Implementing Real-Time Terrain Deformation Using a Volumetric Voxel-Based Algorithm

Benjamin J, A, Mitchell

Bsc Honours Computer Games Technology, 2018

School of Design and Informatics
Abertay University

Table of Contents

Table of Figures 3
Figure 2-1: 15 Fundamental cases of the marching cubes algorithm….. 11
..... 3
Table of Tables 4
Abstract 5
1. Introduction 6
1.1 Project Aims and Research Question 8
2. Literature Review 9
2.1 Terrain Generation 9
2.1.1 Volumetric Terrain Generation 10
2.2 Terrain Deformation Techniques 14
2.3 DirectX11/GPGPU Techniques 16
3 Methodology 21
3.1 Introduction 21
3.2 Evaluation/Performance Factors 21
3.3 Terrain Generation with Marching Cubes 22
3.3.1 Feasibility Demo 22
3.3.2 Compute Shaders 23
3.3.3 Vertex Buffers 23
3.3.4 The Chunk System 24
3.3.5.1 DirectX 11 Resources 25
..... 28
3.4 Terrain Deformation 28
3.4.1 Terrain Ray Intersection 29
3.4.2 Handling Deformation 30
4. Results 35
4.1 Testing Machine 35

4.2 Data and Discussion 36
5. Conclusions 42
5.1 Project Conclusion 42
5.2 Future Work 43
5.2.1.1 Vertex Buffer Re-Initialization 43
5.2.1.2 Compute Shader Optimization 43
Appendices 46
..... 47
List of References 48
Bibliography 53

Table of Figures

Figure 2-1: 15 Fundamental cases of the marching cubes algorithm…..11
Figure 2-2: Generation of the marching cubes case value……....12
Figure 2-3: Edge and corner indexes for a voxel…….13
Figure 2-4: Marching Cubes algorithm pseudocode……...13
Figure 2-5: Voxelization of object geometry within a bounding box……..15
Figure 2-6: Bounding Box deformation pseudocode……...15
Figure 2-7: Liquid-Like deformation using drift-diffusion……16
Figure 2-8: DirectX11 shader stages…….....17
Figure 2-9: Compute Shader semantics description……..…..19

Figure 3-1: Major development advancements of application……..….24
Figure 3-2: Tri-Planar texture mapping visualization……...28
Figure 3-3: Finding the deformation centre pseudocode……29
Figure 3-4: Pseudocode describing the deformation process…….....31
Figure 3-5: Input/Output of compute shaders……...32

Figure 4-1: FPS/Terrain generation time vs chunks……....36
Figure 4-2: FPS/Terrain generation time vs thread groups per chunk….37
Figure 4-3: FPS/Terrain generation time vs threads per thread group….38
Figure 4-4: FPS while under high deformation strain……..39
Figure 4-5: CPU vs GPU process time for a single frame……..40
Figure 4-6: Terrain's capability for arches and overhangs…….46
Figure 4-7: Deformation applied to create a small cavern……..46
Figure 4-8: Formation tool used to create a rock structure……....47

Table of Tables

Table 4-1: Test Machine specifications…….....35

Abstract

Terrains are a pivotal part of the majority of modern games. The shape and design of the terrain used in a game often determines the atmosphere and general feel of the game. Many of the terrains that can currently be observed in games are very static objects, often offering little to no interactions with the player and almost never changing. One way that this might be improved upon is the addition of a volumetric terrain system. A volumetric system would allow the generation of more complex terrain features such as caves or overhangs, and could be built upon to provide interesting methods by which a player could form or deform areas of terrain.

With the combination of DirectX 11's compute shaders and a technique for rendering a volumetric terrain, this project aimed to build and evaluate an adaptation of the marching cubes algorithm to create a game terrain. This terrain was required to enable user-controlled deformation during run-time and run at the level of performance accepted in modern game environments. Ray-casting is used to allow the user to determine where to modify the terrain, with an option to either add to the terrain or remove from it.

Measurements were gathered from the application to investigate the performance of the algorithm under varying terrain sizes and usage circumstances. The results showed that, using GPGPU techniques, a volumetrically generated voxel terrain can be generated using GPGPU techniques, with deformation being handled efficiently and in real time.

1. Introduction

3D games are defined by the atmosphere they present to the player. The way a game feels to its player is defined by a number of factors, not least of which is what the player can currently see on screen. A large part of what a player can see in many games is the land surface, or the terrain. Most modern games make use of one of several known terrain generation methods; producing a static mesh that serves as the floor for the player. Often, this is simply a flat plane distorted by a height-map or a similar displacement technique to give a simple, curved surface. These plane surfaces are fundamentally incapable of producing more complex terrain shapes, such as cave systems, or even overhangs and cliffs.

With the increasing capability of modern GPUs comes an opportunity to improve upon this to produce far more interesting terrains; by allowing a far greater storage and resolution for terrain generation. Additionally, due to the increase in processing power, terrains that provide game features other than simply being a point of collision for the character become possible. This can include terrain deformation. Allowing the player to use the terrain as an interactive object provides a number of possibilities for game mechanics and interactions.

In order to allow for more interesting and complex terrains, a voxel terrain system can be implemented. A number of such algorithms can be seen in use, both for terrain generation and other purposes. The goal of these algorithms is to mathematically generate a mesh surface using a set of input data. One of these mesh generation techniques, the marching cubes algorithm, has proved to be an excellent resource when generating more complex surfaces in computer graphics, allowing it to serve excellently in terrain generation. The iso-surface used to generate the mesh is very malleable in nature, and can be extended into any 3D shape possible. The only drawback being the difficulty calculating this surface can sometimes present. Due to this malleability, shapes more complex than a distorted plain can be generated, with entire sections of the mesh

above others. Therefore complex terrain feature such as caves, overhangs and arches can be generated with ease.

Due to the complex shape generated by a voxel algorithm like marching cubes, texturing cannot be handled by sampling a texture as would normally be appropriate. Texturing can be handled on more complex shapes such as this by applying texture co-ordinates based on the position. Using a method known as tri-planar mapping, the texture can be applied using a multiplicative factor to each axis of a vertex based on its normal direction.

Giving a player the ability to alter the terrain surface is becoming a desirable modern game feature, with games such as Minecraft (Minecraft, 2009) encouraging this rise in popularity. While generally speaking terrain deformation in existing games features as a primary game mechanic, as graphics processors continue to grow more powerful and terrain generation techniques continue to become more documented, terrain deformation may become a more common feature in games.

There are a number of methods presented for deforming a terrain in real-time. However, the technique is still scarce in games due to the high performance cost, and is usually only seen when used as the primary game mechanic. The marching cubes algorithm provides a structure that gives an opportunity for deformation. By storing the previously mentioned iso-surface, then making changes to it and regenerating the mesh around the new iso-surface, the mesh can be updated during run-time. However this is very performance intensive. To allow for the game to run at a frame-rate acceptable for games, measures must be taken to accommodate this.

The evolution of modern GPUs was driven primarily by the video games industry, as developers recognized the power that massively parallelized processors had in generating 3D worlds (D. H. Eberly, 2015). Thus the practice of using the GPU for general purpose programming has arisen,

and has been labelled GPGPU programming (general purpose GPU programming). Many APIs have been released that make use of the power of GPGPU programming, including Microsoft's DirectCompute. The compute shader makes direct use of the parallel processing power of the GPU to complete tasks at a remarkable rate. This same process parallelization can be used to maximize the performance of the marching cubes algorithm, the entirety of which can be run on the GPU.

1.1 Project Aims and Research Question

This project aims to answer the following research question:

Can a volumetrically generated voxel terrain be effectively implemented to allow for deformation at run-time with the use of GPGPU Techniques?

To do this, an application will be built that will make use of the marching cubes algorithm to generate an interesting and unique terrain. The project then aims to allow for this terrain to be modified at run-time with minimal performance impact, allowing a player to form and deform terrain as they wish.

The investigation aims to determine if:

- The marching cubes algorithm can be built entirely on the GPU, and if so, how can the compute shader assist with this.
- The implementation can then be adapted to update the terrain during run-time, and how best to implement this.
- Whether the application produced is capable of running at a high enough performance rate to provide no hindrance to any game it may become a feature of.

2. Literature Review

2.1 Terrain Generation

Terrain generation is a vastly explored subject, especially in 3D games, where the quality of the terrain has a major impact on the impression left by the gameplay. At the basics, a game's terrain can be built using a height map. A height map is an image that can be shaded in, with points on the terrain displaced based on the colour gradient of the height map. Height maps can be drawn by an artist or procedurally generated. Hand-drawn maps can be useful if designers have a particular shape in mind for the game. However, certain styles of games are more interested in procedurally generated height maps.

A number of methods exist to procedurally offset the height of points along a plane to create a terrain. Midpoint Displacement, originally introduced as stochastic parametric surfaces, using Multifractal Terrain Generation can be used to generate a realistic looking fractal terrain (Ramstedt. R and Smed. J, 2016). Midpoint displacement can be described using the steps as follows (Losh. S, 2016):

- Initialize the four height map corners to random values.
- Set the points found half way between each of the edges to the average of the two corners it is between, possibly offset by a random amount.
- Set the centre of the square to the average of points just calculated, with a possible random offset.
- Repeat this method on the four squares within this square, reducing the random offsets.

Midpoint displacement, especially with multifractals, is a powerful method of terrain generation that makes use of a height map. The resulting height map is often quite mountainous due to the fractal nature of the generation. This may be useful for some environments, especially rural

game environments, however is still limited in its usability by a game developer. Fractals are still extremely useful however. By putting emphasis on speed at the cost of physical correctness, fractal terrains can be used to simulate erosion in near real-time (Olsen. J 2004).

Outside of the use of height-maps, there are still a number of well documented techniques for generating terrains, including faulting and particle deposition. Faulting is a simple fractal based technique for building realistically balanced terrains, avoiding the disjointed look pure random height produces. The algorithm can be described as repeatedly drawing a random line across a plane, slightly elevating the terrain on one side of the line (Halford. C, 2015). This eventually results in a mountainous landscape.

Particle deposition is a method of building the terrain using agents. Agents can see the current elevation of any point on the plane and are allowed to modify these points at will (Doran, 2010). By using multiple agents the terrain can change in unpredictable manners. These methods provide generally more complex terrain shapes than the height map methods, however they are still not capable of producing the life-like terrains desired as an ideal solution in many applications.

2.1.1 Volumetric Terrain Generation

While all of these generation methods produce realistic terrains, they all share one pitfall; they are limited by the bounds of a flat plane. This means that without stretching the plane positions at odd angles, creating stretched textures and unnatural spikes, they cannot create more complex terrain features such as caves or overhangs. However, this can be achieved using a volumetric terrain system.

