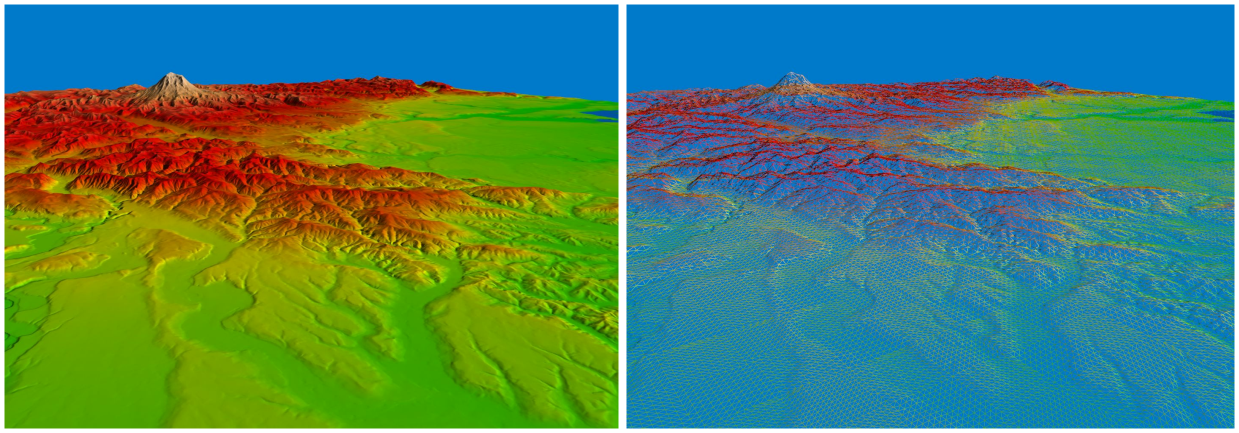


Figure 3.6 - Puget sound dataset rendered using GPU tessellation using the sphere diameter in clip space metric, which yields evenly sized triangles across the screen.

3.4. Unity's Built-in Terrain System

For testing purposes and analysis of a real customer grade application that handles terrain visualization, an application was developed using Unity3D game engine. Even though there is no public information about the LOD algorithm used by Unity, it is acceptable to assume that it is some kind of variant of Urich's Chunked LOD [8], since, by analysing the mesh's wireframe, it clearly adopts a quad-tree for node selection and has settings such as pixel error, which is also used in Urich's approach. Unity3D has a built-in terrain system where each terrain object is capable of rendering a heightmap resolution of 4097 by 4097 at most. But this resolution may not be enough for applications need to visualize massive terrains. Therefore, an algorithm was implemented to chunk the heightmap data and assign each heightmap chunk to a terrain object. Those terrain objects were then placed as a grid, side by side, effectively increasing Unity's maximum terrain resolution.

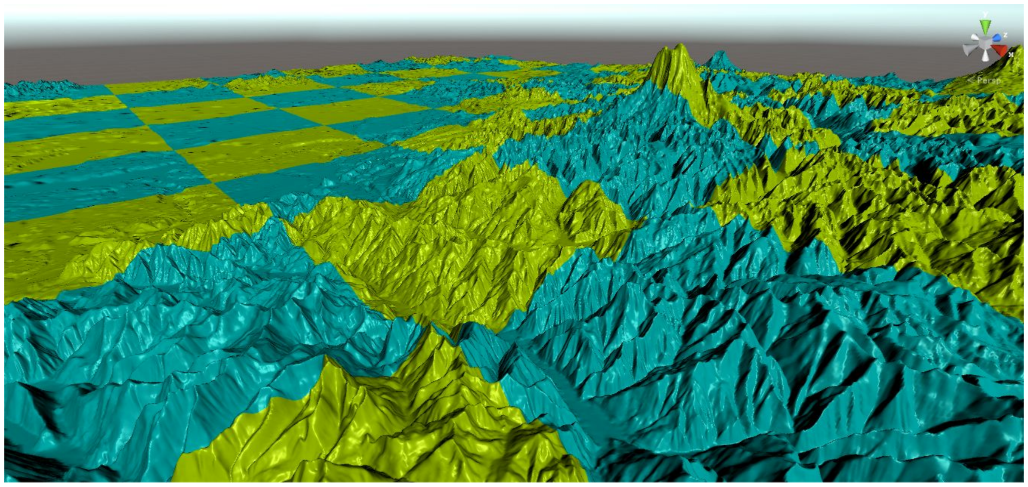


Figure 3.7 - Puget sound dataset rendered by Unity. The checkered pattern shows the individual terrain objects.

