A System for Real-Time Deformable Terrain

Justin Crause
University of Cape Town
jcrause@cs.uct.ac.za

Andrew Flower
University of Cape Town
aflower@cs.uct.ac.za

Patrick Marais
University of Cape Town
patrick@cs.uct.ac.za

ABSTRACT

Terrain constitutes an important part of many virtual environments. In computer games or simulations it is often useful to allow the user to modify the terrain since this can help to foster immersion. Unfortunately, real-time deformation schemes can be expensive and most game engines simply substitute proxy geometry or use texturing to create the illusion of deformation.

We present a new terrain deformation framework which is able to produce persistent, real-time deformation by utilising the capabilities of current generation GPUs. Our method utilises texture storage, a terrain level-of-detail scheme and a tile-based terrain representation to achieve high frame rates. To accommodate a range of hardware, we provide deformation schemes for hardware with and without geometry tessellation units. Deformation using the fragment shader (no tessellation) is significantly faster than the geometry shader (tessellation) approach, although this does come at the cost of some high resolution detail.

Our tests show that both deformation schemes consume a comparatively small proportion of the GPU per frame budget and can thus be integrated into more complex virtual environments.

Categories and Subject Descriptors

I.3.6 [**Computer Graphics**]: Methodology and Techniques - *Graphics data structures and data types*
http://www.acm.org/class/1998/

General Terms

Algorithms, Performance, Design.

Keywords

Terrain Deformation, OpenGL, Geometry Tessellation, Parallax Mapping, Caching

1. INTRODUCTION

Modern computer games exhibit a high level of realism, utilising detailed virtual environments which often respond to player interaction. Some interactions, such as opening a door, can be easily scripted. Other interactions – triggering a landmine, for example - result in more fundamental changes to the world geometry. In the latter case, the world designers often provide replacement geometry, such as a crater, which can be swapped in based on the interaction type. While these methods may give the illusion of a responsive world, more complex interactions and responses are not generally supported. For example, if a tank rolls across terrain, the game engine will usually texture a track pattern onto the ground mesh. Close examination will quickly reveal this, which causes a break in user immersion.

Terrain forms a major component of many game worlds and realistic physical interaction with terrain helps to foster user immersion. Such interaction can be expensive to compute, however, and most game engines use low quality alternatives such as texturing or pre-computed proxy geometry.

In this paper we present a framework to manage the efficient, real-time deformation of terrain on a range of graphics hardware. In this context, deformation refers to the warping of the terrain geometry in response to forces arising in the game world, such as explosions or passing vehicles. In order to accomplish this, we developed a tile-based terrain caching scheme and combined this with extensive use of the programmable Graphical Processing Unit (GPU), which exists on most modern computers. The core innovation is the efficient use of GPU textures to store deformation data – this ensures that all deformations are persistent and that the system response does not degrade when large numbers of deformations are applied. To ensure efficient rendering of geometry, a level-of-detail scheme, geometry clipmaps [1], is used. To cater for both old and new GPUs, we developed two deformation schemes: one based on fragment shaders, and another based on geometry shader tessellation. Current GPUs can support both deformation modes.

The paper is organized as follows: Section 2 discusses background and related work. Our system is presented in Section 3 and Results discussed in Section 4. We conclude in Section 5 and then provide some possible extensions for future work.

2. BACKGROUND AND RELATED WORK

Previous work on deformable terrain systems is not readily available in the public domain. Unfortunately computer game engines use proprietary techniques which are not generally published. Consequently, much information about games or systems that may incorporate such techniques is speculative and based on observation of the systems. Despite an extensive literature search, no specific literature was found pertaining to the kind of caching scheme required for our system. The framework presented in this research was designed specifically to meet the requirements of a deformable terrain system.

Our framework requires an efficient, easily dynamic mesh representation to ensure high frame rates. A number of techniques exist [2] for representing a terrain mesh. Common algorithms include ROAM [3], Geometrical Mip-mapping [4] and Geometrical Clipmaps [5]. These algorithms usually require the terrain mesh to be updated regularly on the CPU and

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SAICSIT '11, October 3–5, 2011, Cape Town, South Africa.
Copyright © 2011 ACM 978-1-4503-0878-6/11/10... $10.00

streamed to the GPU. Our system makes use of Geometrical Clipmapping, which focuses on a Level-of-Detail nested mesh scheme centred on the camera, since it maps readily to a GPU implementation.

Triangle Mesh Tessellation takes an existing triangle mesh and sub-divides each triangle to create additional triangles. The aim of this process is to produce a more detailed model from a coarse mesh. If tessellation is done adaptively, such that areas without elevation aren't tessellated, displacement mapping can be applied to apply fine detail to the mesh in an efficient manner [6].

Many tessellation schemes exist. Some evaluate tensor-product or B-spline surfaces, but these are expensive [8]. Subdivision surfaces, generated by algorithms such as that of Catmull and Clarke [9], are a commonly used approximation to these surfaces. Currently two major schemes exist for computing subdivision on GPUs [10]. One involves multiple passes with intermediate results being stored in graphics memory whilst the other performs direct evaluation in a single pass but requires texture lookup tables for tessellation patterns. Accurate subdivision surface generation is not essential to the goals of this problem. Instead basic refinement schemes [11], defined in barycentric coordinates, are used to tessellate the deformed terrain in this system. An alternative to using tessellation is to use texture mapping methods to provide illusionary detail.

Texture mapping is a well-known and widely used method to enhance the realism of computer generated content [12]. There are a variety of texture-based techniques that would be suitable for our system. These techniques include normal mapping [12], relief mapping [13], [14] and parallax mapping [13], [15]. These are all examples of bump mapping techniques [16] which are used to improve realism of computer generated surfaces.

Bump mapping can use simplified geometry and add back detail using data stored in textures. However, this may still produce unrealistic looking objects. This can be solved by actually displacing geometry through displacement mapping which was introduced by Cook [18]. Displacement mapping actually moves vertices on the objects surface to add real detail [19], [20].

We implemented a number of software prototypes to compare the different techniques and based on these it was decided that parallax mapping was the most suitable method, as it provided the most visually pleasing results. Parallax mapping can be extended to include offset limiting and also implemented with an iterative loop, both of which better the final result. The interested reader is referred to [13], [21], [22] for more details.

