Incremental loading of terrain textures

Eirik Fikkan

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Theoharis Theoharis, IDI

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

Abstract

Graphic Processing Units (GPUs) are getting more powerful and are currently capable of displaying millions of polygons at a steady frame rate. The challenge is however, to display unique and interesting data while navigating in a 3D world. This task consists of both selecting which data to display, as well as storing all the required data.

This thesis focuses on efficient methods for displaying portions of large terrain textures, by loading the textures incrementally, and thereby avoid some of the storage limitations of current graphics hardware and increase the detail level of the terrain rendered. These methods include algorithms for selecting the parts of the terrain to display, loading the data in time when it's needed and present the data in an appropriate quality while maintaining performance. How the camera behaves plays a key role in these algorithms.

The process of displaying large terrains is divided into a multi step process, where coarser algorithms detect the active region of the terrain, then more accurate algorithms calculate which portions to load, and then the most precise algorithms aim to only display the necessary data. Camera movement is also taken into account in this process, and the algorithms attempt to fetch the necessary data in time before it's needed, which is a novel approach. The proposed implementation is compared to related approaches such as Virtual Texturing, a comparable methods which rely on feedback when required data is missing during rendering.

Sammendrag

Grafikkprosessorer blir stadig kraftigere og er i dag i stand til å tegne millioner av polygoner og samtidig oppnå jevn bildeflyt. Utfordringen er derimot å kunne navigere i en stor 3D-verden av unik og interessant data. Denne oppgaven befatter både å velge ut hvilken data som skal vises samtidig som at påkrevd data må lagres.

Denne masteroppgaven fokuserer på effektive metoder for å vise utsnitt av større terrengteksturer. Metodene forsøker å omgå begrensningene i dagens maskinvare ved å begrense hvilke data som er lagret, og laste inn terrengdata ved behov. Metodene omfatter algoritmer for å beregne hvilke utsnitt som er nødvendige. Algoritmene tar hensyn til kamerabevegelser, så utsnittene blir lastet i tide før de trengs.

Oppgaven med å vise større terreng deles opp i en prosess på flere steg. Grovere algoritmer skal avgrense hvilken større del av terrenget som er aktuelt, før mer presise algoritmer bestemmer hvilke utsnitt som skal lastes. Til slutt vil egne algoritmer kun tegne det som vil vises på skjermen. Kamerabevegelser brukes til å bestemme hvor store sikkerhetsmarginer med data som skal lastes inn, for å hindre at data mangler hvis kamera beveger seg. Dette er en nyvinning innen tegneteknikker. Masteroppgavens egen implementasjon sammenlignes med en teknikk som heter *virtuelle teksturer*, som er en sammenlignbar løsning på problemstillingen. Til sammenligning bruker denne metoden tilbakemeldinger fra tegneprosessen til å avgjøre når data skal hentes.

Preface

This thesis describes the project work done in the final year of my Master Degree in Computer Science, conducted in the spring of 2013 based on the research conducted in the fall of 2012.

I wish to thank my supervisor Theoharis Theoharis for advice, guidance and encouragement throughout the course of the Master thesis.

Eirik Fikkan
Trondheim, 2013-06-11

Contents

List of figures

List of graphs

Terminology and abbreviations

First-In-First-Out (FIFO) – A principle of queue processing prioritizing on the time of arrival.

Graphic Processing Unit (GPU) – The electronic component in a computer responsible for rendering graphics.

Hard Disk Drive (HDD) – A storage device based on rotating magnetic discs.

Heightmap – A 2D (image) representation of a terrain, where each pixel's intensity corresponds to the height of a point in the terrain. Heightmaps can be used for creating the entire mesh of a terrain, but is more compact in storage.

Level-of-Detail (LoD) algorithms – A general term for algorithms which adjust the visual detail level based on geometric properties such as distance or angle to the viewer.

Linked list – A data structure where each element points to the next element in sequence, until the last element which terminates the list. Linked lists is a common implementation of queues.

Megapixel (MP) – Millions of pixels.

OpenGL – An open cross-platform application programming interface for rendering of graphics, currently maintained by the Khronos Group[1].

OpenGL Utility (GLU) – A standard utility library for OpenGL available on most platforms[2].

Shader programs – Small programs running on the GPU performing vertex manipulation, fragment shading, tessellation and more. The term “shader” might be a bit confusing since shader programs can perform additional tasks today.

Solid State Drive (SSD) – A storage device which utilizes integrated circuits to store data without any moving parts.

Virtual Texturing – A representation of the visible parts of a large texture mixed together in a single texture storage.

Chapter 1

Introduction

This thesis describes methods of loading terrain texture data efficiently, to allow more realistic presentation of large terrains in great detail while maintaining performance. The following chapter describes the motivation for this project, a precise definition of the problem and the steps of solving this problem.

1.1 Motivation

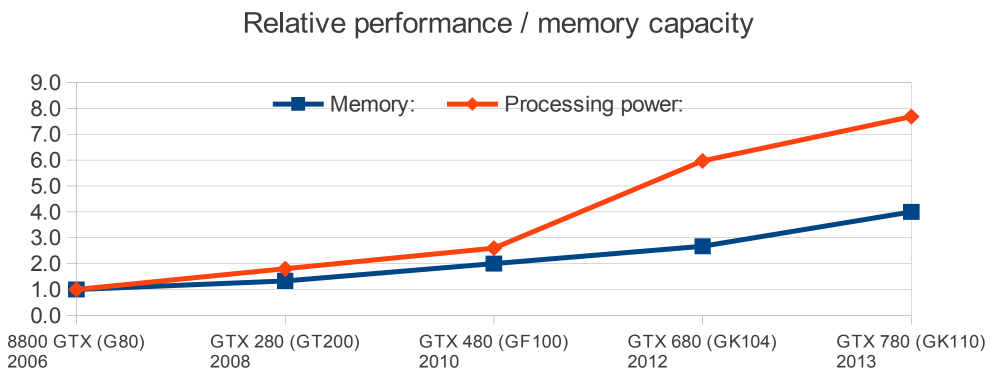
The steady development and innovations in computer hardware and visualization technologies are following the demand for improved visual quality, both to improve realism as well as allowing greater visual creativity. In addition to usage for entertainment purposes, visual quality is also increasingly important for professional use, including commercial and military simulators for training of personnel. Increased realism in simulators might significantly improve the effects of simulator training for aircrafts or military equipment.

The performance improvements of GPUs has been substantial compared to improvements in CPU performance over the last decade. Most of these improvements are related to the massive parallelization made possible by the large numbers of cores available in GPUs. Most graphic operations consists of sequences of simple operations well suited for parallel computation. Recent GPUs such as GK110 which packs 2688 cores[7], provide performance previously reserved for expensive super computers. These products are now available for consumers[1], and pack even more performance when combined with multiple units into larger configurations.

Even though the computational performance of graphics hardware keeps improving at a high rate, the utilization of this performance for increased visual detail is lagging behind in

1 GTX Titan offers 2688 cores, GTX 780 offers 2304 cores.

simulators and games. This lag is due to the limited storage capacity in graphics memory in addition to the relative high cost of transferring data to the graphics memory from external storage. For instance, a recent GPU[2] is capable of rendering over 8 million textured polygons on a HD screen at 60 Hz, giving a tremendous 4:1 ratio in polygons per pixel, which is way higher than most games and simulators are currently utilizing today. In the recent generations of GPUs there has been an increasing gap between graphics memory and computational performance:



Graph 1.1 Relative performance and memory capacity for some high-end graphics cards.[3]

As displayed in graph 1.1, GPUs offer an increasing amount of performance available compared to the amount of data stored in GPU memory. This performance can be harnessed for improving visual quality through LoD algorithms or techniques which create illusions of higher visual quality through “detail” textures[4]. Increasing the detail level of a texture a single step requires 4x the storage capacity[5], so even though the total memory capacity is increasing steadily over the years, the memory has to be used smarter to allow higher visual quality.

The time required to prepare and load meshes and textures into GPU memory from storage might take up to several minutes in simulators and games, and this is an increasing problem with higher detailed content.

2 Benchmarked with a static textured terrain running on a GTX 680 in 1920x1200.
3 Based on [6] [7] [8].
4 Addition of detail illusion through coherent noise such as Perlin noise.
5 Example: 256x256 is 4x the size of 128x128.

The last couple of years several 5-10 MP[6] monitors has entered the market. The trend is likely to continue, and Intel[30] has stated a push for higher resolutions in laptop and desktop computers in the next few years. Increasing monitor resolutions will also impact the requirements for texture details in games and simulators.