3.5. Logarithmic Depth Buffer Precision

One common problem that most programmers will probably someday encounter in computer graphics is the z-buffer precision issue (also known as z-fighting). It usually occurs when two coplanar primitives have almost identical z-buffer values, causing the primitives to be chosen arbitrarily as the closer one, yielding severe artifacts as can be seen on Figure 3.8. That is caused by the way z-buffer works. The values stored in the z-buffer depends on the distance from the camera and on the view frustum's near and far planes. However, the values are divided by the distance from the camera, what makes the z-buffer's precision proportional to the reciprocal of that distance value. This gives amounts of precision near the camera but little off in the distance.

To overcome this issue a logarithmic z-buffer, as proposed by Brano Kemen [11], was implemented. This solution changes the z-value precision distribution by a logarithmic distribution, which gives an exceptional amount of precision when compared to the default z-buffer, especially at long distances.

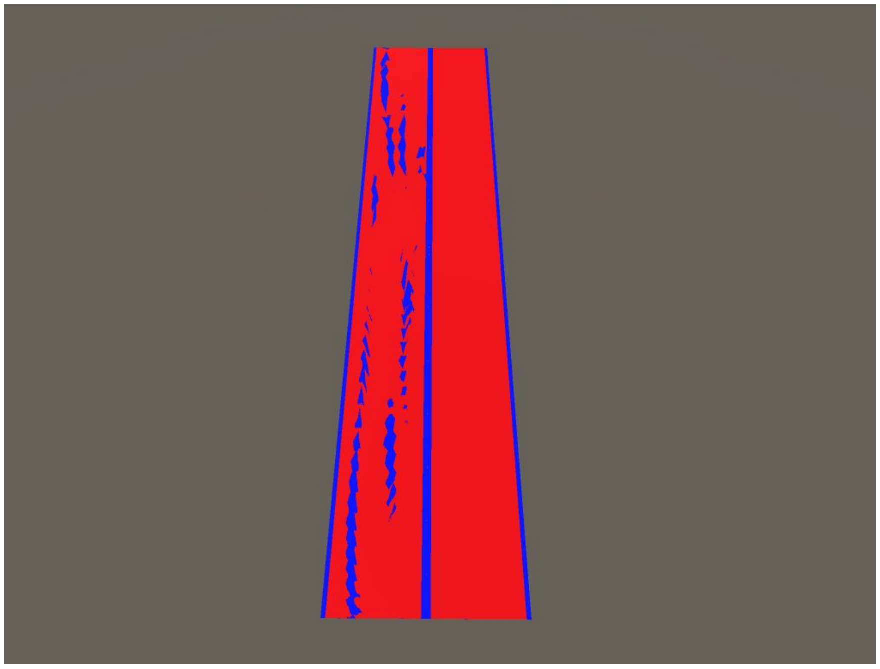


Figure 3.8 - Two identical objects rendered with standard depth buffer (left) and with a logarithmic depth buffer shader (right). The object used on this scene consists of two overlapping planes, one blue and one red, where the red plane is 0.2 units, above the blue plane. On this scene, the camera is approximately 1000 units from the objects.

4. EXPERIMENTS AND RESULTS

This chapter will provide information about the experiments executed with the implemented LOD techniques. All tests have been executed using the Puget Sound data set, which consists of two 4097 by 4097 textures, being one the heightmap, which contains the elevation data, and the second a simple color texture. Moreover, all test also have been executed at a screen resolution of 1280 by 720. The testing machine was composed by an Intel Core i7 4700MQ, running at 2.4GHz, a Nvidia Geforce GT755M and 8GB of RAM.

