A Framework for Real-Time, Deformable Terrain

Andrew Flower[*]
Department of Computer Science
University of Cape Town

November 2, 2010

Abstract

The use of static terrain in modern computer games can detract from the realism, in that users cannot interact with the terrain as they would other game entities. This report proposes a design and implementation of a deformable terrain system that can be used in modern computer games. The system provides two levels-of-detail for the terrain, each of which can be independently deformed. The first, allows large-scale deformations that affect the macro-geometry of the terrain - the geometry that players walk on and collide with. The second deformable level-of-detail represents fine-scale surface detail that affects the appearance of terrain and allows for superficial detail such as footprints and bullet-holes. The proposed deformations occur in real-time and can be predefined or procedurally generated. The system makes use of state-of-the-art techniques such as geometry clipmaps, displacement mapping, render-to-texture and parallax mapping. Successful results are acquired for the large-scale deformable system, but only high-end machines may support the fine-scale deformations.

[*]e-mail: aflower@cs.uct.ac.za

Overview

Contents

List of Figures

Introduction

The focus of modern computer games is on realism. Not only physically-based rendering, but the physics itself as well as interactions with the virtual environment. The combination of these factors results in greater immersion and presence within the environment. In order to satisfy these goals, computer games must be consistent in the way they work and react with the player. If the player performs an action, he/she expects an appropriate reaction to occur. When such a reaction does not occur, the player notices and may lose his/her degree of presence within the game environment. Terrains are used in the majority of modern computer games, and have been for quite some time. They are, however, static in all but a few cases. A static terrain does not react to the actions of the player and it can therefore detract from the sense of realism built up by other environment factors. A player not accustomed to this shortcoming of computer games, would expect feedback when shooting the ground, exploding a bomb or even walking across snow. The vast benefits of the inclusion of a dynamic terrain are thus clear. Besides the feedback a player would experience, dynamic terrain offers a sense of unpredictability where environments do not remain the same. For example, players could dig trenches in which to lie or tunnel underneath enemy barriers. In multiplayer games, this could add an entirely new paradigm.

Although dynamic terrains and destructible environments are very uncommon, they are not non-existent. The games that do possess these features, however, make use of them in a limited manner. The terrain deformations or destruction of objects is usually predefined. Predefined deformations are repetitive and, although better than none at all, can still detract from the realism. It is a lot more common for games to display such detail in more superficial ways by the use of decals. Decals are images with transparent backgrounds that are laid over objects. A common example is the bullet-hole decal which is placed at locations where guns have fired as if the bullet actually pierced the ground or wall. Because they are basic 2-dimensional images, they lack realism. In addition to their lacking realism, decals usually disappear after a specified amount of time. This is done to save processing time because every extra decal brings extra computation for the CPU and Graphics Card. Disappearing decals and dead-bodies are another set of shortcomings within modern computer games.

This report introduces the design and implementation of a dynamic terrain system for use in computer games. The system supports vertical deformation of the terrain on both a coarse and fine scale. Coarse-scale deformations affect the elevation of the terrain whilst the fine-scale deformations change surface detail and appearance. The shape or form of these deformations is customizable and can even be determined on-the-fly if chosen. This allows for unpredictability and removes the problem of repetitiveness. Subsequent deformations do not add any extra computation and therefore an infinite number of deformations can be performed throughout the application's run-time without a drop in performance. Because of this, deformations need not decay or disappear. This system adds a greater sense of realism to computer games. The only cost is that fine-scale deformations require a considerable amount of memory.

Chapter 1 introduces concepts relevant to the understanding of graphics techniques used in modern computer games and the terrain system proposed by this report. Chapter 2 covers the design of the terrain system and brings in some of the techniques mentioned in Chapter 1. Chapter 3 describes the implementation of the deformable terrain system in more depth. Results and timings of certain tests are listed and evaluated in Chapter 4. Chapter 5 concludes the report and considers areas of improvement.

Chapter 1

Background

1.1 The GPU

The Graphics Card or Video Card is a component of modern computers completely dedicated to the generation of images and text on the screen. In the same way the CPU controls a computer, the graphics card is controlled by the Graphics Processing Unit (GPU). The primary driving force for the development of GPUs as coprocessors were computer games which required ever increasing processing power to produce more realistic environments. Because GPUs need to perform only a subset of the operations that the CPUs do, their architecture took a different development route resulting in the ability to perform many floating-point operations quickly and at the same time. Their architecture can be classified as almost SIMD (Single Instruction, Multiple Data) according to Flynn's Taxonomy [Fly72] which means that the same instruction is executed on different data simultaneously. The architecture can be better classified as SIMT (Single Instruction Multiple Threads) [NVIc], however, in that individual threads may opt out of conditional operations and branching. The processing power of a GPU is largely related to the number of parallel cores that it has. The fast scalar processors that execute individual threads are commonly referred to as the *cores* of a GPU. The state of the art NVIDIA chips contain 480 cores (scalar processors) [NVIb]. It is this large number of highly capable parallel cores that allows the graphics card to process visual data at such high rates to produce the visual effects seen in games and simulations today.

1.1.1 Fixed-Function Pipeline

Graphics Pipeline is a term used to refer to the multi-stage process applied to data in order for it to be displayed on the computer screen. In computer games, the most common screen output is that of a character or virtualisation of some real-life object. An on-screen 3D object, the final product of the graphics pipeline, is an amalgamation of a number of raw resources. The most essential of these pipeline inputs is mesh data. A mesh is the description of an object's 3-dimensional shape. Because it is difficult to represent continuous data on a computer, meshes are usually described by a discrete set of polygons or facets. These facets are referred to as primitives and are usually triangles (or quadrangles in other cases). Each triangle that is used to approximate the object mesh is described using three vertices. A vertex is a positional point used to describe the corners or intersection of shapes and, in the case of a triangle, the three corners. The vertex is defined using a 3-vector (with x,y and z coordinates) or 2-vector (with x and y coordinates) depending on whether the environment is 3D or 2D respectively. A mesh is thus defined by an ordered set of vertices specifying triangles.

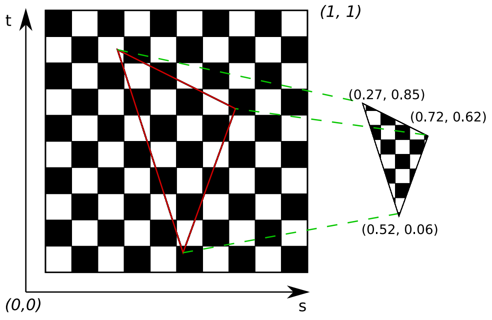


FIGURE 1.1: *Demonstrates the mapping of s and t coordinates on a 2D texture.*

In addition to shape data, an object can have other arbitrary attributes associated with each vertex. It is common to use an image to provide a mesh with colour detail. These images are known as textures and though most often 2-dimensional, can also be 1D or 3D. In order to map a texture onto the 3D mesh, each vertex must have a set of texture-coordinates associated with it such that they define what part of the image should fall on the specified vertex. It is as if that part of the images is stretched over the polygon. Texture coordinates' axes align with the dimensions of the image as seen in Figure 1.1. The coordinates in three dimensions can be labeled u, v and w or, alternatively, s, t and p. The process of texture mapping is thus often referred to as UV-Mapping. Once a triangle has three points of reference within the texture, from the texture coordinates of each vertex, the internal colouring can be interpolated. Each component of a texture coordinate has the range $s, t, p \in [0, 1]$ such that $(0, 0)$ is the lower left corner and $(1, 1)$ is the upper right.

A 3rd popular vertex attribute, after position and texture coordinates, is that of Normal vectors. These help describe the curvature of the smooth surface being modeled, as their mathematical definition states them to be orthogonal to the surface or line. Normals are used later in the pipeline for lighting and shading the polygons and can have a big effect on how soft or hard an edge appears on a model.

There are two major processing stages in the pipeline, those of vertex and pixel processing. The first of these in the graphics (or rendering) pipeline is the *transform stage*. Typically three matrix transforms are applied to vertices: world, view and projection. The world transform controls the orientation and positioning of a mesh in the 3D environment and is usually a concatenation of rotation, translation and scaling matrices. After the world transform, a *view* (or *camera*) transform is applied. Usually the viewer can move around the environment as well rotate their view. To make calculations simpler, the viewer is always kept at the origin with no rotation the rest of the world is moved around it. The view transform transforms the mesh vertices into what is known as *eye space* or *object space*. The final transform is used to project the triangles onto the viewport which is the application window or computer screen. Most 3D games use a perspective projection transform such sizes tapers off with distance, whereas CAD tools may prefer orthographic projection which have no dependence on depth.

The next major stage involves the drawing of the 2D triangles to screen. This process is called rasterization and generates a set of pixels that fill the triangle. The value of each vertex attribute is interpolated, from the three vertices, that corresponds to the generated pixel. A pixel (or fragment) processing phase then takes place that can use these attributes to perform lighting and shading calculations, the result of which is written to the framebuffer.

There are other intermediate stages as seen in Figure 1.2 and are described briefly in the following section.

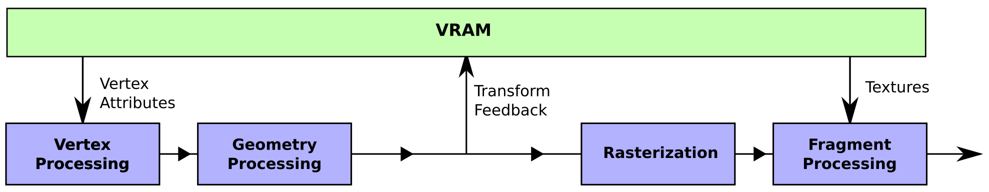


FIGURE 1.2: *Basic illustration of the major stages in the Graphics pipeline. On most modern cards, texture fetch is also supported in the vertex and geometry processing stages*

1.1.2 Modern Graphics Pipeline and Programmable Shaders

In the past decade, the fixed-function pipeline has become programmable, allowing developers to customize the vertex and fragment processing stages. A shader program comprises a set of instructions that execute in parallel on the GPU. In the beginning they needed to be programmed in assembly, but since then a number of high-level languages have arisen such as OpenGL's *GLSL*, DirectX's *HLSL* and NVIDIA's *Cg*. Using a high-level language unfortunately removes the performance impact that was previously obvious from the assembly instruction count. It is therefore still beneficial to view the compiled assembly as an optimisation process. Programming for a GPU is different to that of a CPU. For example, there is minimal branch prediction. Because of the SIMD architecture, each core must follow the same path. When a branch is encountered, two things may happen: Predication [HB05], where all branches are executed and the result of the appropriate one is chosen ; or each branch is executed indepenently in a sequential manner, resulting in idle processors. GPUs can perform floating-point operations on a 4-vector with a single instruction and even have an instruction (MAD) that can multiply and add three operands. It is important to exploit such functionality to lower instruction count and increase overall throughput of the massive quantities of rendered data.

Originally there were only two types of shader programs: the Vertex Shader and the Fragment (or Pixel) Shader. Since then, with DirectX 10 and Shader Model 4.0, a Geometry Shader was added to the pipeline [MSD]. The geometry shader was available in GLSL via the EXT_geometry_shader4 extension until the OpenGL 3.0 specification accepted it. Each shader program type can be used for different purpose as is fit for that stage in the pipeline.

When a primitive, such as a triangle is pushed into the render pipeline, the vertex shader is the first to encounter data. As input, the vertex shader receives an arbitrary vertex from the pipeline along with its attributes such as normals, texture coordinates or colour. The usual job of the vertex shader is to transform the vertices and normals into eye-space and to possibly perform vertex lighting, passing the results down the pipeline. Very often, the same vertex is used more than once. This is done by specifying vertex indices when defining primitives. Indices, when used correctly, help to exploit the post-transform vertex cache which is of a FIFO architecture. Arranging triangles into strips [Hop99] are a means to using the cache, but can still be outperformed by well organised triangle lists [LY06]. When a cache-hit occurs, the vertex does not need to be transformed again which can save significant processing time. A Vertex Shader outputs the transformed vertex along with any associated attributes.

The vertices making up a primitive are then received by the Geometry Shader. In a conventional render pipeline, the GS simply sends the vertices as output in the same state they entered. The GS can be used, however, to further process the primitive as a whole. It has the power to reject the primitive, create more primitives or alter the input primitive. The primary condition is that all input primitives are of the same type and that all output primitives are of the same type. The GS may therefore be used to perform tessellation on primitives though the performance drop is roughly linear with respect to the number of primitives output [NVI08]. There is also a GPU-dependent limit to the number of vertices and attributes that may be output.

At this stage in the rendering pipeline, data may be output to VRAM using Direct3D's Stream-Output or OpenGL's Transform Feedback [Khrb, SA09] functionality, but is usually passed to the next stage in the pipeline, Rasterization. The resulting primitive is clipped against the viewport. This means that if any vertex lies out of the viewport, it is split into smaller primitives such that they are contained within the screen. Primitives that face away from the viewer are also culled here if *back-face removal* is enabled. The primitives are broken up further into basic fragments (or pixels) which are then passed to the final stage.

The fragment shader is executed for each fragment. The vertex attributes of each vertex on a triangle are automatically interpolated across the pixels in the triangle so that the fragment shader receives the correct values. The usual job of the fragment shader is to set the colour of the pixel/fragment or perform pixel lighting. It may also apply bump mapping or textures. After the shader executes, the fixed function performs any blending that was specified for the destination framebuffer. Depth values may also be written to a depth buffer.

1.1.3 OpenGL

OpenGL (Open Graphics Library) is a common cross-platform API that provides high-level control of the GPU. The recently released specification for version 3.2 [SA09] includes functionality relevant to this project which targets Shader Model 4 GPUs such as the NVIDIA GeForce 9 and 200 series cards. OpenGL 3.2 functionality can be compared with that of Direct3D 10.x. The latest specification has removed the fixed-function pipeline transferring the vertex, primitive and fragment processing responsibilities to programmable shaders.

A Vertex Buffer Object represents an allocated section of GPU memory in which vertex data is stored. This allows an object mesh to have its vertices stored in VRAM rather than system memory which decreases render time because data need not be sent across the system bus. Mesh vertices almost always have associated attributes. These attributes may be stored in the same VBO as the vertices, a process known as *interleaving*, or they can each be stored in separate VBOs.

A Vertex Array Object is another OpenGL construct that is used to encapsulate a set of render states and data pertaining to a given renderable object. Prior to the adoption of VAOs, a mesh would require each of it's associated VBOs to be bound independently. This produced many laborious function calls. VAOs are defined by associating a set of VBOs to corresponding attribute location. This VAO need only be bound to set up the state for the given mesh.

The modern graphics pipeline, prior to Shader Model 5, as described in section 1.1.2 contains three programmable stages. OpenGL provides a shading language, OpenGL Shading Language (GLSL or GLSlang), that allows control of the vertex, geometry and fragment processing stages. The 3.2 core specification introduced the Geometry Shader which provides a means to subdivide triangles.

After the Fragment Shader stage, fragments are blended and written to the framebuffer. Framebuffer Objects [Khra] have been introduced into the OpenGL specification. They provide virtual framebuffers that allow rendering to be done to image buffers or textures rather than the system's framebuffer. This is useful for post-render processing and the altering of textures. FBOs also allow MRT (Multiple Render Targets) to be bound so that a single render call can output to multiple textures or buffers.

Finally, OpenGL's Pixel Buffer Objects provide an interface for transfer between RAM and VRAM. Their are two types of PBOs, *Unpack* and *Pack* buffers, that allow transfer to and from the GPU respectively. Rather than wasting CPU and GPU cycles on the memory transfer, PBOs make use of machines DMA to perform the data transfer allowing a higher level of asynchronocity in program flow.

1.1.4 General Purpose Computation on the GPU

Once people realised the parallel computation power of GPUs, they began using shaders in new and interesting ways. The computation-to-price ratio is outstanding and is a great incentive for scientists to invest in powerful GPUs in order to run simulations [Gre05a]. Many algorithms were implemented on the GPU such as N-body simulations, large matrix multiplications and finite difference schemes. Although these produced much faster results, their implementations were non-intuitive and very finicky. Since then NVIDIA has released their GPGPU framework named CUDA (Compute Unified Device Architecture). CUDA creates an abstraction of the GPU hardware so that developers may treat the hardware as a collection of SIMT processors which execute generic kernels rather than hacking graphic shaders together. Many of the graphical effects use a GPGPU-esque approach and CUDA can be used in conjunction with OpenGL to simulate and render fluid simulations or particle simulations.

