NTNU – Trondheim
Norwegian University of
Science and Technology

Terrain Rendering Techniques for the HPC-Lab Snow Simulator

Kjetil Babington

Master of Science in Computer Science
Submission date: June 2012
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

This project builds on current and previous graduate students work on the NTNU HPC-Lab snow simulator. The goal of this work is to make the current simulation more realistic. This will be done by adding in features, such as better terrain, based on one or more sources terrain data. The features may use terrain generation algorithms and/or other appropriate software and techniques. Enhanced snow particle rendering may also be included.

Abstract

This thesis presents a technique for GPU-based terrain rendering and the changes made to the HPC-lab snow simulator to integrate the new terrain rendering technique into the simulator. Our novel terrain rendering technique combines ideas from existing terrain rendering techniques such as CDLOD[19] and Geometry Clipmaps[12] into a hybrid method. The terrain rendering works on patches of quads, that are tessellated, using hardware tessellation based on the level of detail needed. The tessellated patches are then displaced, using a vertex texture fetch of the heightmap in the tessellation shader. The implemented GPU terrain rendering technique is then added to the HPC-lab snow simulator, and changes to the simulator are implemented to facilitate the new terrain rendering technique, all of the old GLSL shaders are updated to the newest standard, the code structure is changed, and the collision detection of the snow simulator is updated to accommodate changes made to the terrain.

The results from our benchmarks show that the tessellation pipeline can be used to facilitate terrain triangle count of over 16 million triangles while maintaining a stable frame rate of over 1400 FPS. When used in combination with the simulator, the implementation is still able to achieve frame rates that are vastly greater than the old implementation in the snow simulator. The visual results acheived from using Perlin noise gives the simulator a more realistic feel, while not degrading the performance of the implementation. Suggestions for futher improvements are also included.

Acknowledgments

This thesis is the result of a semester long master project, done at the Norwegian University of Science and Technology(NTNU) in Trondheim. I would like to thank Dr. Anne C. Elster for providing me with opportunity to do this thesis with her, and for the work she has put in to make the HPC-lab such a great place to work. I would also like to use the opportunity to thank the rest of the HPC-lab members for their support and the positive working environment they provide at the lab. Finally, I would like to thank NVIDIA for their sponsoring of Dr. Elster and her HPC-lab through their Mad Scientist purchase Program.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Terrain geometry is a very important part of rendering outdoor graphical environments. The terrain provides the basis for other parts of the environment to build upon, so creating and rendering the terrain with the realism required, while maintaining a fast frame rate, has always been an important research area. The most widely used representation of the terrain geometry, is the heightmap, a 2D grid of height values. The heightmaps can be created from real world data, or can be procedurally made. The simplest way to render the heightmap is the brute force approach, where each heightmap value corresponds to one vertex. For larger terrains, this is unsuitable, since the number of vertices grows too large to be processed in real-time by the GPU.

The HPC-lab snow simulator is a continuous work at the HPC-lab at the Norwegian University of Science and Technology (NTNU). The simulator started as a smoke simulator, and has since been updated into a full snow simulator. This work builds upon the work of Ingar Saltvik[18], Robin Eidissen[7], Hallgeir Lien[11], changes to the simulator are made. The collision detection system between falling snow particles and terrain is improved, and the all of the code is updated to create a better working environment. Old outdated shader code is updated to the newest GLSL standard, and the old fixed function code for OpenGL is replaced with more modern implementations. The HPC-lab snow simulator has so far used a variation of the brute force approach for rendering terrain. *This thesis aims to increase the rendering performance of the terrain*, by utilizing a more modern approach for terrain rendering. Tessellation capabilities of modern graphics cards are utilized for a dynamic level of detail system. The triangle density of the parts of the terrain further away from the camera are reduced, while the parts that are

close to the camera, are increased so that even more details can be shown at close range. These changes to the implementation enables the HPC-lab snow simulator to have an even higher complexity, in other parts of the simulation, e.g. the wind field resolution could be increased even higher, or the particle count could be increased.

To further increase the realism of the HPC-lab snow simulator, a procedural texturing scheme is implemented. *Perlin noise* is used to create detail textures that can be added to regular textures to increase the realism of natural textures, such as rocks, grass, sand, and mountains. The noise textures are also used to increase the details of heightmaps, e.g. to create mountainous heightmap terrain, and to create a more natural looking experience.

1.1 Outline

This thesis is organized as follows

Chapter 2 introduces the background knowledge related to this master thesis. An introduction to GPUs and GPU programming through CUDA and OpenGL is given, the HPC-lab snow simulator is introduced, state of the art rendering techniques related to terrain and our novel terrain rendering technique is introduced. Lastly, an introduction to the Perlin noise is given.

Chapter 3 gives the details of our novel terrain rendering approach, and details the changes made to the snow simulator to incorporate it in the HPC-lab snow simulator. This chapter also presents the changes made to our snow simulator, to increase the performance of the simulation.

Chapter 4 presents the performance benchmarks of the terrain rendering implementation, and gives a detailed discussion about the results. Benchmark results for the updated snow simulator is also presented. At the end, the visual quality achieved is also presented and evaluated.

Chapter 5 presents the conclusion and potential future improvements.

Chapter 2

Background

This chapter will introduce concepts, background knowledge, and related works used in this thesis. A short introduction to GPUs and GPU computing through CUDA and OpenGL will be given. The HPC-lab snow simulator will be introduced. Terrain rendering methods related to our novel terrain rendering approach will be introduced. An overview of the terrain rendering method detailed in this thesis will be given. Texture noise generation algorithms and applications will be discussed and presented.

2.1 Modern GPUs

The GPU, or the graphical processing unit is a specialized chip designed to offload the work of creating images intended for display, from the CPU. The GPU comes with special hardware for processing and rasterization of huge amounts of vertices and triangles in parallel. This was also one of the main driving forces behind the high-performance nature of the GPU, real-time, realistic graphics for games. But as the power, and parallelism of the GPU increased, so did the interest in the possibility of using the GPU for other applications. The GPU is no longer just a tool for graphics, but is also used for research, simulations, and other applications which require huge amounts of computational power.

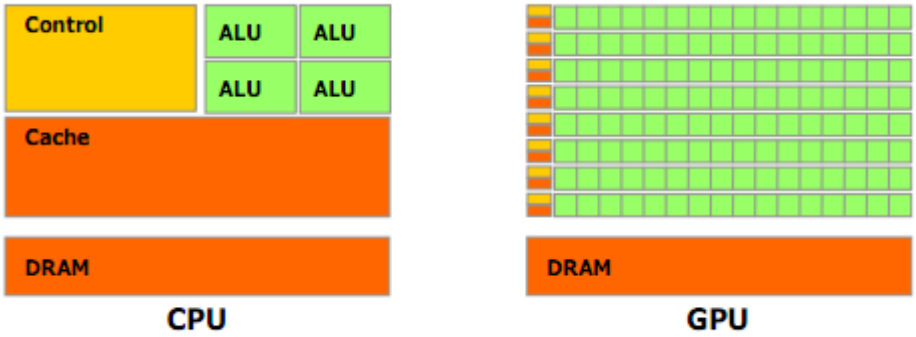


Figure 2.1: The different division of space on the chip of GPUs and CPUs[1]

2.1.1 Differences between the CPU and the GPU

The biggest difference between the CPU and the GPU is how space on the chip is divided. As can be seen from Figure 2.1, a large portion of the CPU space is filled with caches, and control structures, while on the GPU, there are smaller caches and control structures, but a lot more duplicated functionality. Because of this the CPU is a much more flexible unit than the GPU. The CPU can be optimized by taking advantage of instruction level parallelism, and the relatively large caching structures present. The GPU on the other hand does not have this flexibility, but is a lot more focused on performing the same operation simultaneously, and thus focuses a lot more on parallel processing power. Since there is so little space set aside for caching on the GPU, careful consideration for how memory is used is necessary to achieve optimal performance. Optimizations such as memory coalescing, avoiding bank conflicts, and other memory optimizations must performed to reach optimal performance when using the GPU[17].

As can be seen from Figure 2.2, the theoretical GFLOPs of the GPU is rising much faster than for the CPU, and this trend is just going to continue. In June 2011, NVIDIA released their roadmap, Figure 2.3 for GPU architecture, and Kepler, the next version of GPU architecture is state to have a increase of almost 3 over the current Fermi architecture, and Maxwell, Keplers successor is slated to have an even higher increase in power [14].