This thesis focuses only on rendering of terrain textures. However, the methods described can be adopted to be used in general rendering of 3D models, for medical visualization of organs or presentation of volumetric seismic data.

The choice of limiting the scope to terrain textures is due to personal interests for the author, in addition to experience with rendering of terrain meshes and textures. The personal motivation is enhanced by a substantial interest in natural formations in landscapes and foliage, as the author grew up in a region called Sunnmøre in Norway, a region which is famous for a stunning scenery.

1.2 Problem description

The task of this thesis is to describe and implement effective algorithms for rendering of large detailed terrains.

This process can be divided into the following tasks:

- Algorithms for determining which blocks are likely required in the near future.
- Algorithms for limiting rendering to visible parts of the terrain.
- Algorithms for loading, managing and disposal of loaded data.
- Timing and balancing of the algorithms to maintain even performance.

6 Examples: Eizo RadiForce GX1030CL 10MP, Mac Book Pro(2012) 5MP.

1.3 Solution

This thesis proposes an implementation based on three levels of storage pools:

- **Large storage:** Complete terrain stored as small blocks, available in each detail level
- **System memory:** Texture blocks from a wide spherical area around the camera
- **GPU memory:** The visible texture blocks plus a safety margin in case of rapid motion

Due to the bandwidth and latency of transferring data between the pools, the algorithms for block calculation have to be tailored for this usage.

The algorithms for loading, managing and disposal of blocks are just as important as block calculation. Disposal of blocks is required to free space for other required blocks, but as a block targeted for disposal can quickly be needed again, there should be a designed latency before the block is actually freed from the pool. The proposed method for handling of requested and disposal of blocks is through queues (FIFO). The method also has to handle situations where a block is going to be deleted but then is requested again.

The design of the algorithms can take advantage of the transfers between the large storage and system memory can be achieved independently of the GPU workload, unlike transfers to GPU memory. Transfers to the GPU memory pool have to be limited per frame, since rendering has to wait until the transfers are complete.

1.4 Thesis structure

This thesis has the following structure:

- **Chapter 2** – Describes the theory and background for this project
- **Chapter 3** – Presents related work
- **Chapter 4** – Proposed Algorithms
- **Chapter 5** – Implementation
- **Chapter 6** – Results
- **Chapter 7** – Discussion
- **Chapter 8** – Concludes the thesis, and discusses how future work can improve the described solution

Chapter 2

Theory and background

This chapter describes relevant theory and research related to the problem described in this thesis. The theory in this chapter is the basis for the algorithms in *chapter 4*, the implementation in *chapter 5*, and also serves as useful help for understanding the related work in *chapter 3*.

2.1 Viewing Frustum

A viewing frustum is a volumetric representation of the space within the camera field of view. This volume is determined by four parameters: nearest distance, farthest distance (line of sight) and field of view in X and Y dimensions[2].

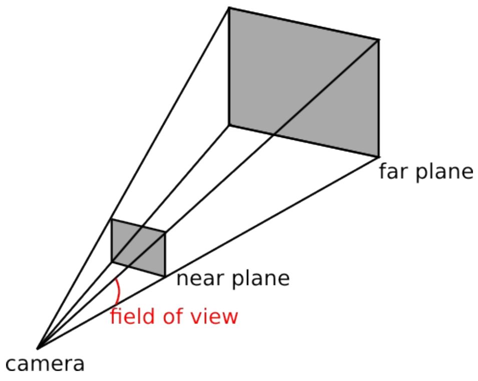


Figure 2.1 Viewing Frustum

To determine if a point is within the field of view, calculate if the point is within each of the six planes of the viewing frustum. This can be used for *frustum culling*[3], a common method of determining if an object will be visible on the screen after rendering.

2.2 Mipmap

Mipmaps are sets of textures stored in multiple detail levels[4], where each subsequent level is half the size of the above level[2]. Lower mipmap levels is usually generated from the original texture. Usage of lower detail levels on distant objects reduce the performance requirements in addition to reduce aliasing artifacts in the texture if the lower levels are blurred. Utilities such as GLU provide functions to generate mipmaps from textures.

2.3 Lockless Queue

Lockless Queues is a type of non-blocking algorithm[31] where one thread may add elements to the queue, and another may remove elements without the risk of data corruption. This is achieved through atomic operations where the producer thread prepares the complete object and then putting it in the queue with one pointer operation. Similarly the consumer thread does one atomic operation to remove the object before using it.

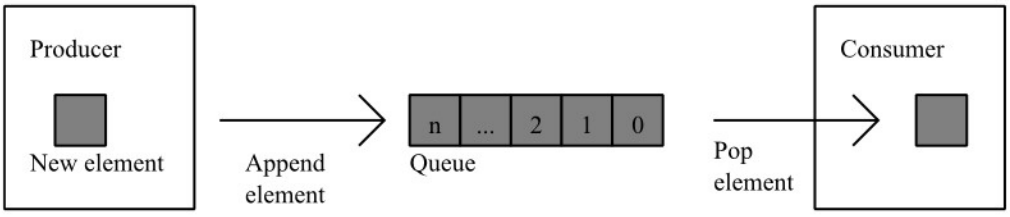


Figure 2.2 Lockless Queue

This way threads may work at different speed without the need for any synchronization, given the consumer can handle an empty queue and the queue does not grow larger than the size of the system memory.

Chapter 3

Related work

This chapter describes previous work related to this thesis.

3.1 ROAM

Early approaches to LoD rendering focused on pre-processing the mesh to reduce the triangle count while maintaining the most significant details. Real-time Optimally Adapting Meshes(ROAM)[25] is a recursive method of merging triangles depending on placement in the view-frustum and the difference between the triangle planes. Several similar algorithms[26] were designed in the 90s, when GPUs were simple processors and CPUs were relatively powerful, and relied on the CPU processing the mesh for every camera movement.

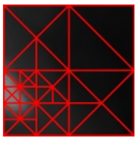
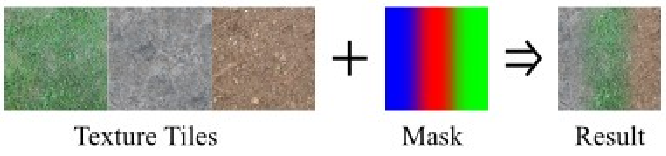


Figure 3.1 – Typical ROAM mesh

ROAM provides a very high detail per polygon, but has become obsolete as faster GPUs render continuous meshes more efficiently, and the CPU overhead is too large with higher detailed meshes. Still there have been recent implementations related to ROAM which apply similar optimizations[27], but don't update the grid depending on the camera.

3.2 Multitextured terrain (mask)

Multitexturing using a mask to blend multiple textures[28] is a simple methods used to blend several textures. Using repeating high resolution terrain detail textures, the GPU is able to create a large *"high detail"* texture using a low resolution mask texture or a heightmap. This method is very simple and effective, but offers a very limited amount of textures to blend, and creates very recognizable repeating patterns of texture tiles.



(a) *Multitexture blending*

(b) *Applied to mesh*

Figure 3.2 – Multitexturing and repeating patterns

3.3 Clipmaps

Clipmaps[18] is a common algorithm for LoD of textures and meshes which has been in use since the 90s. During a time when GPU performance were low compared to CPU performance, GPU memory were very limited and the relative cost of rendering commands to the GPU were low, this algorithm provided an effective method of displaying textures too large to fit in GPU memory.

Clipmaps stores a cutout of the texture centered around the camera at different LoD, which is assembled when rendering the mesh using the highest available texture for each coordinate.

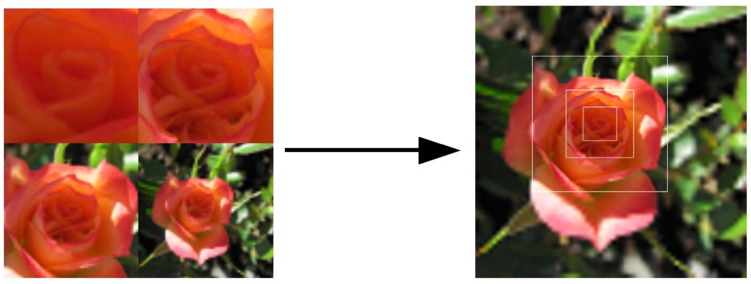


Figure 3.3 – Assembly og clipmap

When the camera moves, the clipmap has to be recalculated. New data is updated as rows or columns in the clipmap. If the new pixel is outside the borders of the clipmap, it's overflows to the opposite side of the texture, which is called *torodial adressing*[18][21]. Loading of a row can be done in a single operation as the data is stored sequential. Updating of columns does however require one operation for each pixel in the column. This causes clipmaps to be less effective on modern hardware with large texture sizes.