The geometry shader can be used as a stream processor [Dia07] performing calculations on vertices containing arbitrary data and writing variable-length results to feedback buffers or arbitrary texture locations. This flexibility enables a wide range of algorithms to be implemented on the GPU and is useful to multi-pass techniques for writing intermediate data.

1.1.5 Memory

Modern GPUs offer a significant amount of memory. The memory can be used directly via the CUDA API but is only accessible implicitly in OpenGL via constructs like textures and VBOs. Storing geometry and images on the GPU is a favoured approach as it minimises CPU to GPU bus transfer which has high latency. This is enforced in OpenGL 3.0 with the deprecation of immediate mode and the fixed function pipeline, leaving Vertex Buffer Objects as the only solution. Although their purpose is to store graphical data, VBOs can be used to store other data in GPGPU applications for example. It is beneficial, though not essential, to understand the different sections of GPU memory [NVIc], their advantanges and disadvantages, and what get stored in each of them.

Image textures are stored in Texture Memory. The benefits of texture memory come from the caching, where spatial locality is exploited by loading adjacent texels into cache. Textures are available to all three shader stages for sampling however texture formats, addressing modes and interpolation may be limited in the earlier shader stages. Textures are a useful alternative to VBOs as they allow radnom access reads, though this use-case may not benefit from caching. It is also possible to dynamically write/render to texture objects from the fragment shader with the use of FrameBuffer Objects (FBOs) [Gre05b]. The location of the written fragments may controlled, for the purpose of random access writes, using a Geometry Shader.

Textures can be used to store arbitrary data such as mesh data as well, where the coordinates are stored in colour channels. Such textures are known as *Geometry Images* as seen in [HR06]. An advantage of GIs is that level-of-detail (LOD) meshes can easily be created using mipmaps of the GI. Mipmaps are sets of scaled-down versions the original texture that accompany it. They are created by repeatedly halving each dimension of the texture, storing each smaller image as another mipmap level, until a size of 1 texel is reached. They help to reduce scaling artefacts.

1.2 Surface Detail

Real world surfaces are never completely smooth. It is therefore important, in computer graphics, to increase realism of objects by adding details to the surfaces. A number of techniques exist that accomplish this task: some modify the geometry whilst others affect the surface shading. In this section a mesh is assumed to have corresponding normal vectors at each vertex which can be used in per-pixel lighting calculations.

1.2.1 Displacement Mapping

Displacement maps [Coo84] are textures containing elevation data. Such height fields can be used to display fine detail on geometry or to represent larger coarser meshes. Displacement maps are mapped to a mesh in the same way as regular colour textures. Displacement maps actually displace the geometry to add the detail rather than performing shading. Each texel contains a value indicating the magnitude of the correspondingvertex's displacement. These maps appears as cloudy grayscale images when viewed as images. The displacement map defines a surface in a plane tangential to original surface. A given vertex looks up its corresponding displacement in the map using its texture coordinates, and the vertex would then be translated accordingly along its normal. Equation 1.1 demonstrate how the vertex's new position **v** is calculate from the original **u** using the height from the displacement map $D(s,t)$ and its normal $\hat{\mathbf{n}}$.

$$\mathbf{v} = \mathbf{u} + D(s,t)\ \hat{\mathbf{n}} \tag{1.1}$$

Because displacement maps operate on vertices, a dense mesh is required in order to represent detail of sufficent resolution otherwise the resulting mesh appears pointy and unrealistic. A common technique is to further tessellate the surface into smaller smaller triangles before applying the displacement map. Displacement maps are commonly used to generate terrain meshes, in which situation they are referred to as heightmaps. This is usually performed as an initialisation step on the CPU, but with modern hardware displacements in the Vertex Shader are possible with minor performance hits. Heightmaps can be generated in a number of ways. One method is to use Perlin noise allowing control of different frequency layers. Another popular method is the Diamond-Square algorithm [FFC82] which is a fractal-based approach.

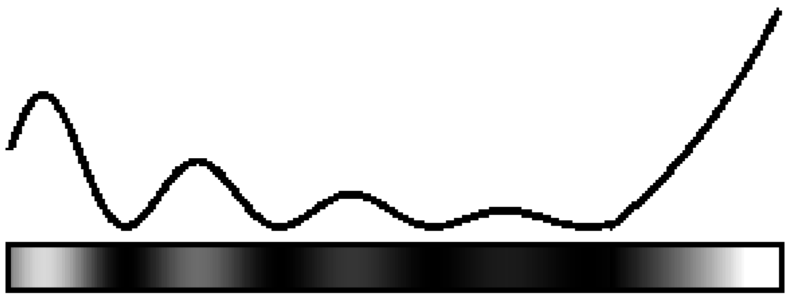


FIGURE 1.3: *An example of a 1D displacement map and the line it represents.*

Once the surface is displaced, the vertex normals are no longer correct and must be recalculated if they are needed for lighting. We know from calculus that the normal at any point of a smooth surface can be computed as the cross-product of the tangent and binormal vectors, as this is their definition. Thus the displacement surface's normal can be calculated by choosing the tangent and binormal vectors to be the changes in the s and t texture coordinates respectively. [SKU08] give an in-depth look at displacement mapping and these calculations.

If the original mesh is just a horizontal plane, then the calculated normal can be used as the normal, otherwise it needs to be combined with the normal of the base mesh to find the actual normal. To avoid the complicated combination, lighting calculations are usually performed in Tangent Space [SKU08]. This space is defined as the space with tangent, binormal and normal as the basis vectors. The paper mentioned covers the math involved. Alternatively, instead of calculating the normals, they can be stored in a normal map texture to save computation.

Displacement mapping is a very useful technique for adding surface detail to dense meshes, coarse terrain meshes or for dynamic surfaces such as ocean waves. Real-time deformation is thus also easily possible by simply editing the displacement map. For uniformly dense meshes rendering becomes an expensive process in which case image techniques like bump mapping are used.

1.2.2 Bump Mapping

Bumping mapping, as introduced by Jim Blinn [Bli78], involves varying normals across a surface in such a way that it appears to have bumps on it when lit. Bump mapping is sometimes used synonymously with *normal mapping*, and on the odd occasion refers to an image heightmap. In Autodesk Maya, for instance, the user can supply a grayscale heightmap as the bumpmap. The software implicitly converts the to a normal map and then applies normal mapping. Bump mapping is used as an alternative to displacement mapping because it is a much less expensive method for producing fine detail. This is due to the fact that it operates on fragments rather than vertices.

Normal Mapping

This was the technique introduced by Jim Blinn whereby a vector field/map of normals is supplied with a mesh. The values in this *normal map* are merely the normals along the surface being modeled. The texture coordinates of this mesh are used to index the normal map and locate the corresponding normal. Using this “fake” normal, the usual lighting calculations are performed. In Blinn’s paper he demonstrates the bumps of an orange skin applied to a sphere. Traditional texture maps and displacement maps can be used in conjunction with Normal maps to create composite effects. Normal mapping provides a computationally inexpensive means to displaying fine detail on surfaces.

Parallax Mapping

This technique [KTI$^+$01] extends the notions of texture and normal mapping by considering the parallax effect. Consider your line of sight to a swimming pool floor. Imagine now that the floor is actually the triangle to render and the water surface is the bumpmap surface. The point on the water surface that the view vector intercects is not orthogonally above the floor point that it intersects. This is the parallax effect. Traditional normal mapping would incorrectly lookup the height using the floor point’s texture coordinates which is not the height of the surface at intersection. Instead parallax mapping locates the point of intersection and its corresponding texture coordinates. These coordinates are used instead for looking up heights, colours and normals from the relevant texture maps.

The original naïve parallax mapping approach uses a constant surface height approximation to simplify calculations. Although this is an approximation, it still yields better results than normal mapping. Other

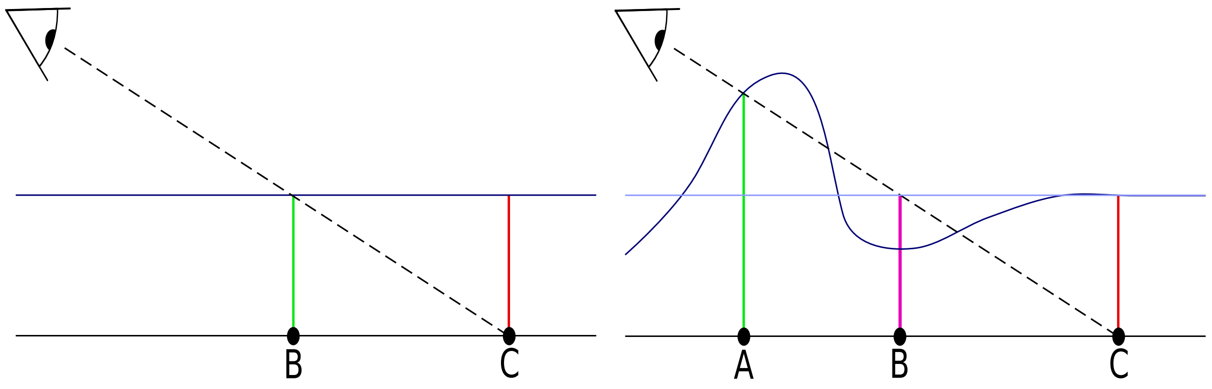


FIGURE 1.4: *Illustrates the principle behind parallax mapping. C is the projection used in normal mapping, B is the naïve parallax mapping using a constant surface, whilst A demonstrate the correct solution.*

approaches exist for locating the actual surface intersection point with varying accuracy and computational cost [SKU08]. These techniques include iterative search, binary search, secant methods and others as heuristic approaches that find the closest intersection sometimes. To ensure that the closest intersection is found expensive searches are required. In addition to extra processing, Parallax Mapping requires both a normal map and a height map.

1.3 Mesh Representation and Level of Detail

The choice of mesh representation and storage is very important due to the fact that large terrain can require vast quantities of vertices. There are various LOD techniques that employ methods to reduce the detail of far away regions. Popular techniques include: ROAM (Real-time Optimally-Adapting Meshes) [DWS$^+$97], Chunked LOD, BDAM (Batched Dynamic Adaptic Meshes) and Geometry Clipmaps. These algorithms are mainly implemented on the CPU, however, and the resulting mesh data passed to the GPU for rendering.

It is difficult, but essential, to acquire seamless transitions when changing the detail level of a region as the camera moves over the terrain. Geomorphing [Hop] is a technique that can solve this problem. When a higher density mesh moves into a region that was represented by a lower density mesh, the extra vertices immediately jump to the intermediate heights that were previously unseen and interpolated on triangle edges. Geomorphing requires the extra vertices to move gradually towards these intermediate heights rather to maintain continuity.

1.3.1 Geometry Clipmaps

Losasso and Hoppe proposed the method of Geometry Clipmaps [LH04]. This is a geometrical parallel to the idea of texture mipmaps and involves the creation of a pyramid of clipmaps or grids. Each subsequent level is has the same number vertices as the previous level, but consumes twice the space in each dimension. The idea is for the camera to be situated at the centre and to nest each of these grids on top of one another such that the resolution of vertices halves in regular intervals away from the viewer. When projected into screen-space, the triangles of subsequent levels consume the same area and thus provide sufficient detail to the viewer.

The algorithm was implemented on the GPU [AH05] and yielded good performance. This method supports heightmap compression and decompression and runtime detail synthesis such fractal detail from uncorrelated Gaussian noise.

1.3.2 Tessellation

Tessellation is a process whereby a given surface is split up into polygons that fully cover the original surface without overlaps. In computer graphics these polygons are usually quads or triangles. Minimising vertex count in meshes is critical to increasing flow through the graphics pipeline. The goal for tessellation in graphics applications is to produce smooth looking models from coarse meshes. This was a very difficult and expensive multi-pass procedure before the introduction of Shader Model 4.0 and the Geometry Shader. Although the geometry shader is limited in comparison to the modern tessellation pipeline it still makes tessellation possible.

Tessellation is an inviting concept for displacement mapping. If tessellation can be done in an adaptive manner, such that areas of greater displacement are more dense, displacement mapping will become a more prominent technique for displaying detail [MM02]. The most simple form of triangle tessellation is to equally subdivide the triangle on a plane using a bilinear interpolation of each of the vertices and their attributes. For an already dense mesh, this is sufficient but for lower resolution meshes, interpolation in the orthogonal dimension is required to produce smooth enough results.

Phong Tessellation [BA08] is a technique that follows a similar process to that of the interpolation of normals in the Phong lighting model. The purpose of the algorithm is to create smooth silhouettes and contours rather than generating formal parametric surface approximations. The algorithm simply requires all the triangle vertices with their normals. From these attributes a new internal point is generated by projecting the point to the vertices' tangent planes and then interpolating using barycentric coordinates. This is a simple heursitic for creating smooth tessellations on top of which further displacement mapping can be performed.

Subdivision Surfaces Like most graphics concepts, the generation of smooth surfaces has been a research topic for many years. A number of mathematical surface representations have been considered such as tensor product surfaces and B-spline surfaces. Evaluating these parametric surfaces is an expensive process [HLW07] and not suitable for real-time implementation. Catmull and Clark [CC98] devised a method for generating approximations to these surfaces with adjustable accuracy, a method termed *Subdivision*. A subdivision surface is defined by a coarse (usually rectangular) mesh known as the *control mesh*. It is akin to the control points in parametric surfaces or splines. Each of the faces can be subdivided according to rules specified by a *refinement scheme* [PEO09]. This refinement process is performed recursively until the desired level of detail is reached.

Catmull-Clark subdivision converges quickly to a bicubic B-spline limit surface after few iterations. The benefits of this method are quick convergence, simplicity of implementation and the fact that each patch can be generated independently with just local information. Less memory is required to store the control mesh than would be needed for usual smoother meshes. The locality of the tessellation allows for relatively easy parallelization. In order to subdivide a quadrilateral, the quad's vertices and the 1-ring neighbourhood of vertices are required.

Currently two major schemes exist for computing subdivision on GPUs [Kaz07]. 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. The algorithm can be simplified if constrained to input control meshes with vertex valences of 4 (quads).

Another approach is to parallelise each step in a breadth-first approach [PEO09]. The different stages are facepoint generation, edgepoint generation, vertex updating and rendering. This approach is more suited to a CUDA implementation where each stage can be shared by all cores. The vertex dependencies of faces on arbitrary meshes is non-deterministic however, thus making memory-coalescing difficult.

A final evaluation method of Catmull-Clark surfaces involves storing the basis functions in a texture along with corresponding 1-ring control points [HLW07]. This would allow exact evaluation of the surface after lookups.

1.4 Summary

The Graphics Processing Unit has become very powerful processor with vast functionality. The programmable pipeline provides the option of performing previously CPU-bound algorithms on the GPU. This frees time on the CPU for performing other intense calculations such as physics and artificial intelligence. Along with these, the time spent waiting on bus transfers to GPU memory can be lessened by storing and computing everything on the GPU.

With regards to deformable terrain, the use of heightmap textures to store elevation data is perfect given that graphics APIs now support render-to-texture operations. Methods such as Geometry Clipmaps provide an efficient solution to representing very large heightmaps and tessellation techniques can allow higher resolution sections of the terrain mesh.

Chapter 2

Design

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 and 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.

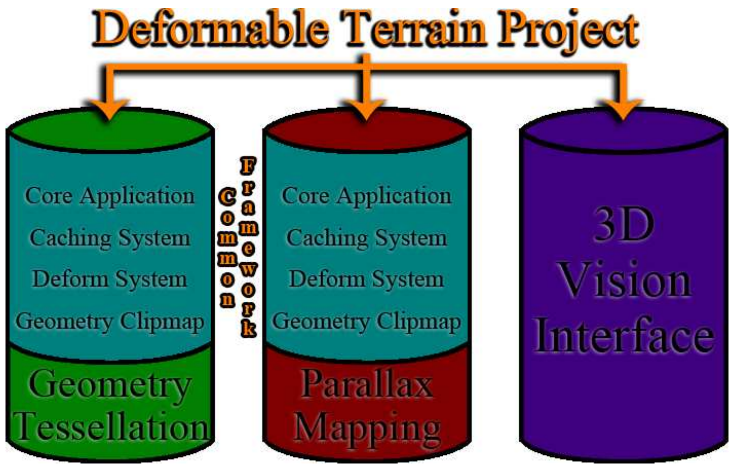


FIGURE 2.1: *Shows the architectural layout of the entire project, with the focus of this report shown as the left (green) cylinder.*

2.0.1 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.

