Novel Technologies for the Manipulation of Meshes on the CPU and GPU

A thesis presented in partial fulfilment of the requirements for the degree of

Masters of Science
in
Computer Science

at Massey University, Palmerston North,
New Zealand.

Richard John Rountree

2007

Abstract

This thesis relates to research and development in the field of 3D mesh data for computer graphics. A review of existing storage and manipulation techniques for mesh data is given followed by a framework for mesh editing. The proposed framework combines complex mesh editing techniques, automatic level of detail generation and mesh compression for storage. These methods work coherently due to the underlying data structure. The problem of storing and manipulating data for 3D models is a highly researched field. Models are usually represented by sparse mesh data which consists of vertex position information, the connectivity information to generate faces from those vertices, surface normal data and texture coordinate information. This sparse data is sent to the graphics hardware for rendering but must be manipulated on the CPU.

The proposed framework is based upon geometry images and is designed to store and manipulate the mesh data entirely on the graphics hardware. By utilizing the highly parallel nature of current graphics hardware and new hardware features, new levels of interactivity with large meshes can be gained. Automatic level of detail rendering can be used to allow models upwards of 2 million polygons to be manipulated in real time while viewing a lower level of detail. Through the use of pixels shaders the high detail is preserved in the surface normals while geometric detail is reduced. A compression scheme is then introduced which utilizes the regular structure of the geometry image to compress the floating point data. A number of existing compression schemes are compared as well as custom bit packing.

This is a TIF funded project which is partnered with Unlimited Realities, a Palmerston North software development company. The project was to design a system to create, manipulate and store 3D meshes in a compressed and easy to manipulate manner. The goal is to create the underlying technologies to allow for a 3D modelling system to become integrated into the Umajin engine, not to create a user interface/stand alone modelling program. The Umajin engine is a 3D engine created by Unlimited Realities which has a strong focus on multimedia. More information on the Umajin engine can be found at www.umajin.com.

In this project we propose a method which gives the user the ability to model with the high level of detail found in packages aimed at creating offline renders but create models which are designed for real time rendering.

Acknowledgements

I would like to thank Dr Ramesh Rayudu, my supervisor for this project for keeping things on track and providing support.

David Brebner for his knowledge of all things 3D, compression and for creating the project in the first place.

Robert Grapes for his patience and consistent help with solving bugs.

Russell Brebner for his management and financial organisation.

I would like to thank my family and friends for their support throughout the project.

Finally I would like to thank The Foundation for Research, Science and Technology (FRST) for funding this Technology for Industry Fellowships (TIF) project.

Table of Contents

Table of Figures

Table of Code Listings

Chapter 1 – Background

The motivation of this research is to provide the real time direct manipulation of a large 3d canvas in terms of high level operations like push, pull and smoothing. The ability to paint surface properties such as diffuse colour and shininess and then be able to save this information at varying levels of detail in a compact storage format. Any interaction methods should support novice users.

This chapter provides a brief background on the history of consumer 3d computer graphics to show how the proposed framework is made feasible by the latest evolution of this GPU technology. Then follows the aims and objectives of this research project and a thesis outline.

The rendering pipeline used in computer graphics has evolved throughout the past few years but has mostly remained very similar in structure. In the early years of computer graphics most rendering was done through software renderers that were very slow. In the 1980's bitblitting hardware became popular which, through direct memory access, could copy regions of pixels to the output screen much faster than could be done through the CPU. This began the trend of offloading graphics related tasks to specific hardware. In the early to mid 1990's CPU based 3D graphics began to gain popularity and so did the demand for dedicated 3D processing hardware.

With the new range of specialized 3D graphics processing units (GPU) two API's were developed, Microsoft's DirectX and the open standard OpenGL by Silicon Graphics Ltd. This class of hardware and software interfaces provided massive speed increases over previous generation hardware at the cost of limiting functionality. The fixed function pipeline meant that only certain number of options could be set when rendering creating a limited possibility of rendering effects.

NVIDIA released the Geforce 3 in March 2001 which was the first consumer hardware with a programmable rendering pipeline. This allowed small programs to be written which would alter the way in which vertex and pixel operations would operate. These programs were referred to as shaders, vertex shaders was the name for those which operated on the vertices before rasterization. The shaders which operated on the pixels after rasterization and before output are called either pixel shaders or fragment shaders depending on the API being

used. As the hardware progressed the flexibility of the shaders increased to allow dynamic branching, looping and increased the maximum length of the shaders.

The design of the rendering pipeline from shader model 1.0 up to 3.0 was similar to Figure 1. The fixed function pipeline was still available however the programmable pipeline could be used instead at either the vertex or pixel shader stages. Access to the GPU memory was limited; the input assembler which was responsible for arranging vertices to pass to the vertex shader could only access vertex buffers. The pixel shaders were limited to accessing texture memory only. On newer hardware the output stage could write back directly into texture memory of the GPU memory through the use of pixel buffers or pbuffers. This was to allow rendering directly to textures so effects such as TV screens could be implemented. This also allowed post process effects to be created which would render to a texture and then a separate pass would edit that texture before displaying it on screen. Post process effects include depth of field, blurs and blooms.

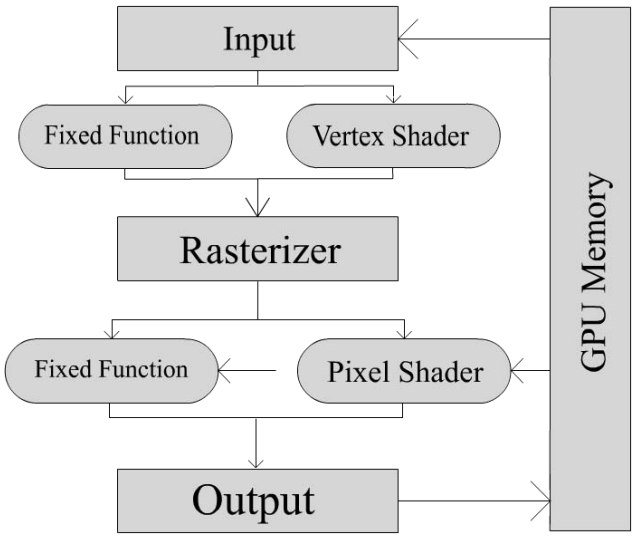

Figure 1: Shader Model 3 Pipeline

In November 2006 Microsoft released DirectX 10 which was a complete redesign to take advantage of shader model 4.0. The most notable changes are the removal of the fixed function pipeline and the addition of the geometry shader. As most graphics applications now take advantage of programmable shaders, the fixed function pipeline was removed to create a more streamlined API. Anything that could be done in the fixed function pipeline can be done in shaders. The new pipeline is shown in Figure 2 (p.3)