3. REAL-TIME DEFORMATION FRAMEWORK

Our system combines several established techniques to achieve real-time performance. The core idea is to use additional textures to store deformations and to re-synthesize the relevant deformations on demand as the viewport moves across the terrain.

As identified earlier, there are two main fields that would benefit from a real-time deformable terrain, namely computer games and 3D world modelling. A games engine utilising our framework would be able to fully support destructible terrain. For example, shooting a weapon or tossing a grenade will result in a plausible, programmatically controllable deformation of the target area. Even the most advanced game engines, such as the engine to be used in the upcoming Battlefield 3 [23] game, still lack fully deformable terrain.

3D model designers who create landscapes for movies could benefit from a system which allows for real-time creation. The biggest challenge when creating landscapes is that they need to be rendered before they can be viewed. Our system allows for changes to be seen in real-time as they are applied. The system would, of course, need to be slightly modified to improve control methods for deformation application, by adding brushes and erosion tools, for example. Figure 1 shows the kind of terrain that can be created using the system.

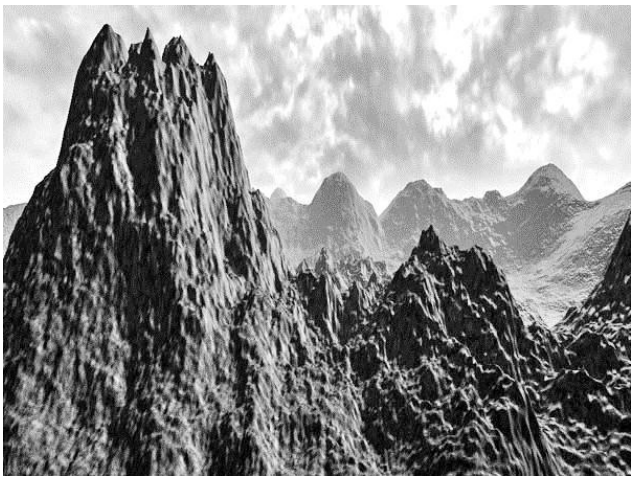


Figure 1: Sample created landscape

Before we developed our final framework, a number of design constraints had to be considered since we wished our system to run on a wide class of modern GPU hardware.

3.1 System Constraints

In order to accommodate a broad range of consumer graphics hardware, our algorithms targeted the NVIDIA 9xxx series of GPUs as our baseline. These devices were the first generation that supported OpenGL 3.x and Geometry Shaders, which are minimal requirements for our framework.

To ensure smooth, real-time performance, we need to ensure that a minimum frame rate of at least 30 FPS is achieved. Given that the deformation system will have to work in concert with several other rendering phases, we arbitrarily set our ideal frame rate as 120 FPS, 4 times the minimum frame rate.

Any deformations that are applied to the terrain need to persist. This requirement motivated the use of the caching system explained in section 3.6. The caching system allows for multiple high-detail maps to cover a single coarse-detail map by swapping map data in when required. Multiple coarse-maps would be possible but in order to limit scope only a single map was utilised in the system prototype.

The system also needs to be free of visible artefacts such as holes in the ground (cracks) and level-of-detail jumps (popping) as this it would break immersion.

3.2 Deformation Frame Work

Based on our analysis, we identified the following major components shown in Figure 2:

1. **Caching System:** The need for a caching system arose when planning the system and realising that multiple texture maps are needed. Both the coarse and high-detail deformation maps have accompanying partial derivative maps; together these require large amounts of GPU memory. The caching system solves this by determining what texture data needs to be loaded onto the GPU at any given time. Based on the system constraints the caching system only handles high-detail maps but could easily be

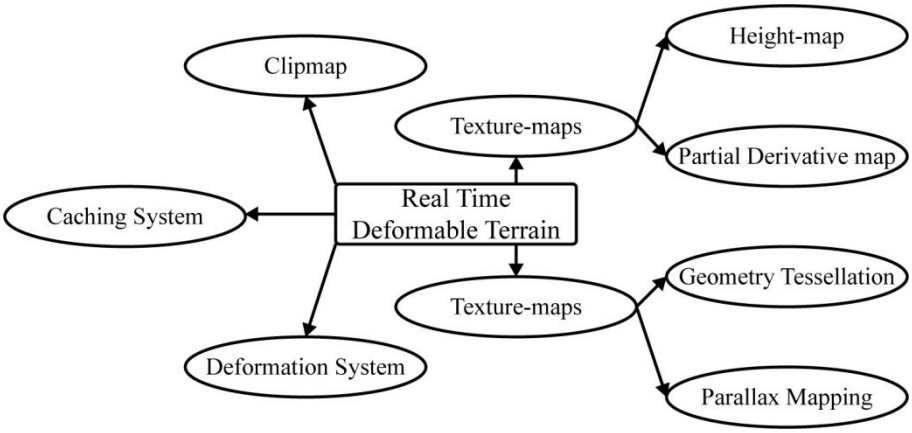


Figure 2: Main components of a deformation system

extended to handle coarse-detail maps. The caching system is discussed in section 3.6.

2. **Deformation System:** The deformation system is responsible for creating the deformations on the terrain. There are two levels of deformation: coarse and high-detail. The deformation system is also responsible for the generation of partial derivative maps. Common methods for terrain deformation use proxy geometry to simulate the effects on terrain. These do not displace terrain accurately and are often pre-computed, thus detracting from the overall realism. However, using proxy geometry is a significant improvement over not applying any form of deformation or simply using a new texture to represent an explosion effect. Details for the deformation system are discussed in section 3.4.
3. **Geometry Clipmap:** Because terrain data further away is harder to see, highly detailed geometry at further distances is not needed. This means using a uniform grid mesh is unsuitable as more detail is needed closer to the camera and less further away. The geometry clipmap was chosen to represent the terrain because on each successive level of the clipmap, the size of the cell doubles in size. Our implementation of the clipmap is covered in section 3.3.3.
4. **Texture-maps:** Textures are used to store the deformation and normal map data. Deformations are stored in the form of height-maps and normals in partial derivative maps. Storing all the data as textures means it can be easily accessed in the shader files. They can also be updated in the shaders to allow for quick deformation, as only small regions are modified and then immediately available for rendering. This removes the heavy cost of transferring the data back and forth between CPU and GPU. Height and partial derivative maps are discussed in sections 3.3.1 and 3.3.2.
5. **Representation:** There are two levels of detail - one representing coarse-detail and one representing high quality deformations. The coarse-detail representation encapsulates large scale deformations, while the high-detail representation is used for small, high resolution deformations. A split multi-resolution representation is sensible since high-detail deformations, such as footprints, will only be visible relatively close to the camera. Two different approaches to rendering the high-detail were developed: one that creates additional geometry and one that relies on texture "tricks" to give the illusion of detail. All the deformation data is stored in texture files as height-map data. Each of these different methods is explained in section 3.5.