2.0.2 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 will be always the same amount of geometry in the scene. The process involves more texture reads and a more complicated lighting calculation; these overheads will 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 will compete directly with the aforementioned geometry tessellation one.

2.0.3 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 one on geometry tessellation. For information pertaining to the other components, see respective documentation.

2.1 Design Constraints

This project focuses on the development of a terrain system for modern computer games. For this reason, there exist certain constraints that the system must satisfy. The first criterion is that the terrain's visual appearance should be acceptable by modern standards. This means that there should be no obvious artefacts such as holes in the terrain nor sudden changes in geometry or shading. In addition, the rendering quality should be of a high standard. The second, and equally important, constraint requires the system to maintain at least real-time frame-rates. Specifically the target frame-rate is chosen to be 60 frames per second. Although this is much higher than a minimum real-time frame-rate, it leaves only a small amount of time for all other rendering and game subsystems such as physics and AI. A third constraint involves the terrain system itself and regards the target resolution of detail that should be achieved: the system should be able to clearly represent deformations on the scale of footprint outlines. The target hardware for this system is the previous generation of graphics cards. These are the Shader Model 4.x cards such as the NVIDIA GeForce 9 and 200 series cards.

High visual quality and the rendering of high detail deformations each affect the frame-rate constraint considerably and thus make it difficult to satisfy all three constraints simultaneously. Aspects that may challenge the frame-rate include the linear performance drop due to Geometry Shader tessellation, the number of triangles in the raw terrain mesh and texture fetch latency. Non-per-frame operations can also affect the frame-rate or cause jerking in the game. The application of deformations is an example, as well the loading of cached heightmap textures and the generation of their normals. Visual quality concerns include slow texture caching, T-junctions in the irregular mesh and sudden changes in level-of-detail. All of these possible problems are considered within the system design.

2.2 Detail by Mesh Refinement

This research focuses on a tessellation method for representing high-detail deformations. High-detail deformations are applied to the mesh in locations near the camera by first tessellating the triangles to a higher resolution and then displacing them according to the deformation. The system can be divided into two distinct subsystems. The first is that which provides a framework for deformation of the terrain data. The second is the rendering component, providing a method for the display and visual representation of this terrain data. The two subsystems may be changed independently provided the underlying, common terrain data structures remain the same. These are described in Section 2.3.1. In this project, however, these two subsystems were designed with each other in mind so as to allow for easy and efficient compatibility and to satisfy the design constraints in the best manner possible. A conceptual relationship diagram is shown in Figure 2.2.

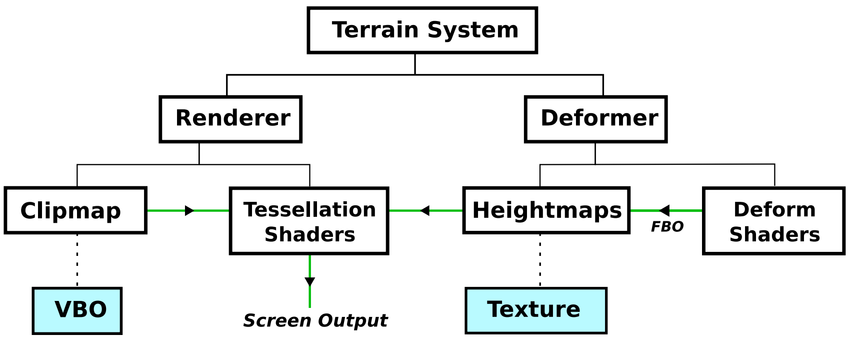


FIGURE 2.2: *An architectural view of the terrain system and its major components. Green arrows indicate the flow of data.*

In the following sections, the design of the terrain is described in greater detail. Section 2.3 covers the use of data structures both for terrain elevation information and the mesh used to represent the terrain during rendering. Section 2.4 details the processes concerning new deformations, caching of height data and rendering. The focus of the design is to create a system that relies as little as possible on transfers across the bus between the CPU and GPU. Most of the data is thus stored on in VRAM and processing of this data is done by the GPU.

2.3 Data Structures

The storage of the terrain data is split into two parts. The terrain elevation data is stored in textures as heightmaps. This means that the terrain representation is 2.5d and does not support any overhangs. The second part of the data storage concerns the presentation, and rendering of the terrain. Here the underlying approximating structure is stored as a flat mesh called a clipmap. The heightmap data is used at render-time, on the GPU, to displace the clipmap and represent the terrain on-screen. More detail follows in the subsections below.

2.3.1 Elevation Data

The stored terrain data consists of elevation values at discrete points. These elevation data are stored in the form of heightmaps in GPU texture memory. Each heightmap has an associated normal map that is used during lighting and displacement, as well as a color texture. There are two sets of textures used to describe the terrain. The first set is referred to as the "coarse-maps" and describe the terrain's elevation on a large scale. These coarse-maps are positioned adjacent to one another in a grid-like formation and can represent a large area due to their relatively low resolution. For the purposes of this project, only one coarse map is being used with the focus being on high-detail. It is accessed toroidally to give the appearance of an infinite terrain. Adjacent texels within the coarse-map map directly to adjacent texels in the finest level of the terrain mesh.

The second set of heightmaps is referred to as "detail-maps" and are used to describe the fine detail, such as footprints, which is the primary focus of this project. The resolution of the detail-maps is dependent on the level to which tessellation can be performed as anything larger would not be noticeable. Thus if the mesh quadrants were divided by a factor of N per side, the detail-maps would have a higher resolution than that of the coarse-map by a factor of N. The detail-maps are positioned in the same grid-like manner so that any arbitrary region can have unique detail. Not every detail-map is be loaded in memory simultaneously, but a caching system instead chooses those closest to the viewer. The detail blends out gradually as it moves away from the viewer. The caching system is described in Section 2.4.2. Covering the entire terrain area with detail-map textures requires considerable memory. The CPU stores a matrix, mapping to the terrain, of texture IDs which identify which detail-maps to use in each section of the terrain.

Depending on the range of height, the bit-depth of the heightmaps need to be larger in order to avoid aliasing. This is certainly relevant for the coarse-map which needs to represent peaks and valleys with reasonable detail. In this case, 16-bit components are used to store elevation data. The detail-maps, however, only represent small deviations from the landscape and only thus only require 8-bit data for a range of 256 values. In addition to the elevation data, each heightmap has a corresponding normal-map texture that is used during the lighting stages. These normal-maps are recalculated after deformation of the heightmap.

Using a heightmap texture as the data structure allows for easy and fast deformations to be performed on the GPU with textures being used as render targets. Both the coarse-maps and detail-maps are deformable. Deformations are applied by changing the intensities of these heightmaps according to predefined or procedural patterns. The deformation process is further discussed in Section 2.4.1.

2.3.2 Mesh Representation

Due to perspective projections used in games, objects become increasingly smaller as their distance from the camera increases and less detail is thus required to represent the same object. This is the primary idea behind the technique of texture mipmaps. The same idea can be applied to geometry meshes where the size of mesh quadrilaterals doubles in size at regular intervals. The clipmap is separated into a number of nested grids. Each successive grid contains quads of double the size and has a hole within it, within which the smaller level can be nested.

Each level contains N vertices along an outer edge. Fitting levels within each other is not trivial and so the method proposed in [AH05] is used to build the clipmap. N is chosen to be $N = 2^k - 1$ where k is an integer. Each level surrounds two sides of the contained level with a line of quads in an L-shape. Square blocks with sides of $M = \frac{n+1}{4}$ vertices are used to surround each corner, whilst *fix-up* blocks of $M \times 3$ vertices fill the centre of each side. Around the edge each level, degenerate triangles are added to join quads of different sizes. This is done to avoid T-junctions, which can lead to obvious artefacts. This layout is shown in figure 2.3. The L-shape alternates sides for each successive level. Although there is some overhead in adding all the degenerate triangles, the hardware is able to recognise and cull them quickly [Spi].

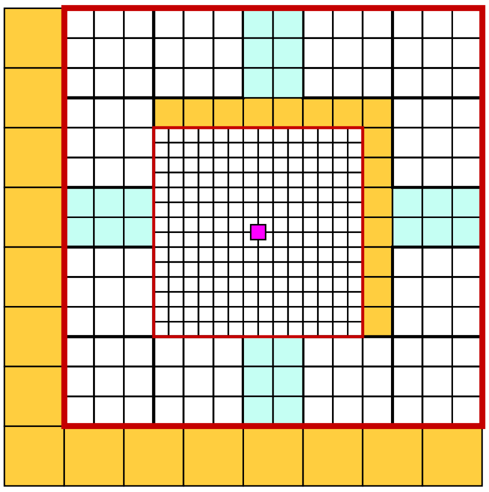


FIGURE 2.3: *This figures illustrates the structure of a 2-level clipmap with $N = 15$ vertices per side and a block of side $M = 4$ vertices. The outer L-shape belongs to the 3^{rd} level. The pink dot represents the camera; orange blocks are the L-shape; blue blocks are fix-up regions ; red lines are degenerate triangles.*

Note that the finest grid is not located at the centre of the clipmap mesh, but this is not noticeable with large values of N. Each vertex has associated texture coordinates to be used for looking up elevation values from the heightmaps. The camera remains centred on the finest level. Rather than shifting the geometry around the camera, during rendering, the camera's location is used to offset the texture coordinates so that the heightmap texture is translated over the clipmap. The vertices of the finest quads sample adjacent texels from the heightmap so that the vertex spacing corresponds to texel spaces. The p^{th} level thus samples every 2^{p-1} texels. Although the clipmap mesh is not translated, it is still rotated about the origin to represent the camera's orientation

In order to present high detail deformations using the detail maps, the finest level is tessellated further in regions where higher detail exists.

2.4 Core Processes

The terrain system has three major areas of processing. The GPU has to render the terrain state every frame and is also responsible for the deformations that could occur any frame. The CPU is responsible for the loading and paging of detail-map textures from and to the hard disk. As with most game systems, everything is controlled by a main looping game thread. Each frame the processes the state of the input devices and responds accordingly. If any deformations are to be made, these are setup and executed by the GPU. A separate thread is create to perform texture paging and the main thread controls their communication. The following three sections cover these processes in depth.

2.4.1 Terrain Deformation

The primary aim of this project is to design a system for computer games in which the terrain can be modified in real-time. These deformations need to occur without noticeably impacting performance and without creating visual artefacts that would detract from the realism and thus break a user's immersion. Section 2.3.1 covered the representation of the terrain's elevation as a heightmap referred to as the coarse-map. This coarse-map is initialised to some default terrain using any arbitrary technique such that the initial state of the environment is realistic. The terrain system also provides data structures to store higher resolution details whose presentation is discussed in Section 2.4.3.

The deformation system thus provides two modes for terrain modification. The first mode allows deformation of the coarse-map which represents the main topology of the terrain. This mode affects the terrain on a large scale in accordance with resolution of the unrefined mesh. The second method provides further control of the terrain, allowing higher resolution modifications. Deformations on this level affect the array of detail-maps and can only be seen by further tessellating the basic terrain mesh. Aside from the maps on which they operate, the two modes differ in a few ways. The first is locality: Because detail-maps must be within a certain range before they are stored in VRAM, as is covered in Section 2.4.2, there is a limit on the distance from the camera at which high detail deformations may be performed. The coarse-map however, is not limited to the same degree and can be deformed as far as possible. The user interface may still limit users on viewable distance however. Secondly, the two types of maps interact differently with the camera. The coarse-map represents the terrain geometry and is frequently required for collision detection. For this reason, when it is modified, the coarse-map must be returned to system memory in order for the CPU to perform collision detection. The detail-maps however, have no physical interactions with the camera and are limited solely to visual feedback. Because there is no collision feedback from high resolution deformations, the maximum amplitude of these displacements should be small enough such that the lack of collision is never noticeable. Their sole purpose is provide high resolution visual detail.

The elevation data for coarse-maps and detail-maps are stored in textures on the GPU. Deformation is performed by using render-to-texture functionality provided by the graphics API. In order to minimise the time taken to render these deformations, the affected region is calculated so that the update may be localised, avoiding unnecessary computation. Once a deformation has occurred the corresponding normal map becomes invalid and must also be updated. The normal map update is also performed in a sub-region of the texture to maximise performance.

The deformation system provides two methods for applying displacements. The first is with the use of predefined textures termed *stamps*. Stamps are regular heightmaps containing grey-scale data, where black indicates a maximum impression and white, a depression. A blend factor can be applied to control the intensity of stamps before their values are added to that of the heightmap. The stamp may be rotated or scaled as is fit for the context. As an alternative to stamps, the system permits the use of procedural stamps for deformation through the use of custom shaders. Rather than simply blending a texture, procedural stamps allow the deformations to be defined programmatically such that their shape may be dependent on any application variables such as time, position and arbitrary values. In order to apply a stamp to the coarse-map or detail maps, the location must be provided in texture coordinates. These may be acquired from the horizontal components of a given position in the environment. In the case of the coarse-map, these coordinates can be used directly, but for high-resolution deformations the detail-map in question must be identified as well as the relative location within this map.

There are a number of issues regarding the consistency of deformation performance. The performance is highly dependent on the size of the region being altered. The time taken increases further when deformations occur on the border between heightmaps. In these cases, separate render-to-texture operations must take place for each heightmap in question. The worst case scenario is when large texture deforms a region that covers the corner between four heightmaps. Detail deformations are limited in size to that of one detail-map.

The functionality provided by this system offers many applications for modern computer games where dynamic terrain would add to the playability and realism. Coarse-map deformations could be used as results of explosions, digging, ploughing. Detail can be used for bullet-holes, footprints or impressions in soft terrain such as snow. The procedural stamps allow variability or even the addition of noise to deformations so that consecutive deformations appear different. Due to the manner in which deformations are stored, that of using textures, no decaying detail is required to save resources. Many computer games remove dead bodies or bullet holes after certain periods of time to save memory and reduce processing. This has an effect of detracting from a game's realism. This deformation system does not remove any detail over time thus maintaining the game's realism.

2.4.2 Caching System

As mentioned in Section 2.3.1, the terrain is divided up into a grid of tiles onto which the detail-maps map. The number of tiles required depends on the resolution of the high detail mesh tessellation, as the texels of the detail-maps are in one-to-one correspondence with tessellated vertices. Consider the example: A terrain consists of a single coarse-map of dimension $N_C = 4096$. 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 $\gamma = 1/3$ yielding a tessellated resolution of $\delta_D = \gamma\delta_C = 0.0\dot{3}$. 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 $N_D = 2048$.

$$\begin{aligned} n &= \frac{\delta_C N_C}{\gamma \delta_C N_D} \\ &= \frac{N_C}{\gamma N_D} \\ &= \frac{(4096)}{(1/3)(2048)} \\ &= 6 \end{aligned} \tag{2.1}$$

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 example illustrates the necessity for a caching scheme.

Because there is only one coarse-map, it is not cached and remains in memory. The caching system stores an array of all the detail-map tiles and their states. A tile's state consists of an *Active* flag and *Loaded* flag. A tile is *loaded* if its texture is currently loaded in GPU memory and is *active* if it is currently visible on-screen. When a tile gets unloaded, the texture's memory is transferred to system memory and immediately written to the hard-disk. The opposite occurs for the loading process. Due to CPU-GPU bus latency, hard-disk read-speeds and system memory copy latency, the loading process is far from immediate. The caching system must therefore issue load requests a reasonable duration before a texture is required. In terms of this caching system and tile states, a tile must thus be loaded well before it becomes active. In order to not stall the game loop, texture caching or paging is performed in a separate thread.

The caching system bases the calculation of load and active states on the camera's location within a tile. The current tile is divided into nine regions. These are formed by creating two bands, one horizontal and the other vertical, spanning the tile across its centre. The band represents a transition region. When the camera sits within the horizontal band, as in Figure 2.4(b), both vertically adjacent tiles must be loaded in memory but neither active. As the camera moves upwards into the top-left corner (Figure 2.4(c)), the bottom adjacent tile is unloaded and the top tile becomes active. In the case of the centre region, where the two bands overlap, all nine adjacent tiles must be loaded. This is the worst case and is shown in Figure 2.4(a). This scheme permits a situation where entering and exiting a region does not result in a unload-load sequence which could cause textures not to be ready due to loading latency. This does however cause a load-unload sequence of events, but this is transparent to the user. The transition period provided by the band can be altered by widening or narrowing the bands.