Two different scenes were chosen in order to test the different case scenarios. In the first one, the camera is set at a ground level position looking at the horizon so that the differences in the terrain silhouette are easily spotted as well as how well the terrain approach can handle up close and far away visualization on the same scene. In the second scene, the camera is set at a higher altitude and is looking downwards in order to simulate a flying object and check the performance when no detailed terrain is needed in order to get a acceptable terrain visualization. A reference image was generated for each scene using GPU based mesh displacement with a grid resolution of 4097, matching the heightmap's resolution, assuming that a higher resolution would not bring any visual quality benefits.

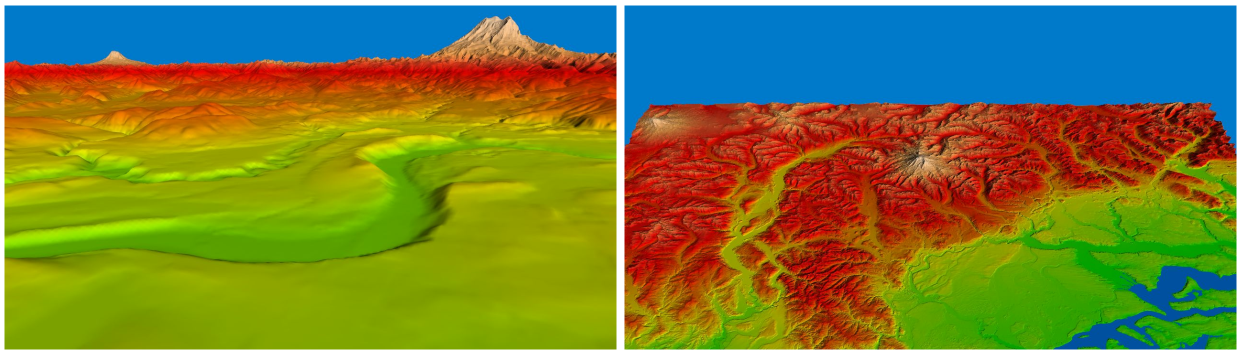
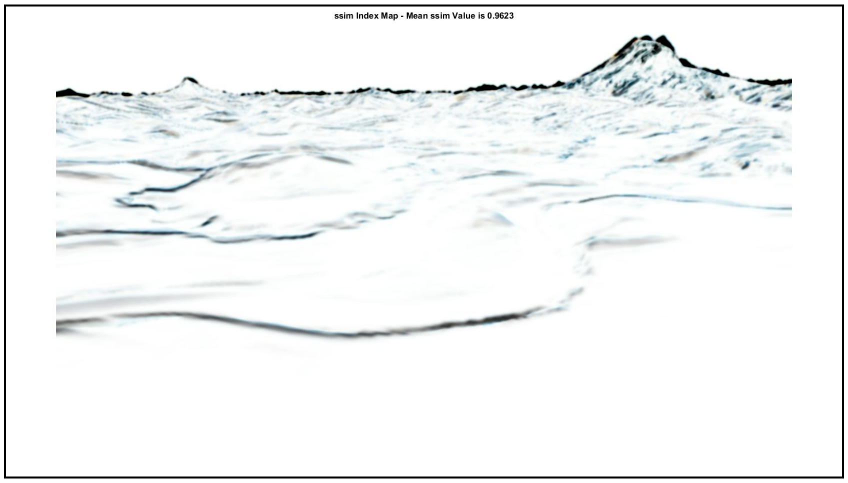


Figure 4.1 - Test scene 1 (left) and test scene 2 (right).

The first tested technique was GPU based mesh displacement. The only tunable setting available was the grid size. Since it has no LOD, it behaved as expected by dropping the performance dramatically as the grid size increased. Being the grid size of 4097 the reference image, it obtained a perfect 1.0 SSIM score. Through the SSIM index map, Figure 4.2 and Figure 4.3, was possible to notice how the detail changes equally throughout the whole terrain.

Grid size		129	257	513	1025	2049	4097
Scene 1	Frames per second	982.225	706.031	451.123	241.772	97.4553	28.1875
	SSIM	0.9623	0.9728	0.9846	0.9939	0.9984	1.0
Scene 2	Frames per second	526.198	364.449	231.025	135.937	70.0379	26.1512
	SSIM	0.9641	0.9804	0.9920	0.9978	0.9995	1.0

Table 1.1 - GPU based mesh displacement test results.