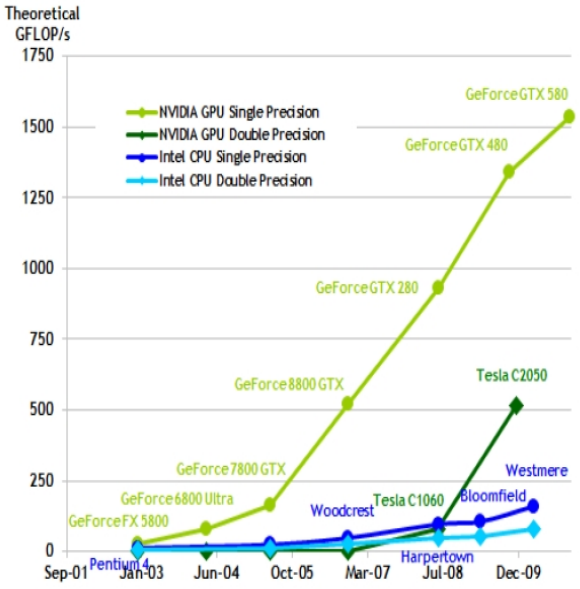


Figure 2.2: A chart showing the development of GFLOPs of different generations of GPUs and CPUs[1]

2.2 CUDA

CUDA, or Compute Unified Device Architecture is a parallel computing environment developed by NVIDIA for general purpose, high performance paral-

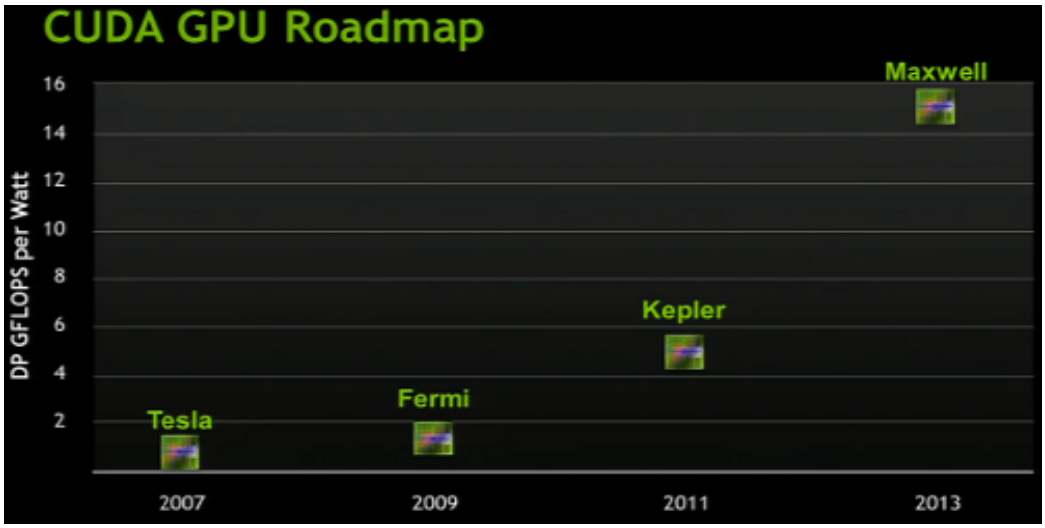


Figure 2.3: The released roadmap of future generations of GPU from NVIDIA

lel computing. CUDA enables developers to write highly parallel code, which can be executed on CUDA enabled devices. CUDA is written in a slightly extended C/C++ language, with some restrictions. As CUDA has matured, a lot of API wrappers have also been created for other languages, such as wrappers for Java, Fortran and Python.

A typical CUDA program is divided into host code, and device code. The host code runs on the CPU of the system. Host code is responsible for memory transfers, to and from the device and kernel invocation. Device, or kernel code is executed on the CUDA-device of the system. To launch a kernel, the host code need to invoke the function through special CUDA syntax. A typical invocation of a kernel specifies the kernel name, the number of blocks, the number of treads per block, and the kernel parameters. $kernelName <<< blocks, threads >>> (param1, param2)$. During compilation the host part of the code, and the device part of the code gets separated. The device code get compiled with NVCC. While the host code get compiled with a traditional C/C++ compiler. The compiled GPU functions then gets embedded as load images in the host object file.

The memory model of CUDA is divided into different levels, each with its own limitations and advantages. Global memory is the largest, but also the slowest memory that threads have access to. So to get optimal speeds, this memory should be accessed as little as possible. Each thread has access to some private local memory, and depending on whether there are free registers or not, the local memory might be placed in global memory. All threads inside a common block also share some fast shared memory. Two additional read-only memory places also exist, Constant and Texture memory. Constant memory is very limited, but is accessible from any thread, and is cached. Texture memory resides in global memory, but is read-only, so it does not suffer the same performance limitations as normal global memory.

2.3 OpenGL

OpenGL (Open Graphics Library) is a standard specification defining a cross-language, multiplatform API for writing applications that produce 2D and 3D computer graphics. OpenGL was developed by Silicon Graphics Inc (SGI) in 1992. The standard is now managed by Khronos Group[9].

2.3.1 The Graphics Pipeline

The graphics pipeline consist of several stages, where some are programmable. Each stage plays a role in creating, lighting, coloring, and presenting graphics to the framebuffer. Figure 2.4 shows the current pipeline for OpenGL 4.2 and DirectX 11, the green colored boxes are stages that are programmable through GLSL or HLSL depending on whetherOpenGL or DirectX is being used.

Vertex shaders are run once for each vertex given to the graphics processor. The purpose is to transform each vertex's 3D position in model space to screen-space. Vertex shaders can manipulate properties such as position, color, and texture coordinate, but cannot create new vertices. The output of the vertex shader goes to the next stage in the pipeline, which is either the tessellation stage, if tessellation shaders are present, geometry stage if a geometry shader is present, or to the fragment stage.

Geometry shaders can generate new graphics primitives, such as points, lines, and triangles, from those primitives that were sent to the beginning of the graphics pipeline.

Tessellation shaders takes as input a patch, and outputs a tessellated patch. Tessellation shaders can increase the vertex count of the input patch, by doing tessellation, thereby increasing the detail level of the input primitive. Figure **??** show how a quad is tessellated.

Fragment shaders compute color and other attributes of each pixel. Fragment can implement a range of different lighting techniques, do bump mapping, shadow creation, specular highlights, translucency and other phenomena. They can alter the depth of the pixel, or output more than one color if multiple render targets are active.

2.3.2 Tessellation stage

Tessellation shaders are used extensively through out this project, and a thorough introduction them are therefor included. The tessellation stage is the newest addition to the OpenGL language. They were added in version 4.0 [13]. The tessellation stage, consist of three sub-stages, control shader, primitive generator, and the evaluation shader.

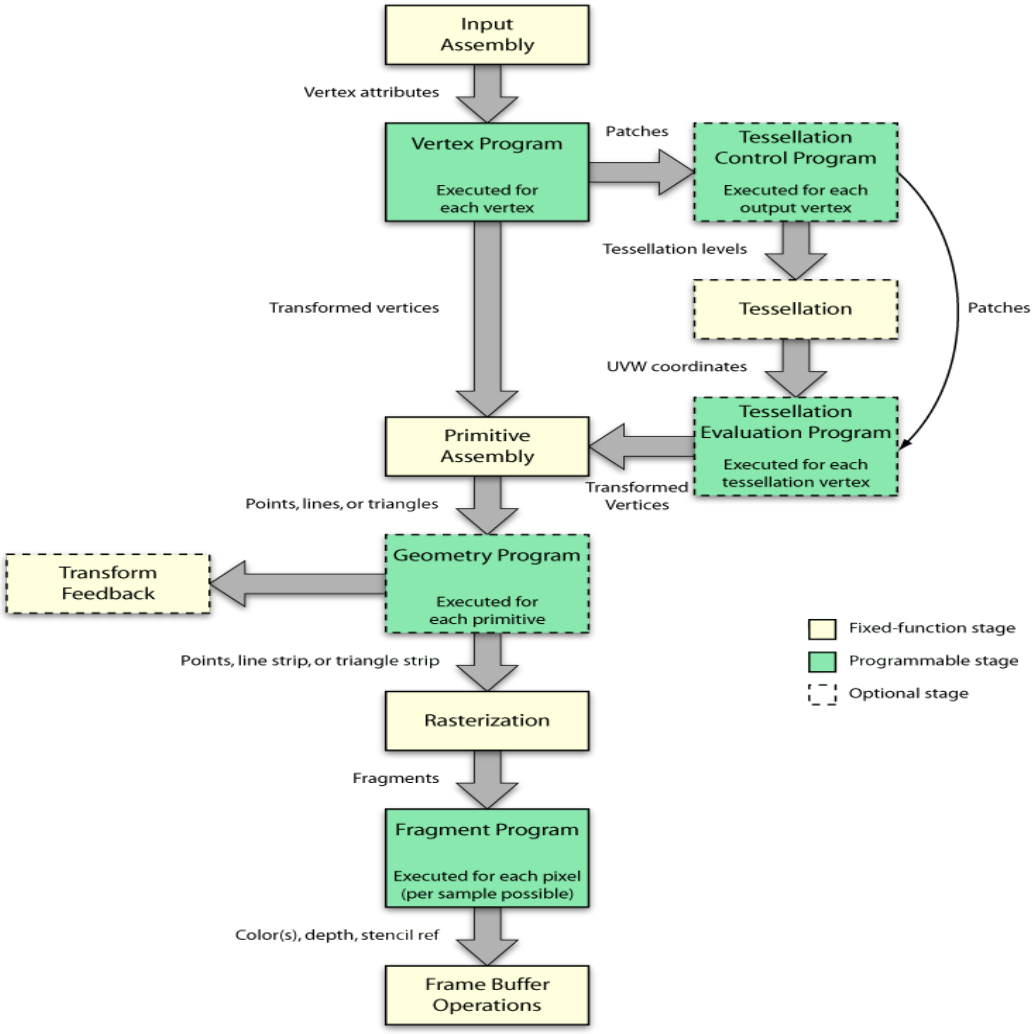


Figure 2.4: The stages of the rendering pipeline

Control shader

The control shader is run once for each vertex in the output patch, and computes the attributes of the vertices, e.g. the tessellation levels. The input to the control shader comes from the vertex shader.