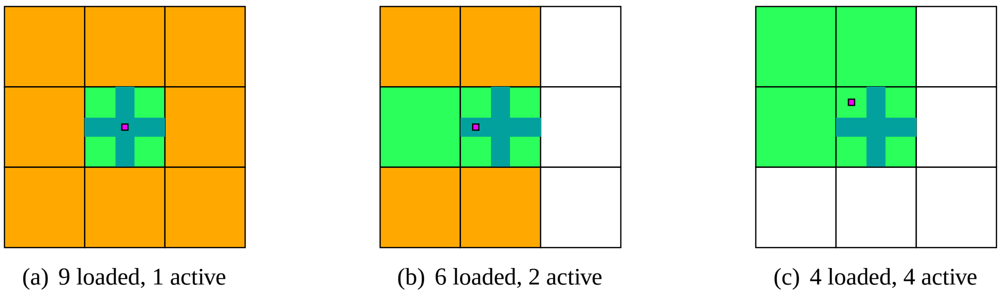

(a) 9 loaded, 1 active (b) 6 loaded, 2 active (c) 4 loaded, 4 active

FIGURE 2.4: *Demonstrates three caching cases for three different regions. Orange tiles represent a loaded but inactive state. Green tiles indicate an active and loaded state.*

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. Another optimization is that textures are not cached to disk unless they have been modified since they were loaded. To save time spent waiting hard-disk requests and bus transfers, normal maps are not cached but are rather recalculated each time a texture is loaded. A final note about the memory usage is that it can be further minimised by making use of compressed texture formats on the GPU.

2.4.3 Rendering Process

This process is responsible for presenting the terrain and environment to the user. This is the most important section as it must satisfy both the frame-rate and visual realism constraints. These two constraints have an inverse effect on one another and thus require the process to include enough optimization so that both criteria may be sufficiently met. The rendering pipeline also includes the process of refining or tessellating regions of higher detail - the primary focus of this research.

A number of simple and common graphics techniques are employed to render the terrain. As mentioned in the previous section on Data Structures, the terrain's mesh is represented using a geometry clipmap with vertical displacements defined by a heightmap. Rendering this clipmap alone produces acceptable detail for a regular terrain and allows for the presentation of coarse-level deformations. The aim of this research, however, is high-detail deformations such as that of footprints and imprints on the scale of tenths of a metre. In order to render these high-resolution displacements the terrain needs to be tessellated to a higher degree. This process refines input triangles into a set of sub-triangles which then sample the detail-maps. Diffuse Lighting is applied to the triangles using the generated normal maps to add to the realism and to improve the visibility of deformations. The terrain's colour is sampled from its colour-map and combined with this diffusive factor.

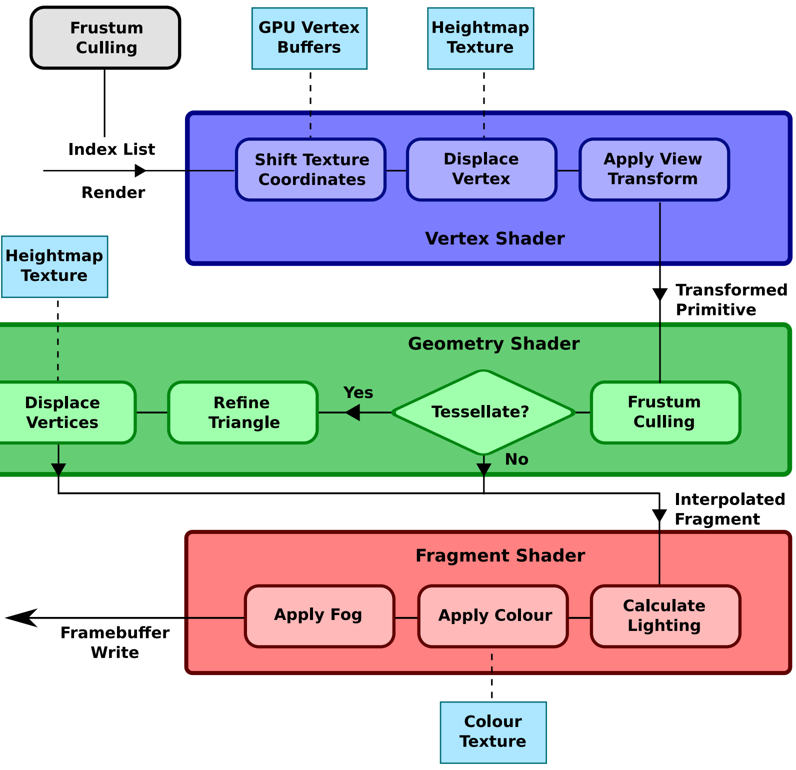


FIGURE 2.5: *The above diagram illustrates the main components of the rendering pipeline and what parts are handled by each shader*

The refinement process tessellates a triangle into many sub-triangles causing the total number of rendered triangles to be considerably larger. This refinement is performed in the Geometry Shader. The performance drop is roughly linear with respect to the number of attributes emitted [NVI08]. For this reason, tessellation should be limited to the smallest number of triangles possible. Because high-detail would not be visible from far, the refinement only takes place in regions within the finest clipmap level and a certain locality to the camera. These details are blended out as they move into the distance. Local refinement is implemented adaptively within the Geometry Shader where full tessellation only occurs if there is high-detail associated with all three triangle vertices. Although the clipmap geometry is indexed and stored on the GPU, the vast number of vertices still takes a relatively long time to render - most of which are not visible. In order to reduce the input to the Vertex Shader, and in turn the Geometry Shader, frustum culling is implemented on the CPU to omit non-visible triangles. Figure 2.5 illustrates the render process.

Frustum Culling

As discussed in Section 2.3.2, the clipmap is split up into blocks of $M \times M$ vertices during creation. All triangles in the clipmap are stored on the GPU as a list of indices. The blocks are recorded along with the range of indices that cover the block and the corner vertices. Each frame, the peripheral vertices marking the bounds of each block are transformed using the camera's view transform. If any of the block's corners fall within the view frustum, the block's indices are appended to list of render-ready indices. Any block that falls out of the frustum completely is omitted from rendering. The $M \times 3$

vertex fix-up regions are also considered as blocks, but the L-shapes, degenerate triangles and the finest clipmap level are not checked and are rendered every frame regardless as they constitute a relatively small percentage of the triangles. This frustum culling result is illustrated in Figure 2.6 in comparison to Figure 2.3 in Section 2.3.2.

Although the pre-render frustum culling removes a considerable amount of work from the Vertex Shader, it is imperative that the minimum number of triangles be processed by the Geometry Shader so as to limit the amount of expensive tessellation. Per-triangle frustum culling is thus performed by the Geometry Shader allowing it to reject non-visible triangles before doing any further processing.

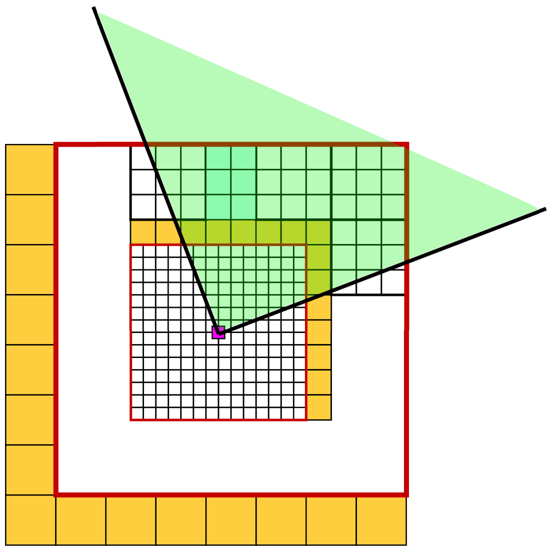


FIGURE 2.6: *$M \times M$ and $M \times 3$ blocks are culled on the CPU before the render call. This clipmap is at a low resolution so as to make it more visible.*

Vertex Shader

The primary job of the Vertex Shader is the displacement of the heightmap. As input, the Vertex Shader receives the raw clipmap un-elevated vertices along with their texture coordinates. Because the clipmap remains centred on the camera, motion is simulated by translating the texture over the clipmap. The Vertex Shader shifts the texture coordinates of the vertex according to the location of the camera, followed sample the elevation value from the heightmap use the translated texture coordinate. After applying the elevation to the vertex, it is translated and rotated according to the camera's height and orientation. The slowest part of the Vertex Shader is the texel fetch from the heightmap. Vertex attributes and shader variables are packed together into fewer data types to maximize performance.

Geometry Shader

The next phase of the pipeline is the Geometry Shader which performs all the per-triangle computation. In the general case, this simply involves testing the triangle against the view frustum. Failure of this test results in culling otherwise the triangle is passed to the next stage. Triangles contained within the finest clipmap level, however, may require further refinement depending on the existence of high-detail deformation data. In the case of these fine triangles, the Geometry Shader then tests whether refinement is necessary and, if so, subdivides it according to a refinement pattern. Triangles on the edge of a tessellated region needs to merge safely with the coarser mesh in order to avoid T-junctions which can cause artefacts. Refinement patterns define exactly how a triangle will be tessellated into sub-triangles. Each pattern is defined as a set of barycentric coordinates which is a general representation allowing the pattern to be applied to an arbitrary triangle.

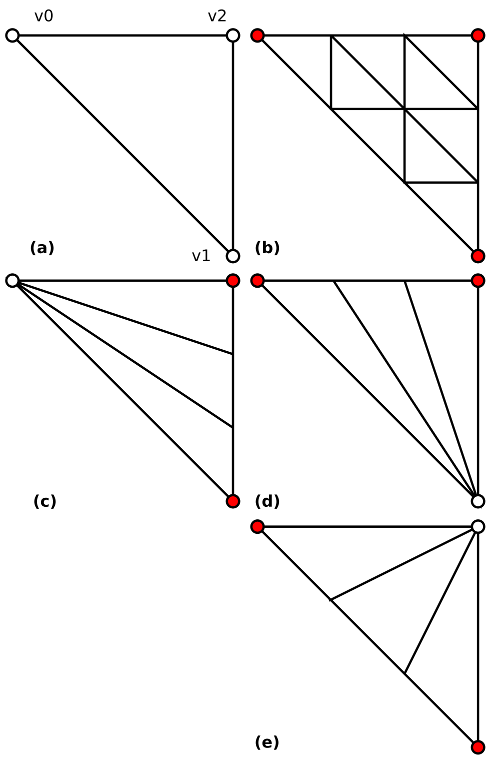


FIGURE 2.7: *The 5 different refinement patterns for different values of ρ. The red dots signify vertices set to tessellate. (a) no tessellation $\rho = 0, 1, 2, 4$. (b) full tessellation $\rho = 7$. (c) $\rho = 6$, (d) $\rho = 5$, (e) $\rho = 3$ one side tessellated*

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. Four of these result in no tessellation - the cases of only one *on-vertex* and zero *on-vertices*. 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.

$$\rho = t_0 + (t_1 << 1) + (t_2 << 2) \tag{2.2}$$

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, $\rho = 0, 1, 2, 4$ are the values resulting in no tessellation. Once tessellated, new vertices can then sample the relevant detail maps and can be offset in the direction of the heightmap's normal. The new composite normal must be calculated using the heightmap and detail map's normals. The vertices are transformed by the view and projection matrices and then sent to the next stage in the pipeline. 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. It is important that the mesh triangles be indexed in a consistent fashion such that the refinement patterns. using barycentric coordinates, match the indices correctly. The vertices are indexed in a counter-clockwise winding order to indicate their front-facing direction. The first two indices always describe the diagonal edge. The indexing and different refinement patterns can be seen in Figure 2.7. Figure 2.8 gives an example of a section of mesh that has been tessellated.

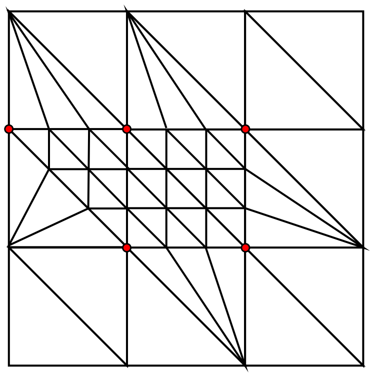


FIGURE 2.8: *A tessellated region of the mesh. Red dots indicate vertices set to tessellate.*

Fragment Shader
The Fragment Shader is the simplest of the steps in the pipeline. It samples the terrain fragment's colour and normal from textures. The normal is used in a diffuse lighting calculation to scale the colour value. This is then output to the framebuffer. The two texture fetches in this shader are the slowest. The culling in the Geometry Shader reduces the number of fragments needing to sample these textures.

2.5 Summary

This project focuses on the creation of deformable terrain on both a large and fine scale. The large-scale terrain detail is presented using displacement heightmaps to offset vertices of the base mesh. The base mesh is composed of nested grids of differing levels-of-detail; a technique known as Geometry Clipmaps. Fine deformations are presented by refining or tessellating the base mesh to higher resolution and displacing the new vertices. Deformations are applied to either the coarse-map or detail-maps by two methods. One method uses predefined texture heightmaps called stamps, adding their values to the existing heightmap. Another method allows procedural shader programs to create a deformation pattern on-the-fly. All deformation, displacements and tessellations are performed on the GPU. In order to minimise the amount of memory required to store the high detail, a tile-based texture caching scheme is used that pages textures to the hard disk if they are far enough from the camera. This system has many applications for computer games. Examples include making impressions in soft, snowy terrains, creating craters from large explosions, simulating erosion or creating deep tyre tracks or footprints.

Chapter 3

Implementation

The terrain system is designed for the purposes of games with dynamic terrain. This project is developed to run on both Windows and Linux systems. It is developed in C++ and makes use of OpenGL 3.2 API with the GLSL 1.5 shading language. SDL 1.3 is used for the windowing system and FreeImage is used for reading and writing PNG images from and to the hard disk.

This chapter covers the implementation of the system in detail. Section 3.1 covers how heightmaps, normal maps and the geometry clipmap are created and stored in GPU memory. Section 3.2 details the processes of deformation, generation of partial derivative normal maps and caching. It also covers the program flow, handling of collisions with terrain, physics system and the process of rendering everything on screen. Finally, Section 3.4 illustrates three examples of different types of deformation stamps.

3.1 Data Storage

Section 2.3 covered the design of data structures used. This section details the implementation of these data structures. Specifically, it covers the texture dimensions and internal formats used to store heightmaps and normal maps and the different VBOs used to store the clipmap vertex data.

3.1.1 Heightmaps

Section 2.3.1 introduced the data structures for storing elevation data of both coarse-maps and detail-maps. These data are stored as textures in VRAM. Representing the elevation data requires a grey-scale texture image where the intensity of each texel denotes the height or elevation at that location. In terms of OpenGL textures, each heightmap maps to a `GL_TEXTURE_2D` with only a single colour channel, `GL_RED`. Heightmaps are commonly 8-bit, providing 256 possible levels of elevation. For terrains of decent detail and large height variance, this is not a sufficient level of accuracy. For this reason, textures with a bit-depth of 16-bits are used, yielding 65536 possible height values. This is very important as it allows very high peaks, whilst still providing finer detail on valley floors. For example, if the maximum height was 5km, a 16-bit range would allow for height deviations of approximately 80mm, whilst an 8-bit range would yield highly aliased deviations of 19.5m. In the case of detail-maps, however, 8-bit textures are sufficient as detail-maps only represent small deviations in the surface. The range of these maps is a maximum of 1m, yielding a perfectly acceptable resolution of 4mm.

The choice of dimension of the coarse-map is co-dependent on the desired distance it should span as well as the desired metre resolution. If a grid of multiple coarse-maps had instead been implemented, the dimension of the coarse-maps would have only depended on the desired resolution. For simplicity's sake, heightmaps are stored using raw texture formats. Coarse-maps use the `GL_R16` internal texture format consisting of `GL_UNSIGNED_SHORT` data and the detail-maps use `GL_R8` and `GL_UNSIGNED_BYTE` respectively. Because there is only one coarse-map, it must be accessed toroidally so as to give an impression of an infinite terrain. The texture address-mode (or wrap-mode) is therefore set to `GL_REPEAT` for both coordinates s and t, so that out-of-bound texture accesses wrap to the other side of the texture. Detail-maps, on the other hand, are organised in a grid formation and should not wrap. To prevent wrapping, their address-mode is set to `GL_CLAMP_TO_EDGE` which treats out-of-bound texture accesses as if they queried the closest edge.

