Honours Project Report

Real-Time Deformable Terrain using Texture-Based Methods

Justin Crause

Supervisor: Dr Patrick Marais

	Category	Min	Max	Chosen
1	Software Engineering/System Analysis	0	15	10
2	Theoretical Analysis	0	25	0
3	Experiment Design and Execution	0	20	0
4	System Development and Implementation	0	15	15
5	Results, Findings and Conclusion	10	20	10
6	Aim Formulation and Background Work	10	15	15
7	Quality of Report Writing and Presentation	10		10
8	Adherence to Project Proposal and Quality of Deliverables	10		10
9	Overall General Project Evaluation	0	10	10
Total marks		**80**		**80**

Department of Computer Science

University of Cape Town

2010

Abstract

This report explores a solution to creating a system capable of real-time deformable terrain that is capable of producing both coarse and high-detail deformations. Such a system provides significant challenges that need to be overcome. This is a reason why deformable terrains are not commonly used in computer game systems and when it is used the deformations are limited to pre-defined ways which lack true realism. Previous work on deformable terrain systems are usually low quality or offline processes.

The application handles two types of deformations; the coarse deformations are applied to the terrain using vertex displacement techniques which affects the underlying surface geometry. In order to allow for high-detail deformations, parallax mapping is used which works on texture trickery to produce illusions of height on the surface of the terrain.

This report shows that such a system has been produced and the results prove that the system can be maintained in real-time on mainstream hardware. A real-time terrain deformation system is proved to be viable but this research prompts further exploration to improve upon it.

Acknowledgements

I would firstly like to thank my supervisor, Dr. Patrick Marais for his support and guidance during the course of the project. Also I would like to thank my project partners, Andrew Flower and Peter Juritz. Andrew has been there at all hours to help and support in developing the project and for his amazing coffee and food during the long days of development at his house. I would like to thank my classmates for all the long nights in the lab and fun meals in the pub. Thanks to my family and friends who have stuck by me even when I had to miss birthdays in order to finish assignments or study. Lastly to my uncle who looked after me for the last two years and was always supportive of me with everything, you will be missed greatly.

BELIEVE

Contents

List of Figures

List of Tables

Chapter 1: Introduction

Graphics based applications such as computer games and animated films have ever-increasing requirements. A large number of these applications require the display of virtual terrains, and up until recently the majority of these terrain systems have been restricted to static geometry. In order to implement deformable and dynamic terrains new techniques must be developed that can produce visually appealing results and are furthermore able to perform under the time constraints that many of these applications include. These results need to be achievable in real-time in order to be of use in gaming and visualisation applications. The project focuses on the creation of such an application.

The goal of this project is to create a terrain deformation test-bed that supports the real-time deformation of a triangle mesh representing a terrain. The system must be capable of handling a large number of deformations whilst maintaining real-time frame-rates. Ultimately this system will serve as basis for creating realistically deforming terrain environment for use in computer games and visual effects. While methods do exist to change geometry on-the-fly, these have not been widely adopted in computer games mainly due to previous limits imposed by graphics hardware. Currently most computer games rely on pre-computed data, meaning that the terrain (or other environmental objects) can only be changed in a small number of predefined ways, thus breaking immersion.

This project aims to evaluate methods to achieve these goals which do not require pre-computed information and can run on mainstream graphics hardware. To this end, the necessary trade-offs in speed and representational errors need to be carefully analysed. Two deformation implementations are examined during this project, one utilising geometry shaders to add new geometry and the other making use of texture methods to achieve similar visual effects but yielding better performance. These two implementations are used to display the high-detail deformations on the terrain. The contents of this report cover the texture-based implementation.

It is shown in this report that the goal of a real-time deformation terrain has been achieved which can produce both coarse-detail and high-detail deformations. The system is capable of producing persistent user generated terrain with high visual quality and can maintain real-time frame-rates which would be required for application in computer games.

Chapter 2: Background

2.1 Overview of Research

The primary objective of this chapter is to review the current techniques that are available and pertain to deformable terrain. Surprisingly, there is very little use of deformable terrain in computer games, which would greatly benefit from such a system. Games that do provide environment destruction usually do so in a predefined manner which lacks realism. With games requiring a real-time environment in order to be playable, key techniques have been identified which aim to allow for real-time terrain deformation. Particular attention is paid to bump mapping and displacement mapping, which is the primary focus of this report. A definition for each of these is presented along with a brief evaluation and comparison. Alternative methods are defined as well as the limiting factors which make them less desirable given the focus of this report.

Bump and displacement mapping techniques have been around for many years in computer graphics. Bump mapping was first introduced by James F. Blinn in 1978 [1] and displacement mapping by Robert L. Cook in 1984 [2]. The techniques presented here cover topics from many years ago and as such, limitations [3] that existed at the time those papers were written may not exist currently. For example, in 1978 the time taken to render a simple sphere with a bump map would amount to several minutes [1] but with the aid of modern computer graphics hardware, this has been reduced to a fraction of a second.

2.2 Texture based methods

Texture mapping is a well known and widely used method to enhance the realism of computer generated content [3]. The ability to wrap a 3D surface in a two-dimensional texture to provide vivid and accurate detail is an integral part in modern computer graphics. The techniques discussed below all utilise textures in some form to provide the additional data required. Texture maps not only store colour information but can also be set to store normal or height values. This capability plays a fundamental role in the techniques described below.

The primary focus is to achieve real-time deformation of a terrain surface by exploiting texture-trickery through the use of bump and displacement maps.

2.3 Bump Mapping Techniques

Smooth, computer generated surfaces are generally not very realistic. Blinn [1] proposed a technique to simulate surface irregularities that are inherent in real objects. A pre-computed image of a wrinkled or bumpy surface is used to compute surface irregularities. The bump mapping technique is applied as a stage in the lighting calculations applied to the scene. On current hardware this occurs in the vertex and fragment shaders of the GPU [4].

Bump mapping works by using additional data that is stored in texture maps. This data is used by the various bump mapping techniques that make use of simplified proxy geometry [3] in place of the original object and then add back the higher detail. This is an important advantage of bump mapping but the simplified geometry is still unrealistic. This can be solved through the use of displacement mapping which is covered in section 2.4.

2.3.1 Normal Mapping

Normal mapping is the first of the bump mapping techniques being considered. Normal mapping works by the perturbation of the normals only and is purely illusory as it does not alter the vertex positions which, is how displacement mapping works discussed in section 2.40. It cannot alter the smooth silhouette edges of the object [1] nor can it account for self-occlusion of the surface features.

In a typical normal map each texel encodes the 3D normal of the corresponding surface point [3]. Bump maps can be useful to render high-frequency geometric detail on a given object. A small normal map can be applied over a large object to add repetitive detail. A common example would be an orange skin or brick wall [3]. An example of the difference between straight texture mapping and normal mapping is shown in Figure 1.

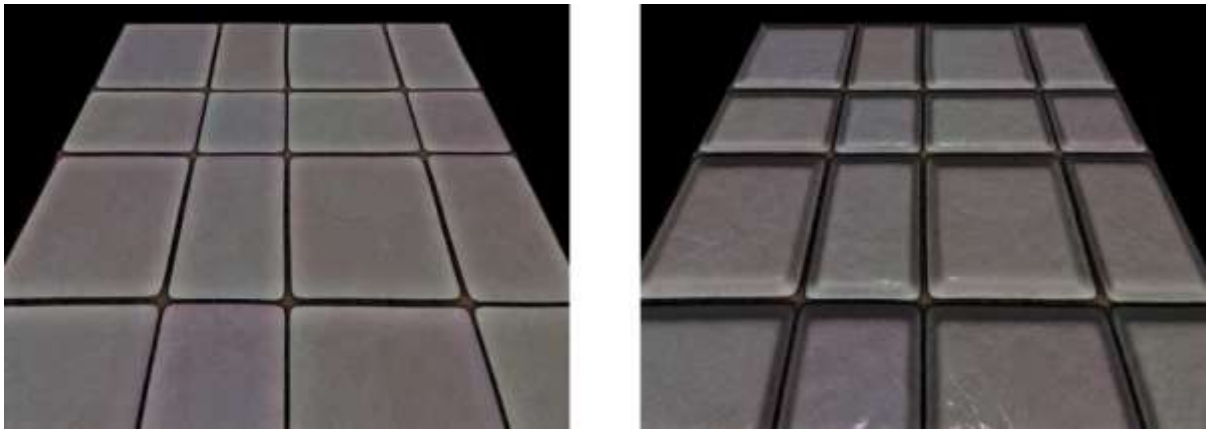

Figure 1: Standard texture mapping (left). Added normal map (right)

Unfortunately, normal mapped surfaces do not provide sufficient detail close up and would not be suitable for standalone implementation in the project. They could, however, be incorporated in a *level of detail* system that switched from higher accuracy displacement maps to normal mapping for more distant geometry.

2.3.2 Relief Mapping

Relief mapping is form of bump mapping that maps relief textures on to polygonal surfaces in real-time. It operates in tangent space, and as such supports self-occlusions, shadowing and per-pixel lighting [9]. Relief mapping works by utilising a root-finding approach on a height map texture [5, 7]. The viewing ray is transformed into tangent space. A linear search is performed to locate the surface intersections then a binary search is done to find the closest intersection. Following this, the shading is performed using the calculated attributes. The linear search has a fixed time step which means the user has to trade accuracy for performance as finer detail needs smaller step sizes.

A separate function that operates on the object's silhouette needs to be implemented [5] to add finer detail to the object boundary. This approach yields significantly better results than bump mapping but still lacks the geometric detail that displacement mapping achieves. Unfortunately this technique is not suitable in the context of this project since of the additional overhead and complexity which would decrease performance.

2.3.3 Parallax Mapping

Parallax mapping is a simple way to augment bump mapping to include parallax effects [7]. It uses per-pixel texture coordinate lookups to achieve high rendering quality [10]. Parallax mapping works by reading the height offset stored in the parallax map at the specified point where the view ray intersects the geometry. Once this height value is obtained, the point at which the view ray intersects this height plane is used for the final texture lookup [5]. Figure 2 illustrates this technique.

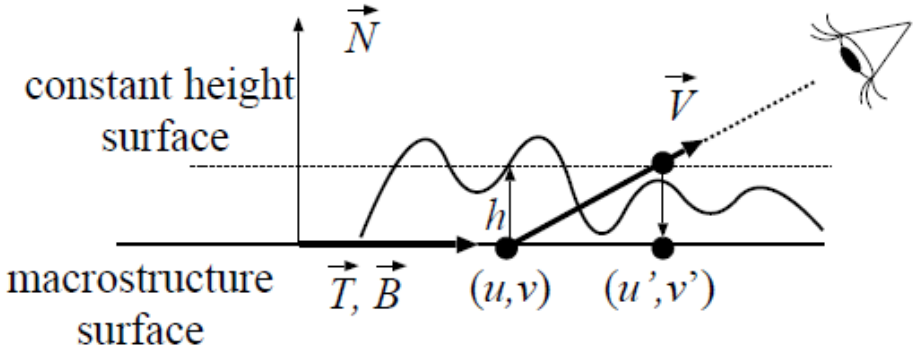

Figure 2: Parallax Mapping [5]

This is a crude technique that provides acceptable results [7] provided a certain set of limitations are adhered to. For instance, this is only valid for smoothly varying height maps; it cannot handle high-frequency features, large displacements and self-occlusion. Parallax mapping has one advantage: implementing it requires only a couple of lines in the fragment shader [7].

2.3.3.1 **Offset Limiting**

Offset limiting [5] is also required to render geometry in the distance because when the view angle becomes too oblique. In this case, the offset approaches infinity, causing random data to be returned and producing rendering errors. These artefacts are demonstrated in Figure 3, the left image shows that distant data gets merged into the background and appears to be floating in the distance. Offset limiting is a simple change in the shaders that produces far superior results and has the added bonus of a frame rate increase.

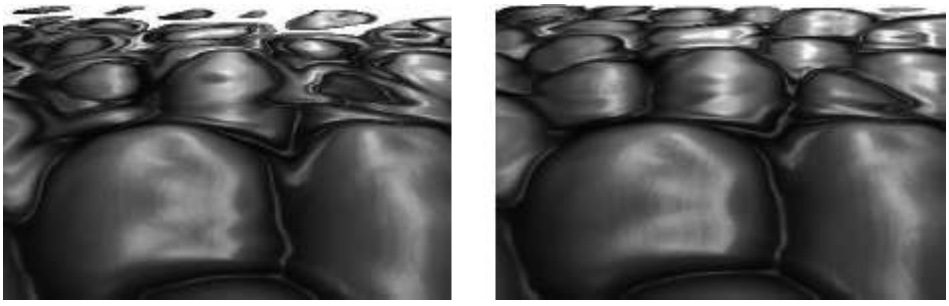

Figure 3: Parallax mapping (left), Parallax mapping with offset limiting (right)

2.3.3.2 **Iterative Parallax Mapping**

Iterative parallax mapping attempts to shift the texture coordinates towards the true height provided by the height map. This process is not guaranteed to be perfectly accurate the first time which is why this process is run several times substituting the new approximation into the equation each time. This is done in the fragment shader and is relatively fast but is an unsafe method; better methods do exist but add to the complexity and affect the performance. Iterative parallax mapping should produce acceptable results for use with the project.

Figure 4 illustrates the difference between normal mapping, offset limiting parallax mapping and iterative parallax mapping running four and eight times.

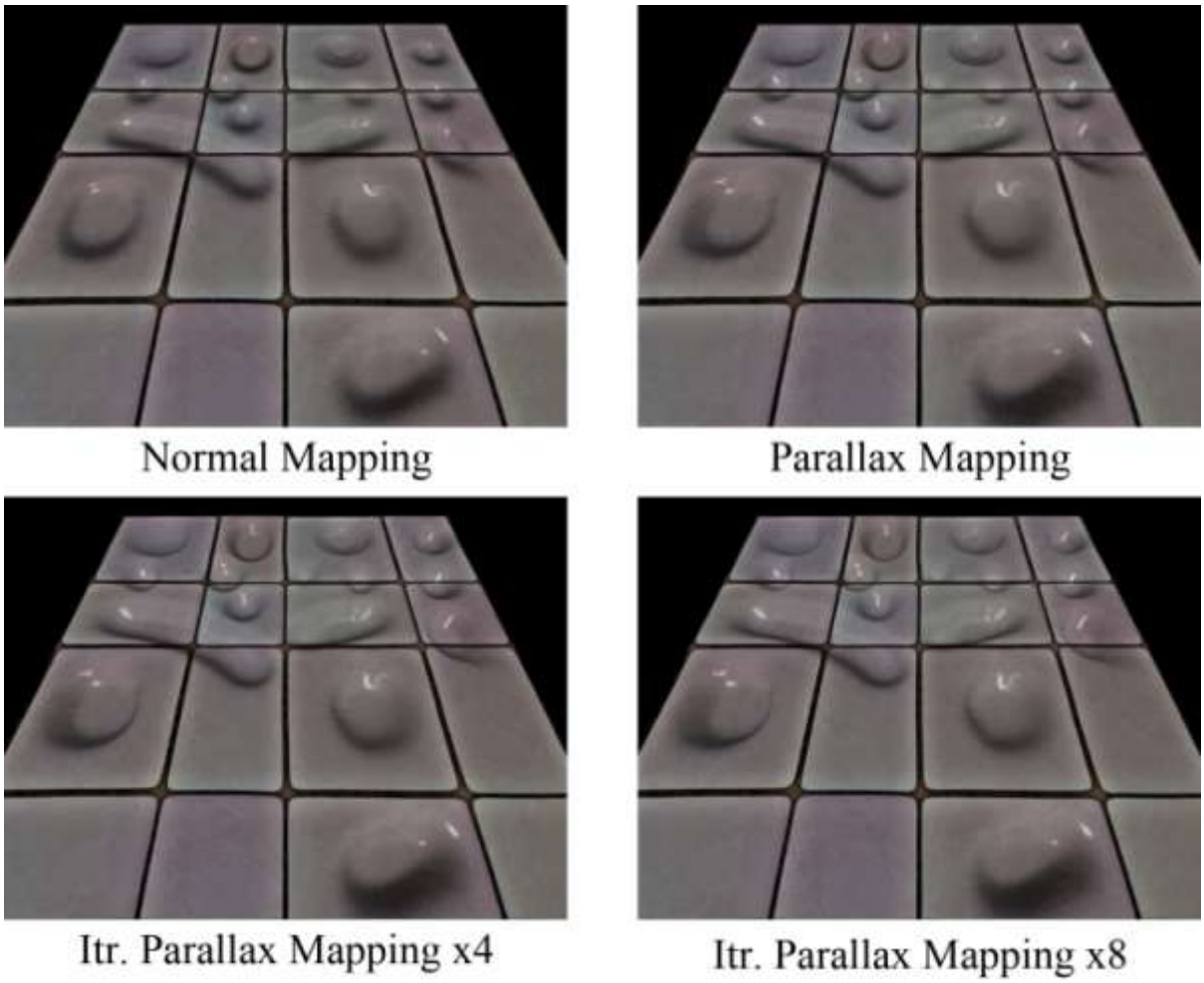

Figure 4: Difference of normal mapping to parallax mapping to iterative parallax mapping

2.4 Displacement Mapping

In 1984 Cook [2] introduced an extension to bump mapping called displacement mapping. Because the location of a vertex is used in the final appearance it is possible to move the location of the vertex as well as perturbing the normal. Displacement mapping was originally designed to solve the issues that arise from the use of bump maps, discussed above, specifically the silhouette effect. Apart from this, displacement maps have been proved useful in other aspects of computer graphics.