Primitive generator

If a Tessellation Evaluation shader is linked, then this stage subdivides the input patch into a collection of points, lines, or triangles according to the

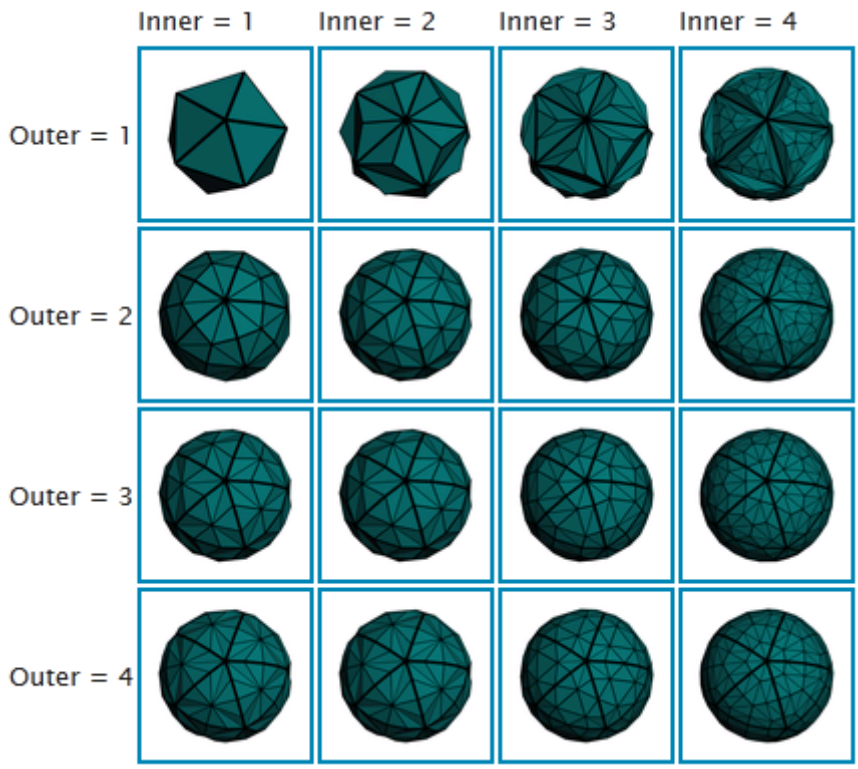


Figure 2.5: A primitive tessellated with different levels of inner and outer tessellation levels, ©Philip Rideout

tessellation levels computed in the control stage.

Evaluation shader

The Evaluation shader takes the location of each vertex generated from the primitive generator, and makes a vertex with a position and different associated values.

2.4 The HPC-lab Snow Simulator

The HPC-lab snow simulator is a project under continues development at the HPC-lab at the Norwegian University of Science and Technology. An image from the simulator can be seen in Figure 2.6. The work started as a smoke simulator[20], but was later updated to a snow simulator by Ingar Saltvik.

His master thesis [18], describes how he implemented a snow simulationer on multicore CPU. Robin Eidissen then built upon Saltvik's work through his master thesis[7], where he implemented, and showed how the GPU could be used to simulate falling snow. The snow simulator was further improved by Gjermundsen[8] and Chellia[4]. The latest addition to the simulator was done by Hallgeir Lien. He added import functionally for real maps created from USGS DEM data and procedural generation of roads[11].

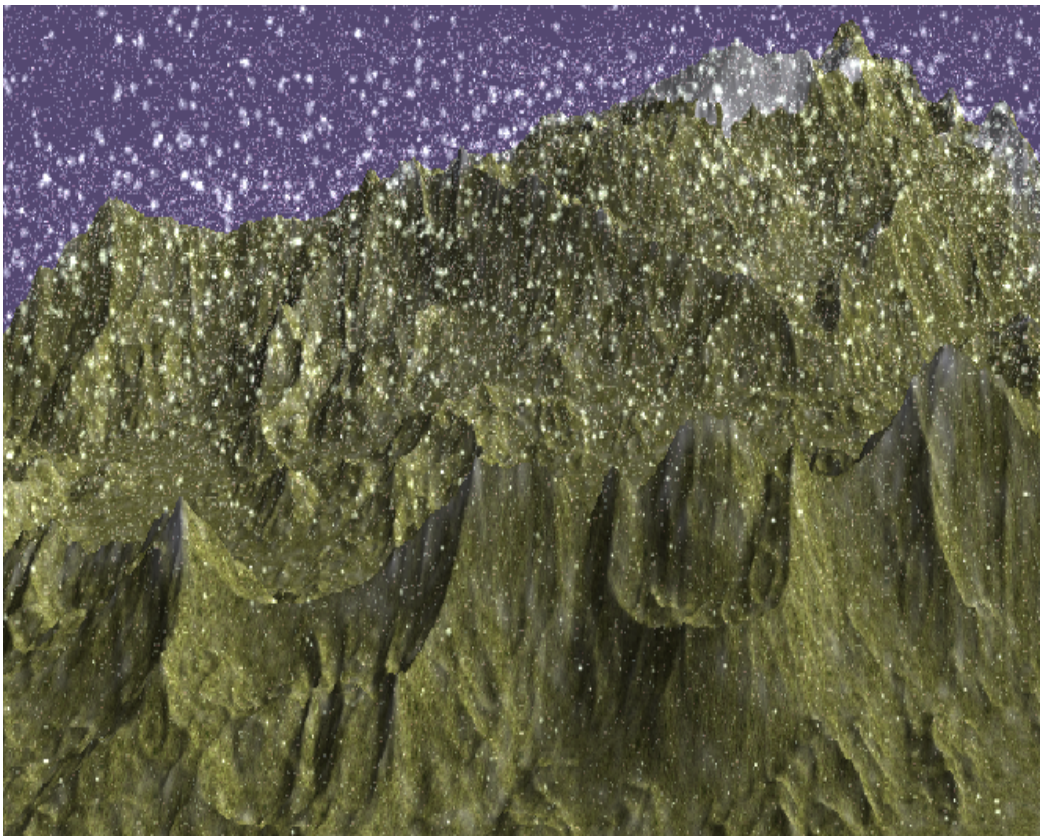
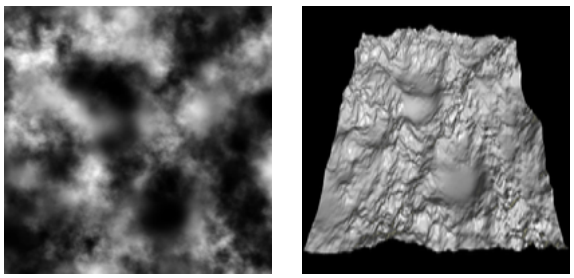


Figure 2.6: The HPC-lab snow simulator

2.4.1 Terrain

In our snow simulator, the terrain geometry is represented as a height map. Each of the grid points in this height map defines a vertex. The height map then gets loaded into OpenGL, using a VBO, and an indexing scheme is calculated. To reduce the number of vertices stored, a triangle strip indexing

scheme is used, where the first triangle is defined by three indices, but subsequent indices only use one new index, plus the two previous indices.



(a) Heightmap image (b) Rendering of the heightmap

Figure 2.7: Heightmap image (a) and the rendering of that heightmap image (b)

As the simulation is run, a snow cover is build upon the terrain. The height of this build up is stored in the W coordinate of the terrain vertices, and used by GLSL during rendering to raise the height of the vertices and to do blending so that the cover appears white.

2.4.2 Wind simulation

The wind simulation is done modeling the wind as a fluid. The flow of the fluid is then solved using the Navier-Stokes Equations for incompressible flow. The equations are then solved by setting the viscosity to zero, and density to one.

$$\nabla \cdot \boldsymbol{u} = 0 \tag{2.1}$$

$$\frac{\partial \boldsymbol{u}}{\partial \boldsymbol{t}} = -(\boldsymbol{u} \cdot \nabla)\boldsymbol{u} - \nabla \boldsymbol{p} \tag{2.2}$$

In Equation 2.1 and 2.2, $\boldsymbol{u}$ represents the velocity field vector, and $\boldsymbol{p}$ represents the pressure. The equation is then solved numerically by first solving the advection step, then the Poisson equation is solved for the pressure $\boldsymbol{p}$. Then finally projecting the velocities into a divergence-free field[8].

2.4.3 Particle updates, collision detection, and rendering

The falling snow particles are affected by the wind velocity and gravity, and their positions are updated for each frame. The wind velocity is found by doing a lookup in the wind velocity texture. The acceleration of the snow particle is given from the drag (The difference between the velocity of the wind and the snow particle), and the gravity affecting the particle. A circular velocity is also added to each snow particle, to make it spin around its own axis.

For each frame, the simulation checks if the snow particles have collided with the terrain. If any snow particle has a position lower than that of height of the heightmap at that coordinate, then the snow particle is reset, to a random location on the top, and the snow height at that position is increased[8]

The rendering of the falling snow particles are done through OpenGL and GLSL shaders. The snow particles are rendered as a points with varying size and a texture. The snow simulator is set-up so that CUDA, and OpenGL share buffers through the CUDA-OpenGL interop interface. By sharing buffers through interop, the contents, like the position of all of the snow particles and the terrain, need not be copied up from the GPU device to the host, and then down to GPU device again for each frame. The different CUDA kernels updates the positions of the snow particles. These changes are then visible to OpenGL, which can then use the updated buffers directly when rendering, this allows for very fast real-time rendering.