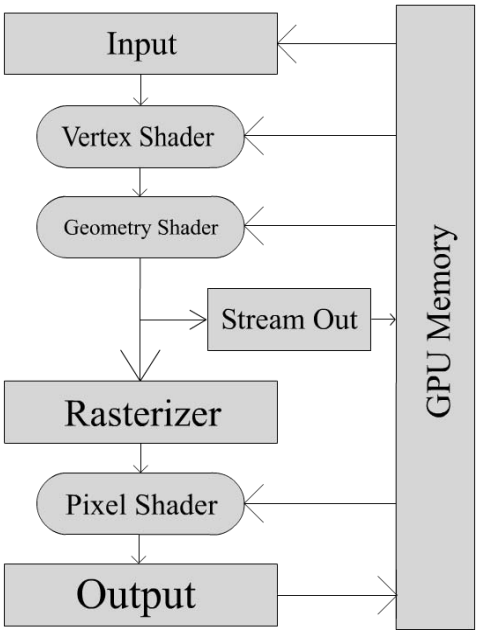

Figure 2: Shader Model 4.0 pipeline

The geometry shader is a new stage which comes after the vertex shader and before the rasterization stage and has the ability to remove or generate polygons, lines and points. The output of the geometry shader can either continue along the graphics pipeline or be sent back into video memory via the stream out mechanism. This allows for vertex buffers to be manipulated and stored entirely on the GPU. The geometry shader can also gain access to the adjacent points of a triangle, this is demonstrated in Figure 3 where V1, V2, V3 are the vertices forming a triangle, and A1, A2, A3 are the adjacent vertices. The geometry shader is capable of performing subdivision on the GPU and then storing the result for later rendering using stream out. Other operations include rendering a cube map in a single pass and storing and manipulating a particle system on the GPU.

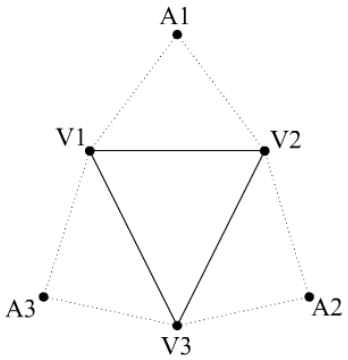

Figure 3: Vertex Adjacency Information

With the ability to store the result of calculations in GPU memory for future processing and output, the bottleneck of sending data to the graphics card is removed. This allows the

GPU to function at maximum capacity while the CPU is free for other tasks. As the performance and capacity of GPUs increase, algorithms which are designed to run solely on the GPU will benefit the most.

Hardware implementations have evolved to create a generalized processing unit, rather than having separate hardware dedicated vertex and pixel shaders. The result is a stream processor which is optimized to work with floating point data and perform vector operations. These stream processors can now be assigned to different tasks depending on the load of the current rendering task. Before scenes could be vertex shader heavy and pixel shader light such as rendering a high polygon count with simple texturing applied. In this case the vertex shader would bottleneck the rendering while the pixel shader would be idle. The opposite would occur on low polygon scenes with highly complex pixel shaders. With stream processors the GPU can allocate different numbers of stream processors to different tasks and increase the rendering performance.

The design of graphics hardware has become increasingly parallel with up to 256 stream processors on a single GPU. This requires parallelism to be a key factor in the design of computer graphics algorithms.

Another benefit of a generalized processing unit is that access to GPU memory can now be uniform throughout the different shader types. Where before the input assembler only had access to vertex buffers and the pixel shader had only access to texture memory, now all stages can access the same memory. This is done by separating resources and resource views. Resource views can be considered as a way of casting a chunk of memory (a resource) for use inside the rendering pipeline. The result is that vertex and geometry shaders also now have access to texture memory. Where shader model 3.0 did have access to texture memory inside the vertex shader, the cost was so severe that the feature was rarely used. With the shader model 4.0 each stage can access memory at the same speed. This is made possible through the use of a common shader core where each type shader stage has a common functionality. The outline of the common shader core is shown in Figure 4 (p.5). Each stage has access to constant buffers for shader input and general buffers which can be generated from within the pipeline via stream out. Textures can either be sampled directly or through sampler states. Sampler states control how the texture lookup is completed and which filters are applied such as mip mapping and texture wrapping states. In addition to the common shader core, each stage can contain specific functionality, such as the geometry shaders ability to add or remove geometry from a buffer.

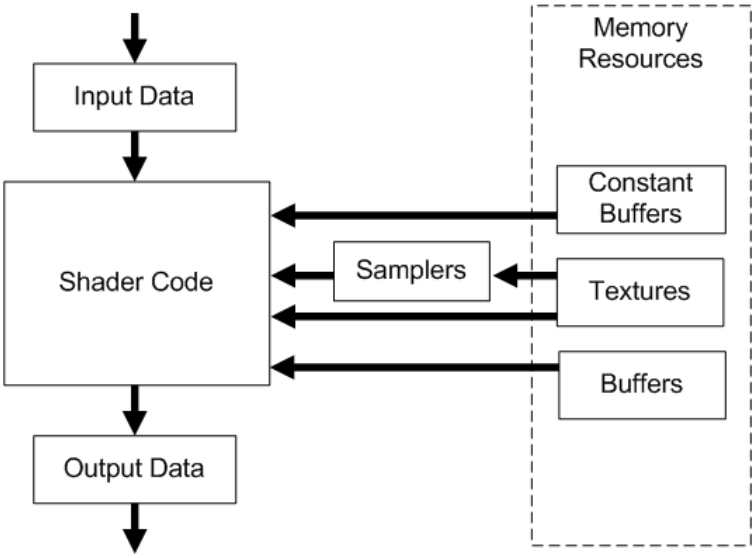

Figure 4: Common Shader Core – Courtesy DirectX SDK

Other new features that have been introduced are integer types and bitwise operations. Within a pixel shader there is now no limit on the number of instructions which can be executed within a shader. This has paved the way for general purpose GPU programming or GPGPU. NVIDIA released CUDA (Compute Unified Device Architecture) which is designed to allow problems which consist of large amounts of vector calculations and can be made highly parallel.

OpenGL 2.0 currently supports some of these features, such as the geometry shader, however OpenGL 3.0 is yet to be released. Version 3.0 is much like DirectX 10 in that it will be a complete redesign of the API and will contain many if not all of the features found in DirectX 10. For this project DirectX 10 was chosen because of its maturity and large amounts of documentation.