Mipmaps are created for the heightmaps, because textures are required to be Mipmap-Complete by OpenGL FBOs which are used during deformation and explained in Section 3.2.1. In addition to this the Heads-Up-Display (HUD) contains a mini-map that uses the coarse-map texture. Because of the small size of the mini-map, mipmaps improve rendering time (more cache hits) and reduce artefacts caused by scaling. The caching system pages detail-map textures to the hard disk. To allow for easy cache viewing, the popular Portable Network Graphics (PNG) image type is used with an 8-bit format. The coarse-map is stored as a 16-bit grey-scale PNG.

3.1.2 Partial Derivative Normal Maps

In order to perform lighting calculations on the terrain, normal vectors are required at each elevation point. In addition to this, when deforming the tessellated coarse terrain with high resolution detail, normals are needed for the direction of displacement. Traditional normal maps are stored with the x, y and z components of a normal vector in the red, green and blue colour channels of the texture. A shader will typically read these in, re-normalize the vector and then use the normal in a lighting calculation. Unfortunately, this requires three components per normal which is three times that of the heightmaps. An alternative to traditional normal maps is that of Partial Derivative Normal Maps. These textures require only two components which reduces both memory usage and texture fetch latency. The details of the usage and theory of partial derivative maps is covered in Section 3.2.2. The textures are bound as `GL_TEXTURE_2D` OpenGL textures in VRAM with two components of 8-bit accuracy each such that the internal format and data type are `GL_RG8` and `GL_UNSIGNED_BYTE` respectively. As the with the corresponding heightmap, coarse-map partial derivative maps are accessed toroidally and thus have a wrap-mode of `GL_REPEAT` whilst that of the detail-maps is set to `GL_CLAMP_TO_EDGE`. Unlike the heightmaps, the partial derivative maps are not paged to the hard disk. Instead, they are recalculated each time a heightmap is loaded onto the GPU.

3.1.3 Geometry Clipmap

The representation of the terrain mesh as a set of geometry clipmap levels was covered in Section 2.3.2. To avoid confusion with foot deformations, the *footprints* mentioned in the original paper [AH05] will be referred to as *blocks*. Every clipmap level can be made up of four distinct set of vertices. For each of these, a VBO is created to store the vertices on the GPU. The first set comprises the vertices that form degenerate triangles around the inner level. The second contains the L-shape around the inner level. Thirdly, there are fix-up blocks at the centre of each side of the inner layer. The final set is the $M \times M$

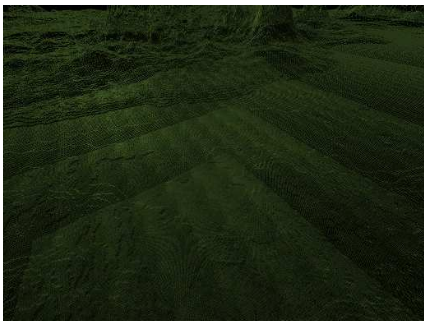


FIGURE 3.1: *Shows the displaced clipmap rendered in the terrain system*

block (footprint), that repeats 12 times in a level. These sets are the same for each level up to a simple scale and translation and they are thus stored only once. This is the method used in [AH05]. The process of rendering these VBOs is covered in 3.2.4. Storing only one $M \times M$ block and only one level's worth of the other patterns, rather than the entire final mesh, saves a considerable amount of memory. An additional VBO must be used to store the finest level grid at the centre of the mesh. Equation 3.1 shows the calculation of vertex count for a clipmap of with N vertices per side.

$$\begin{aligned} V &= \overbrace{M^2}^{block} + \overbrace{4 \times (3 \times M)}^{fix-ups} + \overbrace{2 \times (N+1)}^{L-shape} + \overbrace{4 \times N - 3}^{degenerates} + \overbrace{N^2}^{inner\ grid} \\ &= M^2 + 12M + N^2 + 6N - 1 \end{aligned} \tag{3.1}$$

For a clipmap with $N = 255$ and thus $M = 64$, the vertex count comes to $V = 71418$ for an arbitrary number of levels, which is considerably lower than the almost 2-million that there would be for a 5-level clipmap. Each vertex in the VBO has two components for x and z. No y component is needed as this is acquired by the shader when reading the heightmap. In addition to this VBO, there is a VBO containing 2-component `GL_FLOAT` texture coordinates that correspond to the vertices. Each of the VBO sets also have an associated index buffer that is used to specify a triangle strip so as to make use of the post-transform vertex cache with little effort. Figure 3.2 illustrates the triangle strip used for the $M \times M$ blocks.

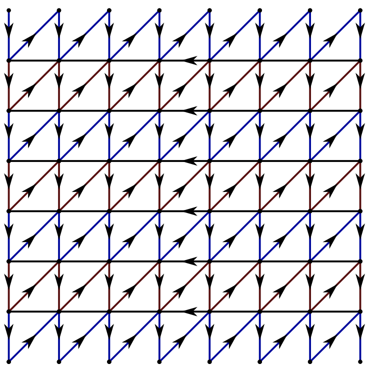


FIGURE 3.2: *Illustrates the indexing of the triangle strip for $M \times M$ blocks with $M = 8$*

3.2 Core Processes

The system be broken up into a number of major sub-systems. Firstly,
the deformation system is responsible for creating modifications in terrain elevation data and may be executed in any frame multiple times. The generation of partial derivative normal maps is also handled by this deformation system. The caching system is responsible for ensuring that the correct detail-maps are loaded in VRAM if and only if they are currently, or may soon be, needed for rendering. This system does processing every frame. The rendering system presents the current state of the terrain, including high-detail local to the camera, every frame. Physics and collision detection give maintain the realism

by performing realistic interaction with the terrain as the camera moves relative to it. The execution of these sub-systems is covered in greater detail in the subsections below.

3.2.1 Deformation

The constraints for deformations, as explained in Section 2.4.1, require deformations to occur without affecting performance. The time between issuing the deformation and noticing a response needs to be minimal. In addition to this, the system maintains all deformations for the duration of the application, or even between executions of the application. This persistence of terrain modification helps increase realism in computer games. Because terrain detail is stored in textures, adding extra detail to the coarse-map does not increase memory usage. The same is true for non-zero detail-maps. Another requirement is that the CPU requires feedback from the coarse-map after a deformation has occurred, in order to perform collision detection.

Heightmap deformations are performed using a render-to-texture operations to modify the texture. The deformation system makes use of a single OpenGL FBO to perform deformations to any of the heightmaps. Results of render-to-texture operations are undefined when reading from and writing to the same region of the same texture. For this reason the deformation system maintains two textures to be used as double-buffers. One of these is used for coarse-map deformations and the other for detail-maps, each set up with the appropriate internal format and dimensions. This unfortunately requires a copy operation to backup the current state of a heightmap to the double buffer before performing the deformation. Ideally a backup could be kept for each texture so that the copy step could be omitted and a ping-pong technique could instead be used.

Initially, the system generates the framebuffer and backup textures and initializes their formats, and dimensions. Deformations are performed using stamps which are also setup during this initialisation phase along with their associated shaders. The different types of stamps are covered in Section 3.4 accompanied by three examples. A VAO is also created to contain a single VBO representing the quad that will be used to apply stamps to regions of the heightmap during the render-to-texture operation. The deformation process can be divided into three stages.

The first stage involves the double-buffering. First the region of interest is computed. This is the region within the detail-map texture to which deformation must be applied. This region is controlled by scale and position parameters. The current state of the heightmap in question, within this computed region, is then copied to the buffer texture so that it may be sampled during the deformation render. The copy is performed using the `glCopyTexSubImage2D` command. This command requires an FBO to bound as the `GL_READ_FRAMEBUFFER` with the current source heightmap as the read-target. The same FBO that will be used for the deformation render is used here. To ensure the texture is mipmap-complete, the mipmaps are then recomputed using `glGenerateMipmap`.

The second stage is where the deformation is performed. The FBO is now bound to `GL_DRAW_FRAMEBUFFER`. The heightmap is bound as the draw target to `GL_COLOR_ATTACHMENT0` and the backup texture is bound to the first texture image unit. The appropriate shader program is then enabled and has its uniform variables set. For regular stamps this will be the generic stamp shader, but procedural stamps may opt for custom shaders. Although procedural stamps have a lot of flexibility in how the render, they are still confined to the region defined by the scale and position parameters. Once all necessary setup for the render-to-texture operation has completed, the `glDrawArrays` call is made. This will render a quad over the chosen region of the heightmap. In the general case it will perform a clamped additive blend between the backup and stamp, writing the result into the heightmap.

The final stage consists post-render cleanup operations. Apart from unbinding buffers and textures and resetting the viewport, it is important that the mipmaps be regenerated as they will be needed in future deformations as well as when rendering a mini-map in the HUD. In the case of the coarse-map, the changed region must be copied back into its backup texture in the same way it was done for detail-maps before the render. This step can be easily avoided if ping-ponging is instead used. Finally after the deformation stages have completed, the partial derivative normal map will need to be updated according to the new height-field. This is also done only within the localised region of modification. The generation of the partial derivative map is a simplified version of the above mentioned stages, but does not require a backup texture as new normal maps are not based on previous normal maps.

In the general case this is all that need be done to apply a deformation. If, however, the region of modification overlaps a heightmap boundary more computation is required. In the case of the coarse-map, deforming over a corner requires a deformation to be performed four times for each of the corners as the texture is wrapped on both axes as if it represented the surface of a torus. If it it only overlaps one of the edges, only two deform operations must be made. The normal maps need to be updated for each of these calls as well. Boundary deformations for detail-maps are currently not handled by the system. A possible improvement to this implementation of coarse-map boundary deformations may be to use a geometry shader to instance a quad for each operation needed. This way no extra render calls are required. If this is combined with ping-ponging of the backup buffer, the coarse-map would also require no extra copy commands.

Collision Detection requires the current state of the coarse-map in order for the player's interaction with the terrain to be in accordance with what is visible. For this reason it is essential that the deformed map be available to the CPU shortly after the operation has been issued. In the case where many deformations are made one after the other in a short period of time, transferring after each command would cause considerable overhead. For this reason, the system waits for a gap of approximately 20 milliseconds before it commences the process of streaming the coarse-map. This method assumes that deformations do not occur anywhere that will affect the player's current position. If this does happen, the player will momentarily be underground. The CPU can account for this using collision heuristics that have prior knowledge regarding the geometry of the deformation, but for arbitrary deformations the system must wait for the streamed data.

The collision detection system uses a floating-point array in system memory to represent the height-field. It is this array that need be updated with the new heightmap data. The array is protected by a mutex so that a separate thread may perform the streaming. The process makes use of a PBO to transfer the coarse-map data. This is done so that the GPU and CPU do not waste cycles on controlling the transfer, but instead pass this responsibility on to the DMA. The coarse-map data is transferred, using a dedicated FBO, into the PBO by the main thread. The main thread then continues with its usual processing. A separate thread maps the PBO to a system memory pointer using `glMapBuffer`, an operation which blocks until the data is ready to be mapped. Once mapped, the data is copied from the PBO's memory into a inactive floating-point array in system memory. A lock is then acquired on the mutex for the collision array. The pointer to the collision array is swapped with that of the inactive array containing the new data. The mutex lock is then released. Because the texture contains `GL_UNSIGNED_SHORT` data, it must be converted to floating-point data during the copy. Using floating-point textures could speed this process up.

3.2.2 Generation of Partial Derivative Normal Maps

Section 3.1.2 introduced Partial Derivatives Normal Maps [Act08] as an alternative to traditional Normal Maps. The primary benefit of these maps is that they only require two components to store. This benefit does come with a cost, however, that the full range of normals cannot be reproduced during the reconstruction phase as is shown in this section. The definition of a normal states that it is the cross-product of a surface's tangent and binormal vectors. Given a point-vector **r** on a height-field field $h(u, v)$ the normal $\hat{\mathbf{N}}$ can be derived from the height-field, by calculating the tangent **t** and binormal **b**, as follows:

$$\begin{aligned}
\mathbf{r} &= \mathbf{f}(u, v) = (u,\ h(u, v),\ v) \\
\mathbf{b} &= \frac{d\mathbf{r}}{du} = \left(1,\ \frac{\partial h}{\partial u},\ 0\right) \\
\mathbf{t} &= \frac{d\mathbf{r}}{dv} = \left(0,\ \frac{\partial h}{\partial v},\ 1\right) \\
\mathbf{N} &= \mathbf{t} \times \mathbf{b} = \left(-\frac{\partial h}{\partial u},\ 1,\ -\frac{\partial h}{\partial v}\right) \\
\hat{\mathbf{N}} &= \mathbf{N}/|\mathbf{N}|
\end{aligned} \tag{3.2}$$

Traditional normal maps would store the x, y and z components of $\hat{\mathbf{N}}$. The idea of partial derivatives normal maps is to only store the $\hat{N}_x/\hat{N}_y$ and $\hat{N}_z/\hat{N}_y$ components, which are the partial derivatives of the unit normal. This project uses finite difference equation approximate the partial derivatives of the heightmap $\frac{\partial h}{\partial u}$ and $\frac{\partial h}{\partial v}$. Specifically a finite difference equation of error order $O(\xi^4)$ is used, where ξ is the horizontal distance between height samples. It can be derived using Taylor expansions.

$$\begin{aligned}
h(x+\xi) &= h(x) + \xi h'(x) + \tfrac{\xi^2}{2}h''(x) + \tfrac{\xi^3}{6}h'''(x) + O(\xi^4) \\
h(x-\xi) &= h(x) - \xi h'(x) + \tfrac{\xi^2}{2}h''(x) - \tfrac{\xi^3}{6}h'''(x) + O(\xi^4) \\
h(x+2\xi) &= h(x) + 2\xi h'(x) + 2\xi^2 h''(x) + \tfrac{8\xi^3}{6}h'''(x) + O(\xi^4) \\
h(x-2\xi) &= h(x) - 2\xi h'(x) + 2\xi^2 h''(x) - \tfrac{8\xi^3}{6}h'''(x) + O(\xi^4)
\end{aligned}$$

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

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

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

$$\begin{aligned}
\frac{2\alpha+16}{6} &= 0 \\
\implies \alpha &= -8
\end{aligned}$$

After eliminating these terms, the finite difference equation 3.3 is acquired by dividing through by -12ξ. This equation yields an error on the order of $O(\xi^4)$.

$$h'(x) = \frac{h(x+2\xi) - 8h(x+\xi) + 8h(x-\xi) - h(x-2\xi)}{-12\xi} + O(\xi^4) \tag{3.3}$$

Although this method requires an extra four texture fetches, compared with usual $O(\xi^2)$ error method, to achieve the smooth and more accurate derivative approximations, these fetches are in close proximity and should thus benefit from the texture cache. During rendering, the shader must reconstruct the normal from this map. This commonly done by creating a vector $(\frac{\partial h}{\partial u},\ 1,\ \frac{\partial h}{\partial v})$ and then normalizing it. This terrain system also employs this method. It is important to note that this technique limits the range of normals to a 45^o cone around the vertical and thus flat normals cannot be represented. This limitation is acceptable, as few terrains exhibit very sharp inclines. The range can be changed, at the sacrifice of quality, by changing the default $N_y = 1$ to a lower value. Another benefit of using this technique is that during implementations of detail algorithms, such as bump-mapping, the formation of a tangent space can avoided. Instead, the composite normal can be created by simply adding the two sets of partial derivatives before normalizing.

3.2.3 Caching

Minimising memory usage is an essential goal of any GPU technique. Because the deformation system covers the playing area in detail-map textures, this is difficult. Texture are therefore cached such that only the necessary textures are kept in VRAM at any point in time. This set of textures is known as the *Working Set*. Section 2.4.2 covers the design of the caching system. This working set is determined by the location of the camera within the current tile. There are nine regions within a tile, formed by a horizontal and vertical band which cross at the centre. The bands denote regions of transitions where adjacent tiles are loaded on both sides of the current tile, but neither are active. The regions were covered in more detail in Section 2.4.2 and illustrated in Figure 2.4.