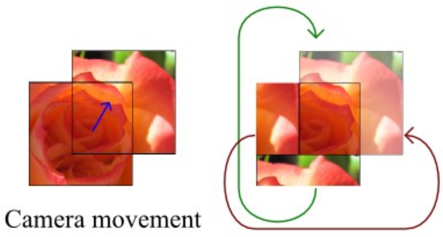


Figure 3.4 – Camera movement and clipmap (torodial addressing on the right)

Geometry clipmaps[20] utilizes the same principle of clipmaps applied to meshes, but does require some method of interpolating the vertex coordinates at the border of two detail levels. Geometry clipmaps suffers from the same issues as textures, as uploading of data with stride is increasingly ineffective on higher detail levels.

Clipmaps may suffer from a lot of data overhead, where typically 60-75%[7] of the data in the clipmap is not in use. In addition, as the camera moves in one direction, it still needs to fetch lower levels of the sections it's moving away from, as clipmaps loads data in a 360 degree angle. The usage of clipmaps may also cause visible artifacts such as sharp corners, as the shape of the clipmap is squared.

Closely related to clipmaps are *geometrical mipmaps*[29], which works similar to clipmaps but loads small blocks instead of single rows or columns of textures and meshes. This causes geometrical mipmaps to update larger blocks of the mesh creating a *"popping effect"* when moving the camera, while clipmaps keep updating smaller rows between each detail level, typically creating visible bands as the camera moves forward.

7 A camera looking forward with a 45 degree field of view will leave approx. 75% of the clipmap unused.

3.4 Virtual Texturing

Virtual Texturing are methods of displaying a texture which may be too large to fit into GPU memory, by storing the active subsections of the texture as small tiles in a single texture storage, which can be reassembled and rendered as if the complete texture was stored in GPU memory.

The first step of Virtual Texturing is to determine which parts of the texture is currently visible[8].

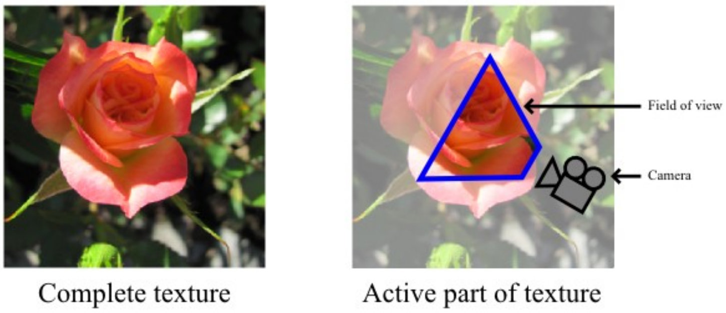


Figure 3.5 – Viewable texture

The each LoD of the complete texture is divided into blocks of constant size, where the total block structure will have resemblance with an Octree structure[2]. This block structure is very similar to geometrical mipmaps described in *chapter 3.3*.

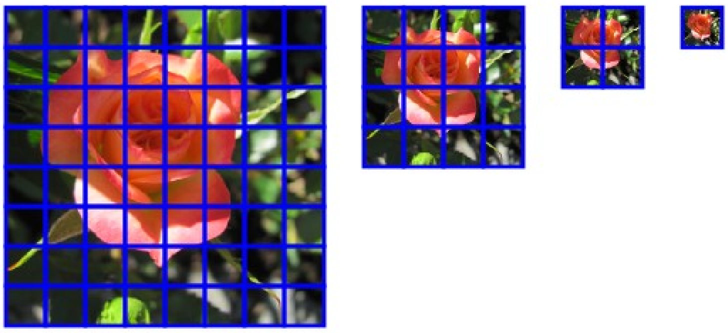


Figure 3.6 – Blocks at different levels

Each of these blocks are checked for intersection with the field of view in order to determine which blocks are required.

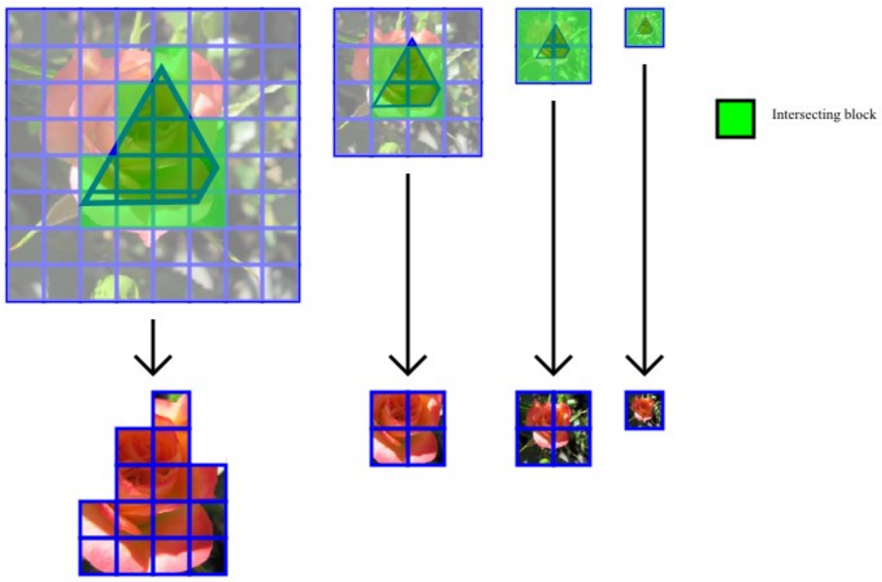


Figure 3.7 – Selection of blocks

The active blocks are then stored in a single texture in GPU memory, and may be in any random order. Since neighboring blocks in the texture storage might not be neighboring in the complete texture, blocks need padding to avoid issues with clipping when the texture is used in rendering. Due to the padding the block sizes are no longer power of two. Usage of texture filtering such as trilinear filtering will increase the need for larger margins, which further increases the storage overhead[9].

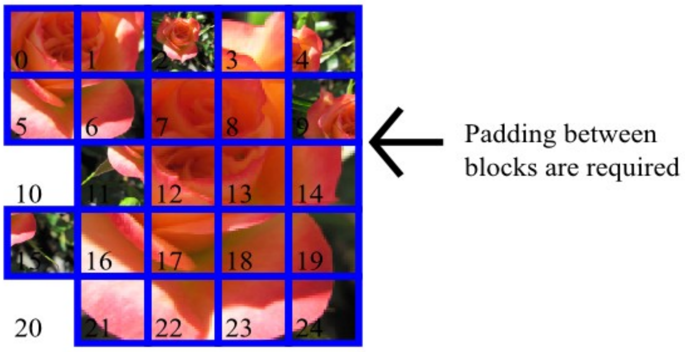


Figure 3.8 – Blocks in texture storage in random order

A paging table is used to map texture coordinates from the complete texture to the virtual texture in the storage during rendering. AMD provides an OpenGL extension *AMD_sparse_texture*[14][11] which may help to reduce the overhead of memory address conversion.

Example - Lookup for texture coordinate (0.7, 0.6):

- Find all sections which contains this texture coordinate: 19, 6, 15 and 2
- Determine which block to use based on distance to camera: camera is close, so use 19

Currently the total size of a virtual texture storage is limited by the maximum texture size for the GPU[8], which determines the algorithm's ability to scale with future hardware.

Sparse Virtual Textures[8], *Megatexture*[10] and *Partial resident textures*[19] are similar implementations of virtual textures. Megatextures are featured in the game engine Id Tech 5[10], which powers games such as Rage(2011), and features textures up to 128000x128000 pixels and 1024 loaded blocks in storage. Id[10] also uses 8 ms for virtual texturing per frame compared to 10 ms of rendering.

8 As of June 2013, the maximum size for a texture is 16384^2 on both NVIDIA and AMD graphics cards, while GPUs have up to 6GB of memory, at maximum 1GB of this can be utilized for a single virtual texture with RGBA channels (256 MB per channel).

3.5 Block handling

The research into loading of texture data in time is limited compared to the efforts for efficient rendering of blocks.

The *Sparse Virtual Textures*[8] presentation suggest to load texture blocks when required for rendering, and to discard blocks by using a *least recently used*(LRU) queue. AMD[19] and Id[10] appears to be using frustum culling to determine which blocks are visible on the screen. If a required block is unavailable, the algorithm will use a lower mipmap level in the meantime, and issue a *LoD warning*[19] to fetch the required part. Tanner[18] suggests a one block *look-ahead* when loading a block into system memory.

3.6 State of current technologies

This section showcases some of the current technologies related to this thesis.