The new capabilities of DirectX10 and OpenGL 3.0 for combining the vertex, geometry and pixel pipelines and allowing multiple pass algorithms provides the key capability to support the proposed framework.

1.1 Scope and Objectives

The aim of this thesis is to develop a unified system for the storage, rendering and high level manipulation of large 3D mesh data. The objectives are to design a system that runs smoothly in real time and is easily accessible to novice users.

The goal of the editing system is to allow for novice users to edit mesh data without having to work at the level of vertices, texture coordinates and surface normals. The resulting framework must present a mode of interaction to the user which is sufficiently high level and abstract away from modelling primitives.

Another objective is to automatically generate different levels of detail from the high detail base mesh. The generation of these levels of detail should be done in real time, to allow instant switching between differing levels. This is an important aspect for real time rendering.

The third goal is to investigate compression of 3d mesh data for storage and transmission. Investigation into lossless and lossy techniques is required.

All these outcomes were achieved and high frame rates even with very large models were obtained (2 million effective polygons) and the accompanying DVD will illustrate the ease of interaction.

The goal of this thesis is to research and create the technology required for mesh editing, not to create a mesh editing program specifically. Creating a user interface and mesh editing program for novice users is outside of the scope of this thesis.

Animation of 3d meshes is outside the scope of this project and hence is not investigated. However integration of animation is to be left as future research so should still be given consideration.

1.2 Significance

Current research into mesh editing, compression and level of detail rendering tend to focus only on a specific problem. As a result research into these fields is disjoint. The goal of this project is to combine the different areas of research and create a unified framework which encompasses all of the issues.

3D modelling in its current state is a difficult process and requires highly skilled artists to specialise in 3d modelling and animation. Specific knowledge regarding vertex positions, texture coordinate assignment and surface normals is required along with knowledge of real time rendering issues. This thesis creates a framework which will allow novice users to edit 3d

mesh data without requiring knowledge of the inner working of the mesh structure or rendering techniques.

Automatic level of detail rendering is important for maintaining performance in real time rendering. It allows applications to scale to suit the current hardware automatically and increases overall performance when used efficiently. The generation of different levels of detail is usually done by the artists who create the original high detail model. Generating the levels of detail by hand is costly and an automatic system which works in real time would be preferred providing the models are good representations of the original mesh.

As performance of computer hardware increases so does the complexity of the 3d mesh data used in rendering. Due to the floating point data used traditional compression techniques do not work well on 3d mesh data. Compressing large meshes for storage and transmission is important to keep file size and transfer times low.

1.3 Original Contributions

Created a unified framework which encompasses mesh editing, storage and level of detail rendering. The framework was designed to solve a range of problems simultaneously which include editing, storage and rendering of 3d mesh data.

Shifted the mesh editing computations to the GPU from the CPU. This provides a significant performance increase in editing while leaving the CPU free. By utilizing the highly parallel structure of the GPU the framework becomes scalable and will upscale as new graphics hardware is released.

Created a simple interaction method for mesh editing which is analogous to editing clay. The simple interaction method of push, pull, smooth etc, makes mesh editing possible for novice users. The brush styles are written as pixel shaders and are highly flexible. New brushes can be created easily and integrated into the framework.

Creation of a brush technique which allows for geometry, surface colour and surface properties such as shininess to be altered with a single brush stroke by projecting detail onto the current surface.

Created automatic real time level of detail creation for large 3d models. This allows for the editing of up to 2 million polygons while viewing how lower level representations will look. The automatic levels of detail created are good representations of the highly level models.

Proved that traditional image compression techniques can be used on 3d mesh data. Investigated floating point image compression techniques for 3d mesh data by implementing a 32 bit floating point JPEG based compression.

1.4 Thesis Organization

The research presented in chapter 2 will show that there are many different methods for storing and rendering 3D mesh information, however each of these have drawbacks and even the de facto standard is far from optimal. Many of the existing techniques focus on only a single area 3D mesh operations at the cost of other factors. As a result of this, there is a wide range of solutions for content creation pipelines, which require significant amounts of input from skilled 3D artists. These content creation pipelines are designed for expert users and do not allow novice users to quickly edit a simple model.

Then in chapter 3 the new framework is presented covering its representation, rendering methods and a range of editing techniques.

In chapter 4 automatic level of detail rendering is investigated and a suitable solution provided.

Next in chapter 5 a range of traditional mesh compression techniques are compared. Bitpacking techniques and image based compression methods specific to this framework are also investigated.

Chapter 6 provides results both visual and numerical which are linked to the accompanied DVD. The GPU based implementation is compared against alternative implementations and the results show the performance difference.

Chapter 7 contains the conclusions of this project including a summary of limitations and future directions for additional research to follow.

Chapter 2 – Representations and Literature Review

Due to the large scope of this project a number of areas are to be covered in the literature review. This encompasses different representations for 3D models, existing software packages, editing techniques and the storage of model data.

2.1 Voxels

Voxels is a term used to refer to volumetric pixels. If pixels are a two dimensional array of colour information then voxels are a 3 dimensional array of colour information. Voxels are still in use for medical data scans such as MRI scans. They have been used in some games however their severe limitations are seen to outweigh their benefits.

Figure 5: 256^2 Voxel Data set

The main drawback with voxels is the large amount of memory which they consume. Figure 5 is an example of a 256^3 voxel data set which consists of 3 channels. To reduce the memory size, no alpha channel is used, instead a fixed value is assigned as the transparent marker. The voxel set still consumes 48 MB of data if the precision is a single 8 bit integer per voxel per channel.

There are a number of approaches to rendering voxel data sets. One such method is to use ray casting on the voxel data set which provides the best quality for rendering. The traditional CPU implementations however are computationally expensive and are generally considered as offline rendering. However by utilizing programmable shaders on modern GPU's the ray tracing approach can become real time due to the highly parallelizable nature of ray tracing problem [16, 19].

Another approach is called splatting in which small view aligned discs are drawn on screen with the appropriate colour. These small disks create a low quality rendering of the voxel set. The splatting technique is simple to implement and performs very quickly. There have been a number of advancements on this such as [1] and [2] which focus on LOD and visual quality respectively.

Another method for rendering voxel data sets is to convert it into a piecewise polygonal mesh and render it using traditional polygon rendering techniques. The first conversion technique to do this was the marching cubes algorithm [3]. A lookup table of 256 possibly voxel combinations is generated from 15 unique cases through reflection and symmetrical rotations. Each voxel is matched to the lookup table and the mesh is generated. The marching cubes algorithm was patented in 1985 and the marching tetrahedron algorithm [20] was created to circumvent the patent restriction. The patent on the marching cubes algorithm has since expired. The marching tetrahedron algorithm also resolves some small ambiguities within the marching cubes algorithm which occurred during some voxel configurations.