The marching cubes algorithm was presented by Lorensen and Cline (1987), originally as a medical tool to create 3 dimensional displays from MRI scans, heart scans etc. The algorithm is now widely accepted to be

an effective method of building volumetric voxel based terrains (McAllister. Z, 2013), with the vast majority of implementations being in some form based off the well documented implementation by Paul Bourke (1994). The Marching Cubes algorithm describes the idea of generating many small meshes inside of small cubes, or voxels, to build one large mesh. There are 256 unique combinations of triangles that can be generated inside of a single voxel. Discounting symmetrical cases, this can be brought down to the 15 cases shown in figure 2-1. (Geiss. R, 2007).

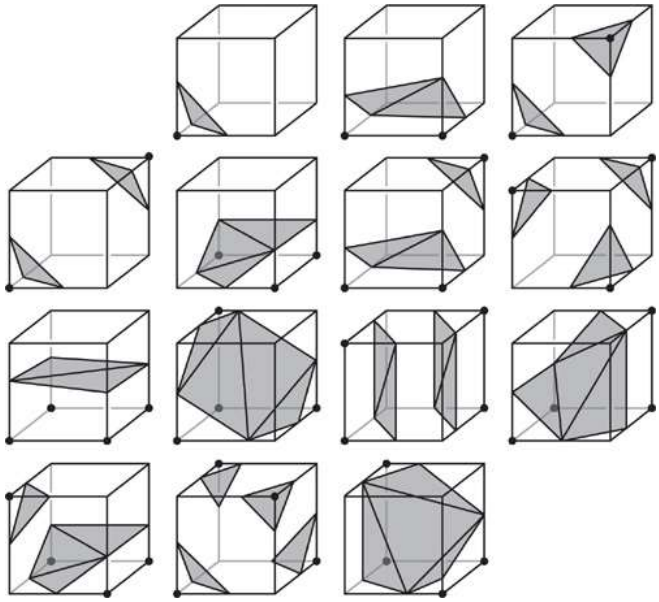

Figure 2-1 – 14 of the fundamental cases of the marching cubes algorithm, the 15th of which is an empty cube (Geiss. R, 2007).

The terrain is built using an iso-surface. Given a scalar field F(P), F being a scalar function in three-dimensional space, the iso-surface is the surface along which F(P) = a, where a is a constant (Newman. T. S & Yi. H, 2006). The iso-value, a, is a value often generated by noise (Perlin, 2002) or a similar pseudo-random number generation function. The iso-surface can also be generated using a number of terrain generation techniques, such as fractals, midpoint displacement, faulting etc. When combined with an effective iso-surface, the marching cubes algorithm can generate a terrain with enough detail to simulate terrains found in the real world (Greeff, 2009).

For calculation purposes, a density value is calculated at each corner of a cube. This density values is an additive combination of the corner's y position and the iso-surface at the corner's 3 dimensional position. If the density value is neither above nor below an arbitrarily set iso-level then the iso-surface runs through that voxel, and geometry should be generated.

The shape of the geometry created depends on which corners' density values were above the arbitrarily defined iso-level and which densities were below (Geiss. R, 2007). To find which exact combination of geometry to generate, a look-up table is used. This look-up table consists of the 256 triangle combinations that make up the marching cubes algorithm. To index this, a case value must be generated. This can be done either by making an 8 bit index, with each bit corresponding to a vertex of the cube (Bourke, P. 1994), or by using a multiplication factor for each vertex.

```
static int multiplier[8] = { 1, 2, 4, 8, 16, 32, 64, 128 };

//calculate final case by which to index marching cubes look up
table
if (Corners.density[i] < isoLevel)
{
    MCCase += multiplier[i];
}
```

Figure 2-2 - Generation of the marching cubes case value

The look-up table is a table of integers, ranging from 0 to 11. As seen in figure 2-3, each of these integers represents an edge of the cube. To convert these integers into 3 dimensional positions, the edge midpoint equivalent of each integer must be summed with the position of the voxel the geometry is being generated within. Additionally, to ensure the terrain is smooth, this value must be interpolated between the two vertices it is between based on the corner's density values.

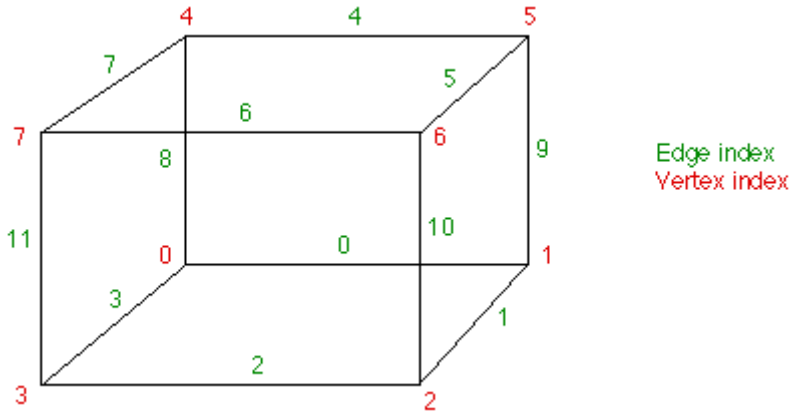

Figure 2-3 Edge/corner indexes for each voxel (Bourke, P. 1994)

To sum up the algorithm, it can be defined very generally using the following pseudo-code;

```
For each terrain chunk
{
   For each voxel
   {
      Calculate positions of the 8 corners of this voxel
      For each corner
      {
         Calculate corner's density value
         If density value is below arbitrary iso-level
         {
            Update case value based on multiplier
         }

                  Get indexed look-up table integers
                  Convert integers into triangle vertices
      }
   }
}
```

Figure 2-4 – Marching cubes algorithm pseudocode

However, the marching cubes algorithm is not the most accurate volumetric terrain generation method documented. The marching cubes

algorithm does not guarantee that all triangles will be topologically consistent, and some triangles may be generated with a poor aspect ratio (Treece et. al, 1999). Paul Bourke also provides a well-documented implementation of the Marching Tetrahedra algorithm (Bourke. P, 1997). Marching Tetrahedra is similar to the marching cubes algorithm, however it uses tetrahedral subdivision instead of cubical (Lysenko. M, 2012). The marching tetrahedra algorithm does produce a more accurate and topographically consistent surface and can be broken down to only 3 symmetrical cases, meaning calculations should be easier and faster. However, due to the tetrahedral subdivision, approximately 4 times more triangles are created in the same space when compared to marching cubes, meaning the performance is approximately 4 times slower.

Due to the low performance costs coupled with the moderately high definition still achievable, the marching cubes algorithm is the focus of this project and is used to generate the deformable terrain.

2.2 Terrain Deformation Techniques

Real-time terrain deformation has seen considerably less investigation than the broad subject of terrain generation. However there are still a number of very interesting methods of deformation already documented. Most commonly, the deformations caused by objects interacting with the terrain are investigated.

Terrain deformations can be calculated by offsetting the terrain surface based on the collision of an object. Using Subdivision Surfaces and bounding box collisions, when an object collides with one of the terrain surfaces, the overlap between the surface and the object's bounding box are calculated (Schäfer et al., 2014). A new bounding box is created around the overlap region created and the geometry within the new bounding box is voxelized. The surface is then deformed so that it is not in contact with the voxels, creating a surface surrounding the object colliding with it.

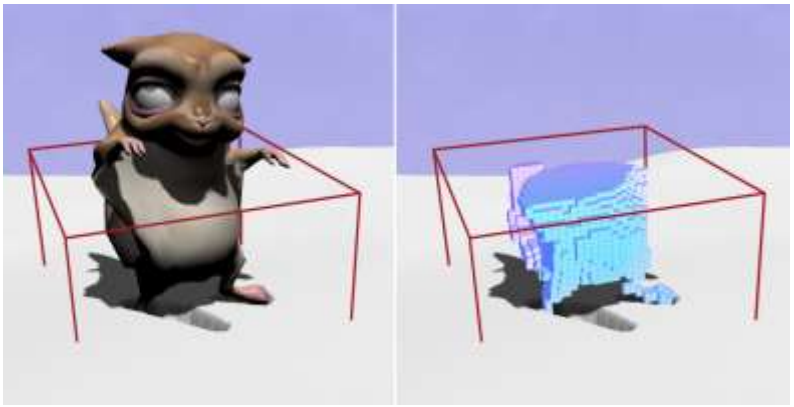

Figure 2-5 - Visualization of voxelization of object geometry within new bounding box (Schäfer et al. 2014)

The algorithm can be defined generally using the pseudocode seen in figure 2-6.

```
If an object and a subdivision of the ground collide
{
    Find where bounding boxes of ground and object overlap
    Create new bounding box using this space.
    Voxelize the geometry of the object within the new box
    Deform the ground around the shape of voxelized geometry
}
```

Figure 2-6 Pseudocode describing Deformation using subdivision surfaces and bounding box collisions

Another similar description is presented by Gilardi et al. (2016). This method has been named Drift-Diffusion. Drift-Diffusion methods deform the terrain by representing it using near-liquid physics. By using liquid calculations on contact with an object, they can distort the terrain just as they could water. However after the terrain has been initially distorted they do not allow the liquid calculations to continue to move the terrain, freezing the terrain into a solid state. This method is also very effective for calculating collision-based deformations. This deformation technique generally makes use of a height map however, limiting the terrain to a single plane.

Figure 2-7 – Liquid-like deformations achieved using the drift-diffusion deformation method (Gilardi et al. 2016)

Due to the innate possibility of complex and non-planar shapes the marching cubes algorithm presents, it makes a near perfect fit for deformation. While it does take a large amount of time to calculate the new surface, this process can be optimized to produce an application that generates real-time deformation using explosive calculations (McAllister. Z, 2013) which could fit perfectly into many interactive entertainment applications. Additionally, the complex surface of the marching cubes terrain algorithm can be used to generate terrain that mimics the terrain shapes seen in real life (Greeff, 2009). By investigating the best ways to optimize the marching cubes algorithm, this project aims to find ways by which terrain of any shape can be generated in real-time by the user at a comfortable frame-rate.

2.3 DirectX11/GPGPU Techniques

Graphics processing units have grown very powerful in recent years. With this rise in power many opportunities to manipulate the processing capabilities of the GPU have presented themselves. NVIDIA's Direct Compute Programming Guide (NVIDIA, 2010) states: the GPU has evolved into a highly capable general purpose processor capable of

improving the performance of a wide variety of parallel applications beyond graphics. General Purpose Graphics Processing Unit (GPGPU) programming is a relatively new concept based on this idea. GPGPU programming focuses on using the GPU to complete programming tasks usually performed by the CPU. Compute shaders are the primary method of accessing this. As shown in figure 2-8, compute shaders have a unique position in the shader pipeline. Being completely separate from the order of the others, the compute shader is allowed to run completely independently, while still having access to pipeline resources.