3.6.1 Unigine

Unigine is a game engine developed by Unigine Corp. The company has released several demos of new features of their game engine the last years, including the *Valley Benchmark*[22] released in 2013. The demo features a large terrain with vegetation and rocks, dynamic weather and environmental effects[22].

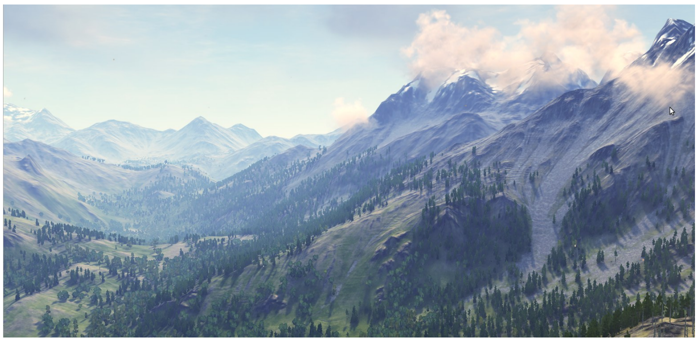


Figure 3.9 – Landscape rendering in Unigine Valley

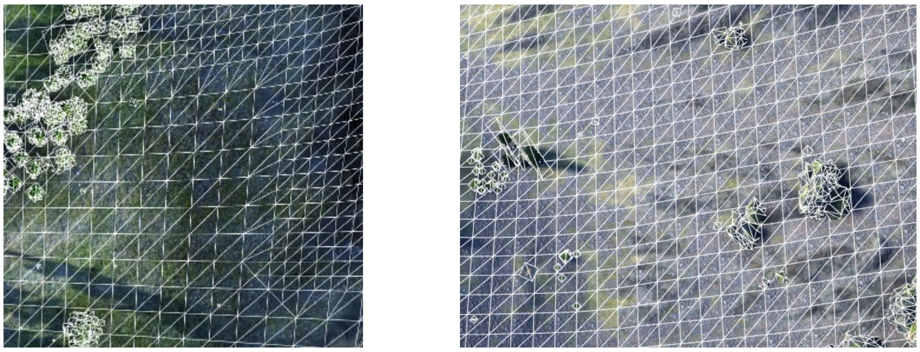

(a) Mesh blocks *(b) Repeating textures blended with low resolution unique*

Figure 3.10 – Unigine terrain features

Some closer inspection reveals this game engine utilizes a block based mesh with a constant block count at each detail level, very similar to geometrical mipmaps described in *chapter 3.3*. The engine does however use repeating textures with a mask revealed in *Figure 3.10 b*, in addition to blending the repeating textures with some unique low resolution details[9]. This methods severely limits the amount of unique detail in the terrain, and the low resolution details become blurred at distance[10].

9 The darker and light green parts of *figure 3.10 b* is low resolution compared to the high resolution repeating rock pattern.

10 The low resolution details is clearly visible in *figure 3.9* near the left side and the center where there is low vegetation.

3.6.2 id tech 5

The game Rage(2011) utilizes the game engine id tech 5, features megatextures to enable large environments with unique textures, including terrain objects and animated characters.

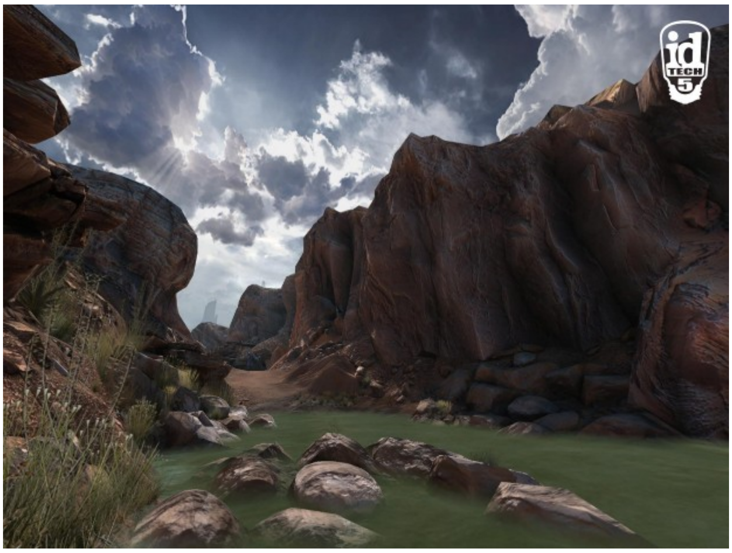


Figure 3.11 – Megatexture in id tech 5[10] © id software 2009

Megatextures proves to enable higher visual quality of textures than simple textures based on mask and detail textures. This implementation does however suffer from two major issues; low polygon count on terrain[11], and visible degradation of texture quality when turning the camera[23]. The last issue is referred to as *texture popping*, and was improved after the release of the game, by increasing the texture cache in GPU memory[24].

11 Notice the edges of the rocks in the bottom and on the left in *figure 3.11*.

Chapter 4

Proposed algorithms

The purpose of this chapter is to introduce the principles of the proposed algorithms in this thesis before giving a detailed explanation of the implementation in chapter 5.

4.1 Separate Textures

This thesis proposes to store the terrain blocks as separate textures in GPU memory, in an effort to improve rendering efficiency and visual quality. Several of the issues of Virtual texturing mentioned in *chapter 3.2* is addressed in this algorithm. Separate textures will eliminate the need for padding between texture blocks by allowing each block to clamp to edges. Separate textures also allows variable block sizes, so a constant block count across detail levels will not require conversion of texture coordinates. For these reasons this approach will require a less complex implementation in shader programs.

To have a high number of active textures has been unachievable until recently, sine OpenGL has a limited amount of available texture units. Traditionally OpenGL have to bind textures to texture unit, and similarly bind vertex buffers and index buffers. Binding of resources to fixed units cause a performance penalty. Some recent OpenGL extensions from NVIDIA called *bindless graphics*[5][16][17] and *bindless textures*[6][15] allows direct access to buffers and textures using pointers inside shader programs. This additional functionality is probably related to the extensive programmability features of CUDA[12], which allows usage of pointers and structures similar to C code. The main drawback for this method is the current limitation to NVIDIA hardware, however NVIDIA suggests it might be supported in future standards[6].

The method of using separate texture blocks is able to fully utilize the total capacity of the GPU memory, and may able to scale with future improvements in memory capacity.

4.2 Block Storage - Pools

This thesis stores blocks in different storage pools since GPU memory is too small to store a large detailed terrain. Even system memory might be too small for a large terrain, and loading of several GB of texture data will create substantial loading times. This thesis suggest division into three block pools; one complete storage pool on a HDD or SSD, and selected block pools in system memory and GPU memory.

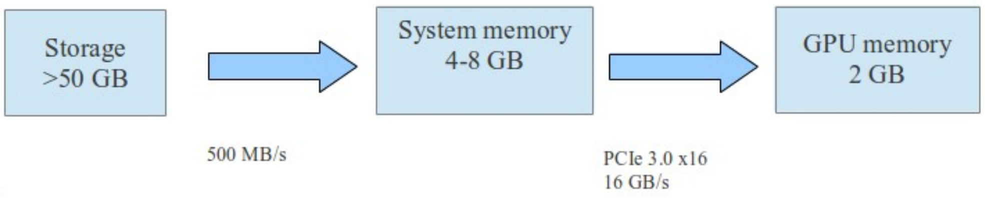


Figure 4.1 - Storage Pools

The large storage pool allows enormous terrains up to several TB in size, while the system memory and GPU memory pools serves as a cache for rendering the active part of the terrain. Using a system memory pool several times larger than the GPU memory pool is a way of handling the slow transfer speed and high latency from the storage pool. Most graphics cards have a small memory capacity compared to system memory, so the system memory may serve as an effective transfer cache between the slow storage and the small GPU memory, making the cost of transferring a block to the GPU far lower provided the block is available in system memory.

4.3 Block handling

The purpose of block handling in this thesis is to reduce some of the issues with virtual texturing mentioned in chapters 3.4 and 3.6.2, by assuming which blocks will be needed in the near future.

The system memory pool and GPU pool will require specialized algorithms for loading. The algorithm for the system memory pool has to be more coarse than the algorithms for the GPU pool, to be able to guarantee a block is available in the system memory pool when the GPU requires it. The loading of textures into the system memory pool will be processed independently of the rendering process.

Chapter 5

Implementation

This chapter describes the specific implementation of the suggested algorithms and strctures.