The main benefit of voxels is that editing operations become extremely simple. Data can be added and subtracted from data set without having to change any of the surrounding voxels. Complex Boolean operations can be done with multiple voxel sets to create complex shapes. The one editing operation that suffers however is rotation of angles which are not multiples of 90^0. The worst case scenario is rotating a thin voxel line by 45^0 around all 3 axes and then rotating back. As the first rotation occurs, voxels from the initial set which do not line up with the target set get discarded. The same occurs when the reverse rotation occurs and the result is holes forming in the voxel set which were not there originally.

Voxel data sets can come from scanned range data or from existing piecewise polygonal data. The process of converting a mesh into a voxel set is called voxelization which was first introduced in [4] and [5].

There have been a number of different methods for compressing voxel data such as [22]. In their paper a hashing technique is proposed for storage of voxel data sets in a small hash cube and offset cube. The resulting representation is heavily compressed and allows for quick reading of data. However editing the voxel data set is difficult as the hash table needs to be rebuilt which is a time consuming task. This method is great for storage and reading however it is unsuitable for real time manipulation.

The use of 3D textures on new graphics hardware has revived interest in voxels for small scale effects. In the NVIDIA SDK [27] an animated mesh is converted to voxels in real time for use with a smoke simulation effect. This can be used for a range of GPU based physics effects.

2.2 Octree

An octree [26] is a tree structure which represents the usage of space within a 3D region. The base node represents the middle of the area and with each successive level of the tree the region is broken up into another 8 regions. It is analogous to the two dimensional structure, the quad tree, which divides each new section into 4 partitions. This is shown Figure 6.

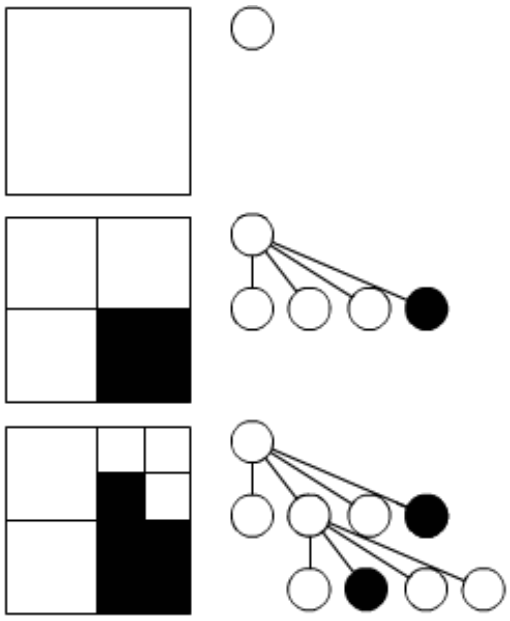

Figure 6: Quadtree Diagram

The octree structure allows for voxel data sets to be encoded to a smaller memory size than a full voxel set. When representing a slanted surface however a large number of small regions are required for a good approximation. The reduced storage size is gained at the cost of lookup time into the octree structure. Where a voxel data set can access any point in one lookup, an octree may require traversal of many levels of the tree structure to locate a specific point. An advantage of the octree structure is that generating different levels of detail can be done automatically by simply stopping at a certain level of the tree structure.

2.3 Adaptively Sampled Distance Fields

Adaptively Sampled Distance Fields (ADFs) [43] are a volume representation which allow for areas of high detail. The structure allows for rendering, editing, level of detail and collision detection and the paper focuses on carving fine detail and rendering volume data. The distance field used is a signed scalar value which represents the minimum distance to the surface of the model. A negative value indicates the point is inside the model and positive represents outside. The use of adaptive sampling as opposed to regular sampling ensures that there is a high amount of samples taken in areas with fine detail. The structure proposed is based on an Octree however mention is made that other formats could also be used such as wavelets and Delaunay tetrahedralizations. The paper states that for 2D shapes ADF's show a 20:1 reduction in memory requirements compared to traditional Quadtrees while retaining the high detail information.

The renderer used in [43] was based upon ray tracing. Later in [44] a dynamic meshing approach was developed which allowed for a polygonal mesh to be generated based upon frame rate requirements. The algorithm creates meshes which have high polygon detail where the camera is pointing and low detail elsewhere.

Later the ADF was revisited and minor changes proposed [45] to provide greater flexibility.

2.4 Normal Meshes

Normal meshes [6] are a technique which allows for the storage of a mesh as one float per vertex. This is possible because each level of the mesh is created by offsetting via the normal from the previous version of the mesh. The offset is made halfway between the points of the coarser level of the mesh. With enough iterations the original mesh detail is gained. A simple example is shown in Figure 7.

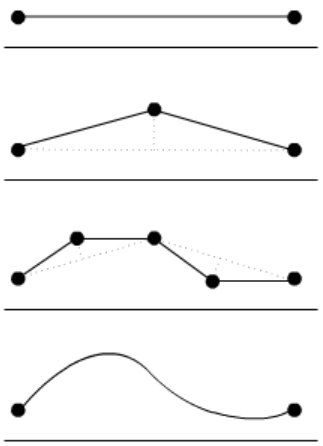

Figure 7: Different LOD on a Normal Mesh

A robust algorithm is provided by the paper which converts arbitrary input meshes into a normal mesh which creates a good triangulation. The semi-regular mesh created is well suited for texturing as the mesh parameterization is relatively smooth inside patches.

Generating levels of details is trivial with this method as it is inherently built in. The storage for the mesh is also very compact due to only one float per vertex being required. Multiresolution mesh editing can also be accomplished easily to edit the overall shape of the mesh while preserving the high frequency detail.

The continuation of [7] was to focus on the compression of normal meshes. This was achieved through a wavelet transform and zerotree coder and provided significant gains in geometry compression based upon normal meshes.

A further algorithm for the creation of normal meshes was later proposed in [8]. A different generation approach assumes a globally smooth parameterization and alters this to create the variational normal mesh. This is in contrast to [6] which finds the base mesh and then builds the normal mesh. Variational normal meshes better preserve the volume and provide better compression.