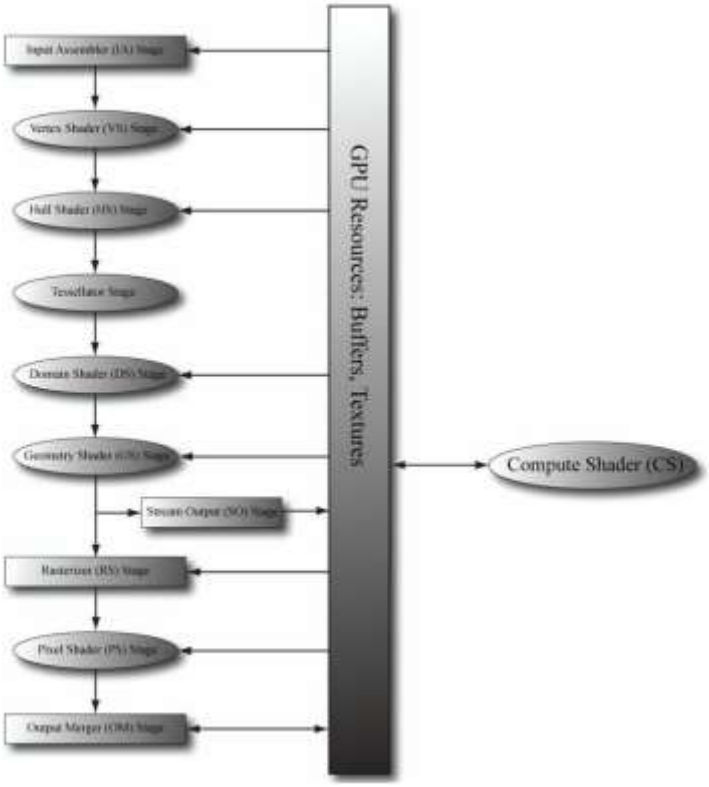

Figure 2-8 – DirectX 11 Shader stages, visualizing the position the compute shader has in the shader pipeline (Luna. F. D, 2012)

A number of frameworks that use the compute shader stage have emerged, including NVIDIA's CUDA (CUDA, 2009) and Direct Compute (Microsoft, 2007). Direct Compute especially was developed with a great focus on the needs of game applications (Boyd, 2010). The Compute shader stage makes use of the GPU's cores to create great numbers of threads, split into thread groups (Young. E, 2010), which allow the GPU

to complete tasks in parallel, meaning they are completed much faster than would otherwise be possible. Not all algorithms can benefit from this however. Only algorithms that are easily parallelizable can make good use of the compute shader stage (Luna F.D 2012). For an algorithm to be easily parallelizable it must have a collection of data elements that each require similar operations calculated on them. A good example of this would be pixel operations. If all pixels are having the same or a similar operation calculated on them then the compute shader can complete the task much faster than the pixel shader. Other examples include numerical analysis, particle systems and physics simulations.

Generally, the compute shader performs large number of calculations, outputting the results to a DirectX resource, such as a buffer or texture. This calculation may require an input. Similar to other shaders, this input is usually a buffer or a texture, passed in as a shader resource view. To retrieve data from the resource output by a compute shader an unordered access view is used (Eberly. D. H, 2015). Resources being used for input and output to compute shaders must, just as in any other shader, be bound with a bind flag defining what their use will be in the application (MSDN, no date).

As mentioned, the compute shader makes use of a collection of threads split into thread groups. The collection of threads can be imagined as a 3D grid, with threads going in x, y and z. When creating a compute shader the number of threads and thread groups in x, y and z may be specified. The threads per group has a maximum value of 1024, therefore the threads in x * threads in y * threads in z must equal no more than 1024 (MSDN, no date).

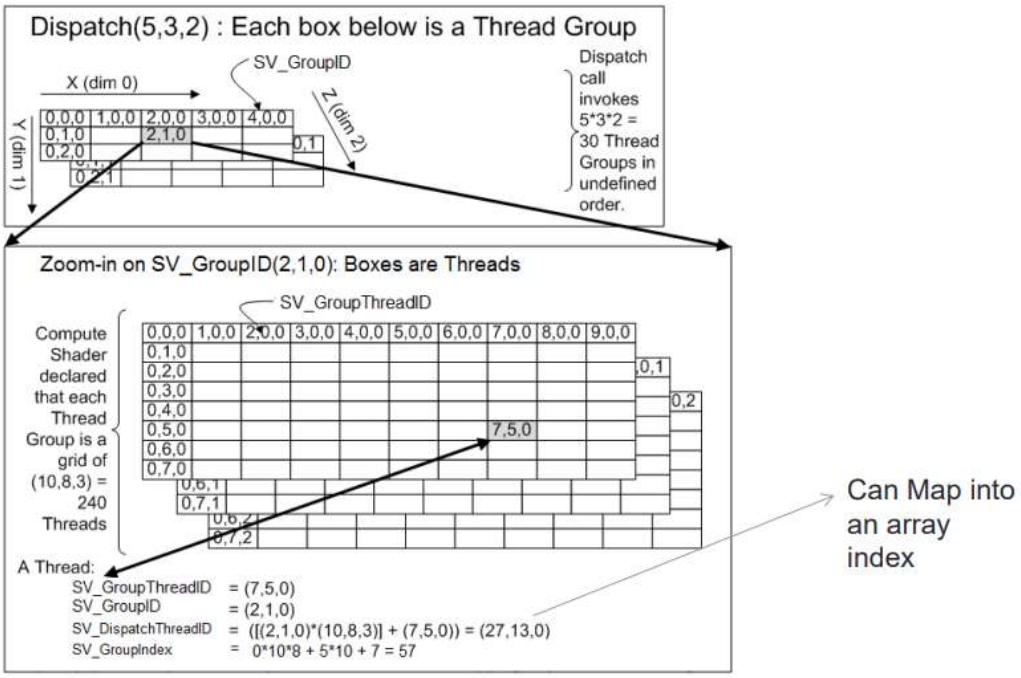

Figure 2-9 – Description of Compute shader semantics, using a dispatch of 5 x 3 x 2 thread groups, each of which contains 10 x 8 x 3 threads (Falconer. R, 2016)

Direct Compute provides the user with 4 semantics by which they may index the thread being currently processed. Using the example shown in figure 2-9, a dispatch call is made using 5 x 3 x 2 thread groups. The SV_GroupThreadID is the 3D index to the thread within the current thread group. The SV_GroupID is the 3D index of the current thread group within the dispatch of groups. The most useful of the indexes, the SV_DispatchThreadID; is a 3D index of the current thread within all thread groups. The Dispatch thread ID is calculated by multiplying the SV_GroupID by the thread group size, then adding the SV_GroupThreadID, as can be seen in figure 2-9. Finally, the SV_GroupIndex is a unique 1D index used for indexing a 1D array or buffer by the current thread. Indexing threads in a 3D grid is particularly useful for geometric visualisations, in this case a visualisation of the terrain (Engström, 2015).

To effectively manage the production of a large scale terrain using GPGPU techniques, it is often better to split the workload however possible. A common method of doing this is to split the terrain into “chunks”, using compute shaders to calculate the surface within a chunk individually. This method also provides a good starting point for deformation, as running a compute shader on the entire terrain may cause significant performance issues. Additionally, using a chunk system allows the voxel data to be processed in more manageable portions (McAllister. Z, 2013).

3 Methodology

3.1 Introduction

In order to answer the initial research question proposed by this project, an application will be developed with the aim to complete the following objective; generation of a highly optimized implementation of a deformable marching cubes terrain. This application will then be used as a basis for performance testing.

The application will be built using DirectX 11, making use of a basic graphics framework by Rastertek (2013). Detailed information on more project specific implementation was more difficult to find. Therefore much of the application was built using reference descriptions, primarily sources such as *GPU Gems 3* (Geiss, R, 2007), the function information available on *MSDN* (Microsoft, no date) and *Introduction to 3D Game Programming with DirectX 11* (F. D, Luna, 2012).

Due to a major focus of the application and project being optimization of the algorithm, the application is built to make the best use of the GPU techniques to handle calculations. This included ensuring all of the data used in the application could be read from on the CPU and generated and used on the GPU. Additionally, it was ensured that all data was easily parallelizable by the compute shader.

3.2 Evaluation/Performance Factors

To determine how successful the application was in meeting its aims, a set of performance tests will be made to gather data. These tests will gather quantitative data, with a focus on the average frame render time and terrain generation speed. These will be tested against a number of independent factors. Tests will include:

- Frames rendered per second based on number of terrain chunks.
- Frames rendered per second based on number of voxels per terrain chunk.

- Average frames rendered per second when the application is not being deformed vs. when it is being deformed continually.
- Time spent generating the terrain based on number of terrain chunks.
- Time spent generating the terrain based on number of voxels per terrain chunk.

Since a large part of the optimization of graphics applications is based on balancing the work-load between the CPU and GPU so that they finish at roughly the same time, the CPU and GPU process times for individual frames will also be measured and averaged. In order to obtain this data, profiling software must be used. *NVIDIA Nsight 5.3* (NVIDIA, 2017) will be used to test these factors due to the extensive information it can provide for single captured frames.

Additionally, to ensure that the terrain meets the intended visual standards of modern games, visual data will be presented of the terrain with varying number of terrain chunks and density generation algorithms.

3.3 Terrain Generation with Marching Cubes

3.3.1 Feasibility Demo

During the first semester of this investigation, a feasibility demo was carried out to ensure that the project was possible and within scope. One of the artefacts generated for this project was an initial Marching cubes terrain generation application. This initial application was used as a basis for the terrain generation for the rest of the project. The algorithm data was generated entirely on the CPU side and was rendered using a geometry shader. The initial thoughts behind this were that the geometry shader would provide an intuitive way to generate new areas of terrain without having to update a vertex buffer.

This initial demo presented a number of performance issues however, with frame rates consistently in single figures even for small areas of

terrain. This provided information on how to generate the final algorithm, showing the main areas that required improvements to get the level of performance that was originally intended for this project.

3.3.2 Compute Shaders

The first major step in overcoming the performance issues presented by the feasibility demo was to offload the calculations to the GPU, allowing the CPU to focus on draw calls and making use of the calculation speed that the GPU provides. The marching cubes algorithm only concerns itself with each cubic space independently, requiring no reference to other voxels around it, making it trivially parallelizable using a thread system. Due to this nature, the marching cubes algorithm is very suited to be implemented inside of a compute shader.

By mapping a thread to each voxel, the terrain can be generated on a per voxel basis. Each thread simply generates the marching cubes output appropriate for the densities generated by the noise algorithms. Since the noise algorithms are deterministic, the noise produced at the upper end of the voxel is guaranteed to be the same as the noise produced at the lower end of the voxel above it, ensuring smooth transitions between the voxels. Additionally, the calculations made by the application are always going to take approximately the same time, leaving little time for threads to be waiting for one-another to complete.