5.1 Block structure

A block is a subsection of the complete terrain texture, and since storing the complete texture as a single file would cause subsections to span over large data areas, each subsection has to be stored as a single file. Each block is squared in size, so a terrain consisting of 128 rows and columns of 256x256 pixel blocks at the top level makes up a total 32768x32768 texture. If the top level block is 256x256 pixels, then lower mipmap levels will be 128x128, 64x64, 32x32 and so on. The files are stored in separate folders for each level, and then one folder for each row of blocks. Folders and block file names are named by their hexadecimal coordinate in the terrain. This structure is chosen both for ease of manual navigation and file access time[12].

A sample file name would look like this:

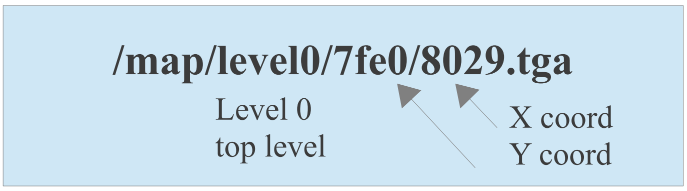


Figure 5.1 – Filename structure

The current limit of blocks are 65536, and the coordinates are centered around 32768 to allow expansion in any direction.

12 Common file systems such as NTFS and EXT are based on binary trees with a limited number of files in each node, splitting each column in a separate directory aims to maintain access time across file systems.

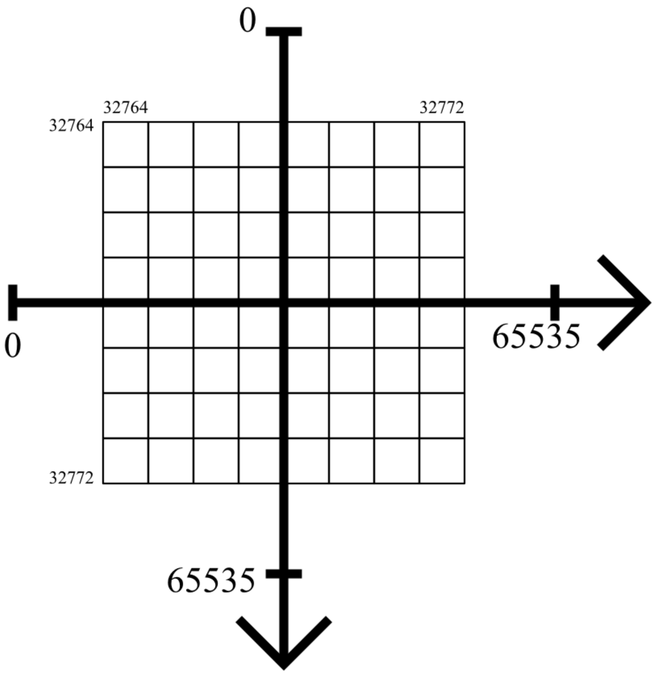


Figure 5.2 – Block Coordinates for a 8x8 terrain

5.2 Threading

In order to achieve uninterrupted rendering, all computation which is not required to be done in the rendering thread[13] is done in separate threads. Only rendering, in addition to uploading and deletion of blocks in GPU memory should be executed within the rendering thread. Instead of synchronization which may cause delays[14], data is exchanged between threads as *lockless queues*. Loading of files to system memory is also performed in a separate thread, to avoid IO latencies affecting other calculations.

13 Rendering thread is the thread holding the context to the GPU, the only thread able to exchange OpenGL-commands

14 During this thesis, latencies is measured up to 1 ms in the regular Linux kernel, so usage of synchronization may cause severe delays, which may be critical since each loop has to execute within 16.67 ms.

5.3 Block Transfers

Graphics cards are currently not capable of rendering and receiving new blocks simultaneously, so each data transfer has to be timed and prioritized to avoid unstable frame rates. This thesis suggest to split the transfers into small packets of blocks between each rendered frame, and limiting the amount of transferred data to a fixed factor of the rendering time of a frame. For a stable frame rate of 60 FPS, each frame has to be completed in less than 16.67 ms. If block transfer is limited to 2 ms per frame, the remaining 14.67 ms can be used for frame rendering. This transfer time can easily be implemented by using a data transfer threshold which limits the side of the data to be transferred in a single rendering thread loop. Larger blocks will require the same amount of time as multiple smaller blocks.

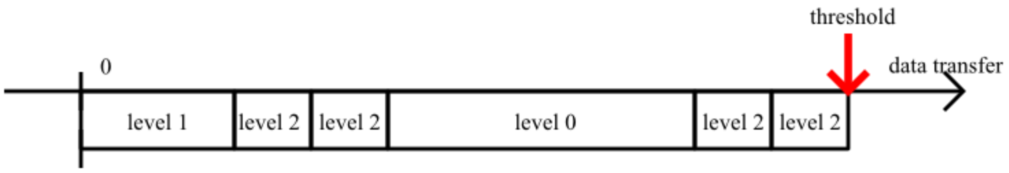


Figure 5.3 – Block Transfer

Adjusting the threshold in real time will make the algorithms flexible to run on different hardware.

Removal of texture data in GPU memory is quicker than loading but still requires some processing time. In addition it could be advantageous to have a small latency in case the block is needed again shortly, because of the high performance cost of re-transfer of blocks.

The system memory and GPU memory pool maintains whether a block is queued, loaded or “marked for deletion”, to avoid multiple requests for the same block and enabling the possible rescue of blocks on the delete queue.

5.4 Block Selection

Both the specialized selection algorithms will generate a list of requested blocks, which in turn will be processed by the block loader algorithms. The lists of block requests for loading and removal are stored as simple linked lists.

5.4.1 System Memory Pool

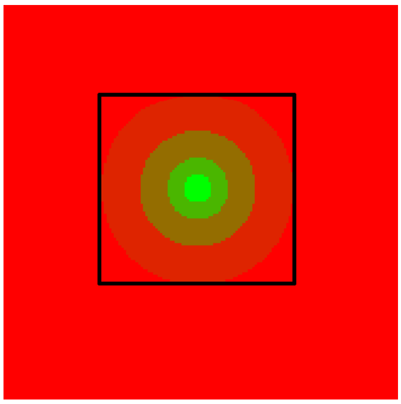


Figure 5.4 System Memory Pool
Black rectangle marks first step
Green colors marks distance to camera in second step

The system memory pool algorithm will run in three steps:

1. Locate a squared area around the camera based on maximum line of sight.
2. Calculate spherical distance from every block within the rectangle, use different thresholds for each LoD.
3. The calculated LoD for a block is compared with the current stored block level. If a block at a higher level is required, the block id is added to the loading queue. If a block is no longer needed, the block id is added to the removal queue.

The first step of the algorithm ensures maximum runtime for the algorithm, regardless of the total size of the terrain.

The thresholds in step (2) has to be adjusted according to the performance of the storage media, which in theory makes an SSD require a smaller safety margin than a HDD.

5.4.2 GPU Memory Pool

Similar to the system memory pool algorithm, the GPU memory pool algorithms will start by limiting the blocks around the camera. This is even more important for the GPU pool since these algorithms are based on frustum culling which is far more computationally intensive.

The GPU algorithms can be adjusted with several parameters depending on camera movement, e.g. a walking camera might want shorter and wider field of view, while an airplane would move faster and might not turn as quickly, so a narrower field of view can be sufficient.

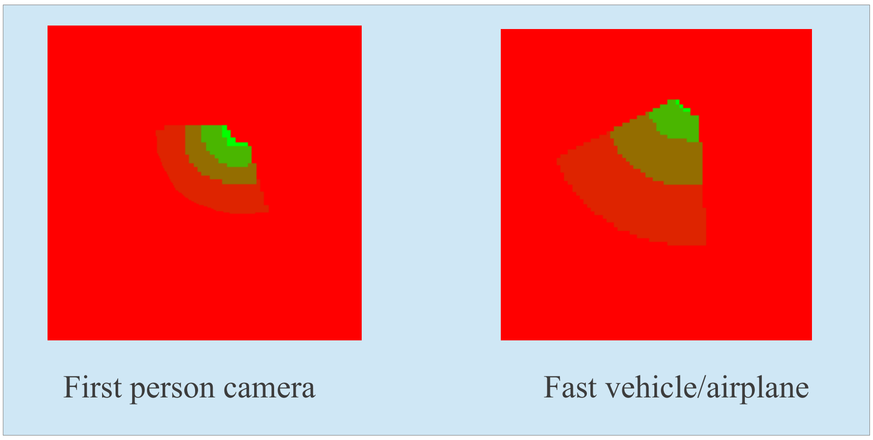


Figure 5.5 Field of View for different camera movements

Adjustment of LoD thresholds and field of view will adjust the algorithm's sensitivity to speed and quick rotations.