2.5 Polygonal Meshes

The polygonal mesh is the de facto standard for model representation in real time rendering. Throughout this thesis the term mesh will refer to a piecewise surface definition consisting of triangles. Quadrilaterals or higher order polygons can be equally used, however when rendering they need to be split to triangles. A mesh is represented as a list of vertices which determine the geometry of the mesh. A list of faces which store the connectivity of the vertices defines the topology of the mesh. Additional data can be encoded such as per vertex or per face normals, texture coordinates and bone weightings. To aid performance when manipulating the mesh, extra connectivity information can also be imbedded in each face, such as the index of neighbouring vertices or defining edges. The information added to the data structure needs to reflect the topological and algorithmic requirements of the mesh. Figure 8 shows the most basic structure for a polygonal mesh.

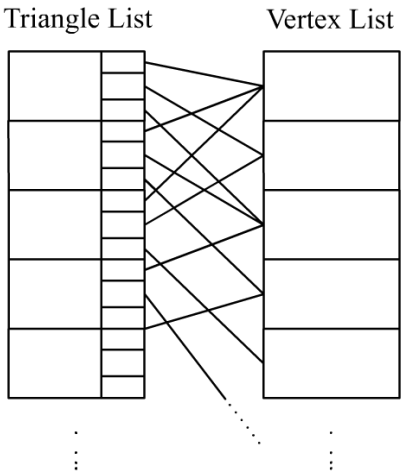

Figure 8: Polygon mesh data structure

Polygonal meshes are closely aligned with current graphics hardware in that they represent the primitives used in rendering. However the storage method is not ideal due to the arrangement of the data structure. To construct a vertex buffer each face must look into another array to find the value stored for each vertex. This must be repeated for other attributes such as texture coordinates and normals.

A polygonal mesh is highly flexible and can be used to define any shape however curved surfaces can require a large amount of polygons to accurately represent a smooth shape. Conversely large flat areas can be represented with only a few polygons. Talented 3D artists can apply a number of tricks when creating models to fully exploit the format. Surface

normals can be altered to create a rounded look for a surface when it is flat and reuse of texture coordinates can reduce the texture map size. Texture coordinate assignment is assisted by automatic solutions, however these are rarely optimal and artistic input is required to create a suitable mapping of texture coordinates to vertices.

Two common methods for storing meshes are Wavefront OBJ [36] and 3D Studio Max's [35] 3DS file formats. There are many different custom formats available, however these two are consistently used standards. Wavefront OBJ is a plain text ASCII format in which lists of vertices, texture coordinates, normals and faces are store in human readable format. Material information and groupings are also possible. It is a simple format to understand and implement however the resulting file size is larger than necessary due to the ACSII encoding and is not optimal. 3DS is a binary file format which is designed to add additional information such as animation data.

Manipulation of polygon meshes is usually done at a very low level, with the artist working directly at the vertex level. However models can be created using higher levels of abstraction such as through subdivision patches or CSG and then converted to polygon meshes once editing is complete. CSG operations on polygon meshes directly are a very difficult task where polygons must be cut correctly and new polygons created to join the gaps.

2.6 Geometry Images

Geometry Images [9] were first introduced in 2002. The proposal was that rather than storing mesh data in an indexed array lookup fashion, to store the data in a regular grid structure. Because of this structure the texture coordinates can be removed as they can be inferred from the position within the geometry image. The difficulty with this technique was that to convert an arbitrary piecewise polygonal mesh into a geometry image the mesh would need to be parameterized correctly. To do this a set of cuts along the surface are defined such that the mesh unfolds into a plane. The seams then need to be taken care of to ensure that the reconstruction of the mesh does not have holes in it.

The result is that a 3D mesh is converted into a 2D image where the colour represented the vertex position. Figure 9 (p.16) shows a geometry image on the left and the model it represents is on the right. The marked letters on the mesh match the positions in the

geometry image. The strengths of this method are that automatic level of detail generation becomes a trivial task, as simple as down sampling the image. Care needs to be taken to ensure that the seams are treated properly. Another benefit was that the mesh structure was now a regular array rather than an array based lookup structure, this meant it was more suited for graphics hardware.

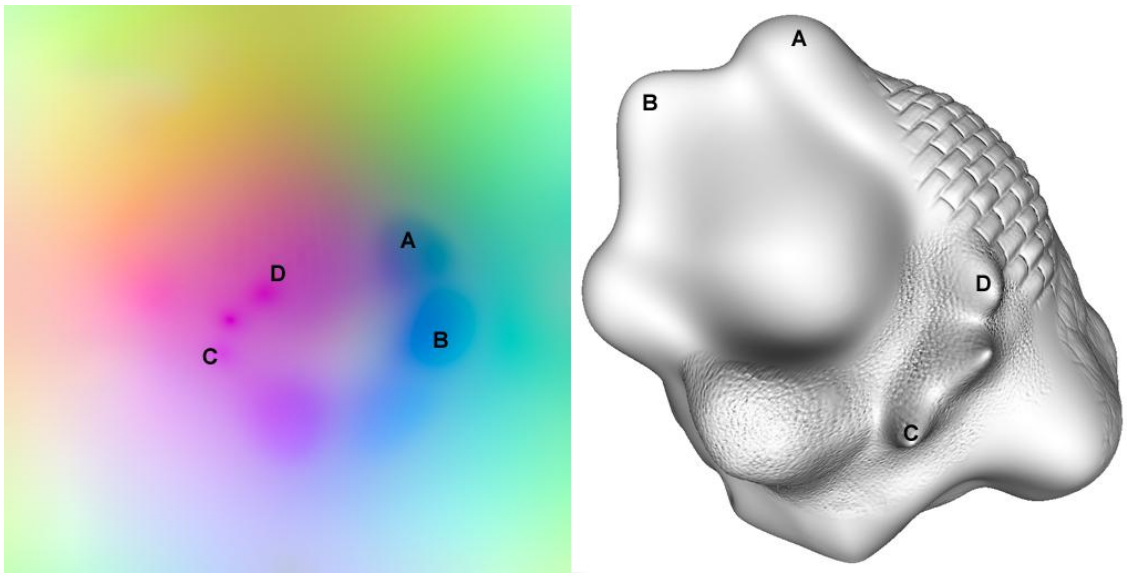

Figure 9: Geometry Image and associated mesh

One area which was left untouched in the geometry images paper was animation. This was later addressed in [10] which proposed the use of 2D video to represent animation. Each frame within the video represents another keyframe in the animation.

Another method proposed by [11] was to create a small geometry image which created the base mesh for subdivision. Once the mesh had been subdivided, a height map is applied which returns the mesh close to its original base value. The geometry image stores the tensor values required for the sub division, which due to the regular structure is now suited for computation on graphics hardware. The fragment shader is used to convert from a coarse octahedron base mesh to a smooth representation of the original mesh.