3.3 Data Structures

The primary data structure employed by our system is a height-map which stores elevation data in a single channel of a texture. Because deformations must persist, the textures which store the data need to be saved and loaded when needed. The caching system manages this process and is discussed in section 3.6. Another important data structure used in the system is the partial derivative (pd) map. Both the height and pd-maps are stored on the GPU for use in the shaders and to provide fast access to their data. Vertex Buffer Objects (VBOs) are used to store the geometry for the clipmap on the GPU - see below.

3.3.1 Height-map

Terrain elevation data is stored in the form of a height-map image textures on the GPU. These textures consist of a single 16-bit channel that allows for 65536 different values of elevation discretisation. This allows for high mountains etc, whilst still retaining adequate detail across a wide range of height values. Height-maps are a very common and inexpensive way to represent a 2.5 dimension mesh. Two types of height-maps are used in our system: *coarse-maps* and *detail-maps*. The coarse-map we use is $4096 \; x \; 4096$ pixels and represents an area of $409.6m \; x \; 409.6m$ based on a scale of $0.1m$ per texel.

The second set of height-maps (detail-maps) are used to represent finer detail on the terrain. The set of detail-maps is created to tile the area covered by a coarse-map. This allows for distant detail-maps to be omitted from the rendering process as they are too far away to be noticed. The process of caching detail-maps is covered in section 3.6. Detail-maps were chosen as 2048 x 2048 textures with each texel representing 0.03m.

3.3.2 Partial derivative map

Lighting is an essential part of surface rendering in modern computer graphics. Lighting uses reflectance functions that operate on surface normals. The most popular way to store surface normals is through the use of *normal-maps*. Storing normal-maps for the large coarse-map and for each of the detail-maps would consume a considerable amount of GPU memory. For this reason an alternative storage technique was used. Partial Derivative (PD) Normal Maps [24] are essentially the same as regular normal-maps, except that the partial derivatives are instead stored for x and z rather than all three components having the y component calculated at render time. This method saves on a third of the memory storage at the small cost of reconstruction as well as reducing texture-fetch latency. PD-maps also benefit bump-mapping, as it removes the need for tangent-space conversions because the partial derivatives can simply be added to form the composite normal.

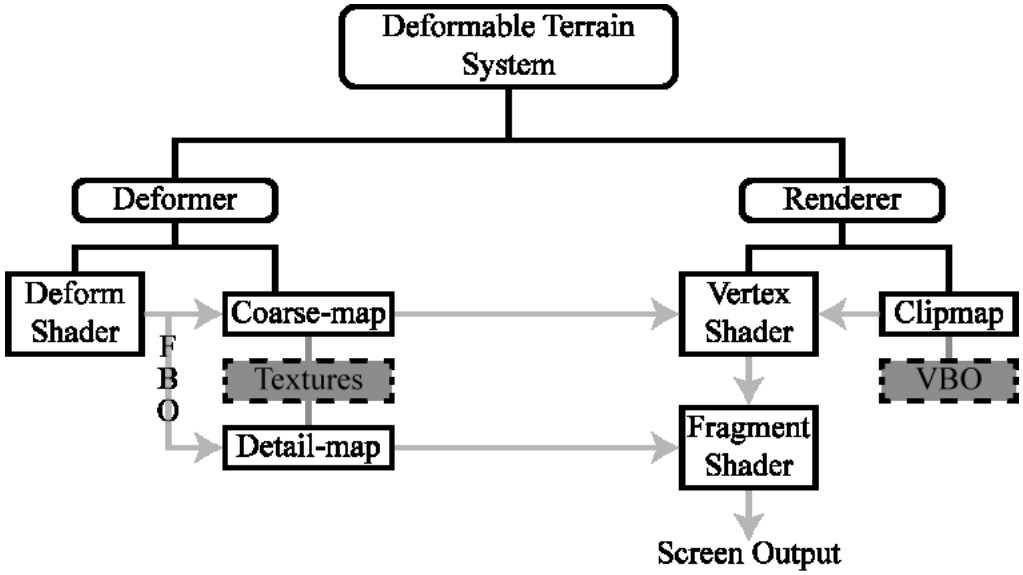


Figure 3: Overview of deformation system

3.3.3 Clipmap

The layout and formation of the set of clipmaps is the same as previous implementations [1]. In order to reduce memory used, the components that exist in each clipmap level are only stored once in VBOs. During render time, they are then scaled and translated into the correct location. In addition to the vertex VBOs, there are corresponding VBOs for texture coordinates and indices. The vertex indexing is ordered in the form of triangle strips to reduce the index buffer size and to make better use of the post-transform vertex cache. While traversing the environment, terrain detail near the edges of each clipmap displays sudden changes as the level-of-detail in a region changes.

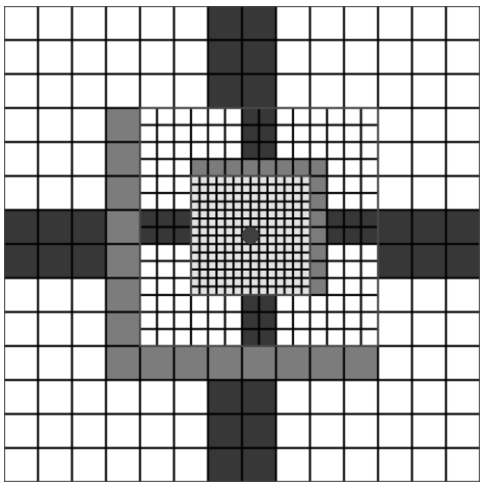


Figure 5: Clipmap with 3-levels, divided into quads:

A sample clipmap is shown in Figure 5. It should be noted that the innermost level is not the centre of the clipmap. However, this does not affect the position of the camera for large size clipmap sizes. Each vertex stores its position information and an associated texture coordinate, which is used to look up elevation data in the shaders to deform the clipmap. The vertices of the innermost level are spaced such that sampling adjacent texels map to adjacent vertices. The clipmap and camera are not translated through the world. Instead the texture coordinates are shifted by the amount the camera gets moved by the user.