Displacement mapping adds real surface detail to objects [6] by shifting the positions of the vertices along the normal to the surface by a distance that is extracted from sampling the displacement map. This differs from bump mapping which only affects the shading of the surfaces [7]. This allows for more advanced effects, not possible with even advanced implementations of bump mapping. These can include surface features that occlude parts as well as self shadowing.

Since displacement maps alter the position of a surface and its normal, these textures can be seen as a means of modelling [8]. Because of this, displacement mapping is well suited to terrain modelling [6] and consequently a technique of particular interest in this report.

Displacement mapping works by perturbing the surface locations by an amount stored in a texel of the displacement map. This process typically takes place in the vertex shaders on GPU's but methods have been proposed that use only fragment shaders [6]. There are, however, some limitations when using fragment shaders only, such as the inability to render large surfaces. Mathematical distance functions are used to smoothly displace vertices based on the samplers input [7]. These functions can be computationally expensive but on modern hardware they can be computed relatively quickly. The process is slower than that of bump mapping; however it produces more realistic results. This is where a level of detail system could be implemented to choose between the two methods. Level of detail is discussed later on in the report.

True displacement mapping works not only by perturbing the vertices already present on the surface geometry but also by adding new ones. This requires tessellation [6] of the surface to provide more points to produce smoother displacements. In current hardware the use of the geometry shader handles the creation of additional vertex information and in latest generation GPU's on-the-fly tessellation can occur with aid of the tessellation shader units.

Figure 5 shows the comparison of normal mapping shown on the left to vertex displacement mapping on the right. The left image is comprised of 4 vertices whereas the right one has 10,000 which represent a dramatic increase geometry complexity.

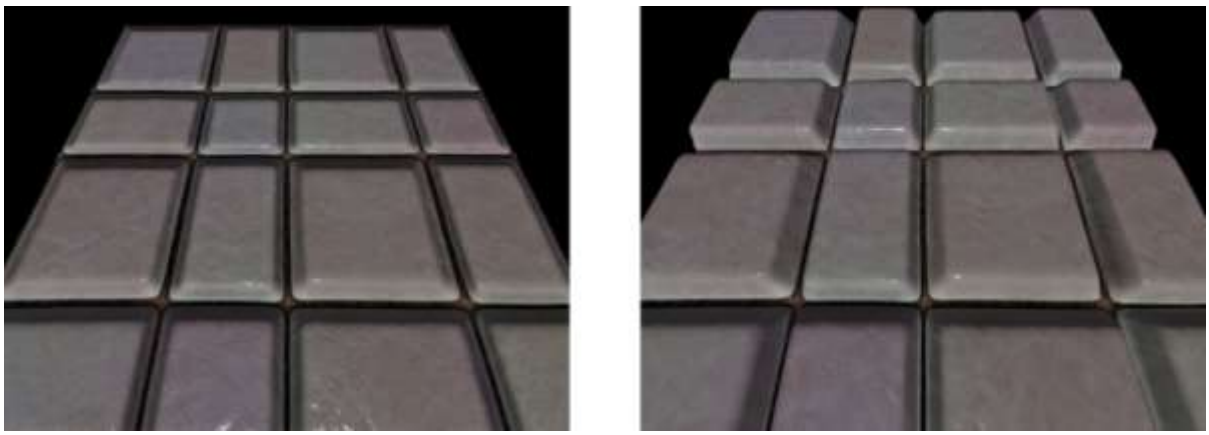

Figure 5: Comparison normal mapping (left) and vertex displacement mapping (right)

Chapter 3: Design

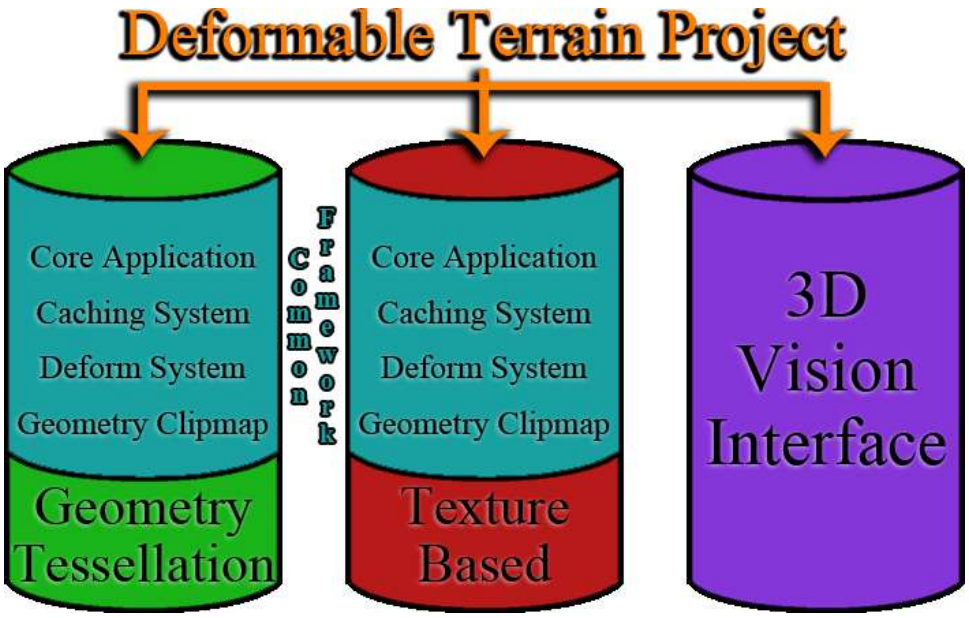

Figure 6: Separate components to the project

The work of the project is divided up into three components of equal weight, Figure 6 shows the divisions. The first and second components share a common framework which is collaboratively developed by the two respective members. This common framework allows for coarse level deformations to take place on the terrain, the implementation of the caching system. The core application also includes the representation of the terrain mesh and basic rendering functionality. The main distinction between the two components comes about with the addition of high-detail deformations to the terrain. The following subsections highlight the work of the different components and how they differentiate from one another.

1. Geometry Tessellation

The first method implemented for the representation of high-detail deformations uses displacement mapping on a finer scale. Due to the coarse granularity of the raw clipmap mesh, simply displacing the existing vertices yields no extra detail. For this reason, the existing mesh is further tessellated, or refined, such that each triangle is subdivided into 9 sub-triangles. The resulting vertices are then displaced according to the high-detail data. High-detail is only considered in regions close to the camera and only the inner grid surrounding the camera is thus tessellated. Ideally, tessellation within this grid would be

adaptive, in that regions would only be tessellated if high-detail data existed. Instead, as this implementation currently stands, tessellation is performed for the entire inner grid within a specified radius. This method does however yield consistent performance hits. The performance of an adaptive approach would decrease in areas of much high-detail, although the average-case performance would be considerably better.

2. **Texture-Based**

This implementation relies on texture trickery techniques such as normal and parallax mapping. The aim of this implementation is to produce the illusion of high-detail on the terrain without the overhead of creating additional geometry. This should allow for unlimited deformations to occur without any noticeable slow-down since there is always the same amount of geometry in the scene. The process involves more texture reads and a more complicated lighting calculation; these overheads reduce the performance of application. However, this penalty is constant irrespective of the number of deformations. This effect is purely illusionary and as a result has certain limitations that must be managed to prevent artefacts becoming noticeable. This implementation competes directly with the aforementioned geometry tessellation one.

3. **3D Vision Interface**

This component forms the front-end to the two separate back-ends described above. This interface is a computer vision based system wherein the user interacts with the application directly through the use of gestures. This component is designed to integrate easily with the two deformation components. The vision system uses object tracking, background subtraction and minimal use of hand pose estimation to create a functional input device from the users hand, hand occlusion, pose and position relative to the computer monitor.

The contents of this report focus on component two on texture-based methods. For information pertaining to one of the other components please see the respective documentation.

3.1 Design Constraints

The project requires the use of specific functions available only to OpenGL 3.2 and higher graphics API's. This functionality is hardware dependant and thus some design constraints around hardware exist. For the purpose of the project the NVIDIA 9xxx series cards are the benchmark hardware since these devices are considered to be mainstream. The use of geometry shaders is fundamental to the implementation of this project and this requires an OpenGL 3.2 context.

The main design goal of this project is to achieve a real-time system and as a result a minimum rendering rate of 30 frames per second needs to be achieved. However, since the possible uses for an on-the-fly terrain deformation system would be to integrate into games this would not be acceptable since there are other more complex systems that need to be run at the same time. Thus a base frame rate of around 100+ frames per second is required for

this system component. These values must be achievable on the base-level hardware (an NVIDIA 9600GT graphics card). Other graphics cards are tested which include high-end enthusiast devices such as the NVIDIA GTX 295.

A second design constraint is that of quality of the final resultant work. High visual quality is needed to support the rendering of high-detail deformations. This directly affects the frame rates and a careful balance needs to be attained between real-time performance and visual quality.

The third constraint is that the user's deformations need to be persistent which means the changes they make are permanent. This process is achieved through saving the height-maps to the hard disk.

Computer games require cutting edge graphics and as a result high visual quality needs to be maintained at the real-time frame rates in order to be of use in modern game engines. This means the terrain must be free of visual artefacts (such as holes in the terrain) and that there should also be no sudden changes in the geometry that can break the sense of realism.

3.2 Common Framework

In the project, there are two separate implementations, which share a common framework. This framework is an extension on the reGL library developed by Andrew Flower [http://people.cs.uct.ac.za/~aflower/]. This library incorporates SDL [http://www.libsdl.org/], and is responsible for handling the majority of the backend instructions to control OpenGL. ReGL includes a math class populated with various functions for vectors and matrices of different sizes. A shader class which is responsible for loading, compiling and binding of shader files.

This common framework was developed collaboratively by the team members and extends the reGL library to support very basic deformations. This includes only coarse level deformations and basic shaders to display these. The caching or Level of Detail (LOD) system has been incorporated in part as its completion is dependent on each of the separate implementations. Work on each of these implementations was conducted concurrently by the respective member and integrated into the main framework. The use of a common framework facilitates ease of integration of the project at the end of development to create one fully functional application. The main difference between the implementations is that one uses tessellation to create additional geometry and the other uses texture based techniques to achieve high detail rendering of terrain.

Differentiation between the two implementations comes when the shaders get implemented. Different shaders were implemented by the members: one making extensive use of the geometry shader to provide tessellation to render in the high detail and the other using parallax mapping which is completed in the fragment shader. These two systems can be toggled while running the application to show the differences between the two implementations.

3.3 Texture Based Deformations

The main focus of this project is to utilise texture based techniques to produce the high detail on the terrain surface. This high detail must be capable of producing fine detail such as footprints on the surface of the terrain. The implementation covered in this report is compared to the reference implementation which makes use of additional geometry to add in the high-detail. Because texture based techniques require less geometry there is less overhead on the shaders and such higher frame rates are achieved. This forms part of the expected outcomes that are evaluated at the end of the report in Chapter 5: Results & Evaluation.

These texture based techniques provide a few challenges which need to be overcome in order to complete this implementation. Normal mapping and parallax mapping require both normals and tangents for each vertex in order to complete the technique. As a result normal and tangent maps must be constructed for the height maps.

Another limitation that arises comes from the storage of three separate maps for each region in the Level of Detail (LOD) system. This means that large amounts of hard disk space are required to store the files. The solution to this would be to store only the height maps on disk and recalculate the normals and tangents at the time the map is loaded by the LOD system. This, however, increases computation required at load time and optimised code is required to keep the application running in real-time.

In order to reduce the amount of computation a compromise is made to choose performance over precision. This is achieved through the use of partial derivative (PD) normal maps [11], these maps store the partial derivative dx and dy in two channels of data on the map and then the z-component is reconstructed in the shader in two instructions. The tangents can also be calculated in the shader on the fly using PD maps. By using this technique only one map need be calculated when loading a region in to the application which saves considerable computation time. Also PD maps require less computation in the first place which aids in keeping the application running in real-time.

The Normal and Parallax mapping techniques are applied over and above the coarse level deformations in the shaders. This process is detailed in section 3.7.

3.4 Data Structures

One of the design constraints mentioned in section 3.1 is that the user's deformations persist which means that when they move away from an area and return later the changes they made are still going to be visible on the terrain. This idea of persistent deformations is handled by the caching system and the files are saved to disk.

The terrain deformation data is stored in two separate parts. The first being the elevation data stored as a single channel, 16-bit height-map to store a large range of height values. This produces the limitation of allowing only a 2.5D environment to be created and does not allow for overhangs or caves to be produced in the application. Height-maps for both the coarse and high-detail modes are stored this way and saved on hard disk. When the application is

terminated or a caching unload request is handled the data is saved. This is required for deformations the user makes to the terrain to persist. The second part is the partial derivative map which is based on the height-map data. However, this is not saved to disk and is recalculated when the height-maps get loaded in to the application. This data is stored in two channel integer format, one for the dx and the other for dy.

The above mentioned maps are stored directly on the GPU and all editing is done directly to GPU memory. This improves performance of the application and loading/unloading is only done when needed to reduce the number of IO operations. This same data is used at render time to display the height-maps and calculate the normals for use in the shaders. By keeping all the data on the GPU there is no overhead of transferring from system memory to GPU memory which aids in the goal of real-time performance.

The other main data structure used in the application is the use of a VBO for storing the clipmap. The clipmap is discussed in greater detail in section 3.6.2. By utilising a VBO data structure the clipmap only needs to be loaded into the GPU once and then is simply called whenever a draw call is made.

3.5 Caching System

In order to improve performance and allow for a scalable system, only the required detail-maps are loaded into GPU memory and made available for deformation. As a result of this the maximum area that the user is then able to deform in high-detail mode is limited to a small area around the user. This ensures that only maps that are currently loaded and active can have deformations applied to them. The coarse-map is divided up into a grid made up of a series of tiles; each of these tiles represents a detail-map. There is only one coarse-map specified in this project and as such wrapping occurs along the edges. This was done to limit the scope of this project to ensure completion in the given time frame. Figure 7 shows how a tile is divided up into a set of regions, each denotes a different state of maps that need to be loaded.

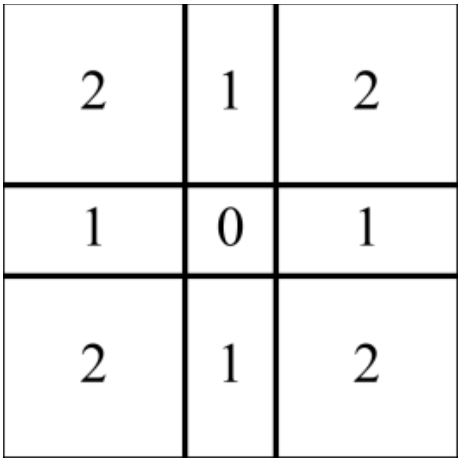

Figure 7: Image showing the different regions of a tile used in caching

When the user is in region 0 all the surrounding tiles need to be loaded into memory as it is unclear in which direction the player intends on moving, but only the current tile is active and allows deformations. If the player is in region 1 then six tiles need to be loaded on the same side that the user is in, if the user is in the top region then the above three tiles, adjacent two and current one are loaded. But deformations can only occur in the current tile and the one directly above. If the user is in region 2 then four tiles need to be loaded and made active which allows for the corner deformations to occur.

3.6 Terrain Deformation System

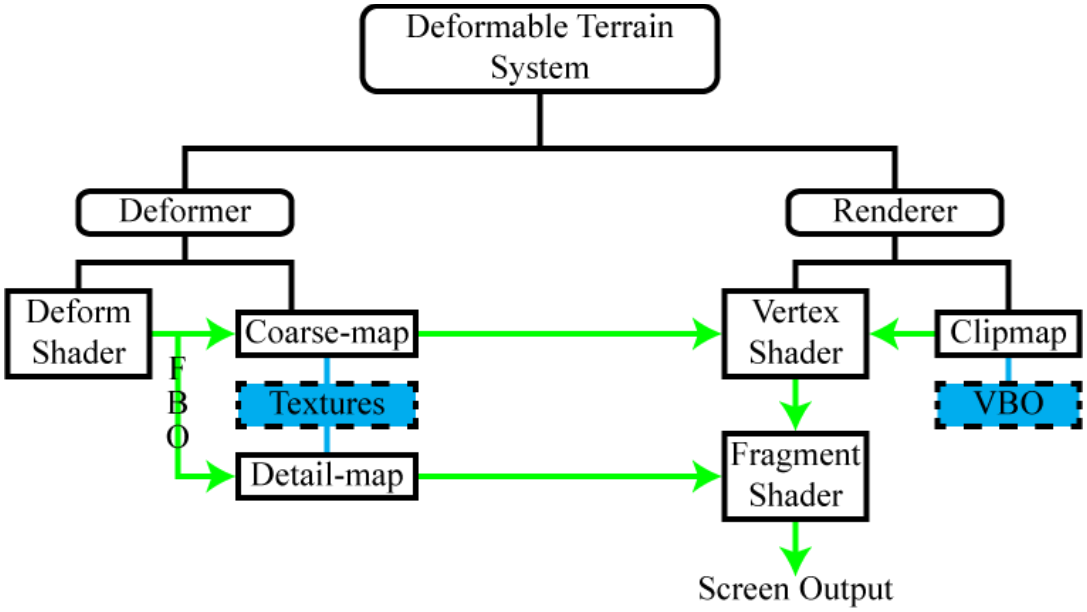

Figure 8: Overview of the terrain deformation system

An overview to the deformable terrain system is provided in Figure 8 which shows how the linking between the deformation and rendering components. These two components share the same set of textures which aids in performance and reduces the memory overhead. The green arrow shows the flow of information in the systems. The blocks highlighted in blue represent data structures used as input.