The primary concern regarding caching is the latency experienced when transferring texture data across the CPU-GPU bus. The purpose of the band-regions is to ensure textures are loaded before they are required. In addition to the bus latency, copying system memory between locations is slow as well as hard-disk writes and reads. These factors all contribute to the overall latency experienced between requesting a texture and it becoming available. A second thread is therefore used to perform the hard-disk and system memory transfers. The main thread still must coordinate the GPU-CPU transfers, as an OpenGL context is not shared across threads. In order to minimise GPU and CPU cycles needed to transfer the data, PBOs are used to invoke the DMA to perform the transfers.

An entity termed *CacheRequest* is used to encapsulate the state and detail of a texture tile transfer. A load or unload request will create a `CacheRequest` struct for the relevant tile and push onto a queue for processing. Every frame, the caching system checks each of the queues and handles the cache requests accordingly. Two pools of four PBOs are created initially to be used during the loading and unloading processes. In addition to this, a pool of texture IDs is created into which loaded textures can be read. If there is no data to load for a specific tile, then the tile shares the zero-texture's IDs. The CacheRequest struct is shown in Listing 3.1.

LISTING 3.1: *Shows the struct used to control load or unload requests for tile detail-maps from the cache.*

```
enum REQUEST_TYPE { LOAD, UNLOAD };
struct CacheRequest
{
    REQUEST_TYPE type;
    GLuint       pbo;
    GLubyte*     ptr;
    Tile*        tile;
    bool         useZero;
    int          waitCount;
    int          cycles;
};
```

When a load `CacheRequest` is created, the tile is assigned and the type is set. It is then pushed into the `readyLoadQueue` where it waits for an `UNPACK` PBO to be assigned to it. Once the main thread has assigned a PBO to the request, `glMapBuffer` is called to map the PBO to system memory into which the image can be loaded. This memory pointer is assigned to the request's `ptr` attribute and the request is pushed onto the `loadQueue`. The helper thread repeatedly checks the `loadQueue` for requests. When a request is popped from the queue, the filename is determined using the tile's row and column and the image is loaded from the PNG file using the *FreeImage* Library. This data is then copied into the PBO's memory pointed to be `ptr`. If the file did not exist, the `useZero` flag of the request is set to `true` to instruct the main thread to let this tile share the zero texture. The load request is then push onto the `doneLoadQueue`. The two queues, load and done, used by both threads, are protected by mutexes to prevent concurrent access. The caching system checks the `doneLoadQueue` every frame for uploading requests. When it pops a request from the queue, it tries to assign a texture ID from the pool. If this fails, it waits till the next frame for a texture ID to be released by an unload request. There are enough texture IDs to store a full working set. Once it has a texture ID for the request, the PBO is unmapped and the data upload commences by issuing the `glCopyTexImage` command. This command should not block and the thread can continue. The PBO is placed back in the pool. The request remains in the queue, with its `cycles` attribute being incremented each frame to allow for the upload to complete. Once the cycle count reaches a predefined constant, the mipmaps are generated for the texture using `glGenerateMipmap` and the partial derivative normal map is calculated. The texture is then ready for use.

A similar process occurs during an unload operation. The primary difference is that the transfer commences before the cache request is passed to the helper thread who writes the texture to disk. Unload requests make use of `PACK` PBOs to transfer data to system memory. The main goals in minimising the latency are to process hard disk and system memory operations as fast as possible and to not let the main thread block.

3.2.4 Rendering

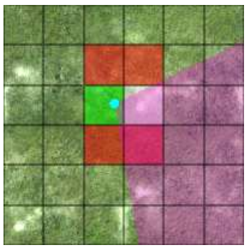
For the purposes of testing the terrain system, the scene is very simple, comprising the terrain, a skybox and the mini-map in the HUD. A perspective projection is used with a 90^o vertical field-of-view and with the horizontal field-of-view adjusted according to the aspect ratio. A basic math library was developed to handle matrix and vector operations during rendering.

The first mesh to be rendered is the skybox. This consists of a quad covering the ceiling, and four walls with a partly cloudy blue sky texture. The sky box does not change position, but remains with the view at the centre and rotates around it. The terrain mesh is rendered next. Before the rendering begins, frustum culling takes place to choose which blocks of triangles are to be rendered. The coarse-map and its associated normal map are then bound to texture image units. The camera's rotation is set as the view transform. No translation is given, because the mesh remains centred about the viewer. The camera's position is passed to the shader, however, and is used to offset the texture coordinates so that the height-field is shifted across the mesh. When high-detail rendering is disabled, the mesh is rendered by a single set of shader programs. The render call first renders the inner grid. Then for each layer it renders the L-shape and fix-up regions. The $M \times M$ blocks are rendered using instancing, and look-up their shift and scale parameters from a uniform array using their `gl_InstanceID` as an index. The $M \times M$ blocks are rendered using instancing, and look-up their shift and scale parameters from a uniform array using the `gl_InstanceID` as an index. The vertex shader performs the displacement of the coarse-map. The geometry shader then does post-transform culling of triangles. Finally, the fragment shader performs lighting calculations. Here the normal is reconstructed from the partial derivative normal map as covered in 3.2.2. Using the normal, diffuse lighting is applied according to the dot product of the light vector and normal. Although a terrain would typically be too rough, a weak specular is also applied as a test on performance using the Phong lighting model. The lighting intensity is used to modulate the colour read from the terrain colour texture before writing it to the framebuffer.

In the case when high-detail is enabled, tessellation needs to be performed. Here, the VBO for the inner grid is rendered separately from the outer layers and uses a different shader. The outer layers are rendered in the manner as the aforementioned coarse-detail rendering case. High-detail is only rendered for the inner grid near the camera. Four detail-maps and their associated normal maps are loaded into texture image units for processing by the shader. The Geometry Shader performs triangle refinement according to the patterns discussed in Section 2.4.3. Adaptive tessellation is not implemented, however, and the triangles of the inner grid within a certain radius are all tessellated into nine sub-triangles. New vertices calculate the tile in which they are located and fetch the displacement from the corresponding detail-map. This value is scaled according to detail heights and then added to the y-coordinate. The normal must then be constructed from detail-maps associated normal map combined with the base mesh's normal. Usually this would require tangent-space conversions, but can be easily computed with the use of partial derivatives. Given the coarse-map partial derivatives $\nabla h(u,\ v)$ and the those of the detail-map $\nabla D(u,\ v)$, the composite normal can be acquired using Equation 3.4.

$$\begin{aligned}\mathbf{N} &= \left(\frac{\partial h}{\partial u} + \frac{\partial D}{\partial u},\ 1,\ \frac{\partial h}{\partial v} + \frac{\partial D}{\partial v}\right) \\ \hat{\mathbf{N}} &= \mathbf{N}/|N|\end{aligned} \tag{3.4}$$

This process is equivalent to adding the two heightmaps together and calculating the resulting normal. The fragment shader is the same for high-detail and works on these computed normals.


(a) Full map mini-map

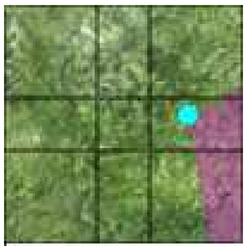

(b) Region mini-map

FIGURE 3.3: *Shows the two mini-maps used for caching visualisation*

After the terrain is rendered, the mini-map is rendered by the caching system. The purpose of the mini-map is to give feedback regarding the loaded and active detail-map tiles. A map is rendered using a quad in four passes. The first pass draws the colour texture map modulated by the coarse-map. The second pass fills tiles with translucent colours to represent their current state. Green indicates the current tile; white, loaded and active tiles; red, loaded but inactive. A third pass draws the projection of the view frustum in a transparent magenta. A final pass overlays grid-lines. A smaller mini-map is rendered below illustrating the camera's region within a tile. These maps help to visualise what the caching system is doing.

3.2.5 Collision Detection and Physics

The camera is governed by simple Newtonian physics. Every frame, user input contributes to the forces acting on the camera. During the logic step these forces are integrated over the time-step using Eulerintegration and are added to the velocity of the camera. This velocity is in turn integrated over time to update the camera's position. In addition to user-applied forces, the environment imposes a gravitational force, which can be toggled off and on. When in motion, frictional forces are applied to the camera. These frictional forces are proportional in magnitude to the velocity but opposite in direction.

After the position has been updated, the horizontal location is used to lookup the height of the terrain at that point. If the vertical position of the camera is below this value, the camera is offset to the height of the terrain. This height-field array must be updated each time a deformation occurs. The streaming of elevation data, for collision, is discussed in Section 3.2.1. This collision response method does not alter velocity in any way.

LISTING 3.2: *Bilinear interpolation of the surrounding heights*

```
int   x0 = int(p.x);
int   z0 = int(p.z);
float fx = p.x - x0;
float fz = p.z - z0;

// wrap out-of-bounds indexing
int x1  = x0 < N - 1 ? x0 + 1 : 0;
int z1  = z0 < N - 1 ? z0 + 1 : 0;

// interpolate in X
float top = (1-fx) * h[x0  + N * (z0)   ]
          + (  fx) * h[x1  + N * (z0)   ];
float bot = (1-fx) * h[x0  + N * (z1)   ]
          + (  fx) * h[x1  + N * (z1)   ];

// result from interpolation in Z
float height = (1-fz) * top + fz * bot + EYE_HEIGHT;
```

Listing 3.2 shows the bilinear interpolation of height values used to calculate terrain height at a given x and z. N represents the dimensions of the elevation data array h and p is the camera's position.

3.3 System Parameters

This section covers the choice of parameters used during the development of the terrain system. The scale is chosen with 1 unit to be equivalent to 1 metre. The quads of the finest, inner grid, level of the clipmap are placed 0.1m apart. Each clipmap level has $N = 255$ vertices along its side with the resulting $M = 64$. The clipmap has $p = 5$ levels apart from the inner grid. The coarse-map textures are set to 4096×4096 which in combination with the clipmap's $0.1m$ granularity, spans $409.6m$. Tessellation is performed at a factor of 3, yielding 9 sub-triangles for every triangle and a detail granularity of $0.0\dot{3}m$. Detail-maps have dimensions 2048×2048. The caching system thus maps $4096 \times 3/2048 = 6$ detail tiles, in one dimension, on top of the coarse-map, according to Equation 2.1.

This choice of parameters is not necessarily the best, but produce satisfactory results. In the case of a game, the values would be tailored to the needs of the game system. Some of the parameters would be based on the user's chosen settings. For example, p, the number of clipmap levels, determines the viewing distance and could therefore be increased if the computer had latent performance.

3.4 Example Stamps

The deformation system uses the paradigm of *stamps* to perform modification to heightmap data. Stamps apply a pattern to the heightmap. There are two main types of stamps. The first is a generic texture stamp which takes a small grey-scale image and adds it on top of the target heightmap. The intensity of impression made by any stamp, texture or other, can be controlled. Specifically, the rotation, position, scale and intensity can all be controlled by parameters passed to the deformation system. Due to the design of the deformation system, a second type of stamp may be created. Because stamps may specify an alternative shader program to that of the generic texture stamp shader, procedural stamps may be created. The impression of such stamps is determined on-the-fly by the associated shader. This allows the impression to be determined by variables such as time or position. A given procedural stamp could thus produce a different result every execution. This allows for much flexibility. Three examples are given to illustrate the possibilities of different stamps.

3.4.1 Footprint

The first example illustrates the use of a generic texture stamp. A grey-scale PNG image of a footprint is stored on the hard disk. A stamp entity is registered with the deformation system. The registration involves the mapping of a unique identification string, such as `"footprint"`, to a struct containing the stamp image path on disk. Other attributes of the struct are required for procedural stamps. Once registered with the deformation system, a deformation using this stamp can be invoked by simply passing the unique identification string to the deformation system's `displace_heightmap` function. The texture image of the stamp is added to the specified location at the specified scale and rotation. The application provides a toggle for automatic footprints, generated as the camera moves across the terrain.

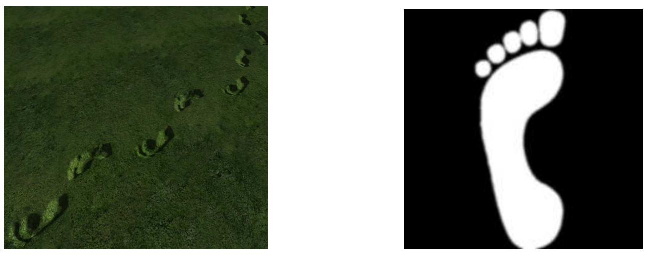

(a) Footprint detail (b) The footprint stamp

FIGURE 3.4: *Shows the footprint stamp and the result it has on detail in the application*

3.4.2 Gaussian

The second example is that of shader-based, procedural stamp. This example is not complicated and does not exploit all possibilities of procedural stamps, but demonstrates the process nonetheless. A Gaussian surface has a bell shape and as a grey-scale texture, appears as circle with soft edges. The most basic Gaussian at location $\mathbf{p} = (x,\ y)$ can be defined by Equation 3.5.

$$f(\mathbf{x}) = \alpha \exp(-\beta|\mathbf{p}\text{-}\mathbf{x}|^2) \tag{3.5}$$

where α controls the intensity, β controls the rate of falloff and **q** is an arbitrary location at which the Gaussian is computed. Section 3.2.1 stated that the region of deformation is set, regardless of the type of texture. This imposes a constraint on the size of the Gaussian. If the parameters are not chosen carefully, the Gaussian would be truncated at the edges of the region. Procedural stamps have a callback function allowing them to perform any pre-deform setup that may be required. The Gaussian shader has a uniform variable to control the falloff β. Using the given scale of the deform region, the falloff can be calculated such that the Gaussian value is near zero at the edge of the deform area. Given Equation 3.5 for the Gaussian, a very small height of ϵ at the edge x, where x is half the dimension of the deform region, is imposed:

$$\begin{aligned} \alpha \exp(-\beta x^2) &< \epsilon \\ -\beta x^2 &< \ln \left| \frac{\epsilon}{\alpha} \right| \\ \beta &> -\frac{\ln \frac{\epsilon}{\alpha}}{x^2} \end{aligned}$$

β can thus be set to $\beta = -\ln \frac{\epsilon}{\alpha} / x^2$ to ensure an intensity of ϵ at the boundary of the region. This falloff value is passed to the stamp's shader and the deformation takes place.

3.4.3 Shockwave

The final example demonstrates how a dynamic texture can be used as a stamp. This is made possible by render-to-texture operations using FBOs. The terrain system provides a means to creates shockwave that travels along the terrain surface, deforming the coarse-map. A separate class handles the state of the shockwave. The wave's texture is initialised to a narrow gaussian. The Wave Equation (Equation 3.6) is used to update the state of the shockwave in an update shader, separate from the deformation system.

$$\frac{\partial^2 u}{\partial t^2} = \nabla^2 u \tag{3.6}$$

The shader uses a finite difference approach to integrating the above equation shown in Equation 3.7.

$$u_{t+1}(x, z) = 2u_t(x, z) - u_{t-1}(x, z) + \gamma(u_t(x - 1, y), u_t(x + 1, y), u_t(x, y + 1), u_t(x, y - 1)) \tag{3.7}$$

In the above equations, u denotes the wave's vertical displacement from zero ; the (x, z) parameters indicate the location in the horizontal plane ; and the $t + i$ subscripts denote the indicated step in time, where $t + 0$ is the current time-step. The parameter, γ, is a combination of wave speed, time-step and mesh-spacing. It has a maximum value of 0.5 to ensure unconditional stability. It is set to 0.5 to yield maximum travelling speed. This update is applied a number of times per frame to spread the wave and is then used as a stamp in a deform operation.

3.5 Summary

A fully functional application has been implemented as a deformable terrain system. Terrain elevation data is stored as OpenGL textures in GPU memory. Terrain data is stored in two levels of detail. The major terrain geometry is stored in a 16bit greyscale textures termed the coarse-map, while surface detail is stored in 8bit greyscale textures termed detail-maps that span the full terrain in a tile-based pattern. Coarse-map data is accessed toroidally using the `GL_REPEAT` wrap-mode to give an effect of infinite terrain. Each heightmap texture has an associated normal map texture. This implementation makes use of partial derivative normal maps which only require two components to store their data at the small cost of limited range. These maps are stored as `GL_RG8` textures in VRAM and are never paged to the hard disk, but rather created each time their associated heightmap is loaded.

Due to the high resolution of detail-maps and value of GPU memory, inactive detail-maps are cached according to the camera's location within the current tile. The caching process is threaded, and invokes the DMA to perform the GPU-CPU transfer. Hard disk operations occur in the separate thread. The state of the caching system is realised by rendering a mini-map as a overlay on the viewport.