The sampling rate of the vertices provides sufficient detail for the coarse deformations but lacks the resolution for high-detail.

Figure 4: Examples of stamps in the system

This is where one of the two high-detail methods comes in and adds the additional resolution to the innermost clipmap region.

3.4 Deformation System

An overview to the deformable terrain system is provided in Figure 3 which shows the linking between the deformations and rendering components. These two components share the same set of textures which aids in performance and reduces the memory overhead. The arrow shows the flow of information in the system. The blocks with dashed outlines represent data structures used as input.

A deformer object is created in the system and is responsible for handling the various deformation tasks for both coarse and high-detail, as well as the calculation of the PD-maps. The deformer makes use of a Frame Buffer Object (FBO) which allows for the deformation to be rendered directly to a texture stored on the GPU. More information on deformations is provided in Section 3.4.1.

On the rendering side of the system, the clipmap, which is stored as a VBO is used as input to the various shaders. The shaders used vary, depending on whether the Geometry Tessellation or Parallax Mapping method is running. Both make use of the clipmap VBO to represent the coarse-detail but utilise the high-detail maps differently - Section 3.51.

3.4.1 Deformations

Deformations to the terrain are applied through the use of stamps. We define three types of stamps. The first is a *functional* stamp, which applies a specific mathematical function to the area over which the stamp is applied. The second stamp is based on a *texture* loaded at start-up. The texture is

stored as a height-map which is used to alter the surface of the terrain. The last type of stamp is *dynamic*, meaning it varies over time.

Figure 4 shows examples of the texture and functional stamps available in the system. Each of the stamps is applied at a specific location on the terrain, based on the location clicked by the user on the terrain. Stamps can have other properties associated with them, for instance they have an intensity value which denotes how strong the deformation should be, and they also have a scale which denotes the size of the area being affected by the deformation. These parameters along with the stamp's ID are needed by the deformation engine to work out where to place the new deformation. Deformations can occur on either the coarse or high-detail maps, both of which use the same functions to create the deformations, they just affect different textures. The dynamic stamp cannot easily be presented in images and as such we refer the reader to our website for video files [27]. Due to limitations of the caching system, deformations have a maximum distance at which they can be applied. For the coarse-map, this distance is much larger than in high-detail mode. We do this because a coarse-map covers $409.6m$ whereas a high-detail map only covers approximately $68.2m$. As such only some maps will be available in GPU memory, so a limit on how far away from the camera deformations can be applied is enforced. One solution to this would be to allow for an offline process where deformations that fall on unavailable tiles are added to a queue and this queue is processed in the background. However, this was beyond the scope of this research.

When deformations occur on the edge of a texture, the neighbouring tiles will also be affected. As such separate deformations need to be applied to each texture, which is loaded in turn. After each deformation has been applied, the corresponding pd-map needs to be updated to reflect the new height values.

Because deformation and elevation data are stored in textures, the deformation process involves the alteration of these textures in GPU memory. This is achieved by making use of render-to-texture functionality provided by OpenGL's Framebuffer Objects (FBOs), which allows one to bind textures as render targets. However, FBO operations cannot write to a bound texture, therefore a copy of the texture must first be made and bound to a texture unit in order for the current height-map state to be read and altered. The deformation process, which is essentially a rendering step, is performed by OpenGL GLSL shader programs. The general deformation shader, which applies an arbitrary stamp, simply reads from both the current tile and stamp textures and adds their height values according to the given intensity, scale and rotation. The final height value is rendered as a texel to the new tile texture. Functional stamps each have a specific fragment shader that performs the terrain deformation based on arbitrary parameters. The Gaussian stamp, for example, determines the extrusion intensity from the gaussian function centred on the clicked location on the terrain. Dynamic stamps, which may change their effect over time, are merely advanced functional stamps that require an independent texture to store the stamp's current state. A shockwave effect can thus be created where the current wave state is stored as the stamp but is updated every frame using a shader that implements the Wave Equation, before being applied to the terrain.

After a deformation or a series of repeated deformations, the coarse-map data is streamed back asynchronously to the CPU for use in the collision detection system. The texture data is read into an OpenGL Pixel Buffer Object (PBOs) which hands control over to the DMA for transfer of the data into system memory. This allows both the CPU and GPU to continue processing during the transfer.

3.4.2 Partial derivative map generation

The primary benefit of partial derivative normal maps [24] is that they only require the storage of two components. The definition of a surface-normal states that it is the cross-product of a surface's tangent and binormal vectors which can be derived from the partial derivatives. The finite difference approximations for derivatives are used to calculate the two first order partial derivatives stored in the R and G channels.

PD-maps are created and recalculated at a number of stages throughout the system's lifetime. Initially, when the coarse-map is generated or loaded from file, the corresponding PD-map is calculated. PD-maps for detail-maps are created when the detail-maps are loaded from the cache, and discarded again when the maps are cached to disk. This is to reduce bus transfer times and time spent writing to disk, because the recalculation is significantly faster and thus worth the trade-off. Finally, PD-map recalculations occur after any deformation of a height-map in order to maintain consistency between topography and its surface shading.

3.5 Representation

There are two modes for displaying the deformation data on the terrain - the first for coarse-detail and second for high-detail. Two sets of shaders are thus used during the rendering process, each for different regions of the clipmap. The innermost region makes use of one of the two different high-detail methods, either Geometry Tessellation or Parallax Mapping. The outer regions all use a standard shader that only represents coarse-detail. The coarse-map is sampled by the vertex shader during displacement of the clipmap vertices. Standard phong lighting is calculated in the fragment shader which completes the process.

For the innermost region things work slightly differently. First displacement mapping handles the coarse deformations and then either the Geometry Tessellation or Parallax Mapping shaders take over. These change the standard geometry and fragment shaders to enable each of the different techniques. More information for each of these methods is provided in the next two sections.