Terrain data used to present the deformations is stored as elevation points at discrete values in the form of height-maps on the GPU. Each of these height-maps has an associated partial derivative map which is generated at first load of the map. There are two sets of height-maps used to present deformations; these are either coarse-maps or detail-maps. The coarse-map is capable of displaying large scale, coarse deformations and has a limited resolution. Coarse-maps can be aligned adjacent to one another but for due to time limitations this project only focuses on a world consisting of a single coarse-map. This single map is wrapped to give the appearance of a seamless world and is large enough that the user needs to cover considerable ground before the wrapping occurs.

The second sets of height-maps are the detail-maps which have a much higher resolution. The detail-maps are positioned in a similar manner to the coarse-maps described above. As part of

the caching system, only the required detail-maps are loaded on to the GPU as it would be infeasible to keep all the textures loaded in at all times. These detail-maps are used by the parallax mapping technique to add in the high-detail to the terrain.

A deformer object is created by the application and is responsible for handling all the deformations. Deformations to the terrain are produced through the use of stamps which are provided by the application. These stamps come in various forms and more details are provided in section 3.6.1. It makes use of an FBO which allows for textures to be rendered to directly which allows for all the data to be kept in GPU memory and thus reduces IO overheads. When rendering the new height-map value, the previous values need to be queried in order to alter them. This requires a secondary backup texture since OpenGL cannot read from and write to the same texture location. A further enhancement to performance is to only modify the small area of the texture which is being deformed. This reduces the total area to render, but it is dependent on the size of the stamp being applied and thus large stamps require more time to process.

When a deformation takes place the partial derivative map needs to be updated. This process also only takes the small region into account when recalculating the partial derivatives in order to increase performance. The maps are rendered in much the same way, except that they do not require use of the previous version and thus the process is simpler than the height-map calculation.

These height-maps are used in the rendering process which is discussed in section 3.7.

3.6.1 Deformations

Deformations can be applied in two ways: the first is in the form of a mathematical function which is coded into shader files, such as a Gaussian function. The second form is that of a texture-based stamp. A single shader is used which gets assigned different textures which can be used to stamp the terrain with a specific pattern. These two deformation types work on both the coarse-map and detail-maps. However, the size of the deformation is limited on the coarse-map due to resolution issues but for the detail maps any size is allowed thanks to the parallax mapping technique.

An initial coarse-map is provided with the application on first use but is overwritten with the user's changes upon application exit. As discussed in the caching system (section 3.5) the coarse-map is divided up into a grid matrix which represents the different detail-maps. Initially they are all set to point to a single zero-map and when a deformation is applied a new texture object is created for that tile, this gets saved to disk and is used to preserve the users deformations.

When a deformation occurs on the boundary of a tile, then multiple deformations need to be applied, one on each side of the boundary. This process is rather slow and optimisations are done in the form of limiting the size of the render region based on how much of the deformation is in that tile. Due to time constraints of the project, only edge deformations are calculated for the coarse-map.

3.6.2 Mesh Representation

One of the main applications of a real-time deformation system is that of computer games, and in computer games there is only a finite distance the player can see. This means that detail can only be seen up close and as such mesh data can be less dense further away from the camera. For textures, this behaviour is provided by use of mipmapped textures. For geometry, this requirement has been implemented in the form of a *geometry clipmap*. The method for fitting the different levels into one another is a difficult task and as such the method proposed by Losasso et. al. [12] is used. Each level contains a grid of quads which in turn are divided up into two triangles each. On each successive level the size of the quads doubles.

Each level is made up of n vertices along the outer edges, where $n = 2^k - 1$ where k is an integer value. The inner-most level contains a solid grid of $n \times n$ vertices with the camera centred on the midpoint of this level. The inside of each successive level contains an L-shape set of quads which offsets inner-level. Blocks of size $m \times m$ are used to surround the corners of the previous level where $m = \frac{(n+1)}{4}$. *Fix-up blocks* of $m \times 3$ vertices are inserted along the centre of each edge in the level. The outer edge of each clipmap is made up of degenerate triangles which link the different size quads together; this is done to prevent T-junctions which would form noticeable artefacts in the terrain.

A sample clipmap with $n = 15$ ($k \to 4$) and $m = 4$ and 3 levels deep is shown in Figure 9. The L-shape set of quads are highlighted in green with the fix-up blocks in blue, degenerate triangles are marked with the red border. The camera is centred on the centre of the finest level.

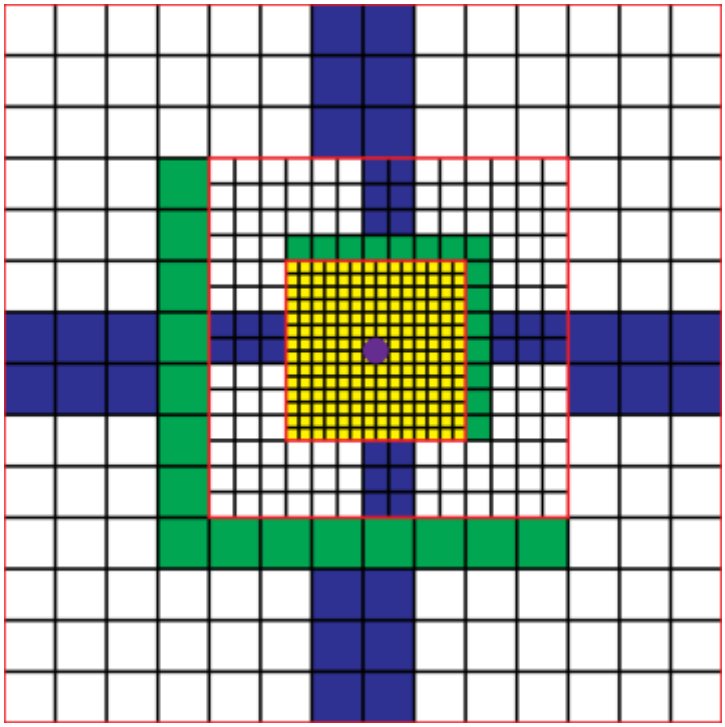

Figure 9: Clipmap with 3-levels, divided into quads

As it can be seen from Figure 9 above the inner-most level is not the centre of the clipmap but this does not affect the position of the camera for large values of n. Along with each vertex having its own position information it has an associated texture coordinate which is used to lookup elevation data in the shaders to deform the clipmap. The vertices of the inner-most level are spaced correctly such that sampling adjacent texels map to adjacent vertices. The p^{th} level is sampled at 2^{p-1} texels. The clipmap and camera are not translated through the world but instead the texture coordinates are shifted by the amount the camera gets moved by the user. On the other hand the clipmap gets rotated about the origin based on the cameras orientation.

The sampling rate of the vertices provides good enough detail for the coarse deformations but lacks the resolution for high-detail. In order to add high-detail deformations to the application, parallax mapping are added. This process greatly increases the resolution at which detail can be seen and allows for footprints to be seen. Parallax mapping is only calculated for a finite area around the user to save on computation since high-detail cannot be seen at large distances from the user. Parallax mapping works with texture coordinates and does not rely on vertices and thus it works best with a smaller number of vertices. This is because fewer vertices mean less overhead to the vertex shader. The resolution achieved is directly linked to the size of the texture used for parallax mapping.

3.7 Rendering Process

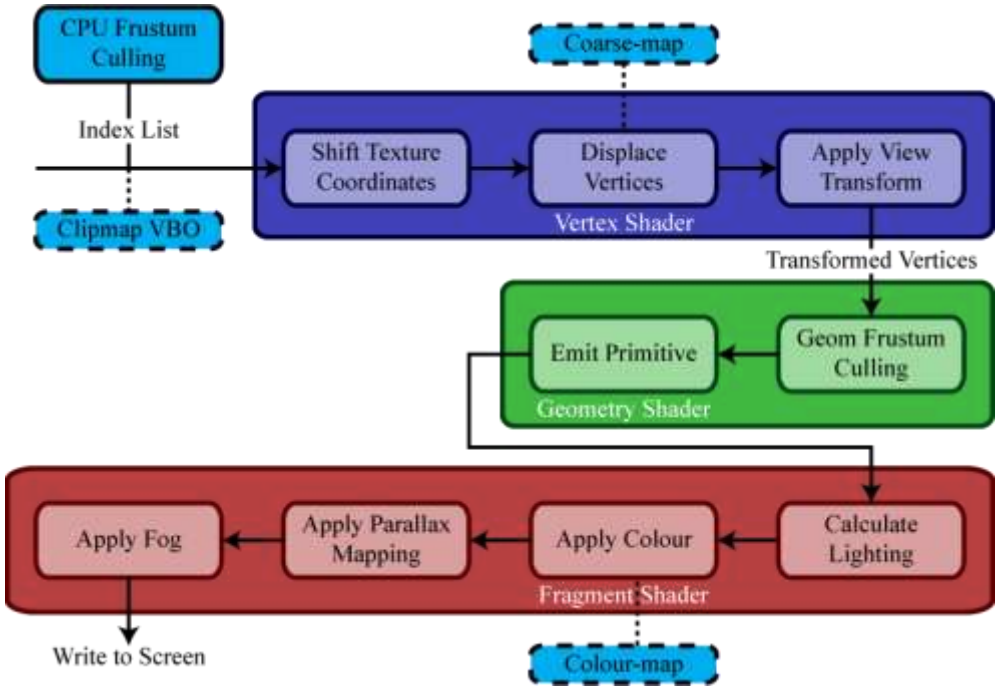

Figure 10: Overview of the rendering pipeline

Figure 10 shows the sequence of operations required to display the deformed terrain to the user. This is the main purpose of the project and is thus highly optimised in order to attain the stated design constraints. Figure 10 shows the main components of the rendering pipeline, the three shaders are highlighted in their own colour and explained in more detail in the

following sections. The rendering stage is also where the parallax mapping technique is applied, which is the main focus of this report.

In section 3.6.2, on Mesh Representation, it was explained how a geometry clipmap is used to provide coarse detail but that there is a need for higher levels of detail. To add this detail the resolution of clipmap would need to be increased significantly which would raise the vertex count of the program and hamper performance. This is where parallax mapping comes in since it allows us to add back high-detail irrespective of the number of vertices. There are a number of other functions that occur during the rendering step which include adding lighting functions which follow the Phong shading model [13]. The colour texture is also applied at this stage and fog is added to the scene, these steps aid in producing a more realistic environment.

3.7.1 Pre-Render Frustum Culling

As discussed above in section 3.6.2 on Mesh Representation, the clipmap is made up mostly of $m \times m$ vertices. An index list of all the triangles is stored on the GPU which is used to render the clipmap. On the system side, the blocks are stored with their range of indices and corner vertices. This data gets queried on every update step against the camera's view frustum by checking the corner vertices to see if they fall within the frustum. If all of the vertices fall outside the frustum they are ignored but if any vertex falls inside, then the index range of the block they correspond to is added to a render-ready list. This render-ready list is used to render only the $m \times m$ regions that are visible to the camera. Later on the geometry shader cuts out these unnecessary regions at a triangle level which further reduces the number of triangles; this is explained in section 3.7.3.

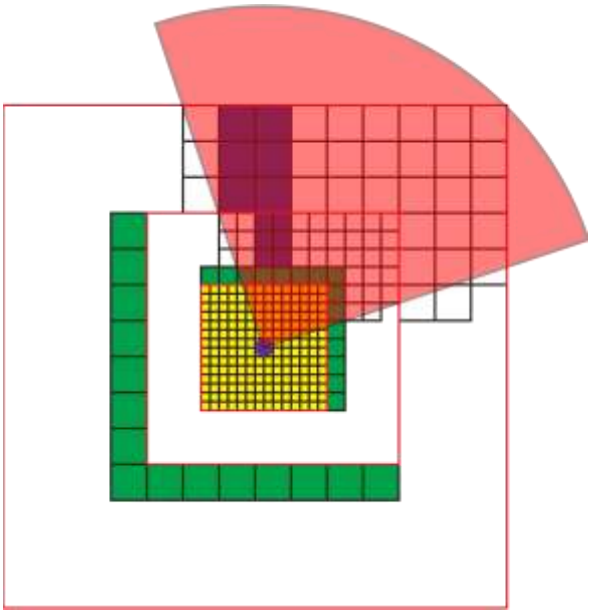

Figure 11: Result of CPU based frustum culling

In Figure 11, the effect of the pre-rendering frustum culling performed by the CPU is illustrated. The $m \times 3$ fix-up blocks are also considered in the frustum culling process, but

the L-shapes, degenerate triangle rings and the finest clipmap level are not checked and are always passed in to the rendering pipeline. These represent a small percentage of the total number of triangles and thus would not provide further performance increase in culling these out before rendering.

3.7.2 Vertex Shader

The vertex shader is responsible for performing the displacement of the coarse-map on a per vertex basis. The vertex shader receives the un-elevated clipmap data together with the texture coordinates of the vertex. The clipmap is centred at the origin of the world and as a result the motion experienced in the application is achieved by shifting the texture coordinates by the amount of camera translation. The shifting of the texture coordinates is completed before a texture lookup is performed which reads in the height value the vertex needs to be displaced to. After this the vertex is further transformed by the cameras view matrix which orientates the vertex in relation to the cameras rotation. Performance is of high importance in this project and certain optimisations are made in this shader. This includes the packing together of uniform values to reduce the number of reads from the uniform buffer. Next the data is outputted to the geometry shader for further computation.

3.7.3 Geometry Shader

The geometry shader is highly simplified in this implementation and is used solely to perform additional frustum culling. It performs per triangle computations and is used to perform additional frustum culling. The incoming triangle is tested against the cameras view frustum and if it falls outside it is culled. If it passes the test the shader continues to execute and outputs the triangle to the fragment shader to continue processing.

Below Figure 12 depicts the final set of geometry that gets passed to the fragment shader.

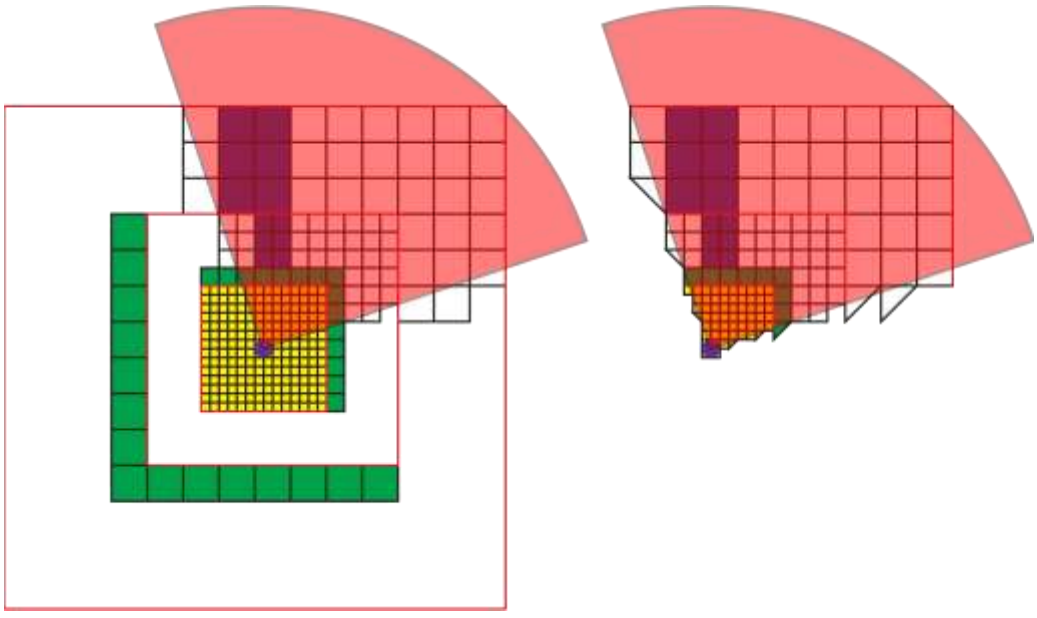

Figure 12: Comparison of CPU vs. Geometry Shader frustum culling

The first round of frustum culling culls only $m \times m$ blocks and the fix-up block whereas the geometry shader culls the triangles that are out of view. As can be seen from the above image the geometry shader significantly reduces the number of triangles that get sent through to the fragment shader.