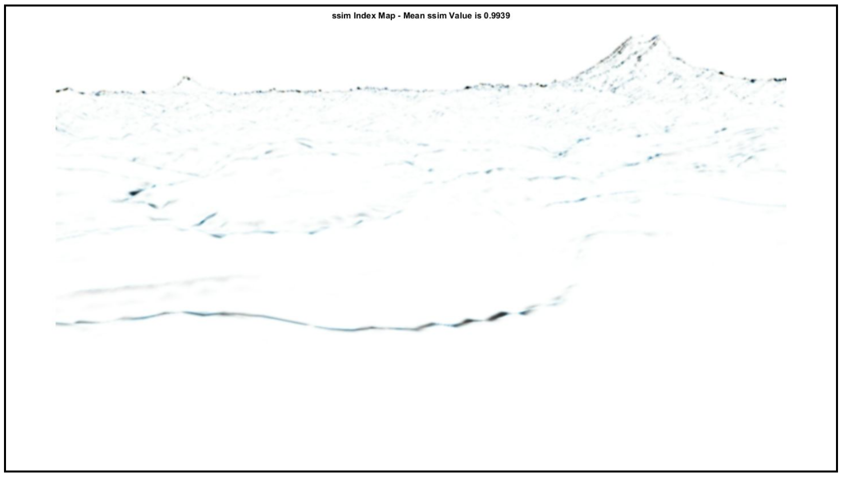
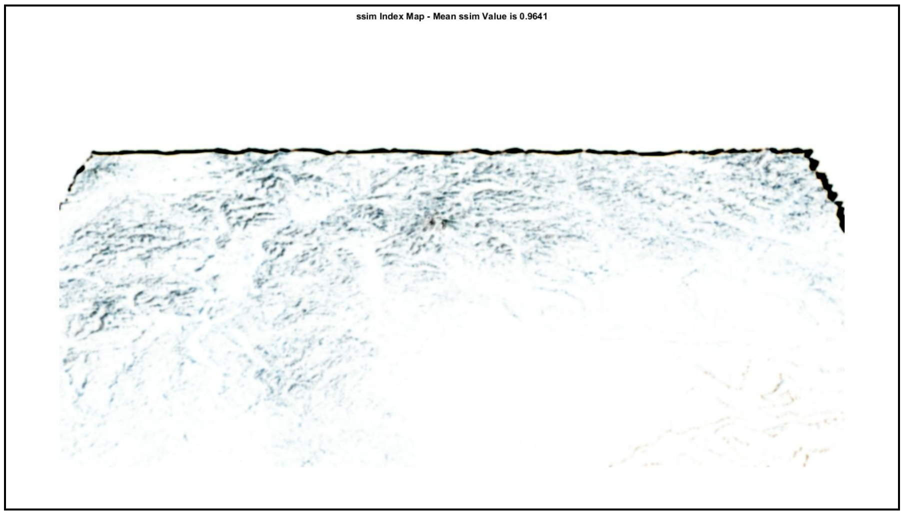


Figure 4.2 - Scene 1 SSIM index maps for GPU based mesh displacement with grid size 129 (top) and 1025 (bottom).



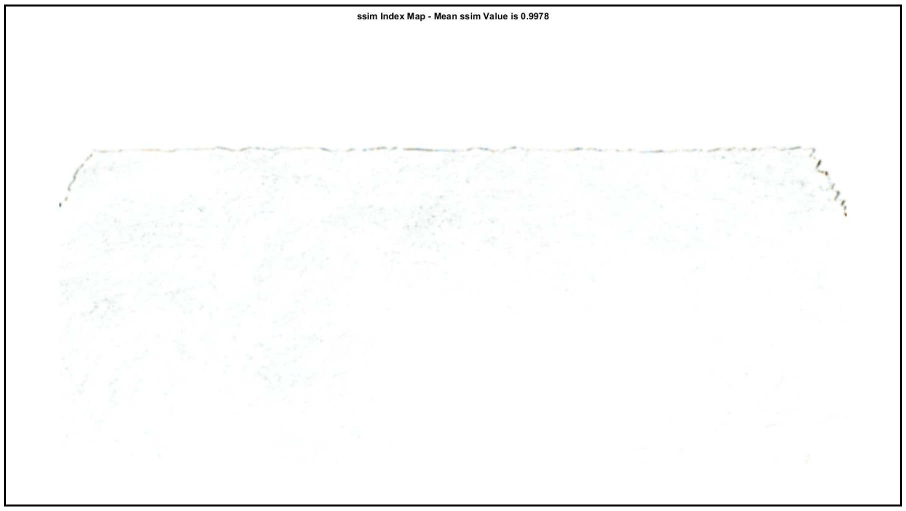


Figure 4.3 - Scene 2 SSIM index maps for GPU based mesh displacement with grid size 129 (top) and 1025 (bottom).