2.5 Terrain Rendering

A number of techniques exists for rendering terrains, they can roughly be classified into three different categories: regular grid or hierarchical algorithms, that recursively divides the terrain data using a common structure such as binary trees or quadtrees. Triangulated irregular networks that represents the terrain surface through a polyhedron with triangular faces, or the relatively new way of rendering terrain through the use of voxels. As the graphics hardware have become faster and faster changes to terrain rendering algorithms from CPU based to GPU based implementations have become more and more common, and the latest state of the art methods are fully or almost fully GPU based implementations.

2.5.1 Regular Grids

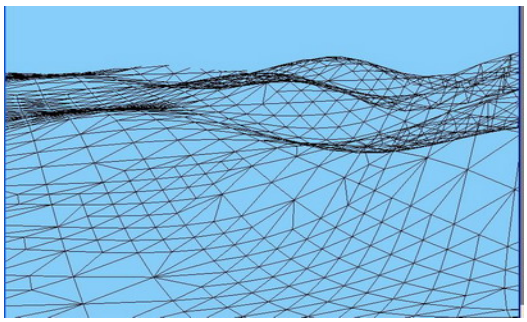
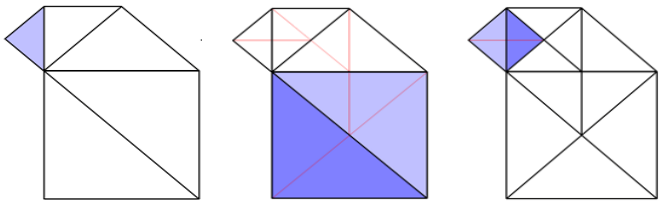


Figure 2.8: Wireframe representation of a ROAM triangulation

ROAM (Real-time Optimally Adapting Meshes) is a well known hierarchical rendering scheme, using a binary tree in the recursive division of the terrain[6]. ROAM triangulates the terrain grid into right-angled isosceles triangles. Each of these triangles can then be divided into two new right-handed isosceles triangles. The triangles may also be merged, if needed. The resulting triangulation is then stored in a binary tree. Two priority queues are used to control which of the triangles are in need of a split or a merge. Care must be taken when splitting triangles, to avoid T-junctions. Figure 2.9(b) shows how a T-junction might be formed. T-junctions are triangles that form T-shapes. This introduce potential cracks in the terrain when rendered, and are therefore undesirable. When a ROAM triangle split introduces a T-junction, a forced split on another triangle is done, to avoid T-junctions. Figure 2.9 shows how a forced split might occur. Since ROAM does most of its work on the CPU, it is considered an outdated technique om the new era of powerful GPUs.



(a) Split (b) Forced splits (c) Complete triangulation

Figure 2.9: Shows how a split can lead to a T-junction (a) Shows how the T-junction is removed by splitting more triangles (b) Shows the complete triangulation (c)

Geometry Clipmaps is another terrain rendering technique based on using regular grids. It was presented by Frank Losass and Hugues Hoppe in the paper, Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids [12]. Their method focuses on doing as much as possible of the processing on the GPU, only falling back on the CPU when absolutely necessary. This is one of the reasons they are able to achieve the high-performance benchmarks for large terrain sizes, with a performance of about 1200 FPS for rendering 450,000 triangles on a ATI 5870 graphics card.[12] Geometry clipmaps rely on using a set of nested regular grids centered around the viewpoint, where grids closer to the viewport are smaller, but have a high detail level, and grids further away are larger, but with reduced detail. As the viewpoint moves around, the grids follow and are updated. The vertices of the clipmaps are stored in toroidal arrays, which allows the buffers to be updated incrementally[3]. These incremental updates to the buffers are one of the few operation done on the CPU. However, an improved implementation using vertex texture, is outlined in *GPU Gems 2*, by Asirvatham and Hoppe [2]. By using vertex textures increased frame rates, and reduced processing time were achieved. [2].

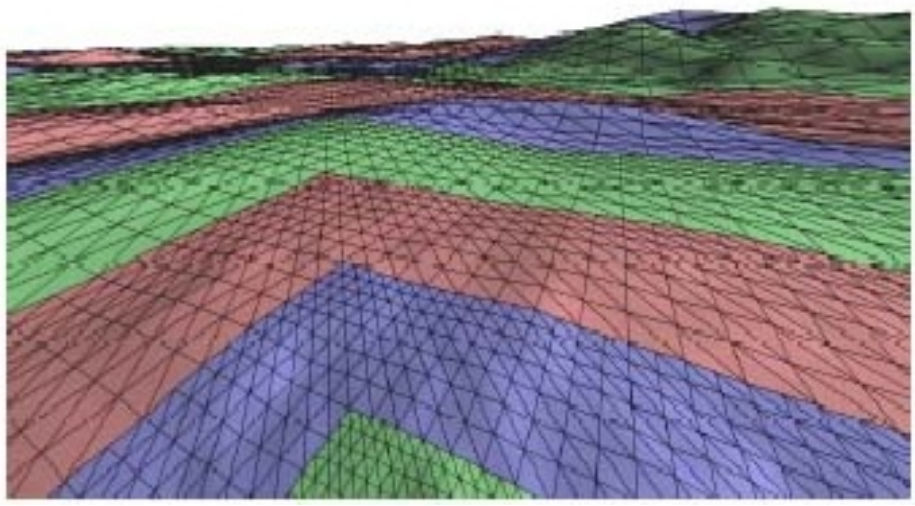


Figure 2.10: The regular grids used in Geometry Clipmaps, the colors of the grid represent the detail level ©Microsoft Research

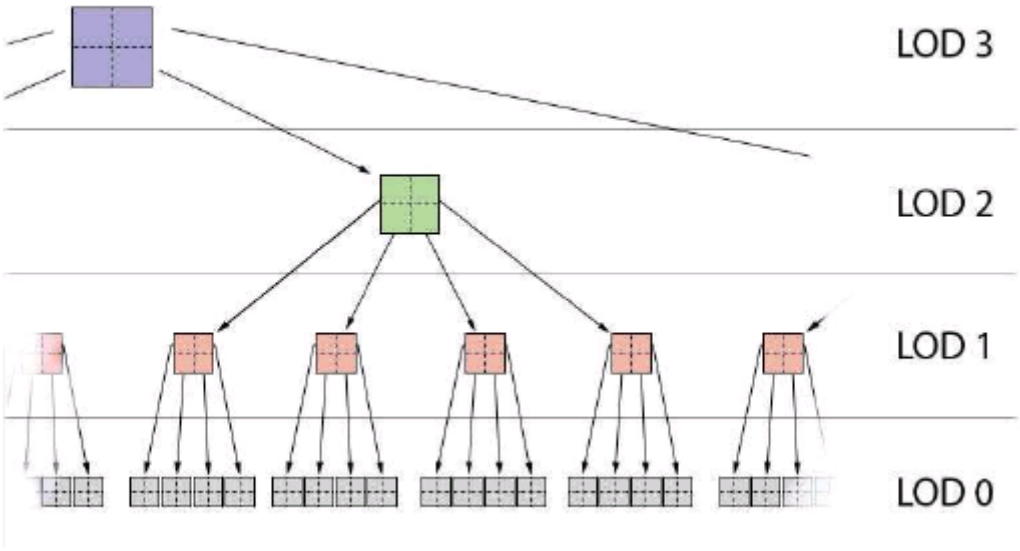


Figure 2.11: Quadtree LOD selection

Continuous Distance-Dependent Level of Detail (CDLOD) is another GPU-based terrain heightmap rendering implementation.[19] It is based around using a quadtree of regular grids. The main focus of the implementation is improving the LOD function from previous implementation. The heightmap is organized into a quadtree, this quadtree is then used to select the appropriate level of detail. At run-time nodes of the quadtree are selected rendering, the depth of the selected nodes, gives the level of detail. For each added depth level in the quadtree, the detail of the rendered node is increased by a factor of four. Effectively this gives a completely continuous LOD function, that provides smooth and accurate results[19]. CDLOD produces a higher quality triangle distribution and utilization, and rendering frame rate than the improved geometry clipmap, presented at the end of last paragraph, but with a n added memory cost for storing the quadtree. [19]

2.5.2 Our Terrain Rendering Approach

This thesis will present a fully GPU-based terrain rendering technique that builds upon ideas of both CDLOD and geometry clipmaps. It uses a fixed grid mesh, that is displace in the vertex and tessellation shader. The quadtree structure of CDLOD is completely removed, there is no utility structure operating on the CPU side, everything is done on the GPU. The level of detail

function uses the idea that the LOD should be completely predictable, and as such the selection of detail level depends only on the distance of the viewpoint to the primitive. The LOD function is a continuous function, and decides the tessellation levels used in the tessellation shader. The technique is intended to render heightmaps that are no larger than the maximum allowed textures size of the graphics hardware used. This is one of the main shortcomings of the implementation, but as the snow simulator is unable to handle larger sizes, without degradation of simulation quality, it is acceptable. Changes to the implementation, so that it can handle larger terrain, are discussed in Chapter 5.2. The implementation requires graphics hardware that have shader model 5.0, with hardware tessellation capabilities.

2.6 Noise

Noise is often used to generate procedural content, such as fire texture, naturally looking grass, water and rock textures, as input to a city generation application[10]. One of the most famous noise generation functions for procedural content generation is the Perlin noise function, and its derivatives.

2.6.1 Perlin Noise

Perlin noise is a mathematical function for generating coherent pseudo-random gradient noise. Since it is coherent, there are no discontinuities when moving from one point to the next. Perlin noise can be generated for multiple dimensions, and the algorithm complexity of the function is $O(2^n)$, where n is the number of dimension. Perlin noise was first described by Ken Perlin in the paper "An image synthesizer" in 1985 [16].