3.7.4 Fragment Shader

The fragment shader is where most of the processing is completed and is the focus of this report. This is where the parallax mapping process takes place to add the high detail to the application. First it reads in and reconstructs the normal from the partial derivative (PD) map; this is required in order to calculate lighting for the coarse-map. Next, the colour-map is read and lighting is calculated for the coarse-map. At this point the parallax mapping stage is entered. Tangents need to be calculated to transfer the coordinate system into tangent space which is required by parallax mapping. The parallax mapping technique is only calculated for a finite distance since further points would be too small for the high-detail to be discerned and waste computational performance. The detail-maps are used to read in the height data and reconstruct the normal stored in the PD map: these are used to complete a second lighting pass which gets blended in with the first calculation. After that fog is calculated and added in to produce the final colour for the fragment. This value is returned from the fragment shader and outputted to the screen.

3.8 Design Summary

The goal of this project is to create a deformable terrain system that is capable of rendering both coarse and high-detail to the same terrain. The user needs to be able to modify the terrain on-the-fly through the use of the stamps provided. Stamps can be applied to both the coarse-map and detail-maps, where the detail is rendered using parallax mapping. The idea is that deformations that the user makes to the terrain need to persist such that they can move away to a new area and come back later and still find their change. This brought about the need for a caching system because it is not possible to store all data in memory. All the deformations take place on the GPU and the application is designed to run on mainstream hardware in order to cater for the majority of users.

Chapter 4: Implementation

The project was implemented as a cross-platform application for both Windows and Linux environments and makes use of free OpenSource libraries. The FreeImage image library is used to provide support for loading and saving of images. SDL was used to provide some core backend calls which handles user input and window setup. There are some differences between the Windows and Linux versions and as such the Glew library needs to be included in the Windows version to provide added support for OpenGL functions.

Some of the concepts presented here have changed since the design phase due to optimisations that were necessary in order to meet the real-time design constraint. The implementation chapter covers the specifics of how the project was implemented; this covers the key techniques as well as presenting a class breakdown with overviews of them.

4.1 Data Structures

This section covers the implementation process of various data structures that were brought up in the design chapter, section 3.4.

4.1.1 Height-maps

Height-maps are the method for storing elevation data used to represent the height values of the terrain at certain points. This notion was introduced in section 3.4 as the method for storage; these maps are stored in GPU memory which provides fast access to the data. These are stored as a GL_TEXTURE_2D structure with a single GL_RED component. Common height-maps use an 8-bit value which only provides 256 possible height values; this is insufficient resolution for the project. Because of this a bit-depth of 16-bits is used which provides a range of heights up to 65,536. At a sample of $5km$ maximum height this results in a single value representing $76mm$ which is a massive improvement over $19.5m$ resolution with 8-bit textures. A 16-bit height-map is only used for the coarse-map as the maximum height that detail-maps can represent is 1m and as such 8-bit textures suffice.

Because of time constraints during the development of this project, only one coarse-map is made available in the program. This results in the dimensions of it having a direct influence on the distance of the world. Because there is only one map the world needs to wrap when an edge is reached, this is done by setting OpenGL's texture mode to GL_REPEAT. Because detail-maps sit next to each other they do not repeat and are set to GL_CLAMP_TO_EDGE,

this clamps out of bounds texture coordinates to the maximum range of between 0 and 1. The internal format and data for coarse-maps is GL_R16 and GL_UNSIGNED_SHORT whereas the detail-maps are stored differently in the format GL_R8 and GL_UNSIGNED_BYTE. Figure 13 shows an example of the coarse height-map used in the application.

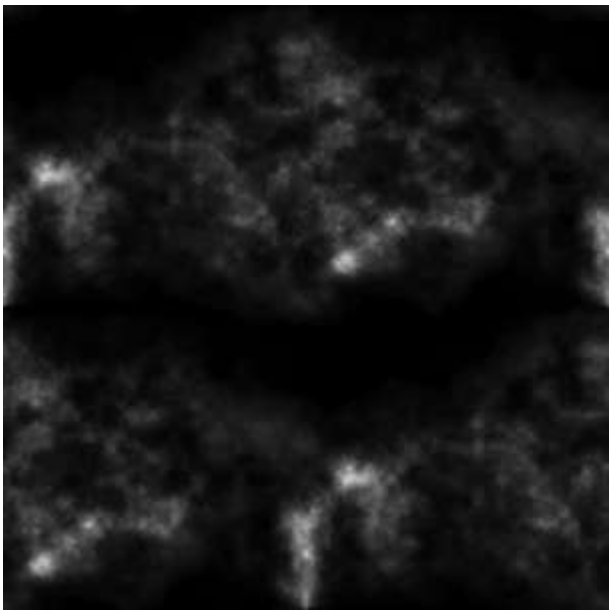

Figure 13: Sample height-map

After the height-maps get rendered, mipmaps are generated for them. This is done as they are required to be mipmapped by the FBO operations used in the deformation process; this is discussed in section 4.2.1. Aside from this process the radar provided by the caching class makes use of the height-map and because it is much smaller the mipmapped version is used which speeds up rendering by reducing the cache misses and scaling artefacts.

The height-maps get stored on disk in order to make the users deformations persist even between application launches. The PNG (Portable Network Graphics) file format is used to store the coarse-map as a single channel 16-bit file and the detail-maps as 8-bit files. This allows for easy display of the files through a custom cache viewer which was written in HTML 5.

4.1.2 Partial Derivative Normal Maps

In the calculation of lighting during the rendering step, normals are required to calculate the diffuse component. Normal maps are a common method for storing normals which represent a distortion of a texture usually as a height-map. Traditional normal maps store the x, y and z components in the three colour channels of a texture map. This requires three components to store the normal; however, by storing the normal as partial derivatives only two components are required. This can be easily reconstructed in the shaders; this is explained more in section 4.2.2. By reducing this number down to two components the latency time to retrieve the texture data is reduced and the memory overhead is reduced. The PD-maps are bound to the GPU as GL_TEXTURE_2D textures that have two 8-bit components, the internal format for these are GL_RG8 and stored as GL_UNSIGNED_BYTE data. The PD-maps, unlike the height-maps, are not saved to disk but instead are regenerated and load-time of the height-map.

4.1.3 Geometry Clipmap

The mesh representation discussed in section 3.6.2 covered how the terrain is represented through the use of a geometry clipmap which was discussed in the paper by Losasso et. al. [12]. The implementation of the clipmap was modified slightly from the design chapter's version. Each level of the clipmap comprises of four distinct arrangements of vertices, each of these different types are stored as a single VBO on the GPU. This changed from the stated design in section 3.6.2 which used a single VBO to store the inner-level and another to store outer-levels.

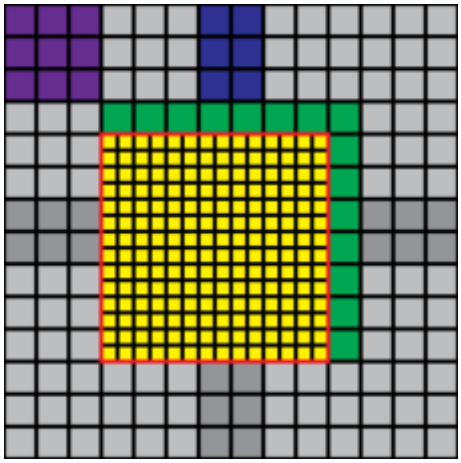

Figure 14: Diagram showing the different VBO components

Each of the different types composes different shapes; Figure 14 shows these different shapes in colour. The first comprises the $m \times m$ block shape (purple); this block is repeated 12 times in a single level. The next VBO stores the $m \times 3$ *fix-up* blocks (blue), there are 4 of these per level. The third VBO stores the L-shape block (green) which is made up of $2 \times (N + 1)$ vertices and used once per level. The forth VBO stores the degenerate triangles (red); there are 4 sets which is $4 \times N - 3$ vertices. The final VBO is used to store the inner-most level

(yellow) of N^2 vertices. The following equation shows the total number of vertices required to store the data in the VBO's with *N* vertices per side. Each of the blocks are translated, rotated and scaled to fit into the various places on the levels.

$$V_{total} = \overbrace{M^2}^{Block} + \overbrace{[M \times 3]}^{fix-up} + \overbrace{[2 \times (N+1)]}^{L-shape} + \overbrace{[4 \times N - 3]}^{degenerates} + \overbrace{N^2}^{inner-level}$$
$$= M^2 + 3M + N^2 + 6N - 1$$

This means that when $N = 255$ then $M = 64$ $\left(M = \frac{N+1}{4}\right)$, this gives a possible total of 70,842 vertices for any level depth chosen. This is considerably lower than if the original concept was chosen which would result in ~320,000 vertices for a 5-level clipmap. Each of the VBO's store only two components for the x and z position values, the y component is calculated in the shaders from the height-map. In addition there is another VBO which stores the texture coordinates that correspond to the vertex positions. Each of these VBO structures has its own index buffer which gets used to draw a triangle strip.

4.2 Main Techniques

This section covers the implementation of core procedures that form the basis of the terrain deformation system, from the deformation process to the rendering of it on the terrain.

4.2.1 Deformations

Deformations can occur in two different modes, the first on a coarse level and then secondly in high-detail mode. Deformations that occur on the coarse-map account for large scale deformations which are displaced on a vertex level and are handled by the *main* class. Because there is only one coarse-map implemented in this project a wrapping effect occurs which creates the illusion of a seamless world. Deformations that occur in high-detail mode are handled by the *Caching* object and do not allow for edge deformations between two tiles.

When a deformation occurs on the edge a series of cases need to be checked to determine where and how to perform the multiple deformations. Firstly a square area is calculated which represents the area of the deformation, this is calculated in texture space and offset to the location the user wants to deform. Then three sets of checks are performed to determine where the deformation lies. The following pseudo code shows how the left-hand deformations are determined. The area is determined and used to see where it overlaps; if it does overlap then a deformation gets pushed into a list with the position being deformed.

```
Left Columns...
if areaMin.x < 0 {
	if areaMin.y < 0
		Left-Top
	if areaMax.y > 0 AND areaMin.y < 1
		Left-Centre
	if areaMax.y > 1
		Left-Bottom
}
```

The list containing all the deformation locations gets processed at the end and the deformer's *displace_heightmap* method is called to displace the height-map. After all the deformations are applied the normal map is regenerated by calling *calculate_pdmap*.

The *displace_heightmap* function uses an FBO which allows for the rendering of the deformation to a texture directly. This requires a backup texture which represents the current state of the height-map, this is done because OpenGL cannot read and write to the same texture file. The coarse-map has an associated backup texture because there is only one coarse-map. This was done to reduce the overhead of copying to it at every deformation. The detail-maps share a single backup file and need to have the current data for the deformation region copied to it before rendering. The backup texture and current height-map textures are bound and then the deformation is rendered. At the end of this step the FBO needs to be unbound and the height-map has mipmaps regenerated. Before the method returns one last step needs to be completed. If the deformation is taking place on the coarse-map then the backup texture for the coarse-map needs to be updated to keep in sync.

4.2.2 PD-map Generation

The notion of partial derivative normal maps was introduced in section 4.1.2 and the main reason for the switch over was only requiring two components over three for traditional normal maps. This reduces the memory overhead and increases the performance of the application. There is however, a trade-off with accuracy for this performance increase; the full range of normals cannot be reproduced. This is limited to a 45° cone about the vertical which means that flat normals cannot be reproduced. This is an acceptable trade-off because terrain does not typically have very steep gradients. A normal is defined by the cross product between the tangent and binormal at a given point. Given a vertex $\boldsymbol{R}$ situated on a height-map $\boldsymbol{h}(\boldsymbol{u}, \boldsymbol{v})$ the normal $\boldsymbol{N}$ is calculated as follows with tangent $\boldsymbol{T}$ and binormal $\boldsymbol{B}$.

$$\boldsymbol{R} = \boldsymbol{f}(u, v) = (x, \boldsymbol{h}(\boldsymbol{u}, \boldsymbol{v}), z)$$
$$\boldsymbol{B} = \frac{d\boldsymbol{R}}{du} = \left(1, \frac{\partial \boldsymbol{f}}{\partial u}, 0\right)$$
$$\boldsymbol{T} = \frac{d\boldsymbol{R}}{dv} = \left(0, \frac{\partial \boldsymbol{f}}{\partial v}, 1\right)$$
$$\boldsymbol{N} = \boldsymbol{T} \times \boldsymbol{B} = \left(-\frac{\partial \boldsymbol{f}}{\partial u}, 1, -\frac{\partial \boldsymbol{f}}{dv}\right)$$
$$\boldsymbol{N} = \frac{\boldsymbol{N}}{|\boldsymbol{N}|}$$

The partial derivate normal maps only store $\boldsymbol{N}_x/\boldsymbol{N}_y$ and $\boldsymbol{N}_z/\boldsymbol{N}_y$ components which are the partial derivatives of the normalized normal. This project uses a finite difference equation to approximate the partial derivatives of the height-map $\frac{\partial h}{\partial u}$ and $\frac{\partial h}{\partial v}$. Specifically a finite difference equation of error order $O(\xi^4)$ is used, where ξ is the distance between height values. It can be derived using Taylor expansions and shown below.

See the Appendix, section 10.1 for the derivation of the partial derivative function.

During the rendering process the shader has to read in the two partial derivates from the PD-map and reconstruct it, this is done by creating and normalising the vector $\left(\frac{\partial f}{\partial u}, 1, \frac{\partial f}{\partial v}\right)$. This has the 45° limitation, but this range can be increased by reducing the N_y value and sacrificing of the quality of the normal.

4.2.3 Caching

The caching system is responsible for managing the detail-maps; the caching system is required because all of the detail-maps cannot be loaded into GPU memory. First the caching system needs to determine the grid size by calculating how many detail-maps are required to cover the coarse-map at a target resolution. In the project the coarse-map has a dimension of $C_{dim} = 4096$ and the clipmap's inner-level has a resolution of $C_{res} = 0.1m$. The desired resolution for parallax mapping is $D_{res} = C_{res} * \frac{1}{3} = 0.0\dot{3}$, with the detail-map dimension at $D_{dim} = 2048$. This gives a grid dimension of $n = \frac{C_{dim} * C_{res}}{D_{dim} * D_{res}} = 6$. At this value a total of 36 tiles are required to cover the coarse-map. At $4MiB$ per detail-map, the full total requires $144MiB$ of GPU memory. This, however, only stores the height-map data and an additional 200% of memory is required to store the 2-component PD-map, giving the total of $432MiB$. This is why a caching system was required for the project.

The caching system works on a region system, wherein the tiles are divided up into nine regions of three different types; these are shown in Figure 15.

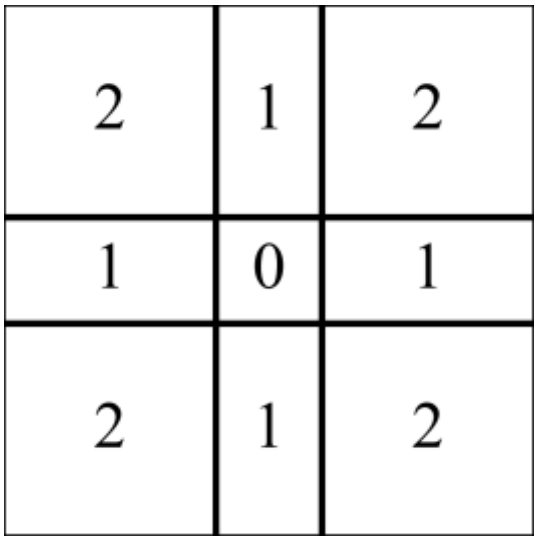

Figure 15: Tile divided up into different regions

When the player crosses one of the boundary lines the caching system changes state, Boolean values are used to store the final caching state. When the state changes an unload command gets sent to all the tiles required by the previous state and then a load command is performed for the current state. This result in a list of tiles that were previously loaded that now need to be unloaded and tiles that were not loaded that are now required to be. These loading and

unloading requests are handled by the caching PBO's which stream the data in and out asynchronously and handled by a separate thread. Figure 16 shows the three different states the caching system can be in. Red blocks represent tiles that are currently loaded in to GPU memory; the green blocks are loaded and made active and white blocks are currently unloaded.