3.5.1 Geometry tessellation

In order to accurately represent the fine terrain detail created by deformation, a geometry tessellation technique was implemented. The method makes use of geometry shaders adhering to the SM3.0 standard. Since rendering performance degrades linearly with respect to the number of triangles [25], it is essential that our tesselator produces the smallest reasonable set of new primitives. We thus utilize an adaptive tessellation scheme, where vertices are only marked for tessellation if there is deformation data at that location in the detail-map.

Each triangle is processed by the geometry shader. If the triangle is within a certain distance from the camera it may be tessellated according to some refinement pattern. Refinement patterns define exactly how a triangle will be tessellated into sub-triangles. Each pattern is defined as a set of barycentric coordinates; the choice of pattern is based on the states of the *tessellation-property* of all three vertices. Each vertex can either be set to tessellate (1) or not (0). There are thus $2^3 = 8$ possible combinations. The chosen pattern is based on a pattern index ρ calculated from the tessellation states t_i of the three vertices v_0, v_1 and v_2 using the following bitwise expression.

$$\rho = t_0 + t_1 \ll 1 + (t_2 \ll 2)$$

This expression produces an integer in the range $[0,8)$. $\rho = 7$ represents the case when all the sides of the triangle require tessellation. $\rho = 3,5,6$ represent the cases when only one side requires tessellation. Finally, $\rho = 0,1,2,4$ results in no tessellation. Once tessellated, new vertices can then sample the relevant detail maps for the elevation values. The new composite normal must be calculated using the coarse-map and detail-map's normals. This simply involves summation of the two sets of partial derivatives. Having the different patterns is essential to creating a smooth merging between tessellated and non-tessellated regions, removing the occurrence of T-junctions which would ruin the visual quality.

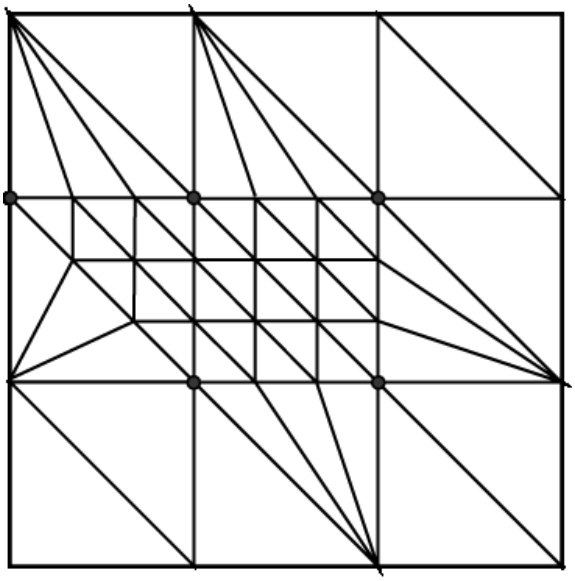


Figure 6: Example of adaptive tessellation

Figure 6 shows how a section of terrain mesh may be tessellated given an area of partially deformed heights. A wireframe rendering of adaptive geometry tessellation is shown in Figure 7.

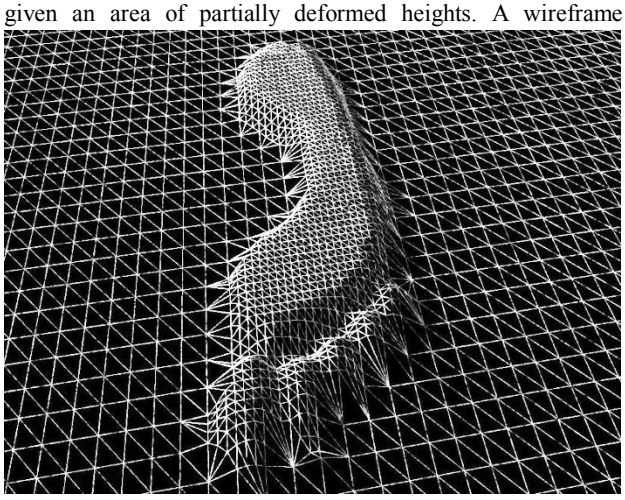


Figure 7: Wireframe of Geometry Tessellation

3.5.2 Parallax mapping

Parallax Mapping is a texture-based alternative to Geometry Tessellation for the representation of high-detail regions of a model. It does not rely on the generation of new geometry. Since parallax mapping only requires code to be inserted in the fragment shader and does not need the geometry shader – this method will also work on older generations of hardware. Because this technique does not create additional geometry, the performance cost is significantly reduced. The number of displacements that can be applied depends on the resolution of the textures being used in the parallax function.

Iterative parallax mapping, with offset limiting, was chosen after extensive testing showed that it produced the best results. Three key variables are used to control this method of parallax mapping: parallax bias, scale and number of iterations. The following values were chosen to be final for the system; bias of -0.004, 0.004 for scale and an iteration count of 4.

Figure 8 shows an example of parallax mapping to produce several deformations. An overlay of the same image in wireframe mode is added to illustrate that no physical geometry has been added or displaced. The deformations produced are purely illusionary [15].

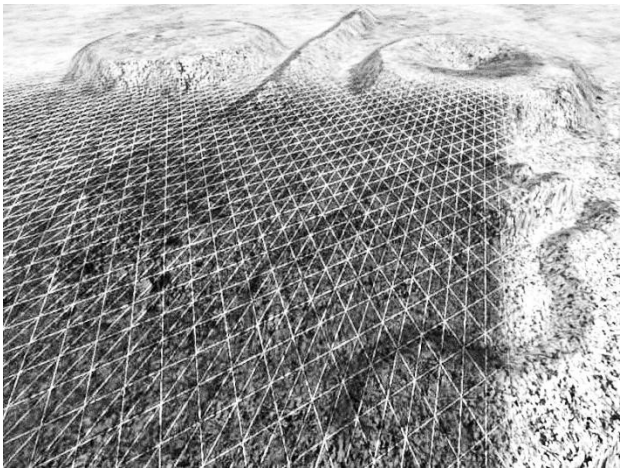


Figure 8: Wireframe overlay of Parallax Mapping

We show in our results that Parallax Mapping significantly out-performs Geometry Tessellation, although the visual quality is not identical. The quality is, however, high enough to be an acceptable means of representing high-detail deformations on terrain. This trade off in quality is balanced by the performance gain.