In the original geometry images paper [9], compression was briefly discussed where the floating point values were quantized to 12-bit integers. Brief mention of a wavelet based compressor was made and this was continued in detail in [13] which focused on the compression of geometry images for storage and transmission. By using a spherical parameterization and then translating to an octahedron the geometry image could be repeated in all directions. When repeated in all directions a globally smooth wavelet transform could be applied. Image wavelets and spherical wavelets were compared with the

spherical wavelets providing better results for geometry image compression. When PSNR analysis was done on the compressed images the results did not coincide with the meshes which look best. The suggestion was made that PSNR is not a suitable metric for mesh comparison and that high frequency data is more important than low. In terms of implementation details mention of quantization to a required bit depth was made, however in the results the bit depth was not stated. A 513^2 geometry image was compressed to between approximately 1,000 and 12,000 bytes however the bit format information is omitted. The visual artefacts which occur when the compression is too high are waves which form in the surface of the mesh.

Multi chart geometry images [28] create an atlas over a model and map sections to a geometry image. A novel technique to zipper sections together was proposed to combat the effect of creating so many seams. The advantage of this approach is that the genus of the model is no longer a problem. Also the geometric fidelity of the mesh is increased when using multi chart geometry images. However problems that arise are the positioning of sections of the geometry images in the charted image. Also creating reduced levels of detail can pose a problem.

A technique proposed by [12] provided inspiration for this project by using geometry image to transfer surface detail. Surface detail of a model could be extracted by smoothing the mesh and then finding the difference between the original and the smooth model. This gave a good representation of the detail on the surface of the model. The information could then be added to a separate mesh to alter the surface of the model. One problem with this method was that the result of applying a surface deformation depends on the underlying texture coordinates of the geometry image.

An extension of this work was GPU mesh painting proposed by [14]. This allowed for the real time editing of a mesh on the GPU based upon a base subdivision surface with a geometry image charted on the surface. Rendering is achieved through the vertex texture fetch operation for each vertex for each frame. The use of pixel shaders was proposed to modify the geometry images as due to the regular structure of the geometry image, it is highly suited for the task.

Other uses of geometry images have been put forth by [15] which demonstrated the use of geometry images for collision detection. Another was [16] which implemented GPU ray tracing of dynamic meshes using geometry images.

2.7 Subdivision

Subdivision is a method for iteratively refining a piecewise linear polygonal mesh to create a highly tessellated smooth surface. The limit surface is the result of subdividing an infinite number of times. A subdivision surface is well suited for automatic level of detail, and the number of iterations determines the quality and polygon count of the resulting mesh.

There are a number of methods for computing the subdivision surface, the first and most common is Catmull-Clark subdivision [34]. This technique makes the mesh smoother, where sharp edges become rounded. Subdivision techniques are common in modelling programs such as Maya, 3ds max and blender.

To add detail to subdivision surfaces [17] proposed adding a scalar displacement field. An algorithm is provided to create a suitable base mesh and height map to represent the original model. This storage method is good for compression, as the mesh topology and parameterization is stored in the base mesh. The detail is stored in a single channel scalar data structure. To edit small detail, the scalar data channel can be manipulated, conversely the coarse base mesh can be altered while the high frequency remains the same.

A GPU based implementation of manipulating fine surface detail of a subdivided surface was proposed later in [37]. By utilizing the GPU the speed of editing operations were increased dramatically, approximately 4-6 times faster.

2.8 Constructive Solid Geometry

Constructive Solid Geometry (CSG) is a technique where Boolean operations are performed on simple primitive shapes such as spheres, cones and cubes to create more complex models. The representation for a CSG model is called a CSG tree and the leaves consist of definitions of simple shapes. The branches are Boolean operators which define the construction and the root node is the final model, as shown in Figure 10 (p.19). CSG is typically used for precisely modelling inorganic objects and is common in Computer Aided Design (CAD) programs.

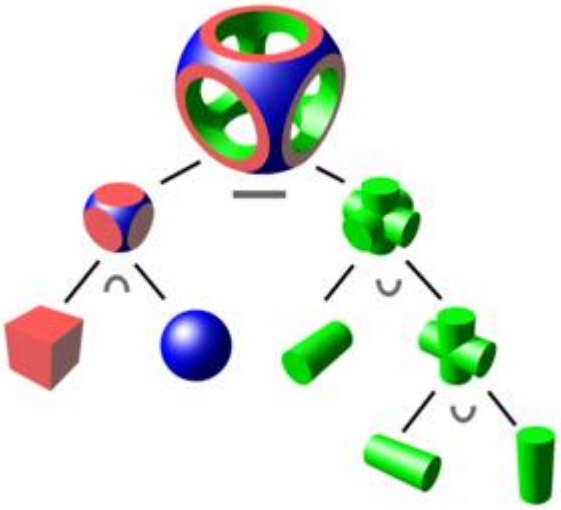

Figure 10: CSG Example courtesy Wikipedia

Rendering of CSG trees can be done in a number of different ways. Ray tracing is a suitable option of offline rendering as the exact volume of the model can be evaluated from the CSG tree. The original Goldfeather [40] algorithm was designed for the real time rendering of CSG trees on a custom graphics rendering system. This was later adapted for modern graphics hardware [41, 42]. This method of rendering employs the use of the z-buffer and the stencil buffer to render only the visible parts of a CSG tree.

Another option for rendering is to convert the CSG tree into a polygonal mesh for traditional rendering. It is best to edit the CSG tree rather than apply Boolean operations to the polygonal mesh directly. It is possible but very complex, as each polygon that is cut needs to be divided to represent the cut and to remove all seams from the mesh.

2.9 Existing modelling packages

There is a wide range of 3D modelling products available, ranging from open source to very expensive commercial packages. These can be categorised into three main categories, polygonal mesh, CSG and multi resolution modelling.

Polygonal mesh modellers such as 3D studio Max [35] generally focus on working with mesh primitives, e.g. quads and triangles. With such a low level focus on model creation artists can create highly optimized models for real time rendering. Models can have the vertex

normals manipulated and careful assignment of texture coordinates can create models which render as efficiently as possible.

CSG style modelling tools are more designed for computer aided design and focus on the creation of inorganic models. These programs are used for the modelling and analysis of machine components, not for the creation of polygonal meshes for real time rendering.

The third style is multi-resolution modelling which can be seen as an extension of polygonal mesh modellers. Examples of these programs are Zbrush [52] by Pixologic and Hexagon [51] by Daz 3D. In these programs the user creates a base mesh which is subdivided. The user can then sculpt fine detail on the surface of the mesh and apply set textures to the geometry such as cloth or skin. These programs create models with very high detail and the resulting meshes are generally used for offline rendering such as computer animation for movies. Some programs have the option to create a low detail model and preserve the high detail information as a normal or displacement map for real time rendering.