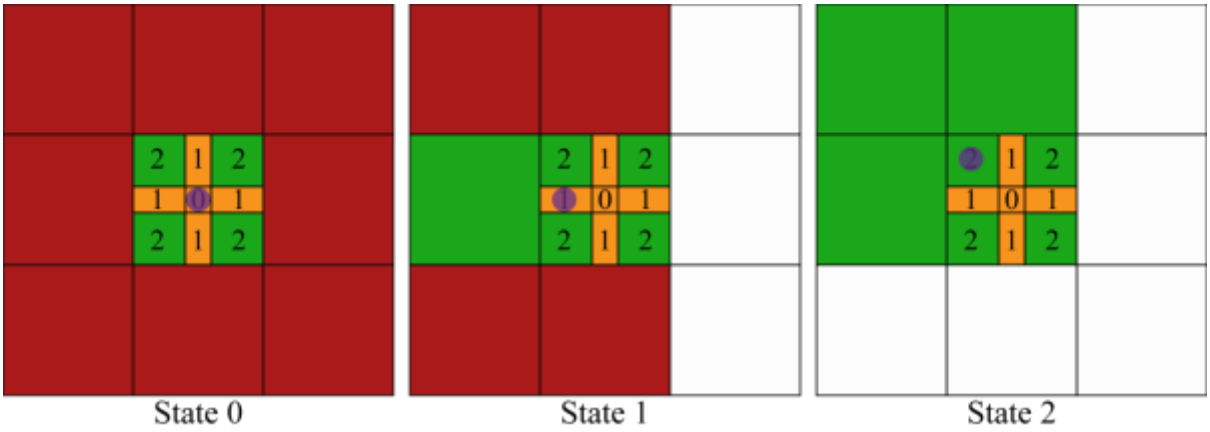

Figure 16: Comparison of clipmap states

As in Figure 16, state 0 has nine textures loaded but has only one active. State 1 has six textures loaded and two marked as active. State 2 has four loaded and active textures. It is noteworthy that the tile that the user is currently residing in is always loaded and made active. With this system there can at most be nine textures loaded in to GPU memory; this gives a total of $108MiB$ with PD-maps irrespective of the grid size.

Initially when the caching class is created a zero texture is created that all the detail-maps share, this gives the maps a default value. This zero texture is used when the hard disk does not contain the required detail-map which indicates that there is no deformation data. By sharing a single texture the memory overhead is reduced and there is a zero load tile for tiles with no deformation data. When a deformation occurs on a tile using the zero texture, then a new texture is created to store the deformation. Each tile has a modification Boolean which gets set to true when a user has made any deformation to the detail-map. This also applies to when maps have been loaded from disk. This means that when a tile unload request is handled the data only gets saved to disk if it needs to be, this saves unnecessary hard disk writes. The PD-maps are never written to disk and the data is simply discarded, this data gets regenerated when the detail-map is loaded.

4.2.4 Rendering Procedure

The rendering procedure is the core of the project and what takes the longest time computationally. Various optimisations were conducted at different stages during development to reduce the render time to improve the frame rate. The shaders were optimised to reduce the instruction count; this was done using the NVIDIA profiling tools which compiles the shaders down to assembly which can then be analysed to optimise them. The shader implementations are discussed in detail in sections 4.4.2 and 4.4.3. Figure 17 shows an overview of the basic rendering stages that are followed by the shader files.

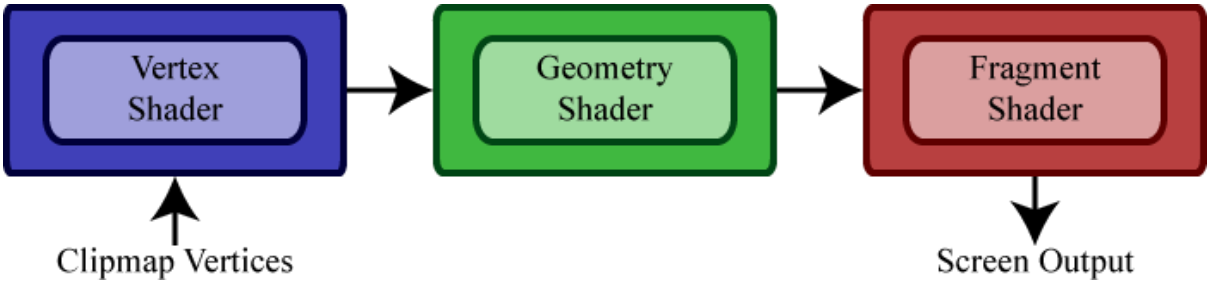

Figure 17: Basic rendering pipeline overview

In Chapter 5: Results & Evaluation, performance results are provided which show the efficiency of the shaders. The rendering process is comprised of two stages the first stage renders the coarse detail to the terrain; this is a shared component and part of the common framework. The second stage involves the rendering of the high-detail deformations which for this report is handled by the parallax shaders discussed in section 4.4.3. The parallax mapping step needs to be blended in with the previous one. Because this detail has a limited range of visibility and to accommodate for this a fade out effect is used to fade detail slowly into view. This prevents the high-detail from appearing to jump in to view suddenly.

4.3 Class Hierarchy

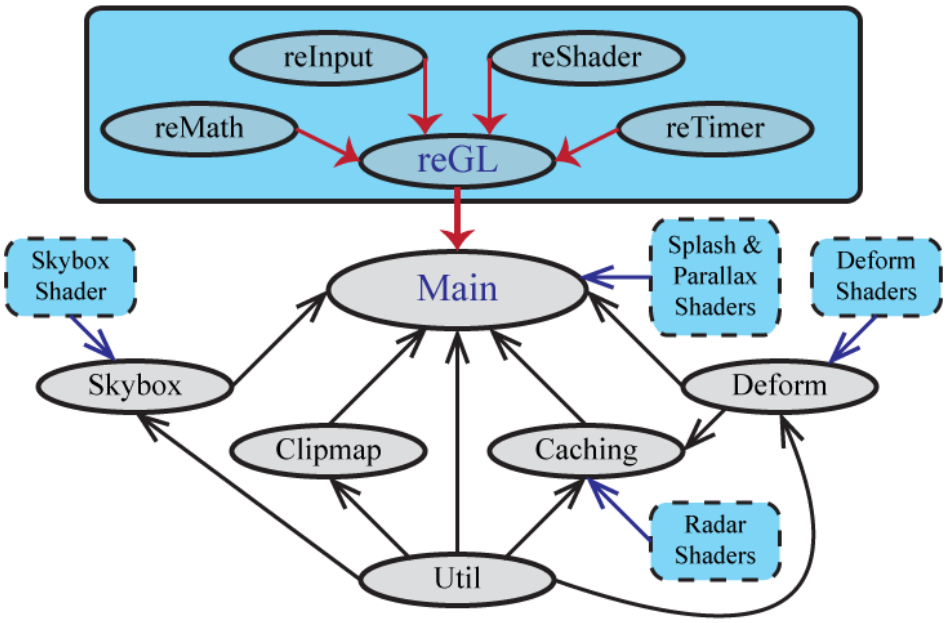

Figure 18: Class hierarchy showing connectivity of application

Figure 18 shows the global class hierarchy and all the connections made by the various classes. The top section in the blue block represents the reGL engine and all the components that make it up, the engine is explained in more detail in section 4.3.1. The main file is the centre of the project and is an inherited class from the reGL engine and extended for the purposes of this project.

The grey bubbles represent the different class elements and the blue dashed blocks represent the various shader files which below to the indicated class. The black arrows show the parent to which the class belongs, all class objects are created in the main file and references get passed through to the classes that need access to others. Each of the classes is discussed in detail in the following sections, in the order of their instantiation in the main class.

4.3.1 reGL Engine

The reGL engine was developed initially by *Andrew Flower* but has been extended and enhanced during project development. This engine provides a number of useful classes that are integral to the working of the deformable terrain application. These classes are vast and brief summaries are provided for each below.

The reMath class provides a number of useful mathematics functions; this includes *Vector* classes for two, three and four component vectors and *Matrix* classes are provided for orders two, three and four. There are also a number of useful mathematical functions which are useful in computer graphics applications, such as *transform matrices* and utility functions.

The reInput class provides easy-to-use methods for checking the state of input devices such as the keyboard and mouse. These methods can be queried to determine if a user is simply pressing and holding down a key or even detect a single press-release action. The mouse state can be checked for button presses and number of mouse wheel ticks. The reInput class needs to have its state updated which is handled by the reGL engine.

reShader provides the necessary controls to compile and link *GLSL* shader files into a shader program, reShader holds a class *ShaderProg*. It also is responsible for checking that the shader files compile successfully and if there are any errors they are reported neatly to the console. A ShaderProg object holds a *ProgramID* which is used when setting uniform values and when enabling the particular shader. The *glUseProgram (**ShaderProg->m_programID**)* command tells OpenGL to enable the use of the specified shader. This command must also be used before attempting to set any uniform variables.

Useful timing functions are provided in the reTimer class which has the ability to *start ()* a timer and then call another method to *getElapsed ()* which returns the time since either the start or the last getElapsed call. An FPS counter is also used to return the average frame-rate that has elapsed since the instantiation of the timer object.

4.3.2 Main

The *main* class is the most important class in the program, it implements the reGL engine. All of the main program constants are defined at the top of the class which control the outcome and properties of the various components, such as the clipmap and caching systems. First the OpenGL settings and states need to be setup which prepares the OpenGL context for use. First the splash screen is loaded and rendered which the other dependencies are created and initialised. The main file is responsible for managing the coarse-map and has various methods which are used to stream the coarse-map from the GPU to a copy in system memory. This

copy is used to provide collision detection to the player so they can appear to walk on the ground. This is only updated after a deformation has been completed and thus a lagged effect is present.

The main file is responsible for managing the users input and updating steps. The input handling uses the reInput class to track users input for a variety of commands, the list of available commands is presented to the user at run-time. The most important input commands relate to the deformations and involve the user clicking on a location on the screen which is un-projected to get the world-space position. When a deformation is applied the user's chosen stamp gets applied. There are two deformation modes, coarse mode and high-detail mode. When in high-detail mode the clicked position, stamp and stamp properties are sent through to the caching class which handles this mode. Coarse-level deformations are handled by main and has a complicated procedure, this allows for deformations to occur along the edges of the coarse-map. The deformer object gets passed the coarse-map, stamp and properties to apply to the map, after which the PD-map is recalculated for the modified region. When a deformation falls over a boundary, multiple deformations of reduced size need to occur on the different regions that are affected. Wrapping occurs along the edges to produce the effect of a seamless world.

The *logic* step is used to update the caching and physics systems. The caching class needs to be made aware of the user's current position in the world in order to update its state, see section 4.3.6 for more information on how caching works. Physics is added to the world to provide a sense of inertia and build a more realistic environment system. It also becomes useful in the inclusion of more advanced effects and dynamic stamps which are discussed in Chapter 7: Future Work. Footprints can automatically be applied to the detail-maps when the user walks around the environment; this can be toggled on and off because it reduces the performance when it is enabled.

The final thing that *main* performs is the *render* step, first the screen needs to be cleared by OpenGL and the camera's view matrix is determined. Next all the required maps need to be bound to the shaders; this includes the coarse height-map and PD-map, the colour-map and the four high-detail height-maps and PD-maps. The clipmap *cull* method is called to complete the pre-render frustum culling, then the rendering process starts. The *Parallax* shader is activated and the inner-level of the clipmap is rendered then the *Simple* shader is activated and the outer-levels of the clipmap get rendered. After this is complete the skybox is rendered and lastly the radar images are drawn on top.

4.3.3 Util

The Util class provides common functions that need to be made available to all of the difference classes. These methods are global and available when a class includes its header file. The Util class provides common image loading and saving and an OpenGL error checking method, these are explained in the following two sub-sections.

1. Image Loading/Saving

The FreeImage library (http://freeimage.sourceforge.net/) is used in this project to handle all the image saving and loading functions. It provides various functions and handles a large amount of formats and is easily integrated and is open-source.

The loading method takes in a few parameters which comprise of a texture destination to store the loaded image, filename relative to the project directory and Boolean values for flipping and scaling. First the image gets loaded into a FreeImage object and properties such as width, height and bit-depth are identified for later use. Once this has been completed it is optionally flipped depending on the value of the incoming parameter.

```
image->flipVertically()
```

If the image is required to be scaled then a set of extra steps are completed. Scaling is done to bring the loaded image to the same size as the current screen resolution. First a scaling value is determined which transforms the image into roughly the same dimensions, the largest of the horizontal and vertical scales is chosen, shown below.

```
targetScale = max(ScreenWidth ÷ ImageWidth, ScreenHeight ÷ ImageHeight)
```

This makes the image share one of the same dimensions as the screen and have the other either the same or larger. The next step crops the image about the centre which means the two edges get chopped off.

```
image->scaleTo (ImageWidth * targetScale, ImageHeight * targetScale)
image->cropTo  (ScreenWidth, ScreenHeight)
```

This technique handles all possible combinations of image and screen sizes and aspect ratios. Next the image properties are re-identified before the next step is conducted.

The final step to image loading is to use the properties to determine the type of texture to be created and then to copy over the pixel data on to the GPU and set the texture ID number that can be used to reference this image. Finally FreeImage unloads the data and mipmaps are generated and success is returned.

Image saving is relatively simple in comparison to the above method. Saving requires a destination filename, pixel data, and bit-depth, number of components, size and whether it must be flipped or not. The bit-depth and number of components are used to determine the type of image that gets saved. After the FreeImage object is created the pixel data is copied over to it and the file is optionally flipped vertically. After the image gets saved out to disk, FreeImage unloads the content and the method returns successful. If an error occurs at any stage prior the error status is returned.

2. Error Checking

The “CheckError” method is used to query the general OpenGL error state to determine if an OpenGL function call was successful. The method takes in a string which is user defined and printed to the console on the determination of an error. If an error is detected then the error

code is looked up against a defined set and then a useful string is reported to the console which aids in diagnosis of a bug. These statements can be used anywhere in the application and are simply ignored if no error is detected but used extensively during bug tracing.

4.3.4 Clipmap

The clipmap class is used to create and manage the geometry clipmap which is the method of mesh representation. This class has a few important methods which are explained in this section. The clipmap requires a set of parameters which are used to describe it; these were discussed in the Design Chapter in section 3.6.2. Values for n, the size of the inner most quads, number of levels and the height-maps dimensions are sent through in the constructor which sets up the clipmap. The inner and outer levels are separated such that two different shaders can be applied to the clipmap, the Parallax shader gets called for the inner-level since this is where high-detail is needed. This allows the complicated Parallax Mapping step to work on only a small area where the remainder of the clipmap is rendered with the simple shader. These shaders are discussed in further detail in section 4.4.

Another method provided by clipmap is *cull*; this is the method responsible for handling the pre-render frustum culling, it requires the view frustum matrix which is used to perform the culling. The method loops over all the positions of the $m \times m$ block and $m \times 3$ blocks and check them against the view frustum to see if they fall inside or partially inside. If the block is determined to be *visible* then its set of indices are added to a render-ready list which is used in the render method to draw the vertices. This process is moderately expensive to compute the visibility of the blocks but dramatically reduces the number of blocks in the average case which decreases the rendering time. The best case for this algorithm is when the camera is looking straight up which results in all blocks being culled and worse case being looking down from a height meaning all blocks are visible.

There are two different render methods provided in the clipmap class, the first renders the inner-level and the second handles the outer-levels. This allows the main file to swap shaders and render the inner-level with the Parallax shader and outer-level with the Simple shader. The outer-level render method loops over the render-ready list and draws each of the blocks as a *triangle strip*. *Triangle strips* require fewer indexes to describe a set of triangles which reduces the overhead of sending an index list which increases performance.

4.3.5 Deform

The deform class is responsible for performing the deformations on the terrain and handles both the coarse and detail maps. The deform class holds an array of stamp objects, which gets populated during the deform class' setup. There are two main methods that need to be executed in order to create a deformation on the terrain; the first is *displace_heightmap* and the second *calculate_pdmap*.

displace_heightmap requires some parameters which are used to describe the deformation and its location. The first is a *TexData* object which stores pointers to the height-maps; the

coarse-map structure is passed in from the *main* (section 4.3.2) class and detail-maps from *caching* (section 4.3.6). Additional parameters for specifying the stamp name, the stamp properties for scale, intensity, rotation and mirroring. First the method needs to setup some parameters which describe the deformation. The stamp is looked-up in the array and its shader is setup. Next a backup of the current texture needs to be created because OpenGL cannot read from and write to the same texture. To increase performance only a small part of the image is copied to a backup location, this region represents the area affected by the deformation. After the backup texture is setup then the backup and current height-maps are bound to the shader program and the stamp properties are sent as uniforms. Next the render process is called which performs the deformation on the current height-map bound then the method returns.

calculate_pdmap gets called by the respective program directly after the *displace_heightmap* method is called, this recalculates the partial derivatives for the area affected by the deformation. The method requires the *TexData* object to specify the PD-map, the clicked position and scale to indicate offset and size of the region needed to be updated. The PD-map is bound and the uniform variables set then the new PD-map is rendered. No backup textures are needed because the values are based on the height-map data and the old values in the PD-map are overwritten.

There is a final method which is provided for convenience which instructs the PD shader to calculate the PD-map for the entire height-map which is called when the height-maps are initially loaded on to the GPU.

4.3.6 Caching

The caching class is responsible for managing the detail-maps; the caching procedure is discussed in detail in section 4.2.3. The caching class is the most complicated in the project, it makes use of multiple threads and PBO's which allow for asynchronous transfer of the height-maps from hard disk to GPU memory and vice versa. This allows for the caching operation to occur in the background while the user continues to navigate the world. During creation of the caching object it requires certain information about the dimensions of the detail-maps and target resolution. These values are used to determine the grid size which is how many detail-maps are needed to cover the coarse-map. At this point the boundaries for the tile regions are calculated which are used to determine what tiles need to be loaded or unloaded by the caching system.