Math and Algorithm

To explain how Perlin noise is generated, an example of 2D Perlin noise generation is presented and explained. Given a Perlin noise function:

$$pnoise2D(x, y) = z \tag{2.3}$$

where x, y, z are real numbers. The noise function can then be defined as a regular 2D grid, where each whole number defines a point on the grid. now

given four grid-points (x_0, y_0), (x_0, y_1), $(x_1, y0)$, (x_1, y_1). Figure 2.12 shows the grid-points.

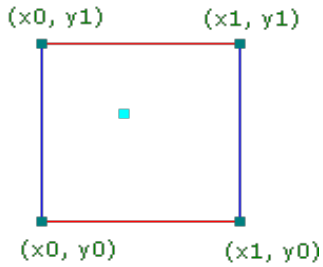


Figure 2.12: The grid corner points

And a function:

$$G(x, y) = (g_x, g_y) \tag{2.4}$$

Which takes as input a grid point and returns a pseudo-random gradient vector with a length of 1. For each of the four given grid points a vector P going from the grid-points to the (x, y) point, are generated. Figure 2.13 Then the influence of each gradient is computed as the dot product between the gradient and the associated P. Figure 2.14 shows the resulting influence vectors.

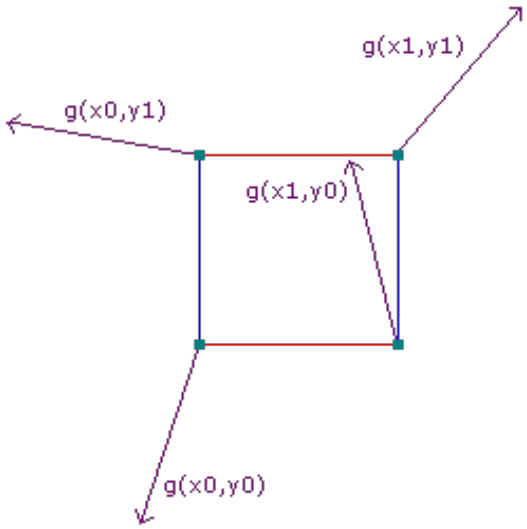


Figure 2.13: The gradients

$$s = g(x_0, y_0) \cdot P_0 \tag{2.5}$$
$$t = g(x_0, y_1) \cdot P_1 \tag{2.6}$$
$$u = g(x_1, y_0) \cdot P_2 \tag{2.7}$$
$$v = g(x_1, y_1) \cdot P_3 \tag{2.8}$$

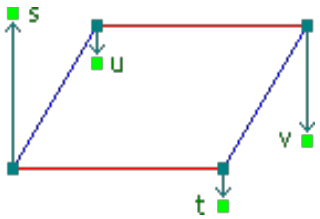


Figure 2.14: The resulting influence vectors

These four values are averaged together using a weight for each value. A blending function is used to give weight, and make the (x, y) point seem closer or further away from the actual grid-points. The blending function first proposed by Perlin was the Hermite blending function:

$$P = 3t^2 - 2t^3 \tag{2.9}$$

This was later revised by Ken Perlin[**?**], and a fifth polynomial blending function was proposed as a replacement. Figure 2.15 shows the graph of both blending functions.

$$P = 6t^5 - 15t^4 + 10t^3 \tag{2.10}$$

The improved blending function is very similar to the first blending function, having a zero first derivative at the end point, but with the added bonus of also having a zero second derivative at the endpoints.This makes the second derivative continuous everywhere in the function. Using the second blending function the value returned from *pnoise2D* then becomes:

$$\begin{aligned} P(t) &= 6t^5 - 15t^4 + 10t^3 \\ v_{x0} &= s * P(x - x_0) + u * (1 - P(x - x_0)) \\ v_{x1} &= t * P(x - x_0) + v * (1 - P(x - x_0)) \\ v_{xy} &= v_{x0} * P(y - y_0) + v_{x1} * P(y - y_0) \end{aligned}$$

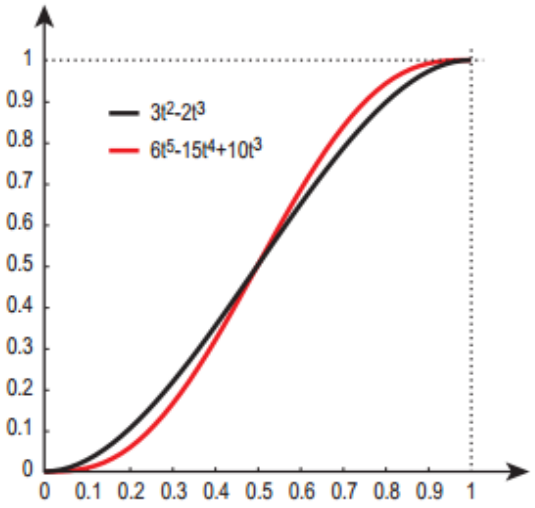


Figure 2.15: The blending functions

v_{xy} is the noise value returned from the $pnoise2D(x, y)$ function.

2.7 Use of Perlin Noise

In our implementation Perlin noise is used to create detailed heightmaps with flowing mountainous terrain. Perlin noise gives the perfect combination of realistic presentation of the terrain needed, and variabiliity of the generate terrain.

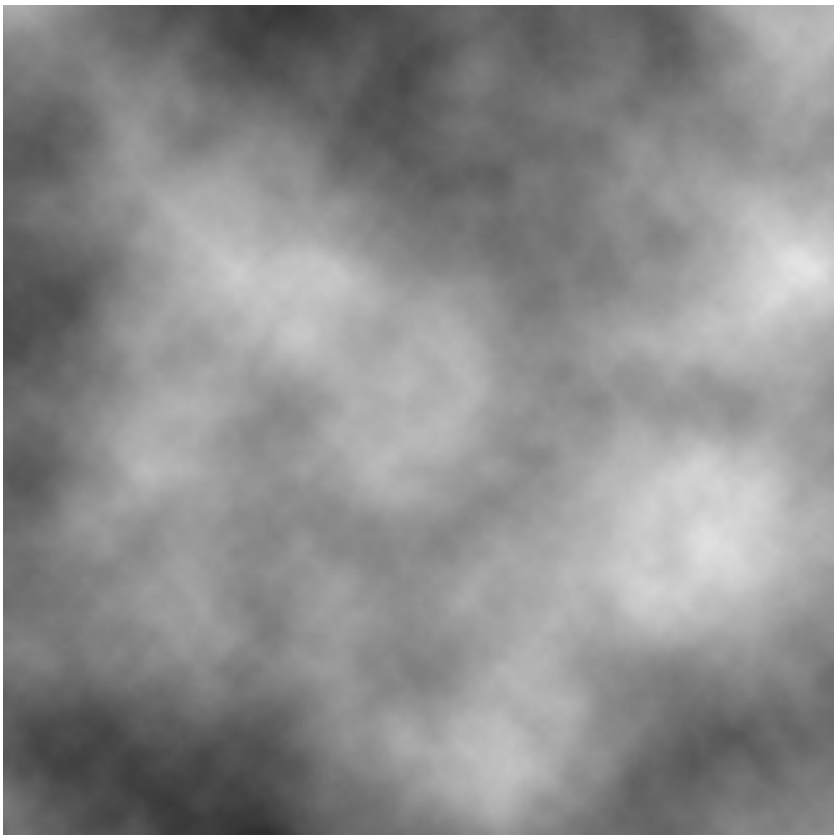


Figure 2.16: Example of generated Perlin noise

Chapter 3

Implementing and Optimizing Terrain Rendering

This chapter will describe how the tessellation based terrain rendering pipeline is implemented, how noise is used to increase the realism of the terrain rendering, how the tessellation based terrain rendering pipeline is integrated with the HPC-lab snow simulator and the changes that need to be made to make it function.

3.1 Terrain Rendering

The platform chosen for the implementation is OpenGL using GLSL 4.2 shaders. The reason for this choice is that our snow simulator is built using OpenGL and CUDA. Changing over to DirectX for rendering is there not a choice. Although a DirectX rendering pipeline, could be considered for a future project, if only to enable the use of quadbuffers for 3D stereo rendering, without using a NVIDIA Quadro grahics card.

3.1.1 Terrain Representation

The terrain is represented as patch of flat quads, which is generated at the start of the application. The patch is then added to a vertex buffer, and stored in the GPU memory. The patch contains a grid of 11×11 quads. An index buffer is also used, but instead of using a similar indexing scheme as

Eidissen did in[7], each vertex of the quad is index explicitly, since this is required when using tessellation shaders with quad tessellation [13]. Since only 121 quads are stored in the VBO, and each quad consists of four vertices, the amount of memory required on the GPU to store the terrain geometry is reduce considerably, and although the heightmap image must be stored as a texture now, this again is offset by the fact that a normal map texture is no long required, since normal can be calculated directly from the heightmap in the shaders. Compared to Eidissen's approach of storing the full triangulation in a VBO, the reduced memory footprint is quite considerable for larger sizes of terrain. The implementation takes advantage of instanced based rendering capabilities of OpenGL, where multiple instance of the same geometry can be rendered without submitting new geometry to the rendering pipeline. The vertex shader takes care of displacing/moving the vertices based on an instance id provided to the shader by the OpenGL pipeline.