3.3.3 Vertex Buffers

Following the addition of a compute shader to generate the terrain, the main performance bottleneck of the application was the geometry shader. Due to DirectX 11’s restrictions on how many vertices could be output per pass, a draw call using the geometry shader was being used for each voxel, causing the CPU to have to send thousands of draw calls. To remediate this, the triangles generated by the compute shader were packed into a giant vertex buffer, creating the first implementation of the terrain mesh.

Drawing the terrain with several large draw calls instead of thousands of smaller ones massively increased the performance of the application. The terrain was able to go to hundreds of times its original size with the application now running at thousands of frames per second.

3.3.4 The Chunk System

The final change major change in structure made to the application was made by splitting the terrain into smaller chunks. While this change proved to actually decrease the frame-rate of the application while the terrain was static, it was done to grant utility and manageability to the application, while also preparing for the deformation system. To implement this change, a chunk class was created, each with its own set of terrain variables, creating a number of independent smaller terrains instead of one larger one.

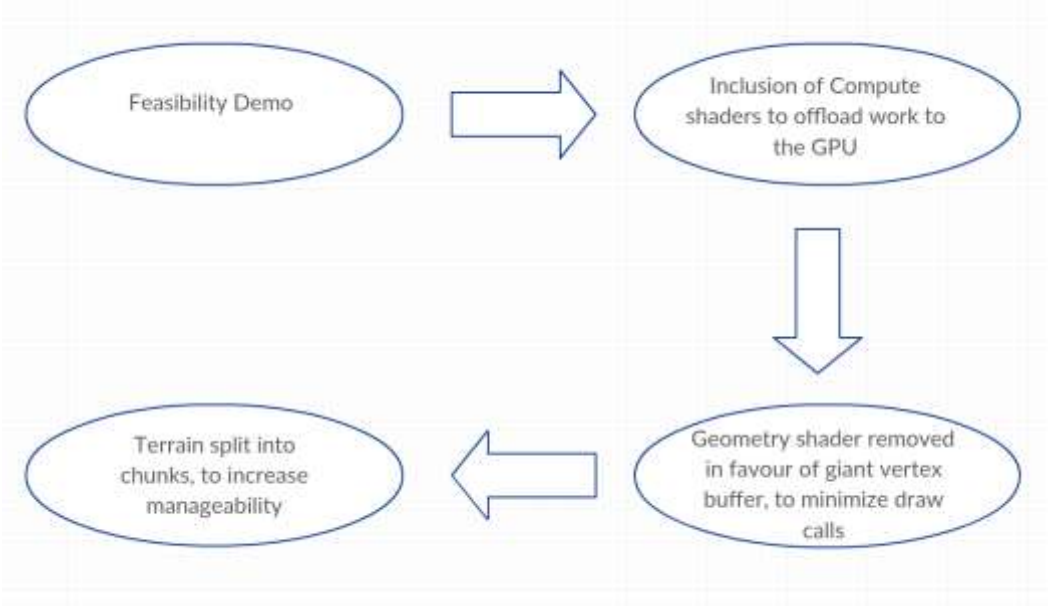

Figure 3-1 Major development advancements of the application

3.3.5 Final Artefact Terrain Generation

3.3.5.1 DirectX 11 Resources

The entire application is based around the generation, usage and manipulation of two DirectX 11 Resources. Resources are the building blocks of any DirectX 11 application. They contain the majority of the data DirectX11 uses to render the scene (Microsoft, no date). Excluding Texture2D resources used as mesh textures, the two primary resources used in the application are the density texture, stored as a 3D texture, and the voxel buffer, stored as a structured buffer.

Each terrain chunk stores its own voxel buffer and density texture. The voxel buffer is written to both during the initial generation of the terrain and whenever deformation occurs. The density texture is written to during the initial generation of the terrain, then read from in order to calculate the position of the deformation, before being written to again to calculate the new terrain surface.

In order for a resource to be both readable from and writeable to, it must be declared using different flags to a standard resource view. Inside of the shader, the resource must be declared using the prefix “RW”, or Read-Write. If an application requires a 3D texture to be read from and written to within a single shader, it would be declared as an RWTexture3D. Similarly, a structured buffer would be declared as an RWStructuredBuffer. The resource would also have to be registered using the “u”, or Unordered Access View type register (MSDN, no date). The voxel buffer and density textures are declared with read-write usage whenever used, as there is currently no scenario where write access is not required.

In order to declare a resource on the CPU which can be handled in this way by the shaders, it must be declared using specific flags. The resource must use the D3D11_BIND_UNORDERED_ACCESS bind flag, with usage remaining as default. An Unordered access view description

must be generated using a UAV view dimension of the same resource type as the resource it will be used to view. An unordered access view must then be generated using these descriptions (MSDN, no date). The application's density texture and voxel buffer are declared as described above to be used in the application's compute shaders.

3.3.5.2 Terrain Generation Shader

The final artefact's terrain's initial generation only occurs once for each chunk, hence this shader is only ever run once. In order to use the marching cubes algorithm, the look-up table had to be usable from within the shader. On the CPU implementation, this was simply stored and used exactly as Paul Bourke (1994) describes. However, DirectX 11 holds a restriction on the size of data held as local variables within a shader, a limit that the look-up table exceeds. Therefore it was decided that the table would be passed in as a constant buffer. The table layout had to be changed in order for the shader to be able to effectively access the data. Where previously the table was stored as 256 arrays of 16 integer variables, the new table is stored as an array of 1024 float4 variables. The look-up table, along with the offset of the current chunk and the diameter of each chunk, are the only inputs required for the initial terrain generation shader.

The density values at the corners of every voxel are calculated using additive variations of a Perlin noise function (K. Perlin, 2002). The shader starts by determining which of the eight densities are above and which are below the iso-level, which in the application is always 0. These values are then used to calculate the integer by which to sample the marching cubes look up table. This is done adding the positive corners' multiplication value to an integer. The multiplication value is two to the power of the current corner number (between 0 and 7 for the 8 corners). This returns a value between 0 and 255, the same size as the marching cubes look up table. The density values calculated for the corners are also placed into the 3D density texture, for use by other compute shaders later in the application's execution.

Once indexed into the look up table, up to 5 sets of 3 integer vertex positions relative to the current voxel are now known. In order to convert these integers into floating point positions, the floating point equivalent of the integers must be added to the global position of the voxel. The density values at the two corners that the vertex is in-between is used to interpolate the position between those two corners. To find the final interpolated point P_f, if P_1 and P_2 are the corners on either end of a cut edge and D_1 and D_2 are the densities at the two corners, the following equation is used:

$$P_f = (Isolevel - D_1)(P_2 - P_1)/(D_2 - D_1)$$

The triangle positions and normals are outputted from the shader in the buffer index appropriate for the current voxel. The number of triangles to be rendered in this voxel is also output as part of the struct.

3.3.5.3 Normals and Texturing

Due to the unusual shape of the terrain, normals and texture coordinates were calculated at run time on the GPU instead of pre-set on the CPU. Surface normals are calculated per-face by taking the vector cross product of two edges of each triangle (OpenGL Wiki). When being packed to the output buffer, these are output once per triangle, with a maximum of 5 normals being output.

Texture co-ordinates are set in the vertex shader used for terrain generation. As the terrain contains an unknown number of triangles, texture co-ordinates cannot be set during initialization. Additionally, due to the odd shape of the terrain, a solution must be found to calculate them at run-time on the GPU. In order to texture the terrain effectively from all angles, tri-planar texture mapping is used. Tri-planar mapping is the name given to the process of applying the texture in each axis based on the amount it is facing said axis. The normals are used to calculate the factor by which a triangle is facing an axis, and texture co-ordinates are set as the vertex's position.

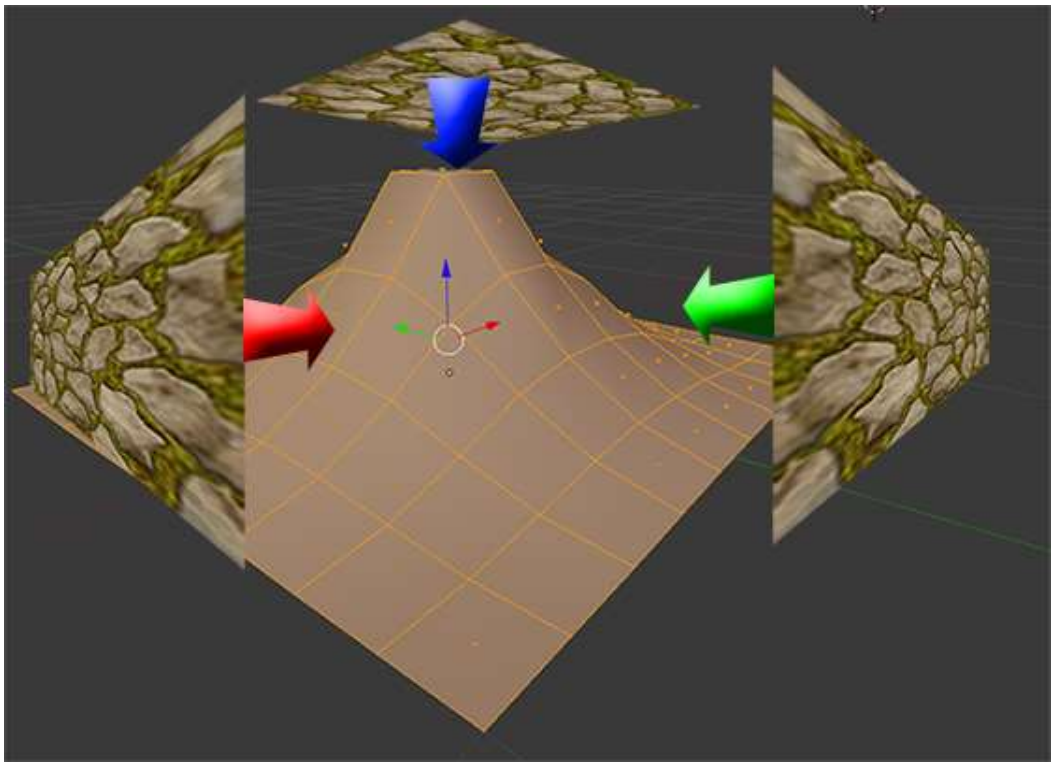

Figure 3-2 – Visualization of the texture being applied by each axis, using normals to calculate the factor by which a vertex is facing an axis (Owens. B, 2014)

3.4 Terrain Deformation

In order to give the user a controlled and enjoyable method of testing the terrain's deformation, a method of applying the deformation at a user-defined position must be developed. This was accomplished by casting a ray from the camera's current position in the direction it is currently facing. This allows the user to choose where to deform by looking at the position at which they wish to apply the deformation. A slider is used as part of the UI to allow the user to determine how large the deformation will be. The slider is given a minimum deformation radius to ensure the deformation covers at least one voxel. A maximum deformation radius is also applied to ensure it covers no more than 8 chunks in order to minimize drastic drops in performance when generating the new terrain.