During the *update* method the user's position is used to determine which tile they are situated in and then the region they are in on that tile. Based on this information the caching algorithm runs which updates the PBO's which is discussed in section 4.2.3. This position values are used in the rendering of the radar images.

The caching class also provides a deformation method *DeformHighDetail* which is called from the main class when a user's deformation request is detected. This method takes as input the clicked position, stamp name and properties. These values are used to determine where

the deformation takes place and thus in what texture it occurs on. Once this is determined the *Deform* class's *displace_heightmap* is called with the reference to the *TexData* object belonging to the chosen tile along with the incoming stamp properties. After this step the *calculate_pdmap* function is called to update it. Currently edge cases are not handled and deformations can only occur in the clicked tile.

The *render* method is used to draw the radar images on the screen, there are two radars. The first represents the user's position on the coarse-map together with the grid divisions and tiles being highlighted to indicate their cache status. More details on the radar can be found in section 4.4.6.

4.3.7 Skybox

A skybox is a method for simulating backgrounds, most notably skies, in computer graphics applications and is used to present distant objects which cannot be navigated to but added in to improve realism. Skyboxes are usually made of a cube structure which surrounds the environment and employs cube-mapping techniques to map the texture on to the cubes surface. Figure 19 shows an example of the skybox being rendered in the application.

Figure 19: Sample of the skybox shown as in the application

The Skybox object is created in main with a simple call to its constructor, no parameters necessary. It does not require updating but has a render method which is used to draw the skybox. The first thing the constructor does is to setup and compile the shader used in rendering. Then the Util class is called to load the sky image which gets projected on to the skybox. The remainder creates the actual geometry for the skybox and stores it all as VBO's on the GPU since the data is persistent and used in every render call.

The render method gets called in every step and binds the sky texture and sends in the transform matrix which is used to orientate the skybox based on cameras movement. Then the VAO is bound and the skybox gets drawn.

4.4 Shaders

Several shaders exist in the project and are used at different times for different functions, these shader files are described in the following sections.

4.4.1 Splash Shader

The splash screen shader is the simplest shader in the application. The vertex shader simply passes on vertex position data and texture coordinates through to the next stage in the pipeline. The texture coordinates are interpolated in the fragment shader to sample the splash image and return the fragment colour.

4.4.2 Simple Shader

This shader is the main one and is responsible for drawing the terrain on to the screen. Three shaders make up the *Simple* shader program, a Vertex, Geometry and Fragment shader. The vertex shader is primarily responsible for performing the initial coarse displacement, then moves on to the geometry shader where frustum culling occurs and finishes off with lighting in the fragment shader. The shaders require a set of uniform variables to be sent through which is handled in the main file, such as camera position and view transforms. The main file is discussed in section 4.3.2. Figure 20 shows the main components of the *Simple* shaders. The *Simple* shaders are discussed in detail in the following sub-sections.

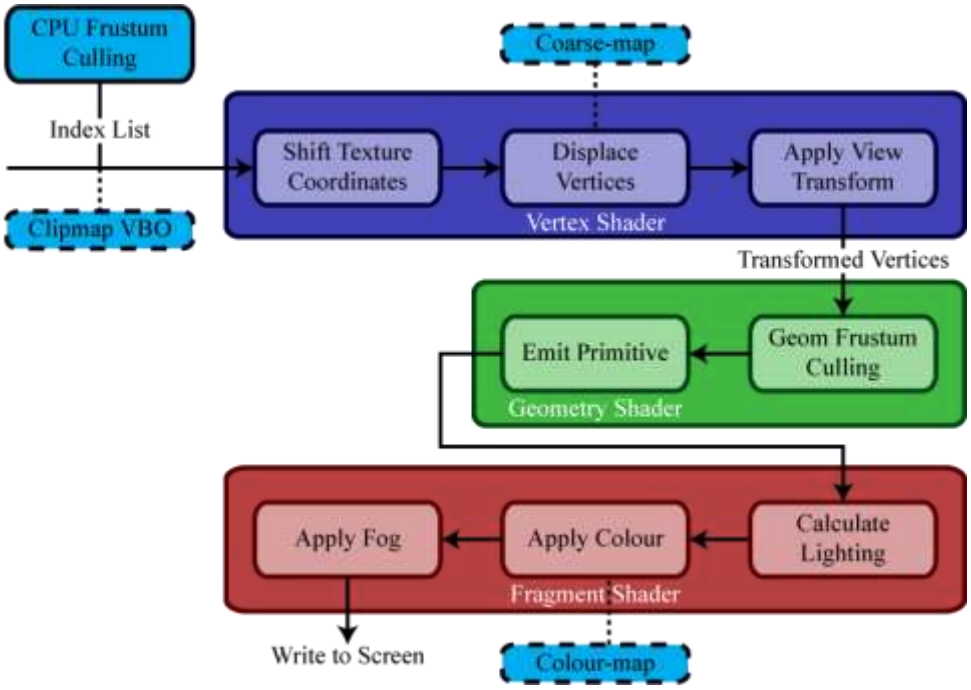

Figure 20: Pipeline of the main components used by the *Simple* shaders

1. Vertex Shader

The vertex shader receives a set of input vertices which come from the clipmap class after the pre-render frustum culling is performed. The first thing the vertex shader does is retrieve the variables from the uniform buffer for shifting the camera and amount to shift the clipmap by. Next the cameras world position (*camera_world*) is calculated by factoring in the texture space to metre conversion ratio. The texture coordinates (*texCoord*) are calculated based on the position of the camera and the shifting amount based off the incoming vertex texture coordinate. This is done because the texture is shifted around rather than moving the actual camera in the world. This set of code is shown in the snippet below.

```
camera_tex     = cam_and_shift.xy;
shift          = cam_and_shift.zw;
texToMetre     = scales.x;
metreToTex     = scales.y;

camera_world = camera_tex * texToMetre;

texCoord = vert_TexCoord + camera_tex + shift * metreToTex;
```

Next the height value is extracted from the coarse-map at the value specified by the *texCoord* and scaled by a constant value. This performs the vertex-based displacement mapping function then the vertex position is updated to reflect this new height value, this is shown in the following code.

```
height = texture(heightmap, texCoord).r;
camera_height = -cam_height;

vec4 pos = vec4( vert_Position.x, height * HEIGHT, vert_Position.y, 1.0 );
```

After this the vertex position (*pos*) gets shifted which is part of the caching and level of detail system to prevent visual artefacts appearing as the texture slides over the clipmap. The final vertex position is dropped by a set amount which is the camera's height; this is such that the camera can always be above the terrain. Finally the view vector is calculated from the vertex position value and then the projection matrix is factored in resulting in the final vertex position in screen space. Then the texture coordinate is sent through and the vertex shader ends. This is shown in the snippet below.

```
pos = pos + const_list.xyxy * shift.sttt;
pos = pos + const_list.yxyy * camera_height;

pos = view * pos;
geom_View = -pos.xyz * (1.0 / pos.w);

gl_Position = projection * pos;

geom_TexCoord = texCoord;
```

2. Geometry Shader

The geometry shader is the next step in the pipeline and is used to perform additional frustum culling by removing triangles that are not visible to the camera. The code below shows the culling process.

```
x =   abs( vec3( vertex[0].x, vertex[1].x, vertex[2].x ) );
y =   abs( vec3( vertex[0].y, vertex[1].y, vertex[2].y ) );
z = -vec3( vertex[0].z, vertex[1].z, vertex[2].z );
w =  vec3( vertex[0].w, vertex[1].w, vertex[2].w );


if ( !any(lessThan(x, w)) )
       return;
if ( !any(lessThan(y, w)) )
       return;
if ( !any(lessThan(z, w)) )
       return;
```

This code is inexpensive and saves only moderate processing in this set of shaders but provides a large average case performance increase in the Parallax Mapping shaders discussed in section 4.4.3.

If the triangle is visible then each of its vertices are emitted and the primitive is ended. This results in the end of the geometry shader and the next stage of the pipeline is then entered.

3. Fragment Shader

The fragment shader is the last step in the pipeline and where the lighting, colour-map and fog are applied to the fragments. A few constants are defined which set the lighting values, fog settings and key variables used in optimisation steps. The first block of code is shown in the snippet below.

```
vec2 dist             = click_pos - frag_TexCoord;
float scale_sq        = scales.y * scales.y;
if (dot(dist, dist) < .25 * scale_sq) {
       frag_Color = vec4(1.0, 0.0, 0.0, 1.0);
       return;
}
```

This code is used to draw the red dot which represents the position on the terrain that the user has clicked. This sets the fragment colour to be red if it is in a set distance from the clicked position and then the fragment shader is returned to avoid unnecessary calculations.

If the fragment is not near the clicked position then the normal must be read in from the coarse-map's partial derivative map and reconstructed. This is illustrated in the following snippet of code. The *cc vector* is applied to the normal value returned to combine the reconstruction process into a single *MAD* instruction which is an optimisation step. Next the normal is transformed into *View Space* and normalised.

```
vec3 pdn = texture( pdmap, frag_TexCoord ).rrg * cc.wyw + cc.zxz;
vec3 normal = normalize( mat3(view) * pdn );
```

In the next step the light's direction is transformed into *View Space*, the *view vector* is read in from the geometry shader and normalised and the colour-map texture value is read in. The texture coordinates are multiplied such to repeat the texture multiple times on the terrain.

```
lightDir = ( mat3(view) * light );
viewVec = normalize( frag_View );
color = texture( colormap, frag_TexCoord * 100.0 );
```

In the next step the Phong lighting model is used to provide per-fragment lighting to the terrain, a direction light is used. The diffuse and specular components are calculated and factored in based on the following lighting equation:

$$I_p = k_a i_a + k_d (L \cdot N) i_d + k_s (R \cdot V)^{\alpha} i_s$$
$$k_{a|d|s} \coloneqq Constant(Ambient \mid Diffuse \mid Specular)$$
$$i_{a|d|s} \coloneqq Intensity\ (Ambient \mid Diffuse \mid Specular)$$
$$L \coloneqq Light\ direction$$
$$N \coloneqq Normal\ vector$$
$$R \coloneqq Reflection\ vector$$
$$V \coloneqq View\ vector$$
$$\alpha \coloneqq\ Shininess\ Factor$$

The code to calculate this is shown in the fragment shader is shown in the snippet below. The fragment colour is calculated to represent the value from the lighting equation.

```
reflec = -reflect(lightDir, normal);
diffuseIntensity = max(0.0, dot(normal, lightDir));
specularIntensity = pow(max(0.0, dot(reflec, viewVec)), 32) * 0.3;

diffuse    = light_diffuse * diffuseIntensity;
specular   = light_specular * specularIntensity;
frag_Color = (ambient + diffuse + specular) * color;
```

The next step is to blend in a blue coloured aura around the player if they are currently in high-detail stamp mode. This is achieved by the code below.

```
dist = cam_and_shift.xy + 0.5 - frag_TexCoord;
if (dot(dist, dist) < hdasq_its.x)
       frag_Color += 2.0 * is_hd_stamp * (color * vec4(0.0, 0.0, 1.0, 1.0));
```

The final step is to calculate the amount of fog to be applied to the terrain and then blend it in to the final fragment colour which is the returned by the fragment shader and the result is output to the screen.

```
fogZ          = gl_FragCoord.z * (1.0 / gl_FragCoord.w);
fogFactor     = exp2(log2_fog_den * fogZ * fogZ);
fogFactor     = clamp(fogFactor, 0.0, 1.0);
frag_Color    = mix(fog_col, frag_Color, fogFactor);
```

4.4.3 Parallax Shader

The *Parallax* shaders are an extension to the *Simple* set described above in section 4.4.2. The vertex shader is identical to the one in *Simple* and thus is not described here. The geometry shader is the same except for additional steps that occurs after the frustum culling but before

the vertex is emitted. There are additional functions added to the end of the fragment shader to add the high-detail data to the terrains surface. The differences in the rendering pipeline between the *Simple* and *Parallax* shaders are highlighted in Figure 21.

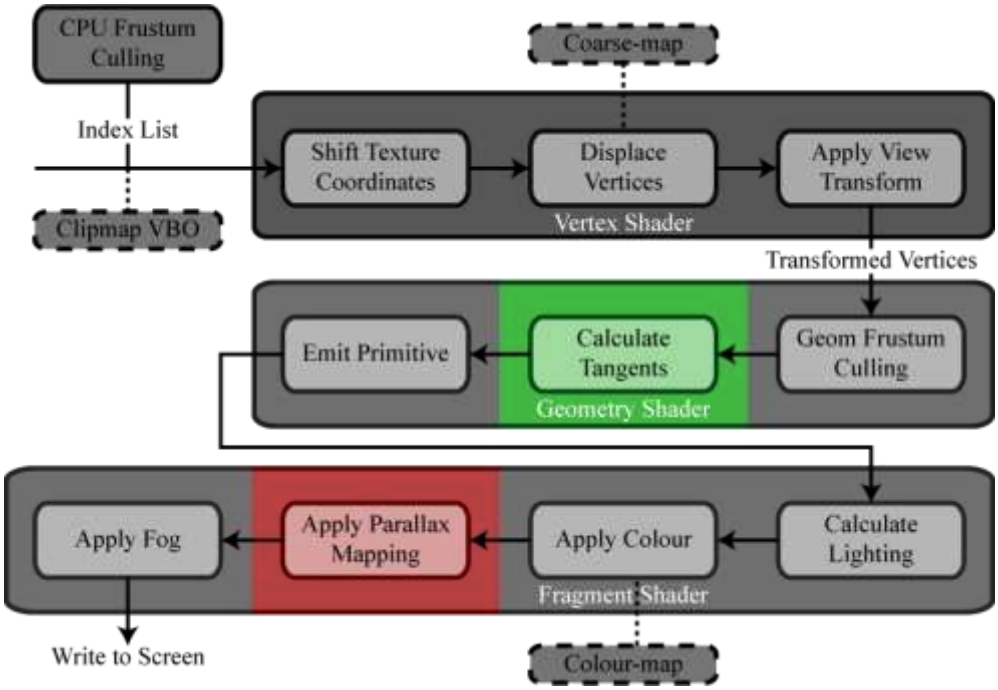

Figure 21: Added steps highlighted in colour

1. Geometry Shader

The only addition to the geometry shader is the calculation of surface tangents; these are required to convert the *viewVec* and *lightDir* in to tangent space. This is fundamental to the calculation of parallax mapping. The calculation of tangents requires edges between sets of vertices that make up an individual triangle, edges for both texture coordinates and vertex positions are created in the following code.

```
vec3 edgeRight        = normalize(geom_Vert[1].xyz - geom_Vert[0].xyz);
vec3 edgeLeft         = normalize(geom_Vert[2].xyz - geom_Vert[0].xyz);
vec2 texEdgeRight     = normalize(geom_TexCoord[1] - geom_TexCoord[0]);
vec2 texEdgeLeft      = normalize(geom_TexCoord[2] - geom_TexCoord[0]);
```

Once these edges are calculated the determinant is calculated based on the texture coordinate edges. This is done in two stages to check that the denominator is not zero. If it is zero then the tangent gets set explicitly $tangent = vec3(1.0, 0.0, 0.0)$. If it is non-zero the determinant is finalised and the tangent gets calculated with respect to the texture and vertex edges. The code to complete this is shown below.

```
float det = (texEdgeRight.x * texEdgeLeft.y) - (texEdgeRight.y * texEdgeLeft.x);
det = 1.0 / det;
tangent.x = (texEdgeLeft.y * edgeRight.x - texEdgeRight.y * edgeLeft.x) * det;
tangent.y = (texEdgeLeft.y * edgeRight.y - texEdgeRight.y * edgeLeft.y) * det;
tangent.z = (texEdgeLeft.y * edgeRight.z - texEdgeRight.y * edgeLeft.z) * det;
```

After this the tangent gets normalised and is sent through to the fragment shader for use.

2. Fragment Shader

The parallax mapping process occurs just after the terrain fog is calculated but before it gets mixed in to the final fragment colour. This process makes use of the high-detail height and partial derivative maps. There can be at most four high-detail maps available to render from and some initial offset values need to be calculated to transform the texture coordinates from coarse-map dimensions to detail-map dimensions. The normal vector is read in from the coarse-map pd-map and the tangent comes from the geometry shader, both of which get multiplied by the *view* matrix. The binormal is calculated from the cross product of the normal and tangent. These three vectors are used to calculate the tangent space matrix (*tbn*) which gets multiplied to the *viewVec* and *lightDir* vectors to move these to tangent space. The code which does this is shown below.

```
vec3 normal    = mat3(view) * pdn;
vec3 tangent   = mat3(view) * frag_Tangent;
vec3 binormal  = cross(normal, tangent);

mat3 tbn       = mat3(tangent.x, binormal.x, normal.x,
                      tangent.y, binormal.y, normal.y,
                      tangent.z, binormal.z, normal.z);

lightDir       = tbn * lightDir;
viewVec        = tbn * viewVec;
```

Once these values have been determined the parallax procedure can execute this is shown in the following code snippet.

```
for (int i = 0; i < parallaxItr; i++) {
       float h = (texture(detail0, parallaxTC.xy) * parallaxScale) + parallaxBias;
       parallaxTexcoords += (h - parallaxTC.z) * viewVec;
}

normal   = normalize(texture(detail0N, parallaxTC.xy).rrg * cc.wyw + cc.zxz);
```

The two maps *detail0* and *detail0N* specify the top-left texture to read from, the choice of map to use is based on the calculations done earlier. After the normal is read in and reconstructed the standard Phong lighting is calculated using this value. After the lighting is calculated it gets blended in with the current fragment colour to which the final blending in of the fog is done then the shader ends.