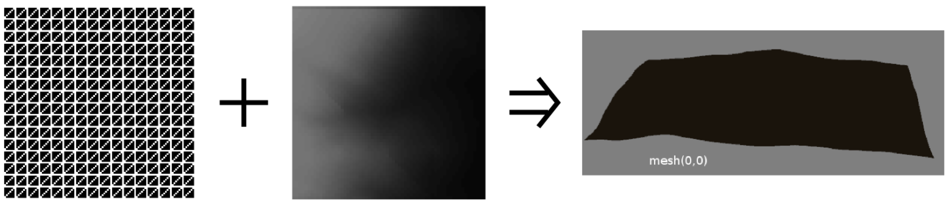
5.5 File Loading

File loading is performed by continuously processing any requests entering in the request queue for the system memory pool. For each block a texture and a mesh is loaded from separate files. The mesh is stored as a *heightmap* where only the height component of the meshes are stored.

5.6 Rendering optimizations

Using a *Vertex Buffer Object* and a *Index Buffer* is one of the most efficient ways of rendering a mesh in OpenGL. The Index Buffer maps vertex coordinates together into triangles, and since the index layout will be equal for all terrain blocks at a specific LoD, Index Buffers only need to be generated once and can be shader between terrain block at a specific detail level.

Similarly, the only difference between different Vertex Buffer Objects is the height component which represent displacement[3] in Y-direction, so the need for unique Vertex Buffer Objects can be circumvented by only storing the height component of each unique terrain block. This optimization works by storing the heightmap as an image buffer[3] for each terrain block instead of generating a complete Vertex Buffer Object.



(a) Mesh (X and Z component) *(b) Y-component* *(c) Finished mesh*

Figure 5.6 Constructing mesh with displacement mapping

To make this work the displacement has to be implemented in the *vertex shader program*.

```
uniform sampler2D displacement; // Heightmap
uniform vec2 translate; // Translation of the whole block

void main() {
        vec2 uv = TexCoord.xy;
        float d = texture(displacement, vec2(uv.x, 1.0f - uv.y)).x;
        gl_Position = modelViewMatrix * vec4(Pos.x + translate.x, d, Pos.z + translate.z, 1.0f);
}
```

Usage of the heighmap stored in GPU memory as a image buffer will ease the use of smooth transitions using tessellation, since GPUs can easily interpolate image data in shader program.

A narrow frustum culling is used to only render the blocks which are really visible. This frustum is smaller than the ones used in the block selection algorithms, and the difference between these algorithms is the safety margin.

The last key optimization which can be performed is usage of *bindless textures*, which is a new feature of NVIDIA's Kepler graphics cards. This allows direct access to textures inside shader programs, allowing usage of large number of active textures without any binding.

5.7 Block Compression

Decompression of blocks can be beneficial for more compact storage, but might increase the loading time depending on compression options:

- Compact lossy in storage pool, uncompressed in system memory and GPU memory.
  - Benefit: Reduce storage size
  - Drawback: Impacts loading speed (~150 MB/s)
- Static compression in all pools.
  - Benefit: No decompression required.
  - Drawback: Compression has to be preprocessed and is time consuming.

This thesis has chosen to use no compression and focus the effort on the algorithmic parts of the problem, due to small benefits from compression.

Chapter 6

Results

This chapter describes the results from this thesis. The terrain used in most of these tests consist of a terrain of ~8GB texture and height data, in total 16 GB with all mipmap levels in the storage pool. The terrain is split into 128x128 blocks, each containing 256x256 of data at the top level.

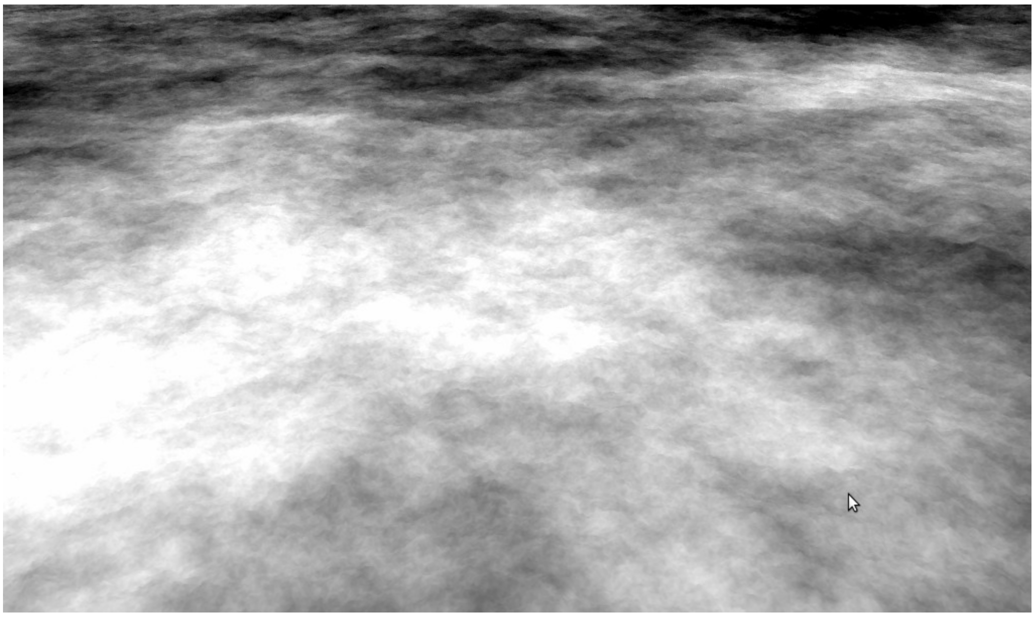
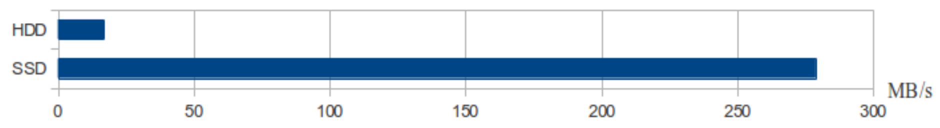


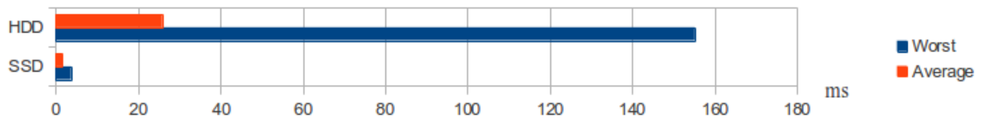
Figure 6.1 – Sample screenshot of giant 32768x32768 textured mesh

6.1 Initial Loading

The initial loading phase is the time to load all required data into the system memory and the GPU memory pool to display the starting position for the camera. The blocks appear on the screen as they are loaded into memory.



Graph 6.1 - Loading performance (MB/s) of 1000 random blocks.



Graph 6.2 – Loading latency (ms) of 1000 random blocks.

SSDs proves to be way faster than HDDs in pure IO-performance, averaging at 279 MB/s compared to 17 MB/s for the slow HHD. *Graph 6.2* shows the importance of having a sufficient threshold for the first selection algorithm. Even though the speed of the SSD is superior, it still requires at average 1.6 ms per block[15] with a total frame time of 16.67 ms, so the algorithm might still need to cache blocks which will be needed several frames ahead. This is the reason for the issues in the game Rage mentioned in *chapter 3.6.2*.

When loading terrain data from a HDD the initial loading completes within 10 seconds regardless of the starting position of the camera. SSDs provide a visible speedup in initial loading time.

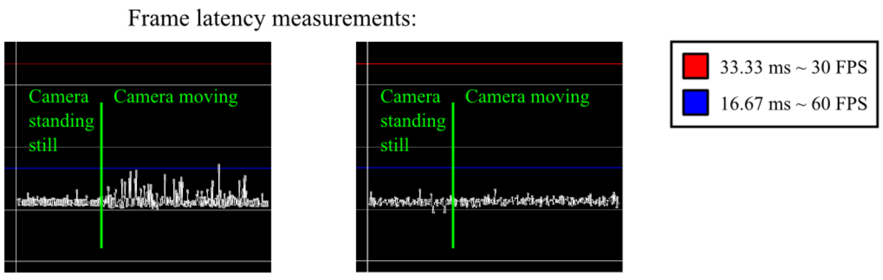
15 Loading of texture and heightmap for a block of 256x256.