Deformations are performed primarily by user input. The process involves render-to-texture operations via OpenGL FBO operations. Only the regions in interest are re-rendered to maximise performance. Deformations are usually followed by regeneration of the associated normal map in that region. Textures, termed *stamps* are used to apply specific patterns during a deformation. Stamps may also be generated procedurally using custom shaders. The implementation provides a number of different stamps to demonstrate their purpose and use. The state of the terrain is presented by displacing vertices of a mesh which is rendered to the viewport. This mesh is formed using the concept of a geometry clipmap and is stored in VRAM as a set of VBOs.

Chapter 4

Results

Chapter 2 specified the design constraints for the terrain system. The system targets modern computer games and the most important constraint is therefore for the system to remain within real-time frame-rates. The target frame-rate was chosen as 60 frames-per-second to allow for other game sub-systems to occupy the remaining frame-time. For this reason most of the tests in this chapter analyse the time taken for certain processes to execute. The tests listed below were performed on two systems with different Shader Model 4.0 GPUs. The lower-end system contains an NVIDIA GeForce 9600GT and the other system houses the GeForce GTX295. This is done in order to analyse the scalability of the terrain system and identify the range of graphics cards it supports.

The terrain system is setup with default parameters, unless otherwise stated, as specified in Section 3.3. The first few tests are performed using only coarse-detail rendering and the final two tests analyse the two different approaches to representing high-detail on the terrain.

Test 1 analyses the maximum render time of the terrain system, rendering only the coarse-map with no high-detail. The results of five test runs and the averages are shown in Table 4.1. Both frame-rates and the time taken to perform a render are recorded. Frame-rates are affected by other processing performed during a frame's execution whereas render-time gives a measurement purely of the time taken for the graphics card to render the scene.

Run #	Time (ms)		Frame-Rate (FPS)	
	9600GT	GTX295	9600GT	GTX295
1	8.725	1.815	95.71	508.07
2	8.730	1.813	95.73	508.70
3	8.829	1.813	92.22	508.91
4	8.799	1.814	93.14	508.98
5	8.649	1.813	96.00	508.12
Avg	8.746	1.814	94.56	508.76
Speedup	4.8x		5.4x	

TABLE 4.1: *Shows the render-time and frame-rate results of 5 tests for both target systems as well as the performance increase from the lower end to higher end machine.*

The *Speedup* row indicates the factor by which the higher-end machine outperforms the lower-end machine on the test. The frame-rate results of 94 FPS and 508 FPS for the lower- and higher-end machine respectively are well above the target frame-rate of 60FPS. The terrain system is thus sufficiently fast when rendering coarse-detail.

Test 2 analyses the dependence of the frame-rate on the size of the viewport. That is, how fast does the frame-rate decrease when the viewport size is increased. Table 4.2 displays the results of the test for five different resolutions. The default resolution, 1024×768, is highlighted and used as the reference result. The *Reference Factor* column represents a comparative factor of each row with the reference row. ie. 640×480 performs $1.3\times$ as fast as the reference row. The *Speedup* column is the same as it was for Test 1, representing the factor by which the higher-end machine outperforms the lower-end machine.

Dimension	9600GT		GTX295		Speedup
	Frame-Rate (FPS)	Ref. Factor	Frame-Rate (FPS)	Ref. Factor	
640× 480	125.76	1.3x	554.57	1.1x	4.4x
800× 600	111.41	1.2x	536.60	1.1x	4.8x
1024× 768	94.56	1.0x	508.87	1.0x	5.4x
1440× 900	75.39	0.8x	438.38	0.9x	5.8x
1920×1080	53.83	0.6x	371.66	0.7x	6.9x

TABLE 4.2: *Shows the effect of viewport dimension on the frame-rate.* 1024×768 *is the reference size that other data are compared with. The speedup column represents how much faster the higher-end card would handle the given resolution.*

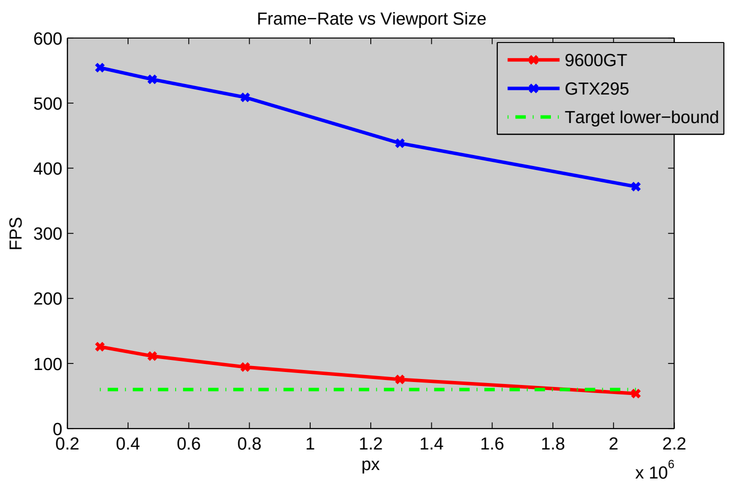


FIGURE 4.1: *Illustrates the dependence of frame-rate on viewport size. The green dotted line indicates the target lower-bound of 60fps.*

The results are represented graphically in Figure 4.1. When lowering the resolution from the default, the lower-end machine yields a greater improvement factor than the higher-end which most likely indicates that the higher-end machine is processing fragments at near peak performance and does not experience any bottle-neck in the fragment shader. It is interesting to note that the same is not true for the 9600GT which experiences a near 60% drop from 125 FPS to 54 FPS, compared to the 33% drop of the GTX295. Improvements and optimizations in the fragment processing stage would thus benefit lower-end machines and should be considered in future iterations of this system. The performance at Full-HD

(1920×1080) resolution is still real-time and only just drops below the requirement of 60 FPS for the 9600GT as is shown in the graph. It is important to note that it is not the drop in frame-rate that is important, but the rather the size of the drop relative to the frame-rate itself, as a percentage. This is due to the fact that frame-rate has an exponential relationship (Equation 4.1) to render-time. Thus a drop from 500 FPS to 400 FPS only represents an increase in render-time of 0.5ms from 2ms to 2.5ms, whereas a drop from 100 FPS to 90 FPS represents a render-time increase of 1.1ms from 10ms to 11.1ms.

$$\text{FPS} = T^{-1} \qquad \text{(T in seconds)} \tag{4.1}$$

Test 3 considers the time taken to perform deformations to the coarse-map. Deformations to the detail-maps are not tested as they will yield the same results or faster. It should be re-iterated that performance does not decrease as more deformations are performed due to no extra data being created - deformations modify existing data. The tests shown in Table 4.3 represent the frame-rate results acquired when increasing the size of the region of deformation. The units of the dimension are given in metres in order to give a practical idea of size. The coarse-map has a resolution of 0.1m per texel and the equivalent texture dimension is 10 times the size in metres. The 200m deformation is thus 2000×2000 texels. The *Deform* column indicates the time taken for the deformation.

Scale (m)	9600GT			GTX295		
	Deform	Normal Gen	Factor	Deform	Normal Gen	Factor
10	6.856	6.195	-	0.899	0.792	-
20	7.027	6.776	0.95	0.913	0.800	0.99
50	7.338	6.898	0.97	0.971	0.828	0.95
100	7.519	7.010	0.98	1.192	0.953	0.84
200	13.15	9.387	0.64	1.875	1.329	0.67
Net Decrease	6.294	3.192	0.58	0.976	0.537	0.53

TABLE 4.3: *Results of deforming regions of different area. The Scale column gives the dimension of the deformed region, thus representing an area of $Scale \times Scale$.*

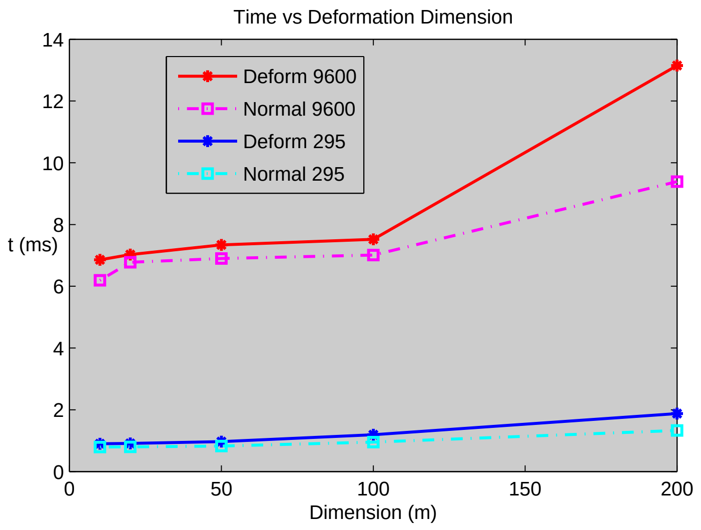


FIGURE 4.2: *Shows the dependence of deformation and normal-generation time on the size of the region being deformed.*

The *Normal Gen* column indicates the time taken to generate the corresponding normals. The *Factor* column gives an indication of the percentage decrease in performance from the above row. Eg. The 200m deformation has 64% performance relative to the 100m deformation, or the 100m deformation takes 64% of the time taken by the 200m deformation. The *Net Decrease* row gives total difference in time between the largest and smallest deformation. Figure 4.2 illustrates the results graphically. The higher-end GPU manages all the deformations with very small performance issues, but the performance of the 9600GT, which is already considerably slower, deteriorates drastically for the 200m deformation. Although such deformations should not be common, except for examples such as the *Shockwave* stamp mentioned in Section 3.4, it is important to note this limit. The difference between the time for generating normals and that of the deformation gives insight to the time spent copying to a double buffer. This increases approximately from 0.5ms to 4ms for the 9600GT when changing from the 100m to the 200m deformation. Eliminating the need for this copy operation would fix this sudden increase. The effect of copying may be greater than the estimated amounts however, as the normal generation requires three additional texture fetches, although the texture cache should hide this. The deformation process should be faster than it is on the 9600GT as such computations have been found to compute in much faster times [AH05].

Test 4 analyses the effect of changing the number of clipmap levels from a small value of 3 to a large number 11. Note that the default is 5. Increasing the number of clipmap levels is one method of increasing the viewing distance. With default resolution 0.1m and $N = 255$, the viewing distances for 3, 7 and 11 -level clipmaps are 100m, 1.6km and 26km respectively.

Clipmap Levels	9600GT		GTX295	
	Frame-Rate	Factor	Frame-Rate	Factor
3	112.25	-	656.26	-
4	95.17	0.87	574.07	0.87
7	80.52	0.85	411.59	0.71
9	70.22	0.87	346.23	0.84
11	63.07	0.90	298.09	0.86
Net Decrease	49.18	0.56	358.17	0.45

TABLE 4.4: *Shows the effect an increase in clipmap levels has on frame-rate.*

These results do not infer too much technical detail about performance, but give an idea as to what a lower-end and higher-end machine can handle. A lower-end machine should probably not go above 7 clipmap levels whereas a higher-end machine could handle 9 levels.

Test 5 concerns the streaming of recently deformed terrain data to the CPU for use in collision detection. This is a difficult process to time as the transfer are performed asynchronously by the DMA. The timing was calculated as follows. The timer was started as the transfer command was issued and stopped immediately after the terrain data had been copied into the collision array. A second timer measured the time taken to copy the data from the PBO's system memory array to the collision array, *Sysmem Copy*. The bus transfer time, *Bus Copy*, was computed as the difference. The results show that the majority of the time is spent copying between system memory arrays. It is important to remember that the copy process is not a simple `memcpy` but also requires scaling, because the raw texture data consists of `unsigned short` elements and the collision array requires the actual heights as floats. This clearly slows the process down. An alternative method should be considered. Possibilities include:

using a floating-point texture (`GL_R32F`) instead of `GL_R16` or storing the collision array as unsigned shorts and only scaling when interpolating heights during collision testing.

Run #	Total (ms)	Sysmem Copy (ms)	Bus Copy (ms)
1	58.8	49.1	9.9
2	56.5	47.2	9.3
3	56.0	46.5	9.5
4	64.5	54.8	9.7
5	66.8	57.3	9.6
Avg.	60.5	51.0	9.6

TABLE 4.5: *Shows the time taken to stream the terrain data to system memory.*

Although the system memory copy is the obvious bottleneck, it may be possible to decrease the bus transfer time. Currently, after initiating the transfer, the algorithm waits an entire frame before mapping the PBO to system memory. The maximum memory bandwidth of a GTX295 [NVIa] is 111.9 GB/s, which would transfer the coarse-map in 0.3ms. This does not account for bus latency, however. Moving the process of mapping the PBO to the beginning of the frame may consume wasted time. A final, obvious optimisation would be to only transfer the modified region of the coarse-map. It is important to minimise this time as best possible so that the physical effects of deformations are realised immediately.

Test 6 The timing of the caching process is very difficult to measure due to bus transfers being asynchronous. It is not possible to tell when a texture has finished uploading to the GPU, for example, as this is controlled by the PBO and DMA. The time taken to unload a texture, however, is measurable to an extent. In order to measure the unload process, the results of which are shown in Table 4.6, a timer is started as an unload request is issued. The timer is stopped - and the elapsed time, recorded - when the data has been written to disk successfully. A second timer measures the combined time taken when copying from the PBO to system memory and writing this data to disk. The first timer's results are shown in the second column of Table 4.6 and those of the second timer, in the third column. The difference in times is computed as the final column and labeled *Bus Copy*. A final note is that these measurements were recorded for the GTX295 and are therefore the best-case results. The hard-drive used was a 7200RPM, and memory was 1066MHz.

Run #	Total (ms)	Sysmem and Hdd (ms)	Bus Copy (ms)
1	159.1	134.1	25.0
2	168.1	148.1	20.0
3	156.2	127.2	29.0
4	162.1	135.6	26.5
5	158.0	125.0	23.0
Avg.	160.7	134.0	24.7

TABLE 4.6: *Shows the duration of time required to page detail-maps to disk*

Although the final column values do contain the time taken to copy across the bus, they also contain time wasted waiting in queues and waiting for PBOs from the pool. It was shown in the previous test that transferring the coarse-map data of $4096 \times 4096 \times 2$ bytes takes under 9ms. A transfer of 2048×2048 bytes should therefore take even less time. The measured values clearly indicate that too much time is wasted waiting in queues and waiting for available PBOs. In order to minimise the caching times, this process will need to be reorganised and optimised so that less time is wasted. An immense time is spent copying system memory and writing to disk, as shown by the third column. Little can be done about disk access speeds, but this is not as much a problem with unloading as it will be with the loading of data. Omitting the memory copy step would mean that the PBO is locked to a cache request for a longer period of time, but would cut down on the total unload time. This possibility should be investigated.

Run #	Sysmem and Hdd (ms)	Est. Bus Copy (ms)	Projected Total (ms)
1	23.5	25.0	48.8
2	20.3	20.0	40.3
3	18.9	29.0	47.9
4	25.3	26.5	51.8
5	23.9	23.0	46.9
Avg.	22.4	24.7	47.1

TABLE 4.7: *Estimates the time it would take to load a detail-map from disk into GPU memory. Only the second column was measured during the load process. The third column is taken from the cache unload table, and the final column is the sum of the two.*

In order to estimate the time taken by the caching system to load a texture, the bus copy time is taken verbatim from the unloading measurements. Assuming that the same time is wasted in queues, waiting on PBOs as well as waiting on available texture IDs, the "bus copy time" must be reduced considerably for load requests as well. Load requests are more important than unload requests as the textures need to be loaded as quickly as possible so as not to cause visual artefacts of partial or unloaded textures. The projected total suggests that a load will take 3 or 4 frames to complete at 60 FPS. This may be sufficient, but only applies for the high-end machine. These times need to be improved so that lower-end machines can achieve decent caching times. Here, the omission of the system memory copy will greatly improve the times.

Test 7 The final test involves the rendering of detail-maps. This report covered the design and implementation of a tessellation method to represent the surface detail. An alternative technique was designed in parallel to this technique. The other technique made use of parallax mapping to represent the detail. The two methods have been measured and are compared in the following two tables. Table 4.8 lists the results of the tessellation technique. The columns are the same as they were for Test 1. The tests were performed with default parameters.