3.4.1 Terrain Ray Intersection

To calculate the point the ray currently intersects the terrain, the first of two compute shaders is run. The shader is run only on each of the chunks that the ray passes through. The shader calculates which of the voxels within the chunk the ray passes through, then which of those voxels contains part of the terrain surface.

The density texture values are used to determine this value. As voxels containing the terrain surface will have a case index which is neither 0 nor 255, terrain is neither completely above nor complete below the voxel, so it must run through it. The closest of these voxels to the camera position is found, the centre position of which is output to a small structured buffer. The buffers returned from each of the chunks are compared to find the closest collision to the camera. This point of collision is then the voxel the user is currently looking at, therefore should be the point at which the deformation should be centred.

```
Initialize deformation position to a high value
For each chunk in terrain
{
    Calculate ray position
    If ray is within this chunk
    {
        Run ray intersection shader, finding closest collision in
current chunk
        If value returned is below current deformation position
        {
            Set deformation position to shader value
        }
    }
}
```

Figure 3-3 Pseudo-code describing the process used to find the centre position of the deformation

This method does not use the exact position the ray collides with the terrain, but rather the centre of the voxel the ray collides with. There are many methods that could have been used to find the exact position the ray collides with the terrain surface, however due to time constraints and a desire to find the simplest, fastest solution, this method was chosen. Using the closest voxel provides a position more than accurate enough for the current use case. At the resolution the application currently runs at, the difference between the two methods would be negligible.

3.4.2 Handling Deformation

With the deformation centre position now calculated, the terrain surface can be altered around that point. Due to the nature of the algorithm, a large number of compute threads must be dispatched in order to calculate the new terrain surface. Additionally, the vertex buffers of the terrain must be updated to accommodate the new values. This can be an extremely costly action for the application. For this reason, checks are made to only apply the compute shaders and re-initialize the buffers of the chunks within the deformation radius. To ensure that the deformation never enters more than four chunks, the maximum deformation radius the player can set is 90 percent of the diameter of each chunk. While the application could deform all of the chunks in the scene at once, it would produce a frame drop that may be uncomfortable for the user.

3.4.2.1 Deformation Shader

With this information, the compute shader can be run on all of the chunks that are within the deformation radius. The deformation algorithm is relatively simple. For each voxel being tested by a compute thread, it is determined if the distance to the voxel from the deformation centre position is less than the deformation radius set by the user. If this condition is met, the density value at that voxel is set to either 1, to deform an area of terrain, or -1, to form an area of terrain. The density of voxels whose distance to the deformation centre is greater than the deformation radius remain untouched.

```
For each voxel/thread in chunk
{
    For each corner of current voxel
    {
        Calculate distance from corner to deformation centre
        If distance is less than the deformation radius
        {
            Set density to 1/-1
        }
    }
}
Re-calculate terrain using new density values
```

Figure 3-4 Pseudocode describing the deformation process

Using a purely distance based check against the deformation radius produces a spherical check, deforming the terrain in a sphere around the centre position. This same method can be expanded to move in any variation of the x, y and z axes to create various other shapes. Additionally, as the density around the edge of the circle is always either 1/-1 or its previous density, the interpolation between densities has no effect, making the terrain very blocky wherever deformation occurs. To resolve this, a more smooth density gradient must be generated around the edge of the deformation sphere. In the case of this application, the equation of a sphere was used. This gives the deformations a very smooth, spherical look.

When both reading and writing to a single resource in the compute shader, it is important to remember that the compute threads run in parallel, however they are also not completely synchronous; some threads may run slightly faster than others. To ensure that one thread does not run faster than another and start reading a value before other threads have finished writing to it, a method of synchronising the threads

must be found. DirectX 11 provides a set of shader functions that force the shader to wait until all threads have reached that line, with different variations of the function forcing different amounts of threads to wait. This can be done by thread group, or by forcing all threads to wait. The application makes use of the latter function;

AllMemoryBarrierWithGroupSync (MSDN, no date).

This function is called after the density values have been written to the Read-Write density texture, forcing all threads to wait until all other threads have finished writing the density values before continuing to read the values back again.

The deform compute shader needs to read the values back again to re-write the voxel buffer with the new terrain vertices, replacing all of the old vertices. This is done via the same process as the initial terrain generation. The density values are used to evaluate a case value by which to index the marching cubes look up table. These integers are then used to calculate world positions for all of the triangles that make up the chunk's terrain mesh.

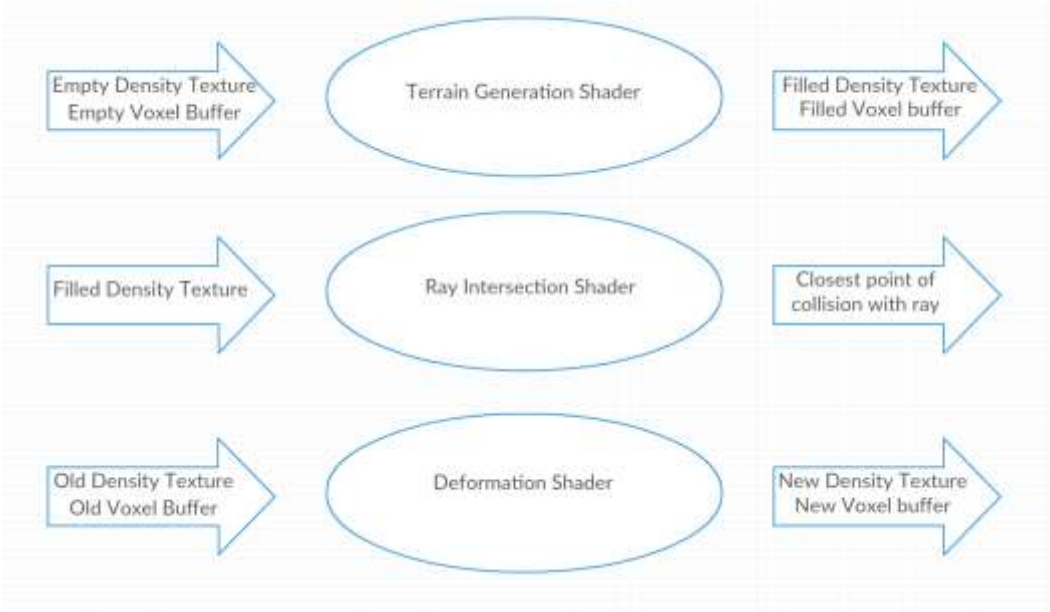

Figure 3-5 – Input/Output of compute shaders, excluding constant buffers

3.4.2.2 Updating the Mesh

In order to manipulate the shape of the terrain, the chunks affected by the deformation shader must now have the terrain mesh associated with them updated. Generally, updating a vertex buffer doesn't have to be a particularly slow operation. However, due to the shader returning an unknown number of triangles, it is extremely unlikely that the new voxel buffer will contain the same number of vertices as the previous one did. Therefore simply mapping the new values to the old vertex buffer is not possible.

To account for this, the application re-makes a new vertex buffer, replacing the information of the old buffer entirely. A new buffer description is made during run-time, the size of which is determined by the number of vertices returned by the compute shader, ensuring that the vertex buffer is precisely the correct size to have the voxel buffer data mapped to it. However, the vertex buffer expects the data to be passed in per vertex, while the compute shaders output a buffer that contains structs describing the terrain data per voxel. For this reason, the data cannot be mapped directly from one buffer to another. The data is unpacked into groups of three vertices. The total number of vertices is calculated by taking the sum of the number of triangles of each voxel multiplied by 3.

It is appreciated that there are more intuitive and perhaps less performance intensive methods of updating a vertex buffer with an undefined size than creating an entirely new buffer. For example, With DirectX 11 came the introduction of the append/consume buffer. This buffer allows you to append elements dynamically to the buffer, with a hidden element count keeping track of how many elements are currently held in the buffer. By using this type of resource for the vertex buffer, it is possible that the vertex buffer could be updated dynamically in the compute shader without having to re-initialize entire buffers. However due to the primary performance issue being related to the shader execution

instead of the vertex buffer initialization, as well as the limited time frame in which the project had to be completed, it was decided to maintain the use of the current system for updating the meshes.

4. Results

4.1 Testing Machine

To determine how effective the application was at fulfilling the aim to be a game ready, volumetric deformable terrain, the terrain was tested against a number of performance factors.

Testing was performed on a machine with the specifications shown in table 4-1.

COMPONENT	TYPE/AMOUNT
CPU	Intel core i5-6500 @ 3.2 GHz
GPU	NVIDIA Geforce GTX 1060 3GB
MEMORY	32GB DDR4
MOTHERBOARD	Gigabyte Intel LGA1151
OS	Windows 10 Home

Table 4-1 – Test Machine Specifications

As the focus of the investigation was on producing a deformable voxel terrain with high enough performance rates for games, the focus of the application testing was on the frames per second and generation time of the application under different test conditions. Much of this testing focuses on how to maximize the efficiency of the compute shaders, as well as testing the effect changing the size and resolution of the terrain makes on the applications performance. The majority of the results were gathered in application, excluding the CPU/CPU processing time comparison, the information for which was gathered using the frame capture feature in NVIDIA Nsight 5.3 (NVIDIA, 2017).

Generation time results were gathered as a timer from the moment the application is run until rendering begins. Frames per second calculations were made as an average of the frames rendered each second over 30 seconds.

4.2 Data and Discussion

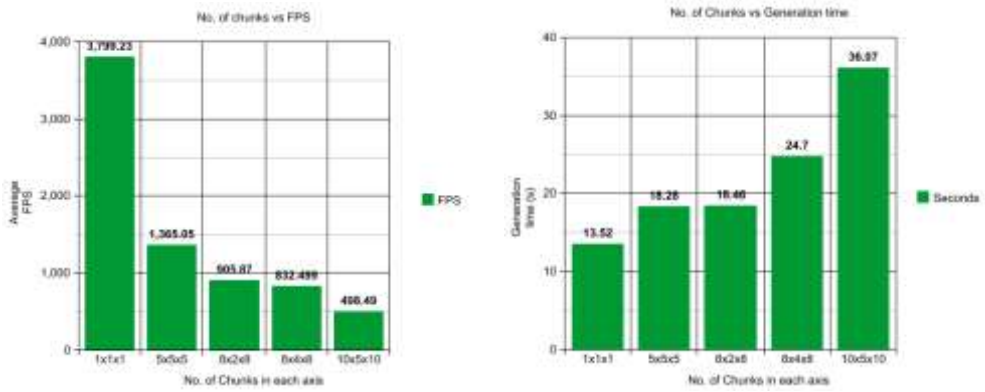

Figure 4-1 – FPS/Terrain Generation time vs Number of chunks per axis

As shown in Figure 4-1, the number of chunks generated and being rendered in the application has a direct linear effect on the frames rendered per second. It is worth noting that increasing the number of chunks generated in the y axis has less effect on the frame-rate of the application than increasing the number of chunks in the x or z axis. This can be seen by the relatively small difference between the results for 8 x 2 x 8 chunks and 8 x 4 x 8 chunks. This can be understood when testing the algorithm, as it can be seen that the majority of the terrain surface upon initial generation is in the first or second chunk in the y-axis. In the context of the terrain, adding additional chunks in the y direction generally adds more empty space, which has little impact on the frame-rate as nothing is rendered.