There are three key variables in the iterative parallax mapping technique that control the results it produces, these are the parallax scale, bias and iteration count. The scale value is used to specify the height of the parallax effect where the bias adds in a standard amount to shift by regardless of the height value read in from the detail-map. The last number is the iteration count which is how many times the method is run. This aims to improve the parallax mapping results; larger numbers generally produce better results but this works to a point. These values need to be tweaked in the program to obtain the best visual results. Experimentation was done to find good values which were then coded into the shader.

Figure 22 shows an example of parallax mapping being applied to the terrain; this provides an illusion of height added to the terrain but in fact the vertices remain unchanged and this effect is purely illusionary.

Figure 22: Example of parallax mapping used on the terrain

4.4.4 Deform Shaders

The deform shaders are the different stamps available to the user, there are two types of stamp. The first being a texture based stamp and the second being a functional stamp. There is one shader for texture based stamps which is shared amongst all the stamps but the texture they bind is different. This texture is loaded by the *Deform* class and stored in the stamp object. The shader used to place the stamp on the height-map texture is relatively simple. The vertex shader outputs two sets of texture coordinates; the first set is the location the stamp is being applied on the height-map and the second are for the stamp. The stamp can be optionally mirrored at this stage before it gets passed on to the fragment shader. The fragment shader needs to read in the current height value and adds the stamps value multiplied by the stamp intensity, this value gets written out into the new height-map. Figure 23 shows an example of a percentage sign stamp; the left shows the stamps height-map and the right image the result on the terrain.

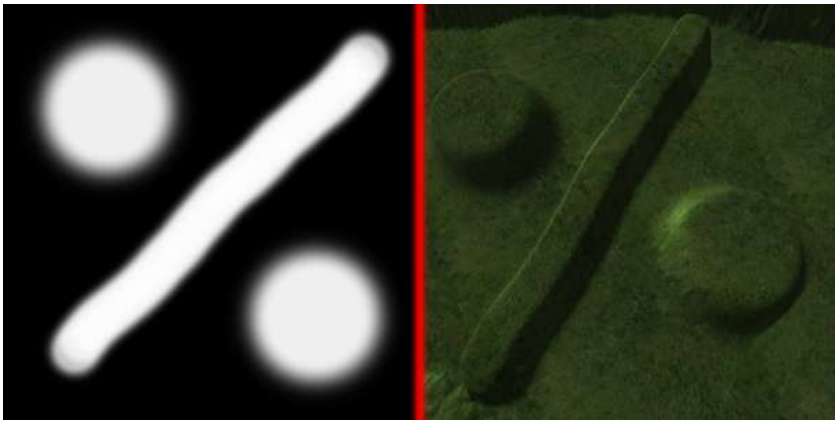

Figure 23: Example of % stamp applied to the terrain

The functional stamps work differently because each type has its own shader associated with it. This shader describes the effect that gets applied to the height-map and is put in the fragment shader, the vertex shader offsets the texture coordinates to the area the deformation is being applied. These are used to render only the region being affected by the deformation, the following code is taken from the *Gaussian* stamps fragment shader.

```
out float height;
void main() {
  vec2 dist = clickPos - frag_TexCoord;
  height = texture( in_heightmap, frag_TexCoord ).r;
  height += exp( - dot( dist, dist )  * falloff ) * intensity;
}
```

The current height is read in from the incoming height-map and then the Gaussian function is added to this value and is returned from the shader.

The use of this stamp technique allows for fully customisable deformations, some possible extensions to this are discussed in Chapter 7: Future Work.

4.4.5 PD Map Shader

The partial derivative shader is used to calculate the partial derivatives which are used to construct the normals in the main shader files. This is the shader which is called after every deformation step and calculates the partial derivatives for only the area being deformed. It uses the height-map to read in the height and calculates *dx* and *dy* values. There are two sets of calculations as shown in the following snippet of code.

```
// Horizontal
dh1.s   = texture( in_heightmap, frag_TexCoord + dst * cc.xy ).r;// Right
dh1.s  -= texture( in_heightmap, frag_TexCoord + dst * cc.zy ).r;// Left
// Vertical
dh1.t   = texture( in_heightmap, frag_TexCoord + dst * cc.yx ).r;// Bottom
dh1.t  -= texture( in_heightmap, frag_TexCoord + dst * cc.yz ).r;// Top

// Horizontal
dh2.s   = texture( in_heightmap, frag_TexCoord + dst * cc2.zy ).r;// Right
dh2.s  -= texture( in_heightmap, frag_TexCoord + dst * cc2.xy ).r;// Left
// Vertical
dh2.t   = texture( in_heightmap, frag_TexCoord + dst * cc2.yz ).r;// Bottom
dh2.t  -= texture( in_heightmap, frag_TexCoord + dst * cc2.yx ).r;// Top
```

dh1 represents the difference in height horizontally and vertically using a distance of one from the centre fragment and *dh2* uses a distance of two from the centre fragment. The *cc* and *cc2* variables store the different distance values and are combined in this way to compile the command down to a *MAD* instruction which was done as an optimisation step to improve performance.

```
vec2 pd = factor.x * dh1.st + factor.y * dh2.st;
vec3 normal = normalize(vec3(pd.s, 1.0, pd.t));
pdmap = normal.xz/normal.y * 0.5 + 0.5;
```

The final step of the shader is shown in the code snippet above which is where the partial derivative (*pd*) is calculated. After which the normal gets calculated and then is divided up and stored as a *vec2* in the *pdmap* variable which can be reconstructed in the main shader files, described in section 4.4.2.

4.4.6 Radar Shader

The radar shader is used to render a 2D representation of the world to indicate the player's position. It shows the players position on the coarse map and grid lines are used to show where the different tiles are situated. The secondary radar is used to indicate the player's position within a grid and shows the region divisions which are used by the caching system. These are provided to illustrate the workings of the caching system. Figure 24 shows a sample of the two radars.

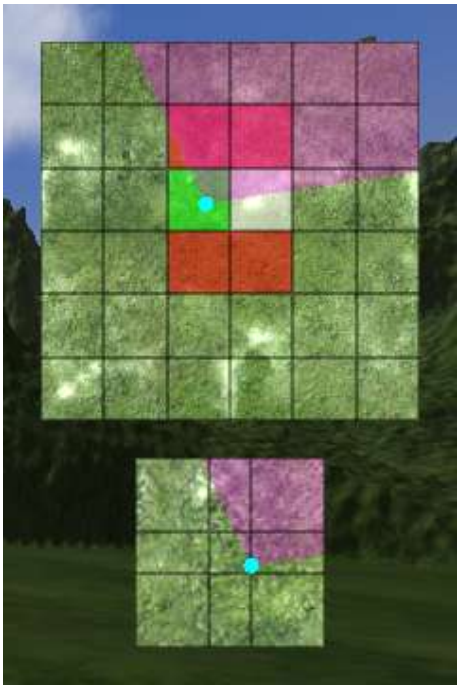

Figure 24: Sample of the two radar images

The vertex shader is simple and is used to pass through the vertex position and texture coordinates to the fragment shader. The fragment shader is where everything happens; there is a switch statement that chooses what set of operations to perform. There are five different states.

State 1: Samples of the coarse-map and colour-map textures are used to provide the background to the radar; the secondary radar samples a region from the specific grid position.

State 2: Is used to render a quad on to the radar and is used to highlight the tile areas as is shown in Figure 24. The green quad represents the current tile the player is in, the white quads show tiles which are loaded and made active by the caching system. The red quads show tiles that are loaded on to the GPU but not made active yet by the caching system.

State 3: Draws in the vision cone offset to the cameras position and indicates the direction the player is looking. This is coloured in purple and shown in Figure 24.

State 4: Draws a set of horizontal and vertical lines; this is done by specifying a 2D position and then two lines are drawn that intersect it.

State 5: Draws a circle offset to the player's location. This is indicated as the cyan dot shown in both the radar images in Figure 24.

These different states are called in their order and alpha blended together to produce the final output. Some of the states are run multiple times to add in all the detail to the radar.

4.4.7 Skybox Shader

The skybox shaders are relatively simple where the vertex shader simply reads in the vertex of the skybox and applies the view transform and outputs it. Then in the fragment shader the sky texture is queried at the fragments texture coordinate, then fog is calculated and the final fragment colour is returned.

```
fogY          = min(0, log2_fog_den * frag_Position.y - log2_fog_den * 10.0);
fogFactor     = exp2(-fogY * fogY);

frag_Color    = mix(texture(sky, frag_TexCoord), fog_col.xxxy, fogFactor);
```

The code above shows how the fog is calculated in the shader. The fog is calculated based on the y-component (height) of the fragments position, a fixed amount is rendered as fog and then a simple linear fade is added for a short distance above that fixed height. This fog helps with the blending in of the terrains fog values.

4.5 Implementation Summary

The implementation of the project proved to be more challenging that initially thought out during the design phase and constant optimisations were needed to keep the application running in real-time. Some concepts such as the clipmap were modified in order to reduce memory overheads which improve the performance of the application. In the end a fully functional deformable terrain system which operates in real-time was produced. The project

was very heavy on implementation and many technical challenges were overcome together with lots of cutting edge techniques improved. These include geometry clipmaps, FBO's, PBO's, geometry tessellation, parallax mapping and partial derivative normal maps, this just names a few. A sample of the applications output is shown in Figure 25, but a live demonstration is needed to showcase it in its glory.

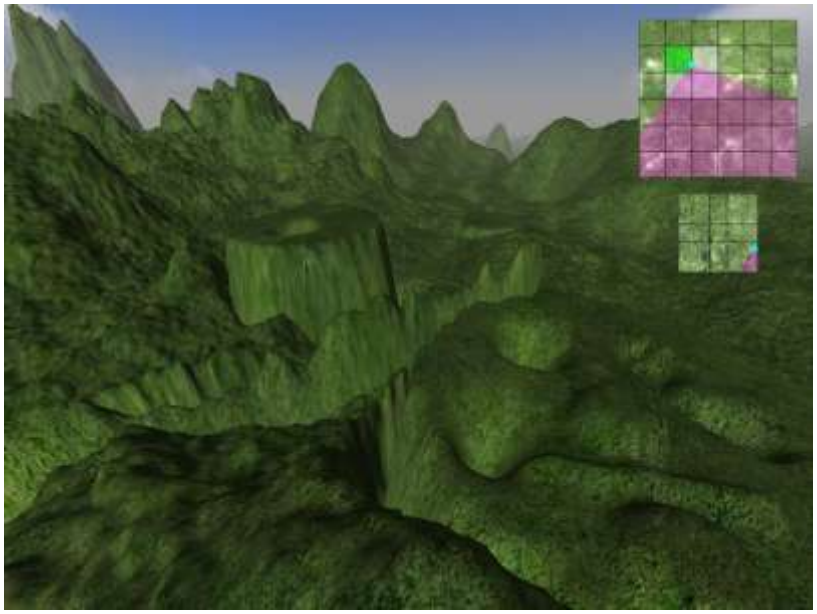

Figure 25: Sample of terrain produced in project

Chapter 5: Results & Evaluation

This chapter lists all the results gathered from the project performed over a variety of different tests. These tests are carried out on two different systems; the first system features an NVIDIA 9600GT GPU coupled with a Core2 Duo CPU and slow hard disk drives (5400RPM) and represents a mainstream PC. The second system features an NVIDIA GTX 295 GPU with a Core2 Quad CPU and fast hard disk drives (7200RPM) and represents a high-end PC. The project is tested on both of these two systems in all of the tests conducted and a comparison is made between the two. The mainstream system is the standard and the project is expected to perform adequately on it. The goal of this chapter is to verify the design constraints mentioned in section 3.1. The applications default configuration is listed below in Table 1.

Attribute	Setting
Screen Width	1024
Screen Height	768
V-Sync	Off
Frame Limiter	Off
Anti-Aliasing Level	0x

Table 1: Application configuration for testing

Six sets of tests are performed on the application and the results are displayed in tables together with a description of the test and a summary of the outcome. After the results are displayed a small overview of the results in general is covered in section 0. Section 5.8 covers an analysis of the differences between the geometry tessellation and parallax mapping implementations and which is the better choice.

In the results an added column which compares the speed increase that is obtained between the mainstream and high-end system. This is done to give an indication of how the application scales performance wise dependant on the hardware configuration. This indicates that the performance is dependent on hardware and that it doesn't contain any massive bottlenecks which hamper performance.

The caching system was not able to be benchmarks because it works asynchronously in the background using a direct memory access bus transfer which provides inconsistent performance data. The performance hit only occurs when changing tiles which results in new tiles requiring being loaded and old ones unloaded.

5.1 Test 1: Average render time / FPS with *Simple* shaders only

This test uses the default configuration on both systems and is designed to test the time in milliseconds (ms) that it takes to render the scene, as well as the average FPS of the application. This makes use of the default *Simple* shaders. The test is run for a total of 60 seconds a total of 5 times for each category. Table 2 shows the results obtained from the tests when measuring the render time in milliseconds. Table 3 shows the results when looking at the average FPS of the application. The test is run on both systems in order to determine the speed increase obtained on a high-end system.

Run Number	Mainstream (ms)	High-End (ms)	Speed Inc
1	8.725	1.815	4.8X
2	8.730	1.813	4.8X
3	8.829	1.813	4.9X
4	8.799	1.814	4.9X
5	8.649	1.813	4.8X
Average	**8.746**	**1.814**	4.8X

Table 2: Results from Test 1a

Run Number	Mainstream (FPS)	High-End (FPS)	Speed Inc
1	95.71	508.07	5.3X
2	95.73	508.70	5.3X
3	92.22	508.91	5.5X
4	93.14	508.98	5.5X
5	96.00	509.12	5.3X
Average	**94.56**	**508.76**	5.4X

Table 3: Results from Test 1b

The results from Table 2 show an average speed increase of 4.8X, this differs from the average FPS increase of 5.4X shown in Table 3. This difference comes from how the average FPS is calculated, this is the average frame-rate calculated over the entire run time of the application. This includes the various other stages of the application which represents a more realistic figure. This is different from the render time, as it only measures the execution time of the shaders used to render the environment. It is worth noting that the original design constraints for a minimum of 30 FPS and even the upper 100+ goal were achieved by both systems with the mainstream one attaining 95% of the target frame-rate. This test makes use only of the coarse-map deformations and is limited in terms of render resolution. See sections 5.3 and 5.4 for a comparison with the geometry tessellation and parallax mapping enabled.

5.2 Test 2: Average FPS with varying screen resolution

This test measures the effect screen resolution has on the average FPS, using the *Simple* shaders. The FPS is recorded over a period of 60 seconds and then the average is reported on application termination. This process is repeated 5 times and the average of them are recorded for the different screen resolutions. The increase or decrease in performance in comparison with the default resolution is reported for each system. The difference between the two systems FPS values is determined and shown in the *Speed Inc* column.

Screen Res	Mainstream (FPS)	Inc/Dec from 1024x768	High-End (FPS)	Inc/Dec from 1024x768	Speed Inc
640x480	125.76	**1.3X**	554.57	**1.1X**	4.4**X**
800x600	111.41	**1.2X**	536.60	**1.1X**	4.8**X**
1024x768	94.56	**1.0X**	508.87	**1.0X**	5.4**X**
1440x900	75.39	**0.8X**	438.38	**0.9X**	5.8**X**
1920x1080	53.83	**0.6X**	371.66	**0.7X**	6.9**X**

Table 4: Results from Test 2

The results here perform as expected; the average FPS should increase as the screen resolution is shrunk down and decrease when the screen resolution is scaled up. This decrease in frame-rate is a result of the increase of the pixel density of the screen. The 1024×768 resolution results in 786,432 pixels, whereas the 1920×1080 resolution has 2,073,600 pixels, which is more than double the number of pixels with only a 40% speed reduction. The increase or decrease amount is shown which compares the average FPS at the current resolution with that of the default configuration of 1024×768. The final column calculates the speed increase of the average FPS between the two systems at the different resolutions. It can be noted that the application scales better at higher resolutions on the high-end system due to the more powerful GPU which can handle larger amounts of data.

5.3 Test 3: Average render time / FPS with *Geometry Tessellation* shaders

This test is run with the Geometry Tessellation shaders enabled. This adds in the high-detail deformations to the terrain by adding in additional geometry by tessellating the inner-most clipmap level. This dramatically increases the amount of vertex data that gets input in to the shaders. The test is run 5 times for a period of 60 seconds in order to guarantee test validity. There are two sets run, the first measures the render time in milliseconds and the second tests the average FPS for the application. These results are shown in Table 5 and Table 6 respectively.