Run #	Time (ms)		Frame-Rate (FPS)	
	9600GT	GTX295	9600GT	GTX295
1	131.291	9.831	7.39	99.73
2	129.329	9.842	7.60	99.65
3	129.561	9.812	7.57	100.02
4	131.479	9.822	7.46	99.75
5	130.211	9.809	7.53	99.99
Avg	130.374	9.823	7.51	99.83
Speedup	13.3x		13.3x	

TABLE 4.8: *Shows the results of representing high-detail by further mesh tessellation.*

Results indicate a large amount of computation is required to perform this method when compared to the results of Test 1 which had no detail rendering. The high-end machine has dropped from a frame-rate of 508 FPS to 100 FPS and low-end from 95 FPS to 7 FPS. Although the high-end machine still runs in real-time, the low-end machine's frame-rate is far below acceptable. The tessellation technique does produce better visual results, with steeper detail and proper occlusion but these are not worth the extra computation required. These results are below the design targets and do not meet the constraint for real-time frame-rates. The implementation needs to be reworked or, if no significant improvement is possible, should be abandoned.

The results of the alternative method for representing high-detail, that of parallax mapping, are shown in Table 4.9. These results are far more pleasing. The render-time for the high-end machine only increase by 0.5ms from the no-detail rendering test and, although it doubles, remains near the constraint of real-time frame-rates for the low-end machine. The parallax shader has not been optimised fully and the 58 FPS should be easily increased. Achieving the target frame-rate for the low-end machine is a success for the representation of high-detail.

The second technique is the obvious choice as a method for representing the surface detail of deformable terrain. If the first technique could be improved significantly it could be an optional alternative, however. For instance, a game could have a setting for *Terrain Detail* with three options of *none*, *parallax* or *tessellated* where owners of high-end computers could opt for the tessellation technique and enjoy the greater visual quality.

Run #	Time (ms)		Frame-Rate (FPS)	
	9600GT	GTX295	9600GT	GTX295
1	14.772	2.331	60.01	401.83
2	15.300	2.330	58.19	401.91
3	15.136	2.329	58.83	402.18
4	15.329	2.327	58.01	403.20
5	15.414	2.338	57.52	401.15
Avg	15.190	2.331	58.51	402.05
Speedup	13.3x		13.3x	

TABLE 4.9: *Shows the results of representing high-detail through the use of parallax mapping.*

In summary, most of the results have met the requirements of real-time frame-rates. Two methods for representing surface detail of terrains have been evaluated. One of these, the tessellation approach covered in this report, did not pass the frame-rate requirement. The other technique was successful. Although the timings were sufficient, many of them do require improvement as was emphasised by the results of the 9600GT. For example, caching and the streaming of collision data can be improved. Streaming of collision data will gain a significant speed increase when only the modified sub-region is copied, rather than the entire terrain and caching will benefit if the copying of system memory can be avoided. Deformation times can be decreased by removing the need for copying to the double-buffer, if possible. As a coarse-scale deformable terrain system, it is successful and with some optimisation and improvements the high-detail representation should be possible.

Chapter 5

Conclusion

The purpose of this project was to investigate, design and implement a framework for real-time, deformable terrain to be used in modern computer games. In order to increase realism and maintain a user's attention, interaction with the game environment is a primary focus. When the user performs an action, he/she expects a reaction. Most games do not provide any many to interact with terrain, however. The addition deformable terrain system to game engines solves this problem and add more depth to a player's experience. Such a system provides a user with immediate visual feedback and gives them a greater sense of freedom. As well as increasing a player's sense of presence, it may also add dynamics where player's can dig their own trenches or burrow under an opponents wall. An additional aim for this project was for all deformations to persist for the duration of the application's lifetime. Many games apply details such as bullet-hole decals that last for a limited amount of time before fading away. This can detract from a game's realism as well. Constraints for the system included real-time minimum frame-rates of 60 FPS and high visual quality. The target architecture for the terrain system was to be Shader Model 4.0 GPUs such as the NVIDIA GeForce 9x and 200 series cards.

The terrain system proposed in this report uses two state-of-the-art techniques. These are Geometry Clipmaps [LH04] and Displacement Mapping [Coo84]. Geometry clipmaps are used to represent the underlying geometry mesh of the terrain whilst displacement maps store the actual elevation data of the terrain. A flat mesh is submitted to the graphics pipeline for rendering. The vertex shaders use the terrain's displacement map to offset vertices. The clipmap is centred on the camera and the displacement map is translated over the mesh in the shader to give the illusion of motion. A deformation subsystem exists that allows modification of regions of the displacement map using texture images termed *stamps*. Stamps, like the displacement map itself, are grey-scale images and are simply added, and possibly scaled, to the displacement map in the specified region. The intensity of the stamp may be varied as well as its rotation and size. In addition to predefined stamps, the deformation system provides the application with the ability to create procedural stamps. Procedural stamps have no actual texture image, but generate the stamp pattern on-the-fly using custom shaders. The rendering process includes per-pixel lighting calculations using ambient, diffuse and Phong specular equations.

The deformable terrain system allows two levels of detail for deformation by means of the coarse-map which is a large-scale displacement map describing the macro-geometry of the terrain, and a set of detail-maps which are laid out across the terrain and describe surface detail such as small bumps, footprints or bullet-holes. The representation of surface detail is done by further tessellating the inner grid and then displacing using the detail-maps. Triangles are tessellated into nine uniform sub-triangles in the geometry shader. Deformations to the coarse-map are taken into account by the collision system. A caching system pages detail-maps to and from the hard disk as needed by the application. All deformations persist throughout the lifetime of the application and even between executions.

The results for the rendering of the coarse-map without surface detail were far above real-time. On a high-end system with an NVIDIA GeForce GTX295, the achieved frame-rates averaged around 508 FPS and the lower-end 9600GT system averaged with 95 FPS. Average-sized deformations of 50 metres took approximately 2ms on the high-end system and 14ms on the low-end machine. It is also important to note that subsequent deformations do not cause any extra performance hits during the rendering process as no extra data is added - deformations modify existing terrain. These are very acceptable results. As a coarse-scale deformable terrain system, the frame-rate constraint is satisfied sufficiently. Due to the fact that all the techniques used - geometry clipmaps, displacement mapping, Phong specular lighting - are state-of-the-art, the visual appearance is of a sufficient standard to satisfy the second constraint.

The results regarding high-detail rendering are not as successful. The geometry tessellation is an expensive process and the geometry shader is known to yield a performance hit that is linearly dependent on the number of output attributes. Although the high-end machine produced an average frame-rate of 100 FPS, the low-end machine only managed 7 FPS. The high-detail could therefore not satisfy the frame-rate constraint. An alternative technique was implemented by a project partner, one that made use of parallax mapping. This alternative technique yielded frame-rates of just under 60 FPS on the low-end machine and over 400 FPS on the high-end machine. Although this technique does not produce as realistic visuals, it satisfies both constraints.

In conclusion, it was possible to successfully create a coarse-scale deformable terrain framework that satisfies the original design constraints. The addition of a surface detail representation using triangle tessellation was, however, not successful.

5.1 Future Improvements

There is much room for improvement in the design and implementation of the terrain system. These include both performance and visual aspects. The most important improvement would be that of adaptive tessellation as proposed in the Design chapter. Only regions containing detail would be tessellated, reducing the number triangles being output from the Geometry Shader from what is currently output. This would, of course, not improve the performance of the worst-case scenario where the entire inner grid requires tessellation. Better design and optimisation of the Geometry Shader could also yield significant improvements. Each tessellated vertex currently performs an additional two texture fetches, to acquire the detail height and normal. This may be reduced to a single read by storing normals in the same texture as the detail elevation data. Using some sort of non-linear interpolation, such as Phong Tessellation, for the tessellated vertices would give the effect of smoother terrain near the viewer and could be used to smooth contours on the horizon as in [BA08]. Future iterations of this system should support multiple coarse-maps to enable larger game worlds and environments.

Render time may also be improved by ordering grid vertices in triangle lists that exploit the post-transform vertex-cache better than raw triangle strips [Cas]. In terms of memory usage, texture size may be reduced through the use of OpenGL's compressed texture formats. It has been shown [AH05] that geometry clipmaps support the presentation of massive elevation data sets that are decompressed and streamed. Whether it is possible to modify and re-compress this data is worth researching as it would yield the possibility of near-infinite coarse-level deformable terrains in computer games.

In its current state the terrain system maps detail textures and colour textures to the mesh in the same manner as it does the coarse-map. When very steep inclines or declines occur, warping artefacts may be seen as the texels are stretched between vertices due to the fact that vertices are only spaced regularly on the horizontal plane. The size and aspect ratio of quads in the mesh may differ greatly, especially in areas of large gradient. It may be beneficial to investigate other mappings of texture coordinates to the mesh, or to rather account for such artefacts in the Geometry Shader by adding or moving vertices.

When deformations are performed, the stamps are applied with uniform intensity to the heightmap. In the case when a footprint is applied to a sharp, but narrow, peak, this would cause an impression on the peak as well as down the slopes. The correct result would be to apply most of the impression to the first point of contact - the tip of the peak. For this to occur, intensity would need some dependence on the current state of the heightmap, but would certainly yield more realistic results. A related addition would be to the support of deformations at arbitrary orientations. Currently all deformations are performed parallel to the vertical. Arbitrary orientation would require extra computation to project the texture coordinates onto the given plane and even further computation if non-uniform intensity were to be supported.

Glossary

Cache

A component or system that transparently stores data in order for future requests to be served faster.

Coarse-map

A heightmap texture used in this project to describe the low-frequency (slow-changing) elevation of the data for the terrain. These maps represent a large area of terrain.

Detail-map

A heightmap texture used in this project for specifying high resolution elevation data for the terrain. These elevation data will describe how tessellated triangles should be displaced from the coarse-map.

DMA

Direct Memory Access. A feature of microprocessors allowing hardware subsystems to access system memory independently of the CPU.

FIFO

First In, First Out. The organisation of data in a data structure such as a queue. The first element to enter the queue, is the first to leave. The last to enter the queue, is the last to leave it.

Framebuffer

The video output device that passes the data representing a complete frame to the video display.

Pixel

Picture Element. The basic element used to display a colour on a computer screen.

SIMD

Single Instruction Multiple Data. This is a class of parallel architectures whereby each processor executes the same instruction, but does so on a different data element. Modern GPUs are of SIMD architecture.

Texel

Texture Element. Similar to a pixel, except that it is the smallest component of given texture image. The width and height dimensions of a texture are equal to the number of horizontal and vertical texels respectively.

T-junction

A location in a mesh where one edge ends in the middle of another edge such that the vertex sits in the middle of an edge resulting in a T-shape. These can cause visual artefacts when the lone vertex's height differs from that of the edge.

VBO

Vertex Buffer Object. An OpenGL resource mapping to an allocated section of GPU memory used for storing vertex data or vertex attributes.

Vertex

A corner of a polygon where two edges meet. Three vertices describe a triangle.

VRAM

Video Random Access Memory. This is the memory on the graphics card.

Bibliography

[Act08] Mike Acton. Ratchet and clank future: Tools of destruction - technical debriefing slides, February 2008.

[AH05] A. Asirvatham and H. Hoppe. Terrain rendering using gpu-based geometry clipmaps. In *GPU Gems 2*, chapter 2. Addison-Wesley, 2005.

[BA08] Tamy Boubekeur and Marc Alexa. Phong tessellation. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, pages 1–5, New York, NY, USA, 2008. ACM.

[Bli78] James F. Blinn. Simulation of wrinkled surfaces. In *SIGGRAPH '78: Proceedings of the 5th annual conference on Computer graphics and interactive techniques*, pages 286–292, New York, NY, USA, 1978. ACM.

[Cas] Ignacio Castano. Optimal grid rendering. `http://www.ludicon.com/castano/blog/2009/02/optimal-grid-rendering/`.

[CC98] E. Catmull and J. Clark. Recursively generated b-spline surfaces on arbitrary topological meshes. pages 183–188, 1998.

[Coo84] Robert L. Cook. Shade trees. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231, New York, NY, USA, 1984. ACM.

[Dia07] Frank Diard. Using the geometry shader for compact and variable-length gpu feedback. In *GPU Gems 3*, chapter 41. Addison-Wesley, 2007.

[DWS$^+$97] Mark Duchaineau, Murray Wolinsky, David E. Sigeti, Mark C. Miller, Charles Aldrich, and Mark B. Mineev-Weinstein. Roaming terrain: Real-time optimally adapting meshes, 1997.

[FFC82] Alain Fournier, Don Fussell, and Loren Carpenter. Computer rendering of stochastic models. *Commun. ACM*, 25(6):371–384, 1982.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, 21(9):948–960, September 1972.

[Gre05a] Simon Green. General-purpose computation on gpus: A primer. In *GPU Gems 2*, page 453. Addison-Wesley, 2005.

[Gre05b] Simon Green. The opengl framebuffer object extension. In *GameDevelopers Conference*, San Fransisco, 2005. NVIDIA.

[HB05] Mark Harris and Ian Buck. Gpu flow-control idioms. In *GPU Gems 2*, chapter 34. Addison-Wesley, 2005.

[HLW07] Xin Huang, Sheng Li, and Guoping Wang. A gpu based interactive modeling approach to designing fine level features. In *GI ’07: Proceedings of Graphics Interface 2007*, pages 305–311, New York, NY, USA, 2007. ACM.

[Hop] Hugues Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering.

[Hop99] Hugues Hoppe. Optimization of mesh locality for transparent vertex caching. In *SIGGRAPH ’99 Conference Proceedings*, pages 269–276. ACM, 1999.

[HR06] Benjamín Hernández and Isaac Rudomin. Simple dynamic lod for geometry images. In *GRAPHITE ’06: Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, pages 157–163, New York, NY, USA, 2006. ACM.

[Kaz07] Maxim Kazakov. Catmull-clark subdivision for geometry shaders. In *AFRIGRAPH ’07: Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 77–84, New York, NY, USA, 2007. ACM.

[Khra] Khronos. Framebuffer objects. `http://www.opengl.org/wiki/Framebuffer_Object`.

[Khrb] Khronos. Rendering pipeline overview. `http://www.opengl3.org/wiki/Rendering_Pipeline_Overview`.

[KTI$^+$01] Tomomichi Kaneko, Toshiyuki Takahei, Masahiko Inami, Naoki Kawakami, Yasuyuki Yanagida, Taro Maeda, and Susumu Tachi. Detailed shape representation with parallax mapping. In *In Proceedings of the ICAT 2001*, pages 205–208, 2001.

[LH04] Frank Losasso and Hugues Hoppe. Geometry clipmaps: terrain rendering using nested regular grids. In *SIGGRAPH ’04: ACM SIGGRAPH 2004 Papers*, pages 769–776, New York, NY, USA, 2004. ACM.

[LY06] Gang Lin and Thomas P. Y. Yu. An improved vertex caching scheme for 3d mesh rendering. *IEEE Transactions on Visualization and Computer Graphics*, 12(4):640–648, 2006.

[MM02] Kevin Moule and Michael D. McCool. Efficient bounded adaptive tessellation of displacement maps. In *In Graphics Interface*, pages 171–180, 2002.

[MSD] MSDN. Direct3d 10 pipeline. `http://msdn.microsoft.com/en-us/library/bb205123(VS.85).aspx`.

[NVIa] NVIDIA. Geforce gtx295 specification. `http://www.nvidia.com/object/product_geforce_gtx_295_us.html`.

[NVIb] NVIDIA. Geforce gtx480 specification. `http://www.nvidia.com/object/product_geforce_gtx_480_us.html`.

[NVIc] NVIDIA. Nvidia cuda c programming guide. `http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_ProgrammingGuide_3.1.pdf`.

[NVI08] NVIDIA. *GPU Programming Guide*, page 28. NVIDIA Corporation, 2008.

[PEO09] Anjul Patney, Mohamed S. Ebeida, and John D. Owens. Parallel view-dependent tessellation of catmull-clark subdivision surfaces. In *HPG ’09: Proceedings of the Conference on High Performance Graphics 2009*, pages 99–108, New York, NY, USA, 2009. ACM.

[SA09] Mark Segal and Kurt Akeley. The opengl graphics system: A specification (version 3.2 (core profile)). `http://www.opengl.org/registry/doc/glspec32.core.20091207.pdf`, 2009.

[SKU08] László Szirmay-Kalos and Tamás Umenhoffer. Displacement mapping on the gpu - state of the art. *Comput. Graph. Forum*, 27(6):1567–1592, 2008.

[Spi] John Spitzer. Graphics performance optimization.