The same trend does not apply to the generation time, however. This can be observed by comparing the difference in the generation time between the 5 x 5 x 5 terrain and the 8 x 2 x 8 terrain. The difference in the total number of chunks generated for these two terrains is only 3, as the 5 x 5 x 5 terrain produces 125 chunks, while the 8 x 2 x 8 terrain produces 128. Therefore, as would be expected, the 8 x 2 x 8 terrain takes marginally longer to generate. In the context of the terrain this might seem odd, due to there being no terrain in the upper y axis chunks. However, the majority of the performance bottleneck at the beginning of the program is generating and allocating memory for the buffer containing the terrain

voxels. These buffers are generated before the terrain vertices themselves, therefore they are generated irrespective of the terrain shape.

Compute shaders are used in the application to maximize the speed at which the terrain data can be processed and generated in real-time. However, despite the power of the compute shaders, they are still a major bottleneck in the application's performance due to the scale of data being produced. Therefore, investigations into maximizing their performance must be undergone.

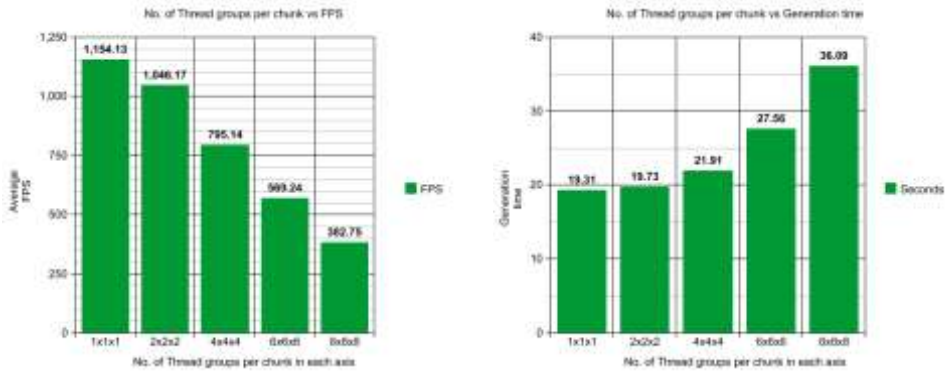

Figure 4-2 – FPS/Terrain Generation time vs Number of thread groups per chunk per axis

In order to determine how the basic features of the compute shader can be best used to gain a good balance of visual complexity and high frame-rate, tests were done on the application, varying compute features. Tests were done similarly to the chunk tests, with results being in average frame-rate and terrain generation time. These results were gathered by varying the number of thread groups per chunk and the number of threads per thread group.

The number of thread groups per chunk describes the resolution of a chunk of terrain. If the number of thread groups is 1 in each axis, then there is only a single group of threads running for that voxel. 8 threads were run in each axis when testing against the number of threads per chunk, therefore a chunk running a single thread group would have 512 voxels, a relatively low resolution. Comparatively, a chunk of 8 x 8 x 8

thread groups would contain 262144 voxels, a very high resolution. Therefore, as can be seen in Figure 4-2, the results for the applications performance when altering the number of thread groups per chunk was as would be expected, with the fps decreasing as the terrain resolution was increased, and the time taken to generate the terrain increasing with the increase in resolution. Unfortunately, due to the nature of the implementation, there was no capacity for testing how resolutions without equal values in each axis would compare to the current results, where thread groups in x = thread groups in y = thread groups in z.

It is also worth noting that generally speaking the same trend can be found by altering the number of threads per thread group. As the number of threads per group increases the frame-rate can be seen to decrease and the generation time can be seen to increase, as shown in figure 4-3. However, a small anomaly can be seen in these results. Where the number of threads in each axes is not equal, for example the results for the 3 x 4 x 5 threads per group, the fps is lower than would be expected and the generation time is higher. When creating compute shaders, it is highly advised to stick to multiple of the warp size when dictating the number of threads to create. The warp size is usually 32 or 64, depending on the API. When there are 4 x 4 x 4 threads per group, this makes a total of 32 threads per group, a multiple of the warp size. This is likely why the 4 x 4 x 4 threads per group simulation runs at a higher frame-rate than the 3 x 4 x 5 simulation, which does not use a number of threads that is a multiple of the warp size.

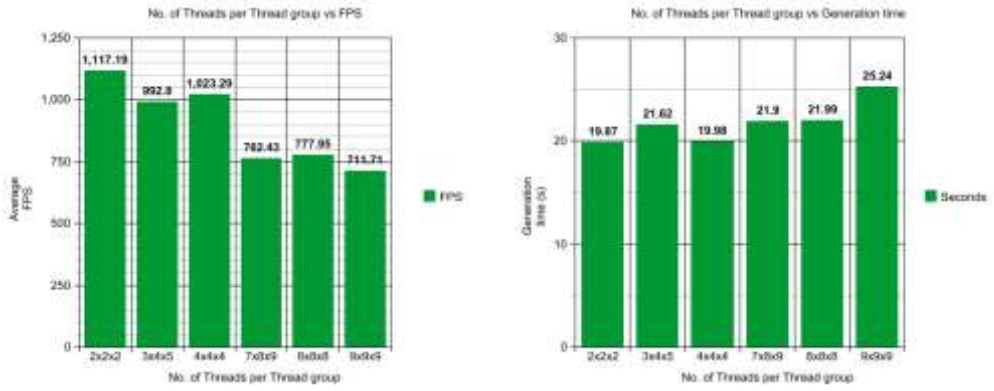

Figure 4-3 – FPS/Generation time vs Number of threads per thread group

These results provide a useful description of the performance of the terrain while static. However to truly define the success of the application, it must be tested while deforming. To do this, the application's frame-rate was tested while undergoing a strenuous amount of deformation. The tests were made using a terrain with 8 x 2 x 8 chunks, 4 x 4 x 4 thread groups per chunk and 8 x 8 x 8 threads per group. When observing these results it is important to consider that the strain put on the application for this test is much higher than the average user is likely to put on it, as the limits of the application were being tested.

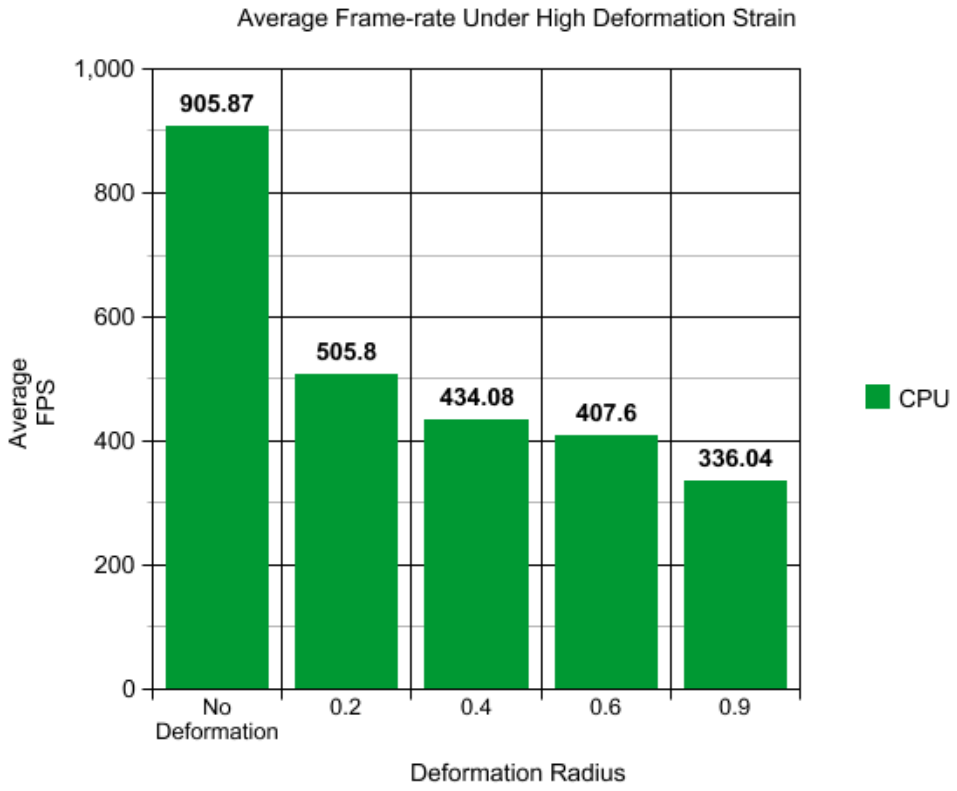

Figure 4-4 – FPS while under high deformation strain

As can be seen in Figure 4-4, continual, large amounts of deformation do have a large effect on the average frame-rate of the application. Larger sizes of deformation are more likely to cover more than one chunk of terrain, with sizes above 0.5 being guaranteed to cover more than one chunk, as they are larger than the radius of the chunk. Therefore, multiple sizes of deformation were tested against to gather how much the effect of

the deformation differs when more chunks are deformed on average per frame. The results shown are useful for displaying the decrease in frame-rate when deformation is applied. However they do not give an entirely accurate account of the effect deformation has on the frame-rate. The user can only press deform so fast, therefore when taking the average frame-rate, many of the frames being tested have no deformation occurring during them. It is worth mentioning that when running the application, large amounts of deformation may cause the application to stutter very slightly. This is expected when running a large number of threads on a compute shader and updating large vertex buffers in a single frame.

As mentioned previously, the application developed in many ways from its initial implementations. The original problems and the improvements which solved them can be seen by testing the duration both the CPU and GPU take to finish rendering a single frame. Figure 4 displays the resulting decrease in process time as the application progressed.