Now geometry clipmapping was capable of maintaining good frame rates, even with a large number of clipmap levels. Moreover, geometry clipmapping also implemented frustum culling, which ended up increasing the frame rates. However, in order to better take advantage of frustum culling, a larger number of clipmap levels had to be used. That happens because, the more clipmap levels are used, the more patches are discarded, increasing the performance difference between the frustum culled and the non frustum culled tests.

Number of clipmap levels			1	2	3	4	5
Scene 1	Frames per second	Culling on	678.643	546.909	481.008	429.637	426.191
		Culling off	590.167	447.44	370.535	316.59	280.395
	SSIM		0.9767	0.9856	0.9903	0.9916	0.9917
Scene 2	Frames per second	Culling on	519.333	442.29	383.962	333.222	335.099
		Culling off	490.98	423.558	369.325	322.346	294.485
	SSIM		0.9826	0.9840	0.9840	0.9840	0.9840

Table 1.2 - Geometry clipmapping test results.

When comparing the SSIM index maps for geometry clipmapping it was possible to notice how the visual quality increases close to the camera when the number of clipmaps was increased. That happens because the clipmap levels are inserted as an inner level, which has more details, leaving the terrain far from the observer untouched. This behavior is visible when analysing the horizon in Figure 4.4.

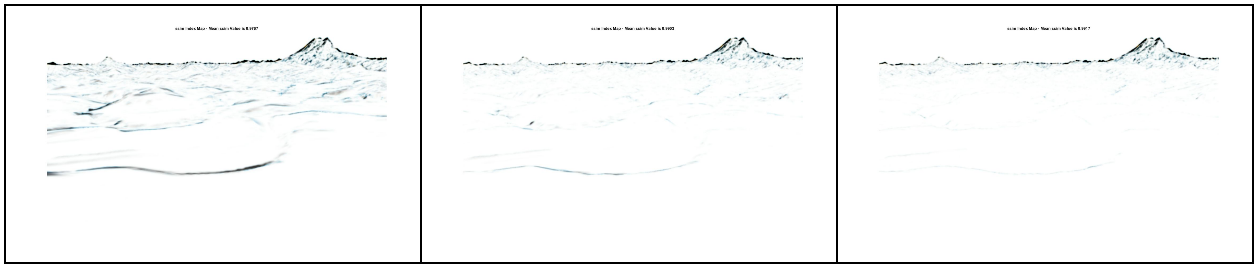

ssim Index Map - Mean ssim Value is 0.9767
ssim Index Map - Mean ssim Value is 0.9903
ssim Index Map - Mean ssim Value is 0.9917

Figure 4.4 - Scene 1 SSIM index maps for geometry clipmapping with, from left to right, 1, 3 and 5 clipmap levels.

Another observation that can be seen on Table 1.2 is how the SSIM indexes doesn't change after 2 clipmap levels. Being the camera at a higher altitude and the algorithm adding more triangles close to the camera's position, there is no difference in image quality, since most new triangles are smaller than a pixel and don't add new information to the scene at the current position.

In order to see how the dynamically tessellated terrain performed, two settings were adjusted. The grid size on this approach have the same objective as on GPU based mesh displacement, however, it was also possible to adjust the desired average triangle screen size, which resulted in a better triangle distribution across the screen.