Run Number	Mainstream (ms)	High-End (ms)	Speed Inc
1	131.291	9.831	13.4**X**
2	129.329	9.842	13.1**X**
3	129.561	9.812	13.2**X**
4	131.479	9.822	13.4**X**
5	130.211	9.809	13.3**X**
Average	**130.374**	**9.823**	13.3**X**

Table 5: Results from Test 3a

Run Number	Mainstream (FPS)	High-End (FPS)	Speed Inc
1	7.39	99.73	13.5**X**
2	7.60	99.65	13.1**X**
3	7.57	100.02	13.2**X**
4	7.46	99.75	13.4**X**
5	7.53	99.99	13.3**X**
Average	**7.51**	**99.83**	13.3**X**

Table 6: Results from Test 3b

As can be seen from Table 5 and Table 6 there is a rather large difference between the two systems which results in an average speed increase of 13.3X. This is mostly due to the dramatic increase in the number of vertices in the application. This increase puts significant strain on the mainstream system which averages only 7.5 FPS which fails all the design constraints and cannot be used as a real-time system. This result is compared with the parallax mapping technique more thoroughly in section 5.8.

5.4 Test 4: Average render time / FPS with *Parallax* shaders

This test evaluates the performance of the parallax mapping shaders, which was the main focus of this project report. The tests are conducted in a similar fashion and run 5 times for a period of 60 seconds to gather data on the render time and average FPS. This test is expected to perform better than the geometry tessellation one. The results are shown in Table 7 and Table 8 below.

Run Number	Mainstream (ms)	High-End (ms)	Speed Inc
1	14.772	2.331	6.3X
2	15.300	2.330	6.6X
3	15.136	2.329	6.5X
4	15.329	2.327	6.6X
5	15.414	2.338	6.6X
Average	**15.19**	**2.331**	6.5X

Table 7: Results from Test 4a

Run Number	Mainstream (FPS)	High-End (FPS)	Speed Inc
1	60.01	401.83	6.7X
2	58.19	401.91	6.9X
3	58.83	402.18	6.8X
4	58.01	403.20	7.0X
5	57.52	401.15	7.0X
Average	**58.51**	**402.05**	6.9X

Table 8: Results from Test 4b

The results above prove interesting in that both systems meet the minimum design constraints set out and obtain the minimum of 30 FPS. While the mainstream system does not meet the secondary goal of the 100+ FPS it only scores 35% slower than in test 1 and Table 3 but this now includes the rendering of high-detail to the terrain which is an acceptable drop for the added detail. The high-end system performs on average 6.9X faster when rendering the parallax mapping. A comparison between these results and the geometry tessellation one is conducted in section 5.8.

5.5 Test 5: Average Deformation time with varying stamp scale

This test measures the time taken to perform the height-map deformation and time taken to recalculate the partial derivative map (PD-map) based on the size of the stamp applied. With each different stamp scale, 5 separate deformations are applied and the times for each are recorded, the average of this is calculated and is recorded in Table 9. The two different systems are compared and the speed increase between them is calculated. The deformations

are applied on to the coarse-map in the centre to avoid the multi deformation caused from edge cases.

Stamp Scale (m)	Mainstream Def Time (ms)	Mainstream PD Time (ms)	High-End Def Time (ms)	High-End PD Time (ms)	Speed Inc
10	6.856	6.195	0.899	0.792	7.7X
20	7.027	6.776	0.913	0.800	8.1X
50	7.338	6.898	0.971	0.828	7.9X
100	7.519	7.010	1.192	0.953	6.8X
200	13.151	9.387	1.875	1.329	7.0X

Table 9: Results from Test 5

As the results in Table 9 show, there is a linear increase in the time to deform the height-map and time to re-calculate the partial derivative maps with an increase of the stamps scale. This is the expected outcome as the area that needs to be updated increases. The speed increase between the two systems is fairly constant with minor fluctuations which come from inconsistent shader executions and is a result of hardware error.

5.6 Test 6: Average FPS with varying clipmap level count

This test measures the effect on FPS when different clipmap levels are applied when using the *Simple* shaders. This changes the amount of vertex information there is in the program and is a good gauge on performance. The higher the clipmap level the further the maximum render distance becomes. The test is run for 60 seconds with the various clipmap levels on both the mainstream and high-end systems and the average FPS is recorded. The speed increase between the two is calculated and shown in Table 10.

Clipmap Level	Mainstream (FPS)	High-End (FPS)	Speed Inc
3	112.25	656.26	5.8X
5	95.17	508.82	5.3X
7	80.52	411.59	5.1X
9	70.22	346.23	4.9X
11	63.07	298.09	4.7X

Table 10: Results from Test 6

The results shown in Table 10 show a linear decrease of the average FPS of the application with the increase of the clipmap levels. This relationship is expected as an increase of the clipmap level results in additional geometry being added to the scene. On the other hand by increasing the clipmaps levels the maximum viewable distance increases at an exponential rate. This increase in geometry has only a moderate performance penalty and the application manages to maintain the minimum FPS of 30. The increase from 5 clipmap levels to 11 results in a 120% increase in both the vertex and index count but only has a 40% decrease in the average FPS which is a good trade off of performance for quality.

5.7 Overview of results

It can be seen from the above results that the high-end system performs between 5 and 8 times faster than the mainstream system. This is accounted for by the added performance that the NVIDIA GTX 295 has over the 9600GT. The application performs as expected and the goal of producing a real-time deformable terrain system has been achieved. All the key components have been tested except for the caching system. This is because it is extremely difficult to accurately time the functions presented as it transfers the data asynchronously via direct memory access (DMA) which cannot be timed with necessary accuracy.

It must be noted that the number of deformations applied to the terrain does not affect the performance in any way due to the level of detail system introduced. This means that an unlimited number of deformations can be applied without performance degradation.

The goals set out in section 3.1 aimed to produce a terrain deformation system that is operable in real-time. This set up a minimum frame-rate of 30 FPS which is required to be maintained and then a secondary part to attempt for 100+ FPS in order to be usable in game engines. The second design constraint is that the visual quality of the terrain can contain high resolution deformations and the third being that persistent deformations to the terrain.

It was proved from the results that such a system has been produced that meets the minimum FPS required, the high detail deformations provided by the parallax mapping. The application only managed 60 FPS which falls short of the desired 100+ goal. Deformations are made persistent with the aid of the caching system.

5.8 Evaluation of Two different implementations

As was shown in Table 6, the results for the geometry tessellation failed all the design constraints and proved to be an unacceptable method for creating a high-detail deformable terrain system. The parallax mapping technique produced a frame-rate of 58.51 compared to 7.51 with the geometry tessellation. The parallax mapping technique performs approximately 7.7 times faster than geometry tessellation and meets the minimum of 30 FPS which means the system can be run in real time. The high-detail is provided by purely illusionary texture based techniques and provides adequate quality for the performance penalty.

The average render time in milliseconds is on average 9 times slower with geometry tessellation which shows that there is a large overhead and slow down experienced in the shader calls. This is because the geometry shader creates additional geometry on the fly and the tessellation process performs many complex operations. Much optimisation was done to improve the performance but given the time constraints of the project not much further could be done.

The results here show clearly that a texture based approach to high-detail rendering of deformations on the terrain prove to be the most efficient method and allow for a real-time system to be produced. Further work can be done to increase the efficiency and performance of the parallax mapping technique to yield better frame-rates.

Chapter 6: Conclusion

This project sought out the development of a terrain deformation system that could be operated in real-time. It required the ability to display both coarse and high-detail deformations on an underlying terrain mesh. This project provided ample challenges that were to be overcome. The reference implementation that was chosen in the project was designed to make use of geometry tessellation which added additional vertices to display the high-detail. This approach has obvious limitations as the creation of additional geometry cannot be sustained indefinitely. This is what brought about the second implementation which was based on texture trickery methods that produce the illusion of height at a fixed cost which allows for near limitless deformations to occur. The contents of this report focused on the later implementation with texture trickery.

The challenges faced included the mesh representation which was achieved through the use of a geometry clipmap and using static geometry that had the texture move about its surface. The ability to have both coarse and high-detail deformations was important to the success; the coarse deformations were handled by vertex displacement and the high-detail through the use of parallax mapping. A caching system was implemented into the program together with a notion of tiles which only requires the neighbouring tiles that can be seen to be loaded which saves valuable GPU memory. This allows for a very large world to be created with only using a small memory footprint. Tiles that are no longer necessary are saved out to disk which can later be loaded back. This method is used to provide persistent deformations in the application such that user's deformations are always available to them even if they move to a new region and come back again later.

This report focused on the implementation of parallax mapping to render the high-detail deformation data. This was implemented to compete with the reference implementation in the attempt to produce an application that outperforms the reference implementation. The results chapter showed how this was achieved and that parallax mapping is a viable alternative to geometry tessellation. The project was also shown to operate fully in real-time on the minimum specification hardware that was chosen.

Chapter 7: Future Work

This chapter highlights some possible improvements to be made in the future to further enhance the project.

- **Dynamic Stamps:**
  Add in the ability to create dynamic stamps which change the terrain in different ways over a small period of time. For example creating shockwaves on the terrain.

- **Add More Stamps:**
  Add in more, different types which could deform the terrain more roughly, perhaps an erosion tool to allow the simulation of more realistic terrain.

- **Add Water:**
  Add in water or other kinds of fluid system which can be added to the terrain and is updated for example when the user adjusts the terrain to lower it the water can overflow and spill out. This could be a method of simulating lakes and such.

- **More Optimisation:**
  Further optimise the code and/or find alternative methods to increase the performance, this can allow for the more advanced systems to be implemented.

- **Multiple Coarse-maps:**
  Add in the ability to have more than one coarse-map, perhaps a semi-infinite world in which new maps get created as they are required. This means that coarse-maps get stored with their absolute position. This can be easily clamped to a predefined limit such that after x maps the world wraps.

- **Automatic Height-map Generator:**
  Add in method that allows for the coarse-map to be automatically generated with a few simple user inputs.

- **Height-map Joining Function:**
  Add in the ability to automatically stitch the height-maps together this is a follow up to the above-mentioned ability.

Chapter 8: Glossary

- **Texture:**
  An image or data file which is used to add detail to the surface of terrain. Textures are used for the colour map as well as for the height-maps for coarse and high-detail.

- **Coarse-map:**
  This is the height-map that stores the large-scale deformations. The height values represent low-frequency (slow changing) elevation data and a large area of terrain in the environment.

- **Detail-map:**
  This is a texture is used to store the high-frequency data that would not be visible accurately on the sparse clipmap due to insufficient vertex data. Parallax mapping is used to render this detail on the terrain.

- **Colour-map:**
  This is a texture that stores colour information, usually a seamless image that can be used to cover the surface of a models surface.

- **Texel:**
  Texture element. Smallest component of a texture, the number of horizontal texels is equal to the horizontal dimension of the texture image.

- **VBO:**
  Vertex Buffer Object. This is a set of memory allocated on the GPU for storing vertex or attribute data.

- **VAO:**
  Vertex Array Object. A collection of state information and VBO's. These are used to identify a set of VBO's and OpenGL state information.

- **FBO:**
  Frame Buffer Object. Contains a list of rendering destinations and state information. Allows for direct rendering of textures as opposed to rendering to the screen.

- **PBO:**
  Pixel Buffer Object. Allows for fast pixel data transfer to and from the GPU through DMA (Direct Memory Access) without needing CPU cycles, this allows for asynchronous transfer of texture data.

- **Normal:**
  A vector which is perpendicular to the surface of the object, this is calculated based off the height-map and updated whenever a deformation takes place.

- **Tangent:**
  Tangents are the vector lines that just touch the surface of the object and are necessary for the calculation of normal and parallax mapping.

- **Shader:**
  A shader is a program which is executed on the GPU that performs specific calculations and is used for the rendering of the scene as well as the main deformations and parallax mapping techniques.

- **GPU:**
  Graphics Processing Unit. Specialised hardware device used to output 2D & 3D graphics data and is fundamental to this project.

- **Uniform Variable:**
  A variable that can be sent through to shader programs that is used to specify parameters required on a per render call.

- **Uniform Buffer:**
  The storage area in GPU memory where uniform variables are stored, the shader queries this when an access call is made to a uniform variable.

- **Header File:**
  This file stores the class declaration in a simple format that allows for quick observation of the members stored in the class.

- **Source File:**
  This file has the definition of the class described in the header file. This is where the source code is found for the program.

- **GLSL:**
  OpenGL Shader Language. This is a high-level shading language used by OpenGL which allows for the direct control of the graphics pipeline.

Chapter 9: Bibliography

[1] Blinn, J. F. 1978. *Simulation of wrinkled surfaces.* SIGGRAPH Computer Graphics. 12, 3 (Aug. 1978), 286-292.

[2] Cook, R. L. 1984. *Shade trees.* In Proceedings of the 11th Annual Conference on Computer Graphics and interactive Techniques H. Christiansen, Ed. SIGGRAPH '84. ACM, New York, NY, 223-231.

[3] Tarini, M., Cignoni, P., Rocchini, C., and Scopigno, R. 2000. *Real Time, Accurate, Multi-Featured Rendering of Bump Mapped Surfaces.* Computer Graphics Forum. 19, 3, 119-130.

[4] Wang, J. and Sun, J. 2004. *Real-time bump mapped texture shading based-on hardware acceleration.* In Proceedings of the 2004 ACM SIGGRAPH international Conference on Virtual Reality Continuum and Its Applications in industry (Singapore, June 16 - 18, 2004). VRCAI '04. ACM, New York, NY, 206-209.

[5] Szirmay-Kalos, L., and Umenhoffer, T. 2008. *Displacement mapping on the GPU - State of the Art.* Computer Graphics Forum. 27, 1.

[6] Hirche, J., Ehlert, A., Guthe, S., and Doggett, M. 2004. *Hardware accelerated per-pixel displacement mapping.* In Proceedings of Graphics interface 2004 (London, Ontario, Canada, May 17 - 19, 2004). ACM International Conference Proceeding Series, vol. 62. Canadian Human-Computer Communications Society, School of Computer Science, University of Waterloo, Waterloo, Ontario, 153-158.

[7] Donnelly, W. 2005. *Per-Pixel Displacement Mapping with Distance Functions.* In GPU Gems 2, M. Pharr, Ed., Addison-Wesley, pp. 123 -136.

[8] Cook, R. L., Carpenter, L., and Catmull, E. 1987. *The Reyes image rendering architecture.* SIGGRAPH Comput. Graph. 21, 4 (Aug. 1987), 95-102.

[9] Policarpo, F., Oliveira, M. M., and Comba, J. L. 2005. *Real-time relief mapping on arbitrary polygonal surfaces.* In Proceedings of the 2005 Symposium on interactive

3D Graphics and Games (Washington, District of Columbia, April 03 - 06, 2005). I3D '05. ACM, New York, NY, 155-162.

[10] Kaneko, T., Takahei, T., Inami, M., Kawakami, N., Yanagida, Y., Maeda, T., and Tachi, S. 2001. *Detailed shape representation with parallax mapping.* In Proceedings of the ICAT 2001, 205-208.

[11] Acton, M. 2008. *Ratchet and Clank Future: Tools of Destruction – Technical Debriefing.* Insomniac Games. http://www.insomniacgames.com/tech/articles/1108/files/Ratchet_and_Clank_WWS_Debrief_Feb_08.pdf

[12] Losasso, F., Hoppe, H. 2004. *Geometry clipmaps: terrain rendering using nested regular grids.* SIGGRAPH '04: ACM SIGGRAPH 2004 Papers, 769-776.

[13] Phong, B. T. 1975. *Illumination for computer generated pictures,* Communications of ACM 18 (1975), no 6, 311-317.

Chapter 10: Appendix

10.1 Partial Derivative Derivation

$$h(x+\xi) = f(x) + \xi f'(x) + \frac{\xi^2}{2} f''(x) + \frac{\xi^3}{6} f'''(x) + O(\xi^4)$$
$$h(x-\xi) = f(x) - \xi f'(x) + \frac{\xi^2}{2} f''(x) - \frac{\xi^3}{6} f'''(x) + O(\xi^4)$$
$$h(x+2\xi) = f(x) + 2\xi f'(x) + 2\xi^2 f''(x) + \frac{8\xi^3}{6} f'''(x) + O(\xi^4)$$
$$h(x-2\xi) = f(x) - 2\xi f'(x) + 2\xi^2 f''(x) - \frac{8\xi^3}{6} f'''(x) + O(\xi^4)$$

$$f(x+2\xi) + f(x+\xi) - f(x-\xi) - f(x-2\xi) = 6\xi f'(x) + 3\xi^3 f'''(x) + O(\xi^5)$$

Using the above expansions, a factor α must be found such that the $O(\xi^3)$ term is eliminated in the following:

$$f(x+2\xi) + \alpha f(x+\xi) - \alpha f(x-\xi) - f(x-2\xi) = (2\alpha+4)\xi f'(x) + \frac{2\alpha+16}{6}\xi^3 f'''(x) + O(\xi^5)$$

After eliminating these terms, the finite difference equation below is acquired by dividing through by -12ξ. This equation has an error with the order $O(\xi^4)$.

$$f'(x) = \frac{f(x+2\xi) - 8f(x+\xi) + 8f(x-\xi) - f(x-2\xi)}{-12\xi} + O(\xi^4)$$