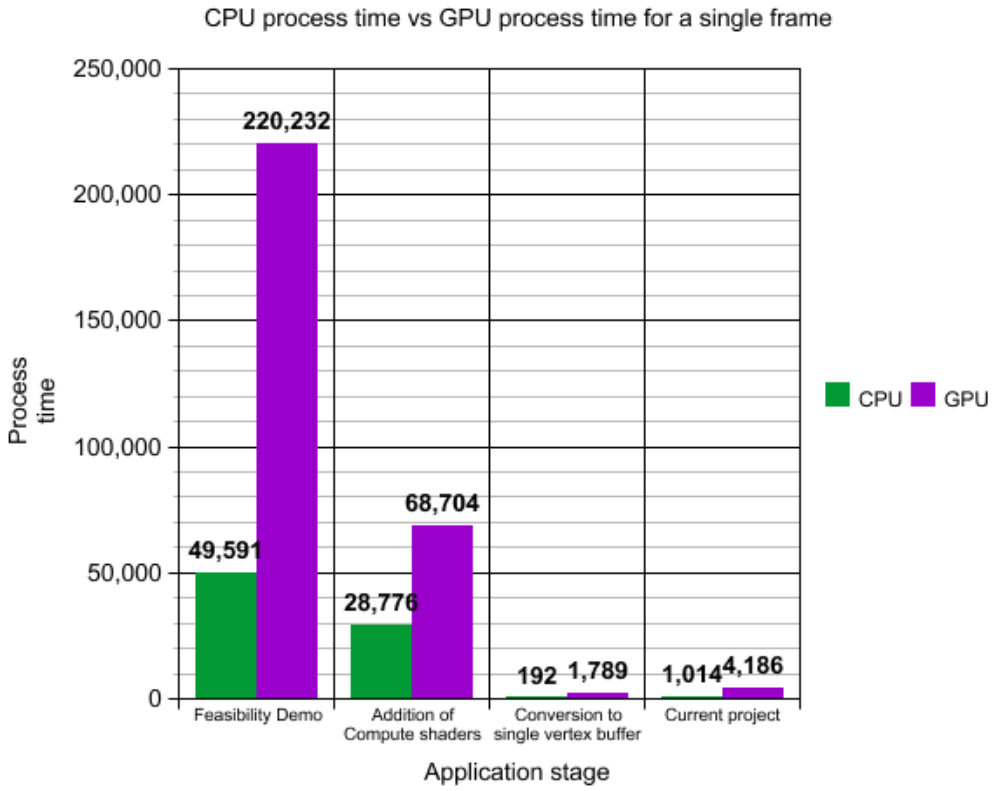

Figure 4-5 – CPU vs GPU (process time for a single frame)

As can be seen, there was a drastic decrease in process time on both the GPU and CPU from the application during the feasibility demo of the project and the current application. By testing the applications CPU and GPU process time in a single frame, ways in which the applications performance could be increased become clear. After the initial demo, testing revealed that while the entire application was slow, the GPU was doing particularly poorly. This lead to the implementation of the compute shaders. Moving the majority of the maths out of the geometry shader and into the compute shader drastically decreased the GPU processing time, bringing the CPU and GPU closer to equal processing times.

To find out why the values were still so high, further investigation took place. Upon further inspection of the function calls slowing down the CPU, it became clear that the Draw function was being called far too often, taxing both the CPU and GPU with many unnecessary render calls. To solve this problem, the geometry shader was removed and replaced with one huge vertex buffer. As can be seen, this brought both the process times to a very manageable level, massively increasing the frame-rate of the application. Notably, the single vertex buffer is faster than the current version both on the CPU and the GPU. Unfortunately this sacrifice had to be made in order to accommodate the deformation aspect of the application. Were deformation not a goal of the project and a static mesh instead desired, a single vertex buffer would serve better as a very efficient method of rendering the terrain mesh, although the difference is negligible in most scenarios.

Finally, figures 4-6, 4-7 and 4-8 (see appendix) display the visual aspect of the application,

5. Conclusions

5.1 Project Conclusion

To give a definition of the success of the project, as well as a conclusion to describe this, the original research question must be answered. The research question asked to start this project was:

Can a volumetrically generated voxel terrain be effectively implemented to allow for deformation at run-time with the use of GPGPU Techniques?

To have accomplished this, an application must have been generated to allow for deformation of a volumetrically generated voxel terrain at run-time, making use of GPGPU techniques. Following this, the project must also meet the criteria set out by the original aims. The first two aims of this project were to create terrain entirely on the GPU, using compute shaders, and to allow this terrain to deform at run-time. Both of these criteria have been met. The final aim of the project was to implement the application in such a manner that allowed for it to perform at a high frame-rate, such that it would be appropriate for use in any game.

To analyse whether the final aim of the project was met, a large selection of results were taken on the frame-rate of the application and generation time of the terrain. The average expected frame-rate of a game application is, at a minimum, 30 frames per second, with 60 frames per second being desired. Both when static and when deformed the application easily meets this criteria, accomplishing all the aims. However note must be taken of the frame drops occasionally seen when deformation takes place. This cannot be seen in the results as the average frame-rate is brought much higher due to the number of frames where deformation is not taking place. However, it does impact how successful the project is as a whole. While this frame-drop is not unexpected when running such a large number of compute threads and

altering large vertex buffers, there are methods which could be further looked into to reduce this frame-drop.

5.2 Future Work

5.2.1 Reducing the frame drop

5.2.1.1 Vertex Buffer Re-Initialization

Were this project to continue development, the first step would be to look into further optimizing the algorithm to remove the potential frame-drops when large amount of deformation are taking place. This could be done, first and foremost, by finding an alternative method of updating the vertex buffer. With DirectX 11 came a new type of buffer, the append/consume buffer. An append buffer enables a shader to append all of the values to the buffer every shader pass, using a hidden dynamic count to keep track of how large the buffer is. Currently a new buffer is created matching the size required to perfectly fit the new number of triangles in a chunk after deformation. However, by using an append buffer to store the terrain's vertices, it is possible to update the mesh without re-initialization of buffers. This was investigated as part of the project. However due to time constraints, the requirement to overhaul much of the current project, and more pressing optimization concerns, it was never implemented.

5.2.1.2 Compute Shader Optimization

While compute shaders are generally a highly optimized method of performing large numbers of calculations, they still provide performance concerns when run at large sizes. Part of this problem is related to branching operations. When a branch operation is performed inside of a compute shader, much of the performance optimizations cannot take place before the code is run, therefore the increase in speed the shader provides is significantly lower. While the project is already highly optimized to avoid these branch operations, a number still exist, including in the deformation. Future Work may look into finding methods of

avoiding these operations to ensure the compute shader can perform all required optimizations.

Additionally, much of the threads run in the compute shaders during deformation are not actually affected, as they are outside of the deformation radius. To maximize the efficiency of the deformation algorithm, methods of only running threads where deformation actually takes place may need to be investigated. With careful indexing of the voxel buffer, it may be possible to reduce the number of threads the deformation shader needs to run.

5.2.2 Threaded Terrain Generation

Currently, the project is restricted to the number of chunks the user sets in the code before compilation. The number of chunks, in turn, is restricted by the processing power of the CPU and CPU of the machine the application is run on. Due to the large amount of memory allocation that takes place on the initial generation of the terrain, running large numbers of chunks will likely crash the application before rendering ever takes place. To remediate this, a method of threading terrain generation may be found. By using parallel processing on the CPU, chunks may be assigned their own thread to continue generating after the initial generation is completed. Using threads would allow the user to begin using the application with the terrain currently generated while terrain continues to generate further away. Due to the large average frame-rate the project is currently managing, a significantly larger amount of terrain could be generated before incurring any performance issues.

Additionally, by applying threads to the terrain generation and allowing terrain generation to continue in run-time, terrain may be generated infinitely around the player. Infinite terrain generation was a large interest when the project began, however it was quickly decided it was out of scope for the time provided. With more time spent on the project, the application could be modified to continually generate terrain in the

direction the player is moving, making use of the memory previously occupied by terrain behind the player by releasing their chunk's data.

5.2.3 Deformation Shape

The deformations seen in the current application are very smooth and extremely spherical. While this produces clean, obvious deformations, a user may desire more rough-edged deformations, to allow them to more effectively reproduce the terrain we see in the real world. There are a number of ways to do this. To add a slight rough-ness to the shape, noise could be added to the density values applied during the deformation, however this would still give a generally spherical shape. In order to apply another shape, the deformation function could be altered to check the distance in each of the axes separately, allowing for a malleable deformation shape. The deformation shape can be altered in any number of ways. As there is no perfect shape for deformation, the shape required is application specific, and should be altered to suit the desired use.

Appendices

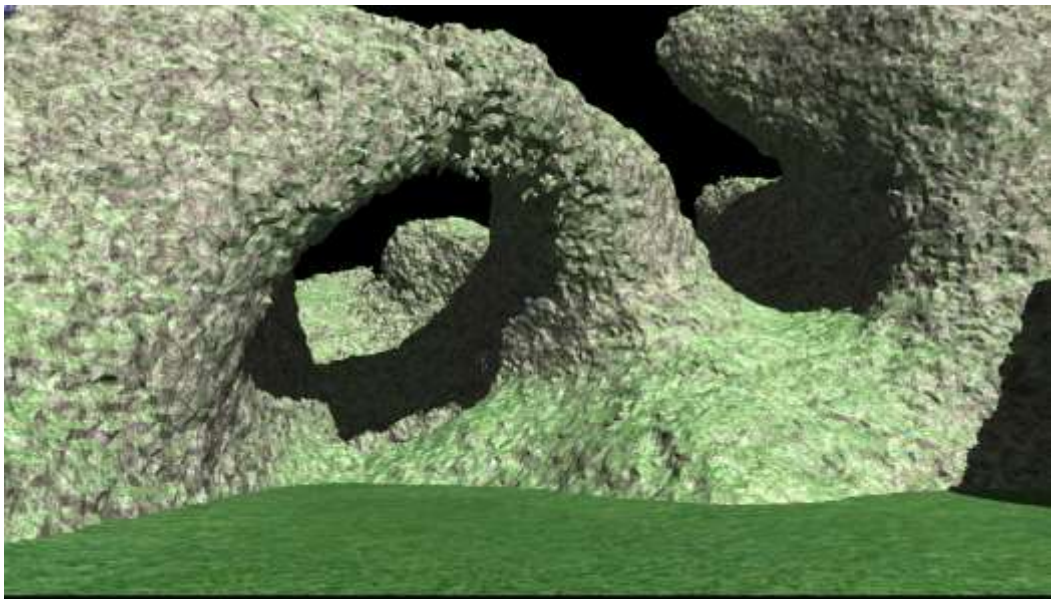

Figure 4-6 Demonstration of terrain's capability for arches and overhangs

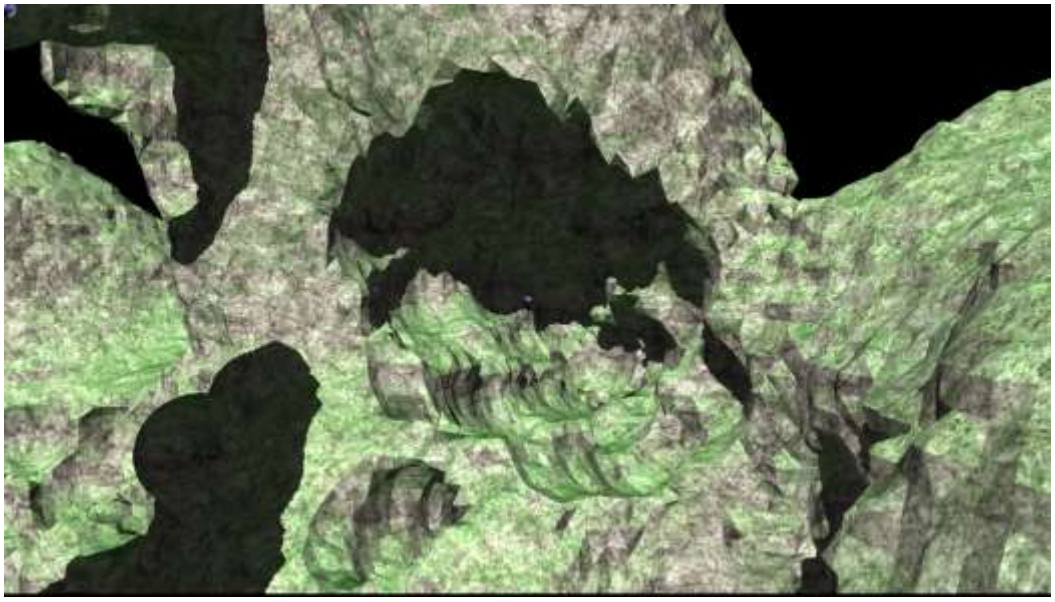

Figure 4-7 - Demonstration of deformation applied to create a small cavern in a rock face