Grid size	**64**	**128**	**256**	**512**
Triangle size 64 pixels	1406fps / 0.9521ssim	921fps / 0.9636ssim	584fps / 0.9756ssim	287fps / 0.9873ssim
Triangle size 32 pixels	1098fps / 0.9551ssim	860fps / 0.9677ssim	555fps / 0.9802ssim	278fps / 0.9902ssim
Triangle size 16 pixels	998fps / 0.9592ssim	786fps / 0.9728ssim	510fps / 0.9841ssim	257fps / 0.9926ssim
Triangle size 8 pixels	848fps / 0.9674ssim	662fps / 0.9794ssim	421fps / 0.9884ssim	206fps / 0.9945ssim

Table 1.3 - Dynamically tessellated terrain test results for scene 1.

Grid size	**64**	**128**	**256**	**512**
Triangle size 64 pixels	759fps / 0.9431ssim	509fps / 0.9639ssim	330fps / 0.9803ssim	194fps / 0.9920ssim
Triangle size 32 pixels	663fps / 0.9431ssim	507fps / 0.9639ssim	331fps / 0.9803ssim	194fps / 0.9920ssim
Triangle size 16 pixels	664fps / 0.9431ssim	510fps / 0.9639ssim	330fps / 0.9803ssim	194fps / 0.9920ssim
Triangle size 8 pixels	662fps / 0.9431ssim	511fps / 0.9639ssim	330fps / 0.9803ssim	194fps / 0.9920ssim

Table 1.4 - Dynamically tessellated terrain test results for scene 2.

From Table 1.4 we can see that, when the observer is far away from the terrain, the base grid size is what gives the terrain quality since it is far enough so that the triangle size won't add more details to the scene.