2.10 Editing Techniques

In [31] a method was presented which made traditional cut and paste available on surface detail information for 3D models. To cut a region of surface detail the selected area is first smoothed to generate the baseline for the detail measurement. The difference between the original and smoothed regions is taken as the surface detail information. To then paste this information is applied to the desired region to add the surface detail. Various detail selection and pasting methods are defined which generate different results. Interactive speeds are claimed however in the worst case the frame rate drops to 5 – 0.5 fps. No details are given as to the polygon count when these editing operations take place.

The GPU mesh painting paper [14] demonstrated some intuitive mesh editing techniques where the user interacts by using the mouse to brush geometry deformations onto the mesh. Different brushes such as pulling, pushing and smoothing were demonstrated. While working on the mesh vertices directly the effect of manipulating a freeform surface is given because the surface is highly subdivided.

Blending between two arbitrary meshes can be a difficult task. Two models will rarely have the same topology and so a simple blending between two meshes is not possible. One method to do this is to ensure a consistent mesh parameterization [39]. This concept was furthered in [38] with a strong focus on blending or morphing between multiple meshes. When using geometry images [9] morphing can be done simply by blending between two different images. However there is no guarantee that the points on the two geometry image will correspond intuitively.

Another method for editing meshes is the posing of existing meshes. One such method for this is described in [32] where a sketching system is used to manipulate a mesh. First a baseline is drawn on the model and then a new target line is drawn. The paper covers determining areas of interest, editing techniques and remeshing when the geometric stretch becomes too large. This technique is well suited for altering an existing mesh with only a few interactions from the user at a high level of abstraction away from vertices and faces.

2.11 Compression

As GPUs become more powerful, the number of polygons able to be rendered increases. With this the storage requirements of 3D model data also increases. Some topics on geometry compression have already been covered such as wavelet compression on geometry images. Here some other compression techniques will be discussed which mostly focus on piecewise polygonal meshes.

The 32-bit floating point format commonly used to represent geometry in a mesh has very high precision. The representation can be heavily quantized to a lower bit format as shown in [47]. This quantization was applied to the position, normal and texture coordinate data and then carefully bit packed to reduce the file size. In their results a compression from 96 bits for position and 96 bits for normals down to 30 bits for position and 12 for normal the visual difference is only slight.

The Edgebreaker [46] algorithm was designed specifically to compress mesh topology information. During compression the mesh connectivity information is stored using *opcodes* which describe the topological connection with the rest of the mesh. The vertex position information is decoupled from the topology and can be compressed separately. In the paper

they suggest quantizing the floating point data to a 10 bit representation which can distributed linearly or non-linearly.

A triangle mesh connectivity compression scheme was presented by [49] which used a parallelogram predictor to determine the connectivity of a mesh. The geometric data was uniformly quantised and compressed. This work was expanded in [50] where the same parallelogram predictor was applied to the geometric information. The position is predicted as a quantized value and the offset from the predicted value to the original value is stored. As this value is usually close to zero it compresses well. For this technique to function well the prediction of vertex positions needs to be good.

The results from these approaches however were still lossy due to the quantization steps required. Later [48] created an entirely lossless mesh compression scheme. Much like the earlier work the topology was encoded with a lossless compression. The geometric information was also calculated using a parallelogram predictor, however all calculations were done with floating point information. The resulting offset value then is split into separate sign, exponent and mantissa. These values are compressed separately with a lossless encoder so the entire floating point data is stored with lossless compression.

2.12 Level of Detail

Generating versions of a mesh which contain different levels of detail is very important for rendering. As a model moves further away from the viewer, fewer pixels are used onscreen and detail is lost from view. To render objects far away a lower amount of polygons can be used to represent the original high detail model. An example of this is the high and low detail spheres in Figure 11 (p.23). When far away from the camera, there is no noticeable difference in the meshes. Performance is critical in real time rendering, and using fewer polygons to represent a model and maintaining the same visual quality is preferred.

Being able to automatically generate different levels of detail which preserve the shape of the original model is preferred as it will reduce the production costs of generating models, as artists do not have to create low polygon versions of models. Also if the different levels of detail are generated on the fly, it will reduce the storage costs as only the high polygon count model needs to be stored.

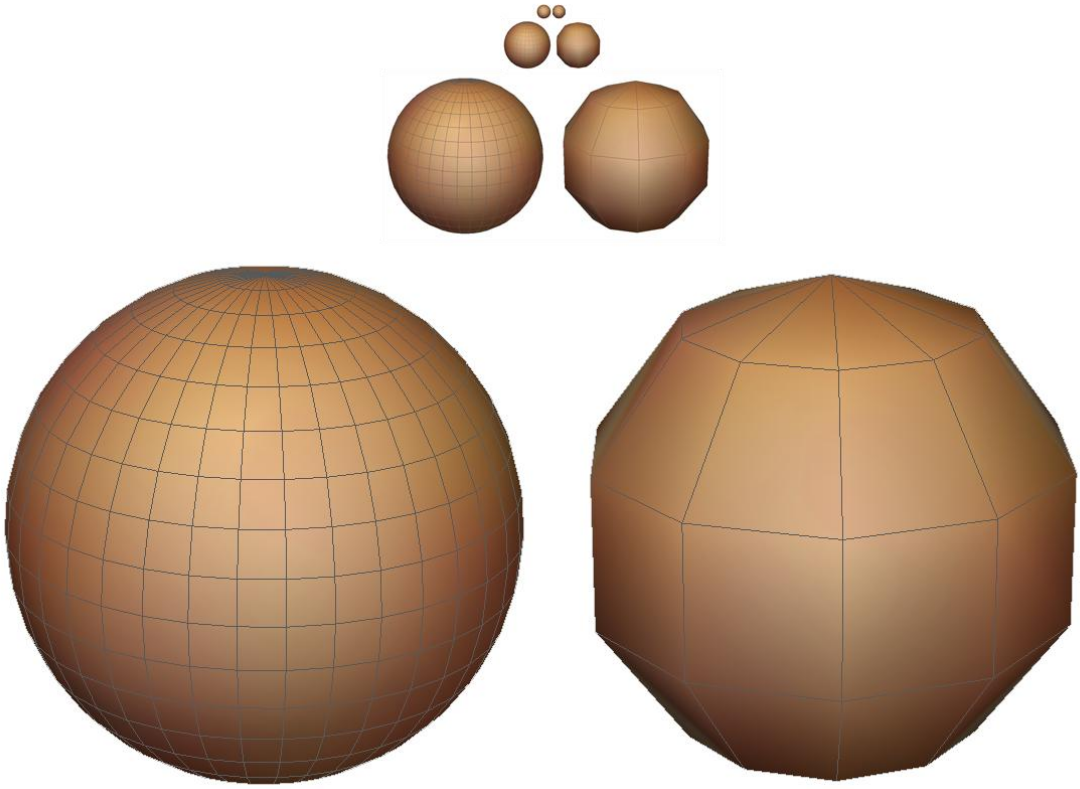

Figure 11: Comparing Different Levels of Detail

Automatic level of detail generation techniques have been well researched and a summary can be found in [54] which focus on mesh based polygon reduction. One method to reduce the polygon count is to search for patterns in the mesh connectivity and replace them. The replacement polygons must have a smaller polygon count and provide a good approximation to the original mesh geometry. These replacements work best when the polygons being replaced are relatively flat in any plane.