6.2 Procedural Loading

Smooth movements in either direction does not present any large problems for the algorithms. An HDD is fast enough to supply a steady stream of blocks, with proper thresholds. Rapid camera rotations does however create a little delay before the required blocks are present in high detail. Limiting camera rotation speed in conjunction with adjusting thresholds of the selection algorithms is important to ensure visual quality. This is probably the weakest point of any form of incremental loading, so certain applications such as *first person games*[16] may consider using a spherical selection algorithm for the GPU pool as well.

6.3 Loading of meshes vs textures

Assembly of vertex data and indices is a time consuming process which greatly affects rendering performance when done in real time. The optimization of displacement mapping (section 4.7) suggests a solution using a single mesh for each mipmap level, and applying the height component in the vertex shader program in real time during rendering instead of during loading. This costs a bit of performance during each rendering, while achieving a more even frame rate since block transfers become much faster. This optimization is considered a great trade off.



(a) VBO Loading *(a) Heightmap loading*

Graph 6.3 – Frame latency. Notice the peaks on (a) are not present in (b).

16 Games in first person camera angle.

As indicated by *graph 6.3*, loading of Vertex Buffer Objects may increase the rendering delay of a single frame in the range of 30-60% for this set of test data. The peaks in *graph 6.3 (a)* is perceivable as noticeable stuttering in the live demo. This graph shows a block transfer size threshold is ineffective when uploading VBOs, while quite effective when only uploading texture data.

6.4 Block Selection

The initially block selection implementation was executed within the rendering thread, which caused major drops in performance. Separating this to a dedicated thread greatly improved performance and stability. Usage of lockless queues enabled usage of multiple threads without the need for synchronization.

The first step of the block selection algorithms proved to be greatly beneficial by reducing the run time to a fixed maximum regardless of the amount of blocks. Without this optimization the current hardware configuration would have problems with more than 256x256 blocks.

6.5 Block Transitions

Mipmaps provide a decent transition between texture LoD, and multiple LoD can be efficiently be blended within the *fragment shader program*. Mesh transitions are however a bit more complicated, but is achievable using flexible tessellation provided on recent GPUs. Tessellation is however not the focus of this thesis.

6.6 Adjustments of Thresholds

The implementation provides a set of thresholds for the algorithms, which sets the camera movement parameters and loading safety margins.

Chapter 7

Discussion

This chapter discusses the different findings during this thesis.

7.1 Test Data Challenges

Creating a decent large terrain for testing was a great challenge for this thesis. To get some large satellite photos of terrain would be ideal, but would still required some work for the heightmaps. For testing purposes the test terrain for this thesis was generated using *perlin noise*[3] and later modified in a graphical editor.

7.2 Implementation Challenges

This section describes various challenges during the implementation of this thesis.

7.2.1 Mipmaps

Even though OpenGL provides several methods of loading and changing a mipmap level of a texture[13], no method of removing a mipmap for freeing GPU memory is provided. Therefore the mipmap functionality has to be manually implemented and needs to rely on separate textures for each mipmap level. The drawback of this is the extra effort in handling of more texture handles and the added complexity in shader programs. It did however provide an important opportunity to design a more perceptial mipmap transistion. The defalt OpenGL mipmap implementation blends the textures like a linear gradient.

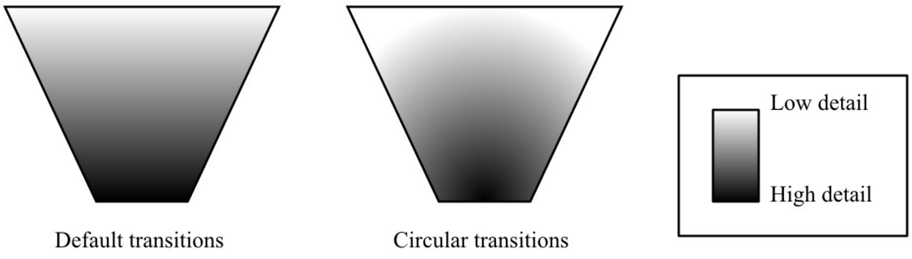


Figure 7.1 – LoD Transitions

This allows the mipmaps to focus more details in the center of the field of view.

7.2.2 Performance instability

Loading of meshes proved to be a cause of variable performance, so the optimization of heighmaps as textures using a single mesh made the performance way more stable. The performance could be further stabilized by making the loading time constant for each frame by uploading “dummy data” if no more blocks are in the queue, but this seems to be a minor problem compared with the variance in performance caused by varying camera orientations. After the optimizations the culling based on the camera orientation remained as the most significant factor of variable performance.

7.2.3 PCIe Speed

This thesis was conducted on the 304 branch of drivers from NVIDIA, which currently only supports PCIe 2.0 speed at 8 GB/s. Experimental driver support for PCIe 3.0 is provided in more recent drivers, but was unable to achieve stability with this configuration. Higher PCIe speed will significantly reduce the cost of transferring data to GPU memory.

7.2.4 Occasional Artifacts

This implementation is not bug free, on occasion there might be a block using the wrong texture in the middle of the terrain. This typically happens after very rapid motions, and is most likely caused by a minor bug in the selection algorithms, since the block may change depending on camera rotation. There is not any reason to suspect any driver bugs, but this has not been investigated extensively.

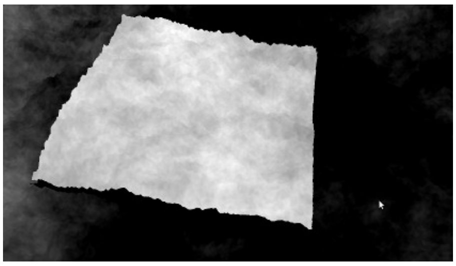


Figure 7.2 – Wrong block

7.2.5 Lookup tables for textures in bindless textures

Current hardware seems to have a allocation limit of sending 8192 texture pointers in table to a shader program, which limits the usability of a *texture lookup table*. Usage of more textures will require passing of texture handles for each block, which might limit some of the performance gains.

7.3 Comparison with Virtual Texturing

Compared with Virtual Texturing, separate blocks provide several benefits. It eliminates the issues related to clipping, which in turn could enable more texture filtering for higher visual quality. The shader program implementation is also very simple and effective. Separate blocks will also be able to use the amount of memory available in the GPU, instead of maximum 1GB (16384^2 pixels) on current GPUs for Virtual Texturing. During testing the implementation was able to use 2-3 GB with the appropriate thresholds for the algorithms. Virtual Texturing also has some padding issues which are not present for separate textures.

The choice of constant block count at each level does not require texture coordinate conversion, but mixing between levels still a bit complicated, and the separate textures can't rely on built-in mipmap since OpenGL does not allow removal of a single mipmap level. The performance of this part is still unknown, as the implementation of LoD transitions is not complete in this thesis.

Chapter 8

Conclusion & Future Work

The goal of this thesis was to implement effective algorithms of incremental loading of terrain textures, by attempting to predict the required texture blocks and render the terrain in high detail at a good framerate without too high storage overhead.

Loading of image buffers such as texture data proved to be far quicker than uploading of vertex buffer objects. SSDs does also provide faster loading speed to the system memory, while HDDs can be fast enough given proper LoD thresholds and restrictions to camera movement. Rapid rotations is the most sensitive movement for the loading algorithms, and delays in the distance can be hidden by applying fog close to the line of sight.

The most interesting find of this thesis is the optimization of using a single Vertex Buffer Object per detail level instead of using unique buffers per terrain block proved to be the most effective optimization. Compared to this the performance gains of using bindless textures are almost negligible due to the relatively low block count visible at any time. At the beginning of this thesis bindless textures held the greatest expectations. Bindless graphics and textures has the most performance gains when rendering large amounts of small meshes, such as grass, terrain objects or foilage. This optimization is usefull for terrain engines, but mostly usefull for the smaller details.

Unfortunately the LoD transitions were not completed in the timeframe of this thesis, so every aspect of the algorithms can't be evaluated. Anyhow, the principle of separate textures has proven to work well, and could be developed further to a complete implementation.

Hopefully this thesis will inspire further development to utilize the large programming potential of modern GPUs for even more sophisticated algorithms, and encourage other GPU makers than NVIDIA to provide these hardware features.

8.1 Future Work

Virtual Texturing can utilize the proposed loading and block selection algorithms. The algorithm may also use the optimization of having heightmaps represented as textures, which will give a speedup even without bindless graphics. If a virtual texture is limited by the maximum texture size of the GPU, textures, heightmap and normalmap can be split into three virtual textures using a single paging table, since each block of the texture is paired with a block of heightmap and normalmap.