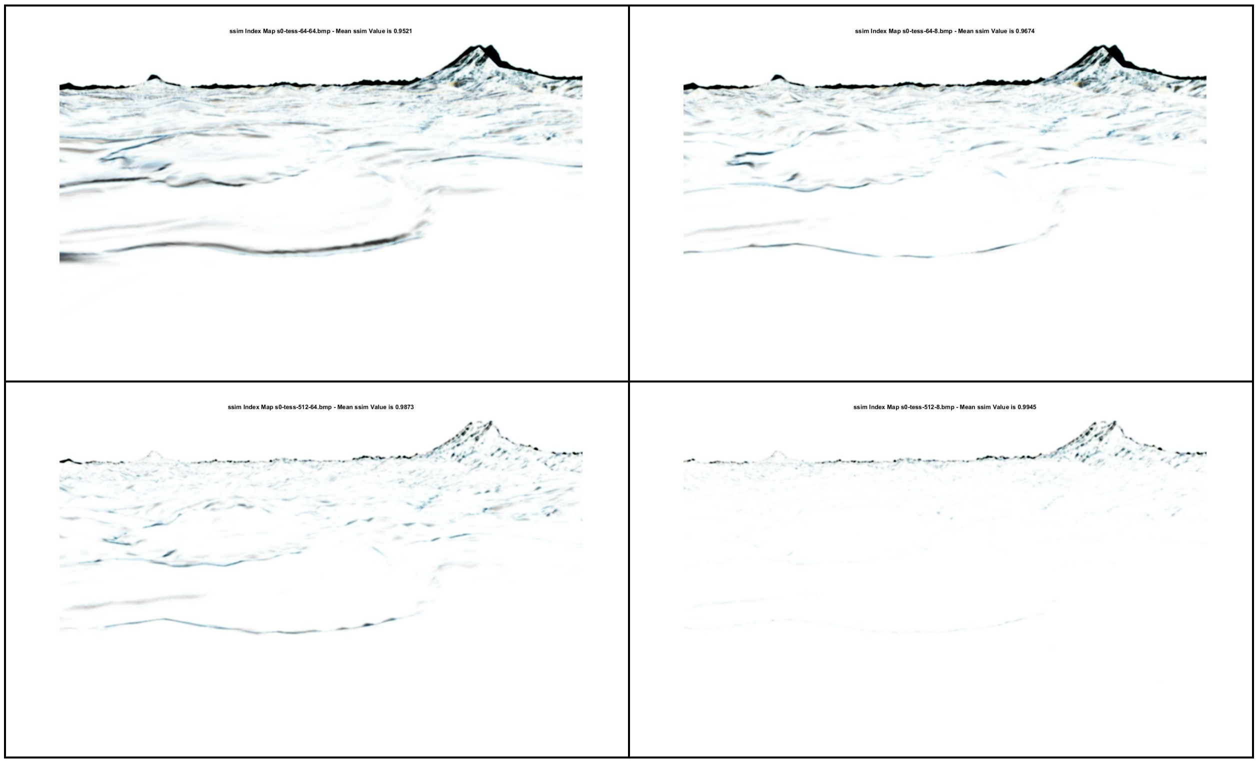

ssim Index Map s0-tess-64-64.bmp - Mean ssim Value is 0.9521
ssim Index Map s0-tess-64-8.bmp - Mean ssim Value is 0.9674
ssim Index Map s0-tess-512-64.bmp - Mean ssim Value is 0.9873
ssim Index Map s0-tess-512-8.bmp - Mean ssim Value is 0.9945

Figure 4.5 - Scene 1 SSIM index maps for dynamically tessellated terrain. Both left images have a triangle size of 64 pixels and both right images have a triangle size of 8 pixels while the top images are using a grid size of 64 and the bottom ones are using a grid size of 512.

As well as in the geometry clipmapping, the quality increases massively close to the observer, specially when the triangle size decreases. At distance only the base grid size brings any changes to the visual quality. However, it still won't achieve high quality at distance since neither one of this approaches is adaptive regarding the terrain slopes, but instead they assume that the terrain complexity only decreases with the distance.

5. CONCLUSION AND FUTURE WORK

This project had the objective of understanding all the difficulties that can be encountered while working with massive terrain visualization. The project ended up being heavily inclined towards LOD algorithms, what is understandable since the most important feature of a terrain rendering engine is how well it can handle level of detail. Many LOD techniques were studied and a few were implemented.

All of the currently implemented LOD techniques on this project made use of a texture small enough to fit into GPU memory. With that in mind, the next step on the project would undoubtedly be the out-of-core topic. An streamable version of the dynamically tessellated terrain could be implemented using the same idea of toroidally texture update proposed by Lossaso and Hoppe [9]. Another intuitive improvement would be to make use of more than one chunk of dynamically tessellated terrain together with frustum culling. This way a higher grid resolution could be used naturally increasing the maximum terrain resolution.

REFERENCES

[1] Goswami, Prashant. "Level-of-detail and Parallel Solutions in Computer Graphics." PhD diss., 2012.

[2] Vitter, Jeffrey Scott. "External memory algorithms and data structures: Dealing with massive data." *ACM Computing surveys (CsUR)* 33, no. 2 (2001): 209-271.

[3] Silva, Claudio, Yi-Jen Chiang, Jihad El-Sana, and Peter Lindstrom. "Out-of-core algorithms for scientific visualization and computer graphics." *IEEE Visualization Course Notes* (2002).

[4] Lauritsen, Thomas, and Steen Lund Nielsen. "Rendering very large, very detailed terrains." (2005).

[5] Bösch, Jonas, Prashant Goswami, and Renato Pajarola. "RASTeR: Simple and efficient terrain rendering on the GPU." Eurographics Areas Papers (2009): 35-42.

[6] Cozzi, Patrick, and Kevin Ring. 3D engine design for virtual globes. CRC Press, 2011.

[7] Bekiaris, Georgios. "Real-Time Planet Rendering." PhD diss., Technological Educational Institute of Piraeus, 2009.

[8] Ulrich, Thatcher. "Rendering massive terrains using chunked level of detail control." SIGGRAPH Course Notes 3, no. 5 (2002).

[9] Losasso, Frank, and Hugues Hoppe. "Geometry clipmaps: terrain rendering using nested regular grids." ACM Transactions on Graphics (TOG) 23, no. 3 (2004): 769-776.

[10] Strugar, Filip. "Continuous distance-dependent level of detail for rendering heightmaps." Journal of graphics, GPU, and game tools 14, no. 4 (2009): 57-74.

[11] KEMEM, BRANO. Logarithmic Depth Buffer. 08 de dez. de 2009. Available at: <http://www.gamasutra.com/blogs/BranoKemen/20090812/85207/Logarithmic_Depth_Buffer.php>. Visited on: 20 august 2015.

[12] Hore, Alain, and Djemel Ziou. "Image quality metrics: PSNR vs. SSIM." In Pattern Recognition (ICPR), 2010 20th International Conference on, pp. 2366-2369. IEEE, 2010.