3.1.2 LOD-function

One of the main drawbacks of how the terrain rendering is done in Eidissen implementation is that the terrain has the same detail level, whether the viewer is close to the triangles, or further away. But the detail level of terrain that is far away could be reduced, without the view even noticing. One way this could have been done is to make triangles far way represent larger terrain geometry, than triangles close to the viewpoint. The new terrain implementation utilizes a precise and continuous level of detail function. The only deciding factor for the level of detail is the distance from the viewport, this means that terrain closer to the viewpoint is tessellated more, and therefor have a higher detail level than terrain which is far way. The LOD selection function is implemented in the tessellation control shader. Listing 3.1 shows the LOD selection function. The input to the function is a position, and the returned value is the selected tessellation level, which corresponds to the LOD.

Listing 3.1: The level of detail selection function

```
float lod(vec3 pos) {
    vec3 distance = cameraPos.xyz-(pos.xyz*0.5);
    distance *= 0.15;
    float d = LOD-clamp(pow(length(distance), 0.57),0.0,LOD-1);
    return d;
}
```

3.1.3 Tessellation

Both the tessellation control and evaluation shader are used in the implementation.The tessellation evaluation shader contains code for calculating the LOD level and removing patches that are outside of the view (frustrum culling). Listing ?? shows the complete tessellation control shader code.

Listing 3.2: The tessellation evaluation code

```
float lod(vec3 pos) {
    vec3 distance = cameraPos.xyz-(pos.xyz*0.5);
    distance *= 0.15;
    float d = LOD-clamp(pow(length(distance), 0.57),0.0,LOD-1);
    return d;
}
```

The evaluation shader contains code for displacement of the vertex based on the terrain height and for calculates attributes such as texture coordinates, and depth. Listing ?? shows the complete tessellation control shader.

Listing 3.3: The tessellation evaluation shader

```
float lod(vec3 pos) {
    vec3 distance = cameraPos.xyz-(pos.xyz*0.5);
    distance *= 0.15;
    float d = LOD-clamp(pow(length(distance), 0.57),0.0,LOD-1);
    return d;
}
```

3.1.4 Rendering

Since the geometry contained in the VBO sent for rendering contains only flat patches, a method for adding height and geometry detail to these patches is needed. Vertex textures and displacement mapping provides the required functionality to make this happen. Vertex texture is a feature that was added with shader model 3.0. It adds texture sampling capabilities to shaders other than the fragment shader. Before SM 3.0 only the fragment shader had access to texture sampling capabilities. Since the introduction, vertex texture fetching hardware has only improved, with more new modern graphics hardware. Listing

Vertex textures are often used to provide displacement mapping. In the case of heightmap rendering, the height of the terrain can be sampled in the vertex shader, and then the vertex could be displaced by changing the

position of the vertex. With the advent of tessellation shaders, displacement mapping and vertex textures can also be done in the tessellation stage, this functionality is utilized in the implementation to displace the new tessellated primitives to the correct height sampled from the heightmap texture.

3.1.5 Texturing

To texture the terrain, three base texture are used. One for grass, one for mountain terrain, and the last for the snow cover. The terrain is then textured base on the height, slope and the height of the snow cover. The blending function used is very simple and could be improved to create even better texturing. Figure 3.1 shows the new texturing scheme. Listing **??** shows the complete fragment shader, used to texture the terrain.

3.2 Noise Implementation

A variation of the implementation presented by Simon Green in *GPU Gems 2*[2], is used as the base noise generating function for our implementation. Noise is generated on the GPU by rendering the output from two perlin noise shaders to framebuffers. The content of the framebuffers are then copied to textures. These texture can then be used later, for instance as a base for a heightmap. The solution of pre-generating the noise texture at the start of the program, and then storing them in memory for later use, decrease the computational power that must be expended on doing costly noise generation in real-time. The noise can now simply be sampled as a texture. The downside by storing the noise texture in memory, is that there are limits to how large and how many textures can be stored in memory on the graphics card.

3.2.1 Blending Noise

To create more interesting noise, the base function can be calculated more than once, with varying input, and then blending the results. This creates noise that can have lots of different properties.

TODO:EXAMPLES.

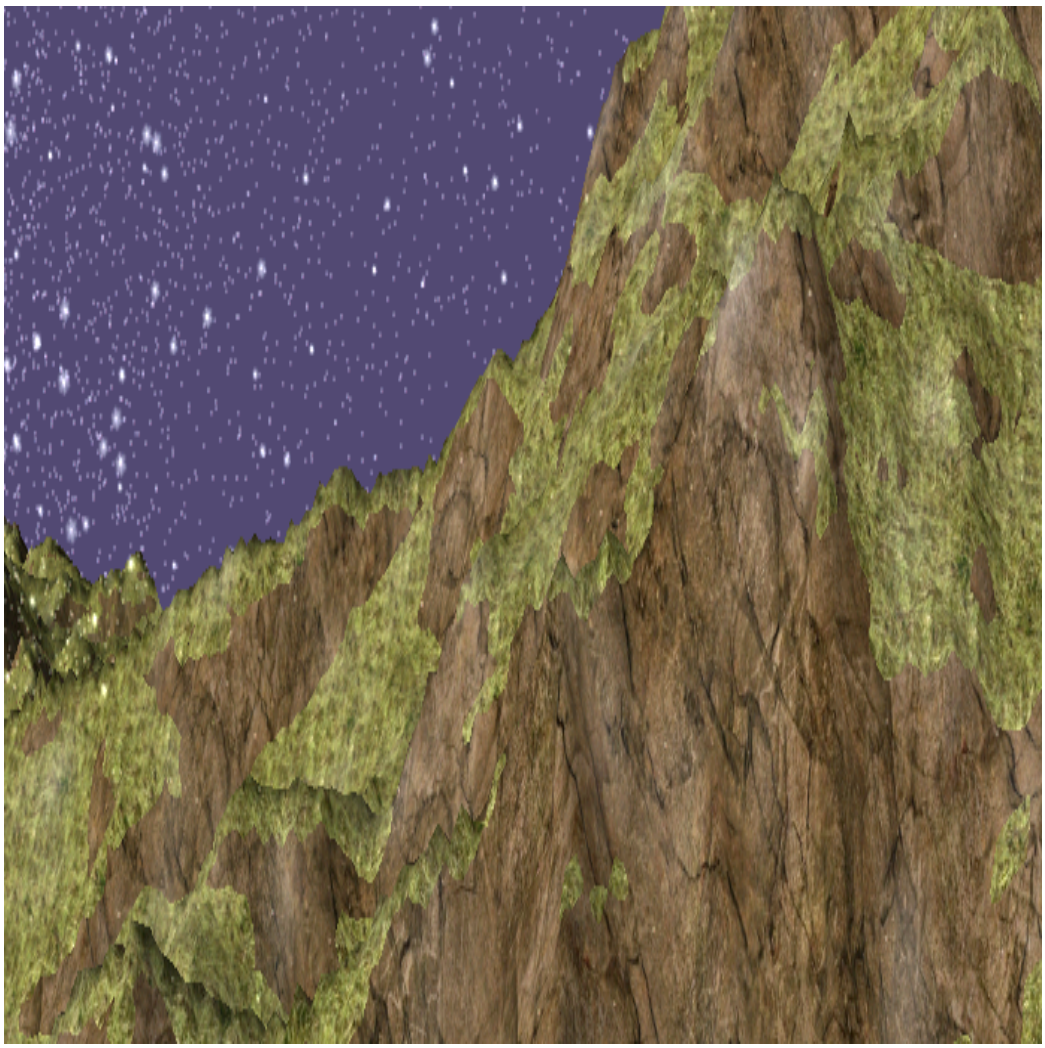


Figure 3.1: The new texturing scheme

3.3 The Snow Simulator

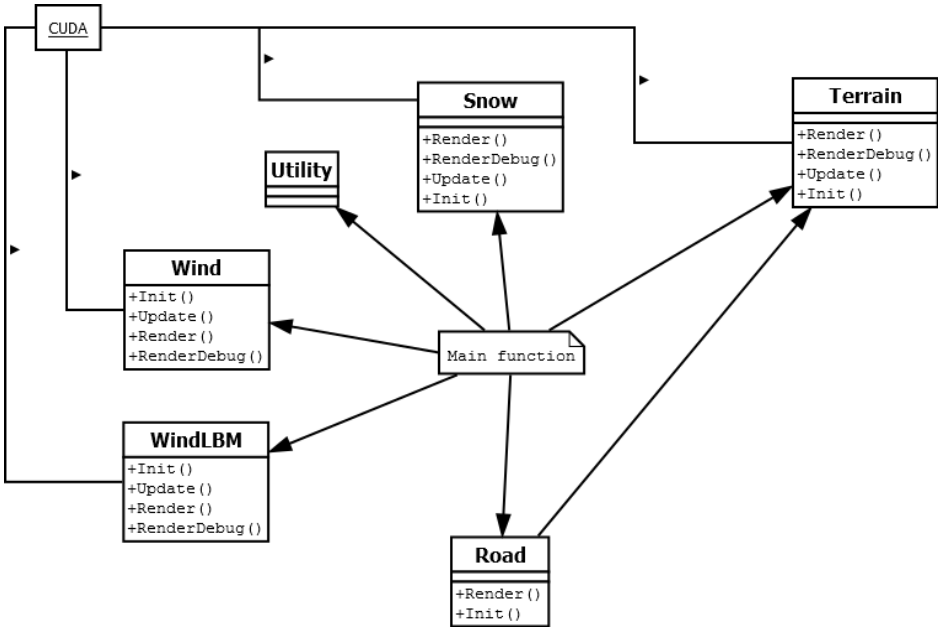
Parts of the snow simulator are outdated, and needs to be updated to use current state of the arts techniques. To enable the new terrain rendering pipeline to function with the snow simulator, some of the CUDA code needs to be updated. The old terrain rendering pipeline is also in need of a face lift to adequately perform using the updated CUDA code.