3.6 Caching System

The caching system is responsible for managing the detail-maps. It is required because all of the detail-maps cannot be loaded into GPU memory at the same time. The coarse-map is divided up into a grid with each cell representing a high-detail texture or tile. The number of tiles used to span the coarse-map depends on the resolution of the high-detail textures. This resolution is chosen so as to achieve minimal noticeable aliasing in both the tessellation and texture-based approaches.

For our prototype, the terrain consists of a single coarse-map of dimension $NC = 4096$. The underlying clipmap mesh has a resolution of $\delta_C = 0.1m$ at the finest level. The finest level is refined by a factor of $\gamma = \frac{1}{3}$ yielding a resolution of $\delta D = \gamma\delta_C = 0.03m$. Texel distance for the detail-maps is thus δ_D and the number of tiles n can then be calculated using the dimensions of the detail-map textures, $N_D = 2048$. This calculation is shown below which yields $n = 6$.

$$
\begin{aligned}
n &= \frac{\delta_C N_C}{\gamma \delta_C N_D} \\
&= \frac{N_C}{\gamma N_D} \\
&= \frac{4096}{1 \div 3 \times 2048} \\
&= 6
\end{aligned}
$$

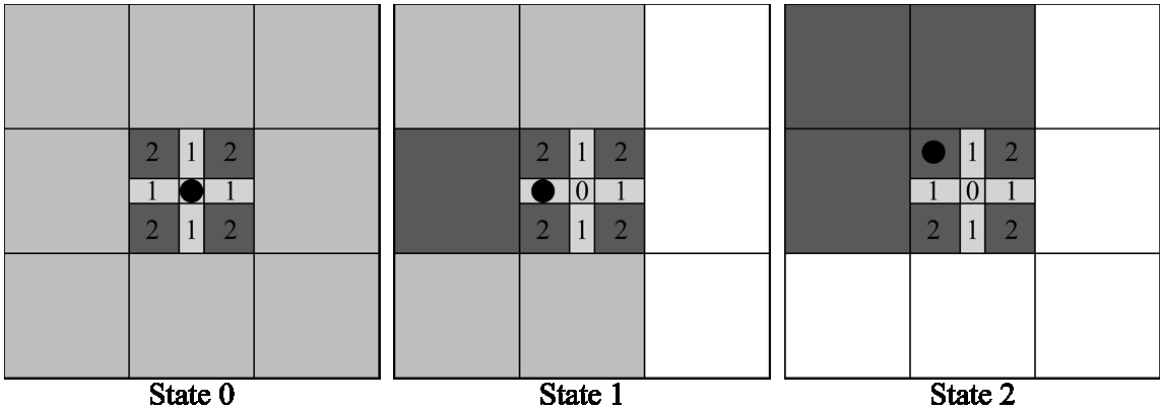


Figure 9: Comparison of caching states

A total of $n \times n = 36$ tiles are thus required to cover a small coarse-map. Storing these on the GPU would consume 144 MiB of VRAM, excluding the normal-map textures which would require an additional 200% of memory resulting in 432 MiB. This is an unacceptable amount of memory for the terrain system to consume alone. In addition to this, computer games would need to store the terrain mesh, game models and other textures. This also puts a very low upper bound on the supported size of terrains. This is the main reason why an efficient caching system was developed.

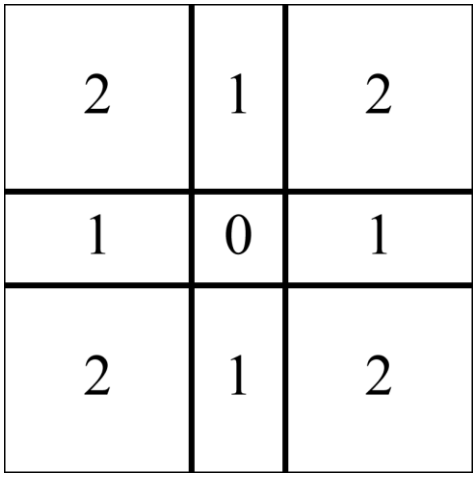


Figure 10: Caching boundaries of a tile

The caching system works on a region system, whereby the tiles are divided up into nine regions of three different types - these are shown in Figure 10. 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 is sent to all the tiles required by the previous state and then a load command is performed for the current state. This results 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 present. These loading and unloading requests are handled by the caching PBO's, which stream the data in and out asynchronously, and are handled by a separate thread. Figure 9 shows the three different states the caching system can be in. The lightly shaded blocks represent tiles that are currently loaded in to GPU memory; the dark shaded blocks are loaded and made active and white blocks are currently unloaded.

As shown in Figure 9, 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 active. Under this system, there can at most be nine textures loaded in to GPU memory; this gives a total of 108 MiB, including PD-maps, irrespective of the grid size.

Initially, if no existing cache is to be loaded from disk, all tiles share a texture storing zero deformation data. When transitioning between regions, no loading and unloading is necessary. This saves processing time that would have been spent wastefully. When a deformation operation is performed on a tile using the zero texture, a new texture is created for the tile as a copy of the zero texture and the deformation is then performed. As an additional optimization, textures are not cached to disk unless they have been modified since they were loaded. To save time spent on waiting for hard-disk requests and bus transfers, normal maps are not cached but are rather recalculated each time a texture is loaded.

4. RESULTS

Our testing setup for the system comprised two differently configured desktop computers. The first system used has a Core i7 950 CPU, 6GB DDR3 RAM and a 7200RPM Hard Drive. Three different NVIDIA GPUs were tested in this computer: GTX 295 using only one GPU core, GTX 480 and a GTX 580. The secondary system forms the bottom of the range for the tests as it uses lower performing hardware. It has a Core 2 Duo P8600 CPU, 4GB RAM and a 5400RPM Hard Drive, and a NVIDIA 9600GT GPU. Tests were conducted on the four different computer configurations and the results are discussed below.

Four different tests were conducted to test the various aspects of the system. The first test measures the render time and an average FPS. The second test shows how the system performs at different screen resolutions. The third compares how various deformation stamp sizes affect the total time to produce a deformation in the system. Finally, some benchmarks for the caching system are presented.