The implementation of this thesis is prepared for smooth tessellation by using displacement maps which can easily be utilized in tessellation shader programs. Normal maps can also easily be added, and loaded in the same was as textures. The mechanisms of the selection algorithms will remain exactly the same.

Usage of BPTC/BC7 compression will give a static block compression ratio of 1:3. It will however demand an immense amount of time to compress a large terrain, and it might introduce artifacts or noise at lower LoD.

Future development to utilize the features of bindless graphics has great potential. The support for pointer types allows complex structures such as linked lists inside shaders[5]. Combined with compute shaders, CUDA and some of the latest features like *Dynamic Parallelism* of the Kepler architecture[7], this might allow a LoD algoirthm implemented almost completely on the GPU, including block selection, culling and requests for new data. *NVIDIA GPUDirect*[7] also allows streaming of data directly from third party devices such as SSDs, this might be utilized in a clever way to enable higher visual quality.

Appendix A

A.1 Testing hardware specifications

CPU: Intel(R) Core(TM) i7-3930K (stock speed)

Motherboard: ASUS P9X79 WS

RAM: 64GB Corsair DDR3 Vengeance 1600MHz CL10 (x8, stock speed)

GPU: MSI GeForce GTX 680 4GB TwinFrozr III (stock speed)

SSD: Intel 520 Series 240 GB

HDD: WD Black 1TB

Monitor #1: Eizo FlexScan S2401W, 1920x1200 60 Hz, TN-panel.

Monitor #2: Eizo FlexScan EV2416W, 1920x1200 60 Hz, TN-panel.

OS1: Ubuntu 12.04 64-bit (normal kernel)

OS2: Ubuntu Studio 12.04 64-bit (real time kernel)

Drivers: NVDIA 304.88 and 304.43

References

(1) Sagal M., Akeley K.,
The OpenGL Graphics System: A Specification Version 4.3 (Compatibility Profile)
Khronos Group Inc., 2013, Retrieved March 5th, 2013 from
http://www.opengl.org/registry/doc/glspec43.compatibility.20130214.pdf

(2) Hearn, Baker, Carithers,
Computer Graphcis with OpenGL: Fourth Edition. (International Edition)
Pearson, 2011.

(3) T. Theoharis, G. Papaionannou, N. Platis, N. M. Patrikalakis.
Graphics & Visualization – Principles and Algorithms
A K Peters Ltd, Wellesley, Massachusetts. / CRC Press 2008.

(4) H. JungHyun.
3D Graphics for Game Programming. CRC Press, 2011.

(5) Boltz, Jeff. OpenGL Bindless Extensions. NVIDIA Corporation. 2009. Retrieved February 2nd, 2013 from
http://developer.download.nvidia.com/opengl/tutorials/bindless_graphics.pdf

(6) *NVIDIA GeForce GTX 680 (Whitepaper)*. NVIDIA Corporation. 2012. Retrieved February 2nd, 2013 from http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf

(7) Kepler GK110 (Whitepaper). NVIDIA Corporation. 2012. Retrieved February 2nd, 2013 from http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

(8) NVIDIA GF100(White paper). NVIDIA Corporation. 2010. Retrieved on February 2nd, 2013 from http://www.nvidia.com/attach/3626515.html?type=support&primitive=0

(9) Barret, Sean. Sparse Virtual Textures. Retrieved January 10th, 2013 from http://silverspaceship.com/src/svt/

(10) van Waveren, J. M. P. *id Tech 5 Challenges: From Texture Virtualization to Massive Parallelization*. 2009. SIGGRAPH. Retrieved January 10th, 2013 from http://s09.idav.ucdavis.edu/talks/05-JP_id_Tech_5_Challenges.pdf

(11) Obert, Juraj, van Waveren, J. M. P., Sellers, Graham. Virtual Texturing in Software and Hardware. 2012. SIGGRAPH. Retrieved June 10th, 2013 from http://mrelusive.com/publications/presentations/2012_siggraph/Virtual_Texturing_in_Software_and_Hardware_final.pdf

(12) NVIDIA CUDA Library: Unified Addressing. Retrieved May 10th, 2013 from http://developer.download.nvidia.com/compute/cuda/4_2/rel/toolkit/docs/online/group__CUDART__UNIFIED.html

(13) OpenGL 4 Reference Pages – glTexImage2D Retrieved February 2nd, 2013 from http://www.opengl.org/sdk/docs/man/xhtml/glTexImage2D.xml

(14) AMD_sparse_texture. OpenGL Registry. Retrieved January 10th, 2013 from http://www.opengl.org/registry/specs/AMD/sparse_texture.txt

(15) NV_bindless_texture. OpenGL Registry. Retrieved January 10th, 2013 from http://www.opengl.org/registry/specs/NV/bindless_texture.txt

(16) NV_shader_buffer_load. OpenGL Registry. Retrieved January 10th, 2013 from http://www.opengl.org/registry/specs/NV/shader_buffer_load.txt

(17) GL_NV_shader_buffer_load. OpenGL Registry. Retrieved January 10th, 2013 from http://www.opengl.org/registry/specs/NV/shader_buffer_load.txt

(18) Tanner, Christopher C., Migdal, Christopher J., Jones, Michel T. Clipmap: A Virtual Mipmap. 1998. Silicon Graphics Computer Systems. Retrieved February 12th, 2013 from http://www.graphicon.ru/oldgr/library/siggraph/98/papers/tanner/tanner.pdf

(19) Bilodeau, Bill, Sellers, Graham, Hillesland, Karl. Partially Resident Textures on Next-Generation GPUS. 2012. Retrieved March 8th, 2013 from http://amddevcentral.com/gpu_assets/Partially%20Resident%20Textures%20on%20Next-Generation%20GPUs.v04.pps

(20) Losasso, Frank, Hoppe, Hugues. Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids. Stanford University and Microsoft Research. Retrieved April 13th, 2013 from http://research.microsoft.com/en-us/um/people/hoppe/geomclipmap.pdf

(21) Clipmaps(White paper). NVIDIA Corporation. 2007. Retrieved May 2nd, 2013 from http://developer.download.nvidia.com/SDK/10/direct3d/Source/Clipmaps/doc/Clipmaps.pdf

(22) Unigine Valley. Unigine Corp. 2013.
Retrieved May 14th, 2013 from http://unigine.com/products/valley/#features

(23) T Senior. Rage PC players experience laggy textures and low framerates.
PC Gamer, 2011. Retrieved May 20th, 2013 from http://www.pcgamer.com/2011/10/04/rage-pc-players-experiencing-laggy-textures-and-low-framerates/

(24) Rage Updated. 2011. Retrieved May 20th, 2013 from http://store.steampowered.com/news/6464/

(25) Duchaineau, Mark, Wolinsky, Murray, Sigeti, David E., Miller, Mark. C., Aldrich, Charles, Mineev-Weinstein, Mark B. ROAMing Terrain: Real-time Optimally Adapting Meshes. Los Alamos National Laboratory, Lawrence Livermore National Laboratory. 1997. Retrieved May 20th, 2013 from http://www.classes.cs.uchicago.edu/archive/2003/fall/23700/docs/roam.pdf

(26) Hoppe, Hugues. Smooth View-Dependent Level-of-Detail Control and its Application to Terrain Rendering. Microsoft Research. 1998. Retrieved May 20th, 2013 from http://research.microsoft.com/en-us/um/people/hoppe/svdlod.pdf

(27) Isola, Philips. Off the Grid Terrain. Wolfire Games. 2008. Retrieved on May 20th, 2013 from http://blog.wolfire.com/2008/12/off-the-grid-terrain/

(28) ARB_multitexture. OpenGL Registry 1998. Retrieved June 5th, 2013 from http://www.opengl.org/registry/specs/ARB/multitexture.txt

(29) Boer W. H.. Fast Rendering Using Geometrical MipMapping. 2000. Retrieved June 5th, 2013 from http://www.flipcode.com/archives/article_geomipmaps.pdf

(30) Linder B.. Intel: Retina laptop, desktop displays coming in 2013. Liliputing, 2012. Fetched June 5th, 2013 from http://liliputing.com/2012/04/intel-retina-laptop-desktop-displays-coming-in-2013.html

(31) Herlihy, M., Luchangco, V., Moir, M. Obstruction-Free Synchronization: Double-Ended Queues as an Example. Brown University, Providence. 2003. Retrieved June 5th, 2013 from http://cs.brown.edu/people/mph/HerlihyLM03/main.pdf

(32) GL_NV_vertex_buffer_unified_memory. OpenGL Registry, 2009. Retrieved June 10th, 2013 from http://www.opengl.org/registry/specs/NV/vertex_buffer_unified_memory.txt