3.3.1 Overhaul

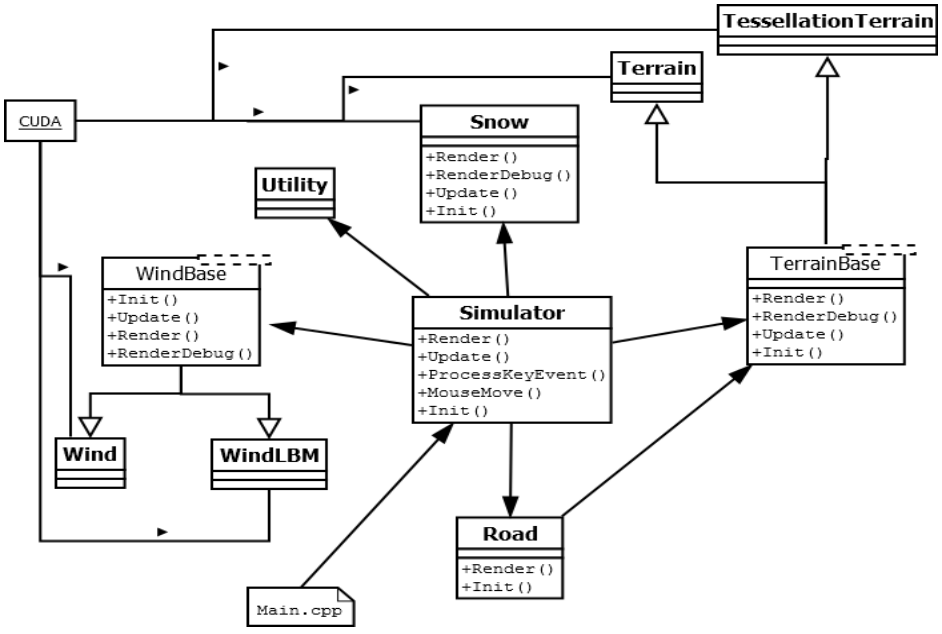
The code base of the HPC-lab snow simulator has steadily increased in size over the past few years, as new master students have used it as a basis for either their thesis or fall projects. As more and more functionality is added to the simulator, the complexity increases. To better handle current and future projects, a overhaul of the structure of the code is necessary. Figure 3.2(a) shows the class diagram of the old code. The classes controlling the wind simulation, Wind and WindLBM, are two separate classes, even though they have functions that are conceptually the same, there is no code reuse between them. The lack of an abstract base class to hold functions that are shared between all inherited classes, really shows in the main loop of the program. The main loop is cluttered with with branching function calls depending on what kind of wind simulation should be done. Figure 3.2(b) shows the new and updated class diagram of the snow simulator. The wind, terrain, snow classes are now abstract classes, and concrete classes that inherits from these base classes are added. A simulator class is added to control the simulation. It instantiates the correct subclass for the snow, wind, and terrain, and polymorphism will take care of calling the correct virtual function, in the instantiated class. The main function contains the code for creating a OpenGL window, and instantiating the simulator class, and running the main loop. The main loop calls the update function in the simulator class, and then the update function of the simulator class, passes this call on to any simulation part (wind, snow, terrain) that needs it. Likewise, when it is time to render, the main function calls the appropriate function in the simulator class, which passes this on. The change from using normal function calls, to virtual function calls is expected to decrease the by a tiny amount [5], but the performance decrease is also expected to be offset by the removal of the branches in the main loop of the program, and even if that is not enough, the time spent executing these virtual functions are minuscule compared to the rest of the program. The utility label on the class diagrams Figure 3.2(a) and Figure 3.2(b) contains utility classes, e.g. a shader class, responsible for loading, compiling, and linking shaders, different configuration classes, and cuda helper classes.

3.3.2 Shader overhaul

Many of the shaders that are used in the snow simulator were created using GLSL 1.2. This poses a problem now that the simulator is being overhauled, so all of the old shader code had to be updated to current standards. All of



(a) old class-diagram



(b) new class-diagram

Figure 3.2: Class diagrams of the old simulator structure (a) class-diagram of the new simulator structure (b)

the new shaders can be seen in the Appendix. The vertex shader for the snow and terrain are supplemented with access to the projection matrix and view matrix from the camera class of the simulator, since the deprecated matrix functions for OpenGL and GLSL were removed.

Because of the changes to the CUDA code, detailed in the next section 3.3.3. Displacement mapping is added to the vertex shader of the terrain. The heightmap texture is sampled for the height of the terrain and the height of the snow cover at that position. The returned sample vector, contains the height of the terrain in the x, position of the vector, and the height of the snow cover in the w position. The snow height is then used as a blending factor between the base texture, and the snow cover texture in the fragmentshader. The fragment shader for the terrain is also update with multitexturing, using splatting. The slope of the terrain is used as a procedural texturing technique. The height and the slope of the terrain are used as blending factors, between a grass texture, and a rock texture, to create a more realistic look.

3.3.3 CUDA Changes

Geometry Collision

The new terrain rendering pipeline makes the old geometry collision detection code obsolete. Since the full geometry of the terrain no longer exists in a VBO, the CUDA code checking for collision between snow particles and the terrain, will fail. Eidissen suggested an improved collision detection method[?], but was not able to implement it because of time constraints. The method uses the heightmap texture directly, to do height comparison between the snow particles and the terrain. This way the uncoalesced reads of the terrain buffer is removed, and the texture memory cache can be exploited to increase the performance of the collision tests. The heightmap texture is shared between CUDA and OpenGL, since it is also used to do displacement mapping in the OpenGL shaders. The snow particle re-spawning, that happens when the snow particle is lower than the terrain height, is unchanged.

Listing 3.4: The new collision detection code

```
float lod(vec3 pos) {
    vec3 distance = cameraPos.xyz-(pos.xyz*0.5);
    distance *= 0.15;
    float d = LOD-clamp(pow(length(distance), 0.57),0.0,LOD-1);
    return d;
```

```
}
```

Updating the Obstacle Field

Another part of the CUDA code that is changed, is how the obstacle field is updated. Because of the changes to the terrain rendering, the full geometry is not present in any buffers on the GPU. So the code that handles the updating of the obstacle field must use the heightmap, since snow build up and terrain geometry is only reflected on that texture. It is still done on the CPU, but this could possibly be changed, to make it even faster. Updates of the obstacle map is still done at one step per 0.1 seconds. The updating cycle is now:

1. Copy the heightmap texture from the GPU. (Changed)
2. Zero local obstacle map memory.
3. Set *self bit* of each cell to 1 if below terrain, 0 otherwise.
4. Set the remaining 26 bits to 1 or 0, according to which neighbors are obstacles.
5. Copy the obstacle map to the GPU

Chapter 4

Results

In this chapter, results from the performance benchmarks of the terrain rendering implementation and the updated snow simulator implementation will be presented and evaluated. Performance characteristics and scaling properties of the rendering implementation and snow simulator, under different configurations and input will be examined. Lastly the visual results achieved will be presented and evaluated.

4.1 Test Setup

To test the performance of our terrain rendering method, two system were selected, Table **??** shows the specifications of those test systems. The GT 520 card was chosen since it is a low end consumer card, this enables us to show how the performance of the terrain rendering will be affected by lower hardware specifications. The Tesla card was chosen to give feedback on how the rendering method scales with better graphics hardware. The other card was a high end NVIDIA Tesla C2070.

4.2 Terrain Rendering Benchmarks

Two different benchmarks are performed, on where the viewpoint during the rendering is static, the other one where the viewpoint move with a predetermined path.

4.2.1 Static Viewpoint

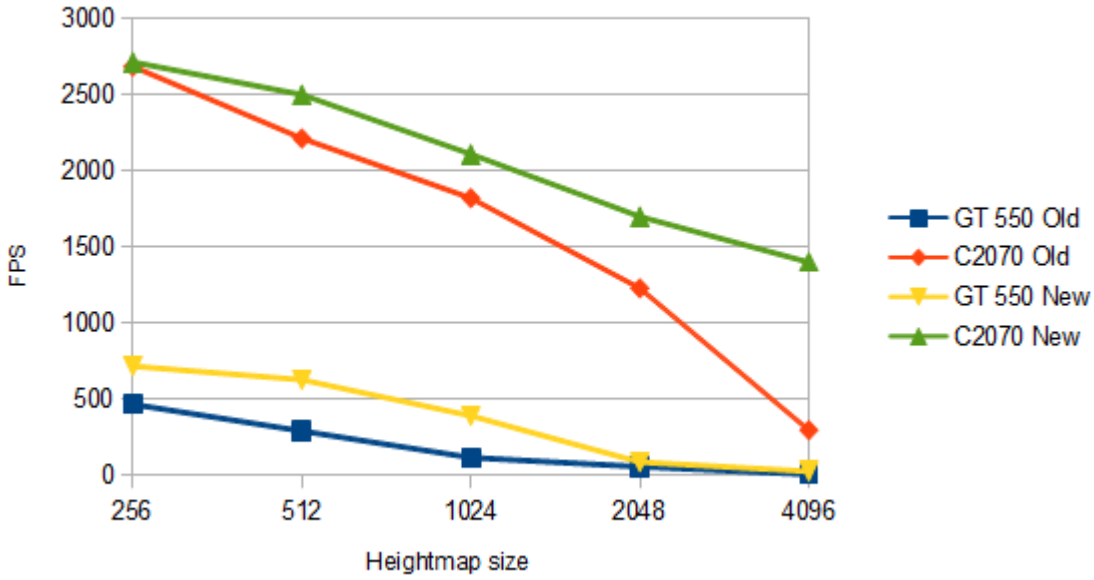


Figure 4.1: Frame rate for static viewpoint

Benchmarks

Discussion

When the viewpoint is static, the whole terrain is visible, and the culling effect of the our new implementation thus does not come into effect. As we can see from Figure ?? when and the map sizes are small, the old brute force approach is not too far behind our new method. However, as the map sizes increases the old brute force approach quickly falls off. The reason for this is that when the terrain is small the LOD of our implementation, is not able to fully reduce the complexity of the terrain as it is too small. The LOD function works on distance and when the distance is small the effects of using LOD become unnoticeable.