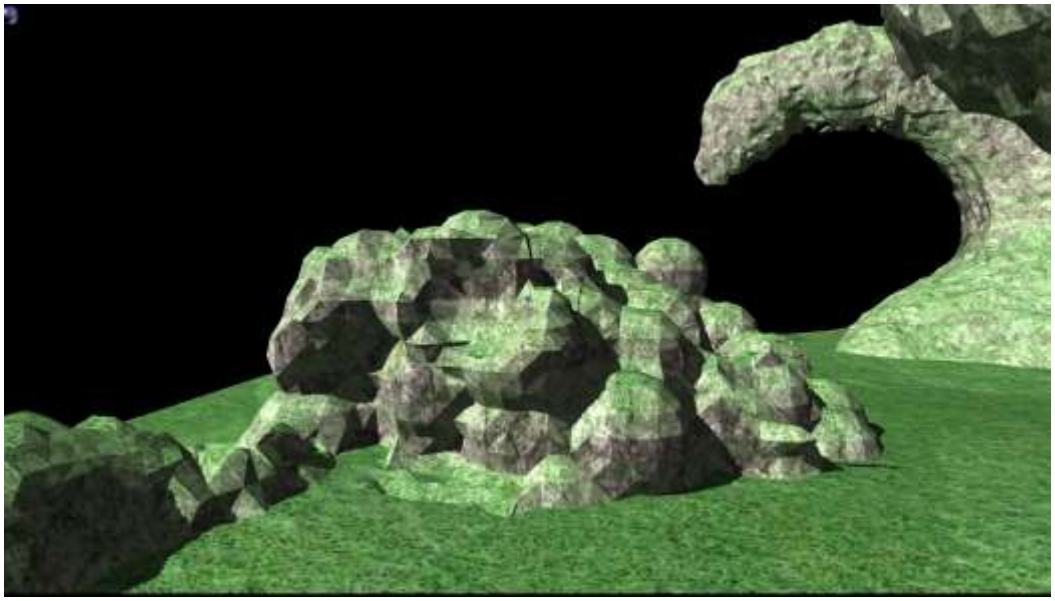

Figure 4-8 – Demonstration of formation tool used to create a rock structure

List of References

Bourke, P. (1994) Polygonising a scalar field. Available from: http://paulbourke.net/geometry/polygonise/ [Accessed 18 October 2017]

Bourke, P. (1997) Polygonising a scalar field Using Tetrahedrons. Available from: http://paulbourke.net/geometry/polygonise/ [Accessed 18 October 2017]

Boyd. C (2010) Direct Compute lecture series 101: introduction to Direct Compute. Available from: https://channel9.msdn.com/Blogs/gclassy/DirectCompute-Lecture-Series-101-Introduction-to-DirectCompute [Accessed 3 February 2018]

Doran. J (2010) Controlled Procedural Terrain Generation Using Software Agents. Available from: https://pdfs.semanticscholar.org/dad9/513db0dd0c6e424e10e2f9065333fa96ea25.pdf [Accessed 03 December 2017]

Eberly. D. H, 2015 GPGPU programming for Games and Science. Redmond Washington, Geometric Tools LLC.

Engström (2015) Volumetric Terrain Generation on the GPU. Available from: https://www.diva-portal.org/smash/get/diva2:846354/FULLTEXT01.pdf [Accessed 03 December 2017]

Falconer. R (2016) GPU Mandelbrot. Dundee. Abertay University.

Geiss. R, NVIDIA (2007) Generating Complex Procedural Terrains Using the GPU. Available from: https://developer.nvidia.com/gpugems/GPUGems3/gpugems3_ch01.html [Accessed 18 October 2017]

Gilardi et al (2016) Drift-Diffusion based Real-Time Dynamic Terrain Deformation. Available from: https://www.researchgate.net/publication/292995143_Drift-Diffusion_Based_Real-Time_Dynamic_Terrain_Deformation [Accessed 10 November 2017]

Greeff, G (2009). Interactive Voxel Terrain Design Using Procedural Techniques. Stellanbosch: Stellanbosch University.

Halford. C (2015) Procedural Methods: Additive Terrain Faulting. Available from: http://codetrip.weebly.com/blog/year-3-semester-2-procedural-methods-additive-terrain-faulting [Accessed 03 December 2017]

Luna. F. D (2012) Introduction to 3D Game Programming with DirectX11. Dulles, Mercury Learning and Information.

Losh, S (2016) Terrain Generation with Midpoint Displacement. Available from: http://stevelosh.com/blog/2016/02/midpoint-displacement/ [Accessed 23 March 2018]

Lorensen, W. E., Cline, H E. (1987). Marching cubes: A high resolution 3d surface construction algorithm. Available from: https://dl.acm.org/citation.cfm?id=37422 [Accessed 18 October 2017]

Lysenko. M (2012) Smooth Voxel Terrain (part 2). Available from: https://0fps.net/2012/07/12/smooth-voxel-terrain-part-2/ [Accessed 15 January 2018]

McAllister. Z (2013) Real-Time Smooth Deformation of Voxel Terrain with Direct Compute. Dundee. Abertay University.

Microsoft (2009) Direct Compute [Software] Microsoft.

Mojang (2009) Minecraft [video game]. Microsoft

Microsoft (no date) MSDN. AllMemoryBarrierWithGroupSync function. Available from: https://msdn.microsoft.com/en-us/library/windows/desktop/ff471351(v=vs.85).aspx [Accessed 18 March 2018]

Microsoft (no date) MSDN. Available at: https://msdn.microsoft.com/en-us/dn308572.aspx [Accessed 20 September 2018]

Microsoft (no date) MSDN. Compute Shader Overview. Available at https://msdn.microsoft.com/en-us/library/windows/desktop/ff476331(v=vs.85).aspx [Accessed 16 December 2017]

Microsoft (no date) MSDN. D3D11_BIND_FLAG enumeration. Available at: https://msdn.microsoft.com/en-us/library/windows/desktop/ff476085(v=vs.85).aspx [Accessed 20 February 2018]

Microsoft (no date) MSDN. ID3D11Device::CreateUnorderedAccessView method. Available from: https://msdn.microsoft.com/en-us/library/windows/desktop/ff476523(v=vs.85).aspx [Accessed 12 December 2017]

Microsoft (no date) MSDN. Introduction to a Resource in Direct3D 11. Available from: https://msdn.microsoft.com/en-us/library/windows/desktop/ff476900(v=vs.85).aspx [Accessed 12 December 2017]

Microsoft (no date) MSDN. Register. Available from: https://msdn.microsoft.com/en-

us/library/windows/desktop/dd607359(v=vs.85).aspx [Accessed 14 February 2018]

Newman T. S & Yi. H, (2006) A survey of the marching cubes algorithm. Computer & graphics. 30(5): pp 854 – 879. Available from: https://www-sciencedirect-com.libproxy.abertay.ac.uk/science/article/pii/S0097849306001336 [Accessed 5 March 2018]

NVIDIA (2007) CUDA [Software] NVIDIA.

NVIDIA (2010) DirectCompute Programming Guide. Available from: http://developer.download.nvidia.com/compute/DevZone/docs/html/DirectCompute/doc/DirectCompute_Programming_Guide.pdf [Accessed 03 December 2017]

NVIDIA (2017) NVIDIA Nsight 5.3 [Software] NVIDIA

Olsen. J (2004) Realtime Procedural Terrain Generation. Available from: http://web.mit.edu/cesium/Public/terrain.pdf [Accessed 03 December 2017]

Open GL Wiki (2013) Calculating a Surface Normal. Available from: https://www.khronos.org/opengl/wiki/Calculating_a_Surface_Normal [Accessed 23 April 2018]

Owens. B (2014) Use Tri-Planar Texture Mapping for Better Terrain. Available from: https://gamedevelopment.tutsplus.com/articles/use-tri-planar-texture-mapping-for-better-terrain--gamedev-13821 [Accessed 29 October 2017]

Perlin. K (2002) Improved Noise Reference Implementation. Available from: http://mrl.nyu.edu/~perlin/noise/ [Accessed 01 May 2018]

Ramstedt. R & Smed. J, 2016 Midpoint Displacement in Multifractal Terrain Generation. Available from: https://www.researchgate.net/profile/Jouni_Smed/publication/313367166_Midpoint_Displacement_in_Multifractal_Terrain_Generation/links/58982bd84585158bf6f5a168/Midpoint-Displacement-in-Multifractal-Terrain-Generation.pdf [Accessed 03 December 2017]

RasterTek (2016) DirectX 11 Tutorials. Available from: http://www.rastertek.com/tutdx11.html [Accessed 30 April 2018]

Schäfer et al. (2014) Real-Time Deformation of Subdivision Surfaces from Object Collisions. Available from: https://graphics.stanford.edu/~niessner/papers/2014/4subddef/schaefer2014subddef.pdf [Accessed 03 December 2017]

Treece. G et. al (1999) Regularised Marching Tetrahedra: Improved Iso-surface Extraction. Computer & Graphics 23(4): pp 583 – 598. Available from: https://www-sciencedirect-com.libproxy.abertay.ac.uk/science/article/pii/S009784939900076X [Accessed 10 February 2018]

Young. E (2010) Direct Compute Optimizations and Best Practices. Available from: http://www.nvidia.com/content/GTC-2010/pdfs/2260_GTC2010.pdf [Accessed 1 February 2018]

Bibliography

Falconer. R (2016) GPU Memory: Image Convolution Optimisation example. Dundee. Abertay University.

Microsoft (no date) MSDN. How To: Create a Compute Shader. Available at https://msdn.microsoft.com/en-us/library/windows/desktop/ff476330(v=vs.85).aspx [Accessed 16 December 2017]