The following system parameters were chosen for the testing process. A screen size of 1600x900 was used for all the tests except when testing different screen resolutions in test 2. V-Sync and Anti-Aliasing were both disabled. The standard coarse-detail mode was used except when testing the high-detail method's performance in test 1. In tests 1 and 2, the player walks forward through the world for a total of 60 seconds, this covers ⅓ of the terrain, while making measurements constantly. The overall average for this period is presented in the results.

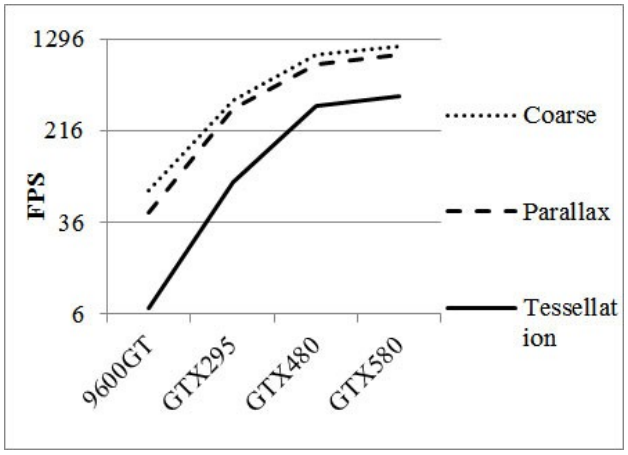
The frame rates are presented in graphs that use a logarithmic scale; this was chosen due to the substantial differences between the various devices. It is shown on the newer hardware that there is a large amount of unused performance. This would allow for other systems to be run as part of game engine.

Test 1 compares the performance of the coarse-detail only method with the two different high-detail methods, all using the four different system configurations. The total time taken to perform a single render step is recorded in ms and presented in Table 1.

Graph 1 plots the average frame rate against the different GPU types for each of the rendering techniques. From the results it is clear that the performance is increased when using newer GPUs, it is also noteworthy that there is a much larger increase between GPU architecture generations which explains the large increase from the 9600GT to the GTX295 and from the GTX295 to the GTX480. A smaller increase is noted between the GTX480 and the GTX580 as they are both based on the *Fermi* architecture. Based on the results in Graph 1 it is seen that on the lowest level of hardware, the 9600GT, the only method that fails to achieve real-time is the method based on geometry tessellation, on all other devices and methods the system operates with more than 30 frames per second. This shows that geometry tessellation is not suitable for use on lower end GPUs.

Table 1: Time (ms) to render scene

	9600GT	GTX295	GTX480	GTX580
Coarse	15.425	2.666	1.164	0.976
Parallax	23.466	3.036	1.364	1.149
Tessellation	148.447	12.576	2.934	2.456

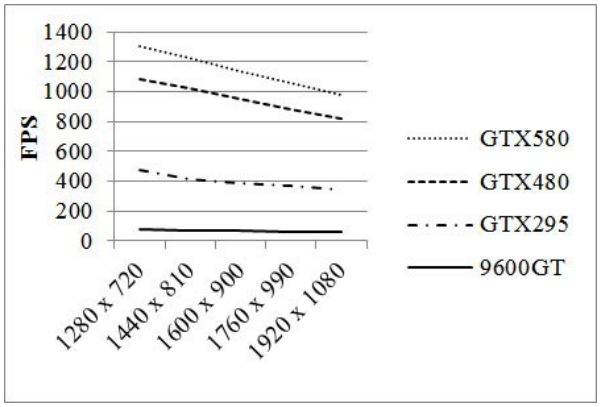


Graph 1: FPS with using different rendering techniques

Test 2 presents the render time in ms of the system when using the coarse-detail only method, with varying screen resolutions. This shows how the system scales when the total number of pixels to process is increased. As seen in Table 2, the time to render the scene increases as the screen size is increased. This is shown as a linear relationship in Graph 2 which plots the average frame rate against the different screen resolutions. The performance decreases at a linear rate as the number of pixels that need to be rendered increase. This is more noticeable on the newer GPUs, GTX480 and GTX580; this is because of the high magnitude of the frame rate for these devices when comparing on a logarithmic scale graph.

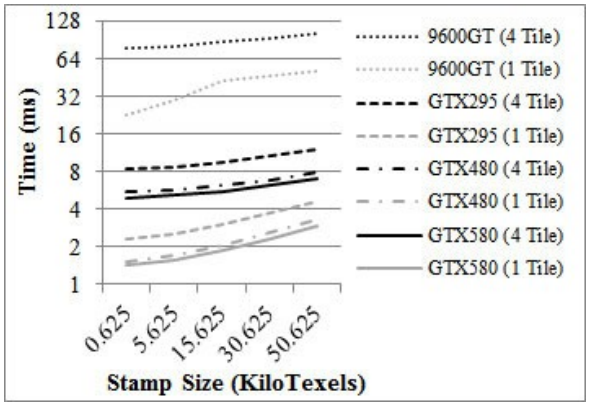
Table 2: Time (ms) to render based on different screen sizes

	9600GT	GTX295	GTX480	GTX580
1280 x 720	12.626	2.399	1.023	0.863
1440 x 810	13.898	2.509	1.088	0.913
1600 x 900	15.425	2.666	1.164	0.976
1760 x 990	16.759	2.823	1.239	1.041
1920 x 1080	17.536	3.002	1.313	1.113



Graph 2: FPS using different screen sizes

In test 3 the deformation component of the system is tested. We tested each of the four configurations against an increasing stamp size when applying a deformation in the centre of a tile and then also when deforming a corner which requires the three neighbouring tiles to also be deformed. The total times to complete a deformation on both a single tile and on a corner which affects the three neighbouring tiles is recorded in ms and presented in Graph 3. A deformation includes the adjustment of the height-map as well as the recalculation of the associated PD-map. It is noted that the time to perform a corner deformation which involves 4 tiles is less than 4 times the cost of performing a single tile. This is attributed to an optimised solution which removes redundant state change operations and employs asynchronous memory transfers to save computational time.



Graph 3: Deformation time with varying stamp size

There was no easy way to measure the performance of the caching system and as such no results can be displayed. It was noted during experimentation that no lag or stall was experienced when moving through the world as a result of caching.