The effects of the LOD function becomes even more apparent with the lower end consumer graphics card. As can be seen from Figure 4.2. The increased frame rate of our approach becomes more important faster since the lower-end graphics card is unable to process larger quantities of triangles.

4.2.2 Moving Viewpoint

Benchmarks

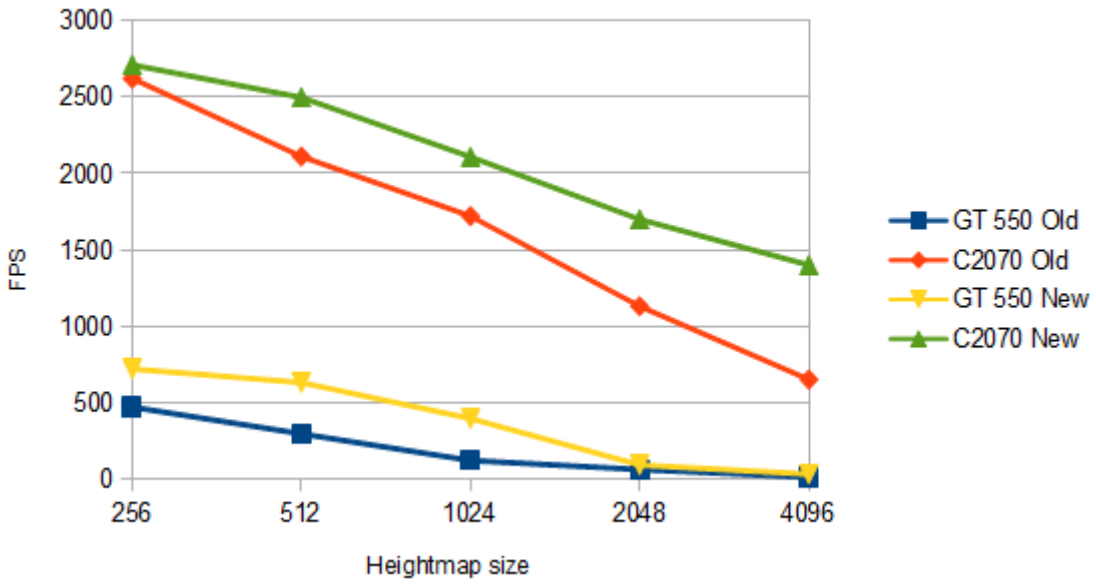


Figure 4.2: Frame rate for moving viewpoint

Discussion

When the viewpoint is moving around through the terrain the effect of culling away quads that are outside the view, become apparent very fast. From Figure ?? we can see that for small map sizes the brute force approach is still not to far behind, since there just is not enough detail in the terrain. But while the brute force approach frame rate is reduce almost as fast as in the static viewpoint test, our new method falls off much slower. This is the effect of culling away quads that are outside the view, it greatly reduces the amount of work the GPU has to do each frame, since none of the quads that are culled, needs to be tessellated.This shows how important it is to remove unneeded primitives from being sent down to the graphics unit without even being rendered in the end.

4.3 Kernel Profiling Comparison

Time constraints prevented us from doing a full comparison of the update CUDA code, but the preliminary results were promising. Results from the CUDA visual profiler shows, that the particle update, where the collision detection code resides, have been reduced, but a comparison that used equal hardware as Eidissen did is not done. But the texture cache hits were a bit high still. An interesting notes, is that we can see from the time distribution that the distribution have changed from what Eidissen reported in **??**. This either means that the new generation of hardware have different characteristics and therefore the distribution is different, or someone have performed updates to the CUDA code, without reporting it. This poses a problem since if there is no report about changes done do the simulator then it become difficult to accurately compare implementations and improvements. All code changes should therefore be documented adequately, either by writing a report about the changes, or documenting changes with a comment.

4.4 Visual results

Large detailed heightmaps can be rendered while maintaining a stable frame rate. Even when the full simulation is run the frame rates are still acceptable even with the largest map sizes. Figure **??** shows a rendering of a very large heightmap, and Figure **??** shows the same rendering with a snow cover. Perlin noise can create very realistic looking terrain. Figure **??** shows a rendering of a terrain created by Perlin noise and Figure **??** show the same terrain with a snow cover.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

The goal of this thesis was to improve the HPC-lab snow simulator by improving the rendering of the terrain, a major bottleneck in the last version. A new terrain rendering pipeline was added to the simulator, capable of rendering large and detailed landscapes, with good performance. In addition, introducing generated Perlin noise combined with premade heightmaps, increased the realism of the simulation. The texturing of the terrain was also improved, by using multiple texture and blending them together based on the slope and height of the terrain, immersing the viewer.

The improvements made to the code structure of the simulator, proved successful, although they did not directly give a performance increase. However, they provide useful abstractions, that give future students working on the simulator a better structure to work with. The cleanup of old shader code, and the removal of all old fixed function OpenGL code, gave the simulator renewed life, and a more stable graphics code base to build upon.

The changes to the CUDA geometry-particle collision detection code, proved successful, reducing the time spent on collision detection significantly. There could even be further improvements when combined with particle sorting to further exploit the texture caching present within texture memory of the GPU.

The visual quality achieved with the new terrain rendering pipeline, in combination with the simulation of wind and snow particles, provides the simulator with a very convincing presentation of landscapes, ranging from snowy

mountainous terrain to flat grassland.

5.2 Future Work

The novel terrain implementation presented in this thesis, functions well for relativly small map sizes, (256-256 - 8192x8192), but for anything larger smarter solutions could be considered. Combining the finding here with solutions presented in the background chapter **??**, could provide solutions giving both the performance and visual quality needed for realistic rendering of huge terrain.

To facilitate larger terrain, the wind simulation code would have to be improved. One way to possibly improve the wind simulation code could be to apply level of detail to the wind simulation voxels, so that voxels that are further away are combined to one, or something like the solution presented in [15]. Where a novel framework for automatically simplifying the dynamics computation of particle system is presented.

Bibliography

[1] *NVIDIA CUDA Programmning Guide 4.0.*

[2] *GPU Gems 2.* Addison-Wesley, 2005.

[3] Nick Brettell. Terrain rendering using geometry clipmaps. Master's thesis, Cosc, Canterbury, ?

[4] Joel Chellia. The ntnu hpc snow simulator on the fermi gpu. Master's thesis, Norwegian University of Science and Technology, Department of Compute Science, 2010.

[5] Karel Driesen and Urs Holzle. The direct cost of virtual function calls in c++. *SIGPLAN Not.*, 31(10):306–323, October 1996.

[6] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. Roaming terrain: Real-time optimally adapting meshes, 1997.

[7] Robin Eidissen. Utilizing gpus for real-time visualization of snow. Master's thesis, Norwegian University of Science and Technology, Department of Computer Science, 2009.

[8] Aleksander Gjermundsen. Lbm vs sor solver on gpus for real-time snow simulations. Master's thesis, Norwegian University of Science and Technology, Department of Compture Science, 2009.

[9] The Khronos Group. http://www.khronos.org. last accessed.

[10] Praveen Kumar Ilangovan. Procedural city generator. Master's thesis, Bourne mouth, .

[11] Hallgeir Lien. Procedural generation of road for use in the snow simulator. Specialization Project. Department of Computer Science NTNU Trondheim Norway.

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

[13] Kurt Akeley Mark Segal. The opengl graphics system: A specification (version 4.0 (core profile)), March 2010.

[14] NVIDIA. Isc-briefing-sumit-june11-final, June 2011. Last accessed at 13/12 - 2011.

[15] D. O'Brien, S. Fisher, and M.C. Lin. Automatic simplification of particle system dynamics. In *Computer Animation, 2001. The Fourteenth Conference on Computer Animation. Proceedings*, pages 210 –257, 2001.

[16] Ken Perlin. An image synthesizer. *SIGGRAPH Comput. Graph.*, 19(3):287–296, July 1985.

[17] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[18] Ingar Saltvik. Parallel methods for real-time visualization of snow. Master's thesis, Norwegian University of Science and Technology, 2006.

[19] Filip Strugar. Continuous distance-dependent level of detail for rendering heightmaps (cdlod). -.

[20] Torbjørn Vik. Real-time simulation of smoke through parallelizations. Master's thesis, Norwegian University of Science and Technology, Department of Compute Science, 2003.