More extensive testing could have been conducted (measure GPU utilisation, memory transfer speeds, etc) but this testing would not change the overall observation that the deformation framework is extremely efficient and imposes a marginal overhead per rendered frame.

5. CONCLUSION & FUTURE WORK

Terrain forms an important part of many computer game environments. Although game worlds support some dynamic modification, the world terrain itself tends to remain static. To address this problem, we have developed a tile-based terrain deformation system which is efficient and supports persistent deformations. Extensive use of GPU shaders and texture storage ensures that our framework is able to deliver high frame rates

for any number of deformation operations on a given terrain tile. Persistence is achieved by means of texture data stores and a pre-emptive loading scheme which ensures that the appropriate deformation information is always resident when required by the renderer. Our framework supports two deformation schemes: one based on parallax mapping, and another on tessellation. The minimum frame rate of 30 was obtained for all but the geometry tessellation method on the minimum hardware system. Based on the tests it was clear that the parallax mapping method produces much better results than geometry tessellation. Although the texture stores do require additional resources, the tiling scheme limits the number of textures which need to be resident on the graphics card at any one time. The overhead of maintaining the deformation infrastructure is thus minimal and will not consume an excessive amount of GPU resources.

There are a number of ways in which our core deformation framework could be improved in the future: The ability to have more than one coarse-map could allow 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. A limit could easily be imposed which starts to wrap the maps after a certain amount.

New graphics cards support Shader Model 5.0, which has better support for tessellation as part of the rendering pipeline. Additional optimizations are certainly possible.

In locations such as the horizon and hills or contours, the piece-wise linear edges can be easily seen. In order to reduce this visual artefact, smoothing should be done on contours by use of a fast technique such as Phong Tessellation [7].

Additional optimizations could include the use of compressed texture formats and geometry orderings that are more vertex-cache-friendly. Further optimisations of the code to increase the performance, this can allow for the more advanced systems to be implemented. Finally, a better cache management strategy, such as that used in Virtual Texturing [26], could be implemented.

Implementing the system in a complete game engine would allow us to evaluate the system more thoroughly. Multiple players could be present in the game where each could place mines around the environment and trigger them remotely. This would test the true performance of the system with multiple explosions causing deformations across the map.

The detail could further be improved by a hybrid approach whereby vertices in the immediate vicinity of the player are tessellated to create high-detail. A new level between the high and coarse-detail could be introduced that uses parallax mapping for vertices slightly further out before the coarse-detail level handles the rest.

6. REFERENCES

[1] Asirvatham, A. and Hoppe, H. 2005. *Terrain rendering using GPU-based geometry clipmaps*. In GPU Gems 2, chapter 2. Addison-Wesley

[2] Barton, R. 2010. *Modern Algorithms for Real-Time Terrain Visualization on Commodity Hardware*. http://geoinformatics.fsv.cvut.cz/gwiki/Modern_Algorithms_for_Real-Time_Terrain_Visualization_on_Commodity_Hardware

[3] Hwa, L. M., Duchaineau, M. A. and Joy, K. I. 2005. *Real-time optimal adaptation for planetary geometry and texture: 4-8 tile hierarchies*. IEEE Transactions on Visualization and Computer Graphics. Vol. 11. No. 4. 355–368

[4] de Boer, W. H. 2000. *Fast terrain rendering using geometrical mipmapping*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.58.2737&rep=rep1&type=pdf

[5] Losasso, F. and Hoppe, H. 2004. *Geometry clipmaps: terrain rendering using nested regular grids*. In SIGGRAPH '04: ACM SIGGRAPH 2004 Papers, pp 769–776, New York, NY, USA

[6] Moule, K. and McCool, M. D. 2002. *Efficient bounded adaptive tessellation of displacement maps*. In Graphics Interface, pp 171–180

[7] Boubekeur, T. and Alexa, M. 2008. *Phong Tessellation*. In ACM SIGGRAPH Asia 2008 papers (SIGGRAPH Asia '08), John C. Hart (Ed.). ACM, New York, NY, USA, Article 141

[8] Huang, X., Li, S., and Wang, G. 2007. *A GPU based interactive modeling approach to designing fine level features*. In GI '07: Proceedings of Graphics Interface 2007, pp 305–311, New York, NY, USA

[9] Catmull, E. and Clark, J. 1978. Recursively generated B-spline surfaces on arbitrary topological meshes, Computer-Aided Design, Volume 10, Issue 6, Pages 350-355

[10] Kazakov, M. 2007. *Catmull-clark subdivision for geometry shaders*. In AFRIGRAPH '07: Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisationand interaction in Africa, pp 77–84, New York, NY, USA

[11] Patney, A, Ebeida, M. S., and Owens, J. D. 2009. *Parallel view-dependent tessellation of catmull-clark subdivision surfaces*. In HPG '09: Proceedings of the Conference on High Performance Graphics 2009, pp 99–108, New York, NY, USA

[12] 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

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

[14] 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

[15] 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

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

[17] 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

[18] 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

[19] Hirche, J., Ehlert, A., Guthe, S., and Doggett, M. 2004. *Hardware accelerated perpixel 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

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

[21] Engel, W. 2004. *Shaderx3: Advanced Rendering with DirectX and OpenGL, (Shaderx Series)*. Charles River Media, Inc., Rockland, MA, USA

[22] Welsh, T. 2004. *Parallax Mapping with Offset Limiting: A Per-Pixel Approximation of Uneven Surfaces*. https://www8.cs.umu.se/kurser/5DV051/VT09/lab/parallax_mapping.pdf

[23] Battlefield 3. 2011. http://www.ea.com/battlefield3, Last Accessed: *31 May 2011*

[24] 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

[25] NVIDIA, 2008. *GPU Programming Guide*. pg 28. http://developer.download.nvidia.com/GPU_Programming_Guide/GPU_Programming_Guide.pdf, Last Accessed: *01 June 2011*

[26] Wavere, J. M. P. van. 2009. *From Texture Virtualization to Massive Parallelization*. Siggraph 2009: id Tech 5 Challenges. http://s09.idav.ucdavis.edu/talks/05-JP_id_Tech_5_Challenges.pdf

[27] Crause, J., Flower, A. 2010. *Real-Time Deformable Terrain* Website. http://people.cs.uct.ac.za/~jcrause/defter/index.php, Last Accessed: *01 June 2011*