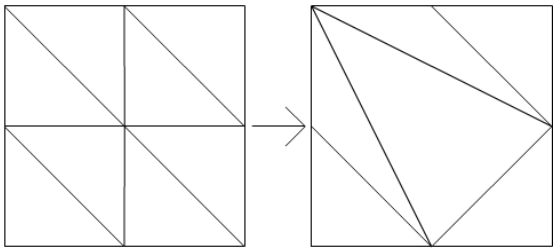

Figure 12: Polygon Reduction

One example of polygon reduction is shown in Figure 12 where an 8 polygon section is reduced to 6. This approach generates another pattern in the mesh topology to which successive matching techniques can be applied.

A more direct method is shown in Figure 13 however this approach requires more calculation to prevent the formation of any holes. This technique must first check that the normals of the 8 polygons are sufficiently close. Then the vectors of the sides must be tested to ensure they are the same. If the vectors of the edges are the same then they can be replaced with the edge of a larger triangle. This method gives an 8:2 reduction of polygons in the best case.

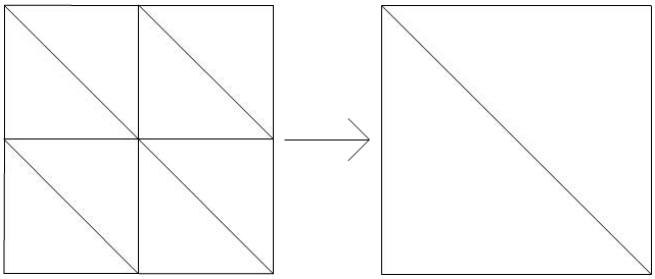

Figure 13: Square based Polygon Reduction

Another polygon reduction technique is the vertex collapse [54]. This works by joining vertices which are sufficiently close together which results in the removal of polygons. To remove the polygons, each face is tested to see if two or more of the vertices are in the same position, if they are then the polygon has become redundant and is removed. To ensure the mesh looks the same care must be taken with the texture coordinates. While the stretch of the texture coordinates might be minimal due to the small movement of vertices, this can create visible seams in the texture across the surface of the mesh. An example of the vertex collapse is shown in Figure 14 where the centre two vertices join together to create one in the middle.

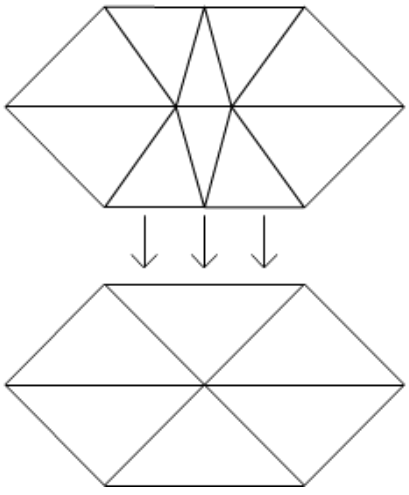

Figure 14: Vertex Collapse

The vertex collapse was most effective in regions of a mesh where small triangles are densely packed.

An important concept for automatic level of detail generation is mesh saliency [29], a measure of importance for all points on a mesh. In their paper they define the metric for mesh saliency as surface curvature, both low frequency smooth curves and high frequency sharp edges. To calculate the saliency a centre-surround filter with Gaussian-weighted curvature is used. This is a combination of surface curvature measurements each with a different radius. However areas are left out from this research and mention is made for future research such as including lighting information which could be done through ambient occlusion. Texture information is also noticeably absent from their research.

Progressive meshes [30] are a format for representing arbitrary triangle meshes which provides mesh compression, progressive transmission and level of detail. The progressive mesh stores the coarsest level of detail and then a list of detailed instructions about how to restore the mesh to its full level of detail. Each instruction details a vertex split, where an additional vertex is added to the mesh. The paper describes the progressive mesh format and how to convert arbitrary meshes into progressive meshes.

2.13 Summary

A wide range of alternatives to the de facto standard, the piecewise polygonal mesh exist for mesh representation and compression. Voxels provide fast lookup to any point and simple editing but the memory requirements are far too large to be practical. Octrees reduce the memory requirements at the cost of the uniform structure and lookup time. Normal meshes provide a compact representation and automatic level of detail. Polygonal meshes are awkward to edit but there are a wide range of compression schemes and they are suited for rendering hardware. Geometry images will become the focus of this research as they provide good options for editing, compression and are very well suited for rendering.

Chapter 3 – A Mesh Editing Framework for the GPU using Geometry Images

This chapter will begin by detailing how the mesh editing framework is presented to the user and how the interactions occur. The mesh structure, rendering techniques and editing techniques will then be discussed. Further information into specific brush techniques will be given as well as details on how level of detail rendering can be utilised during editing to create models with more detail.

Ritschel et al.[14] proposed a framework for mesh editing, which required both GPU and CPU to edit geometry image wrapped around a base mesh. In this chapter, we propose a framework which utilizing features found in the DirectX 10 pipeline runs entirely on the GPU. This leaves the CPU free for other tasks and reduces the cost of communication from the CPU to GPU and vice versa. The design is likewise based on geometry images which can be stored and edited on the GPU and converted to a mesh for rendering.

Figure 15 (p.27) shows the overview of the entire framework at a high level of abstraction. In section 3.1 the user interaction with the framework will be described which identifies the simplicity of the system for users. Figure 15 shows how the rendering and editing of the framework work together and will be detailed further in sections 3.3 and 3.4 respectively. Chapter 4 covers Level of Detail reduction for rendering and storage in traditional mesh based formats. Chapter 5 Investigates the application of traditional and new image compression techniques for the storage of geometry images.

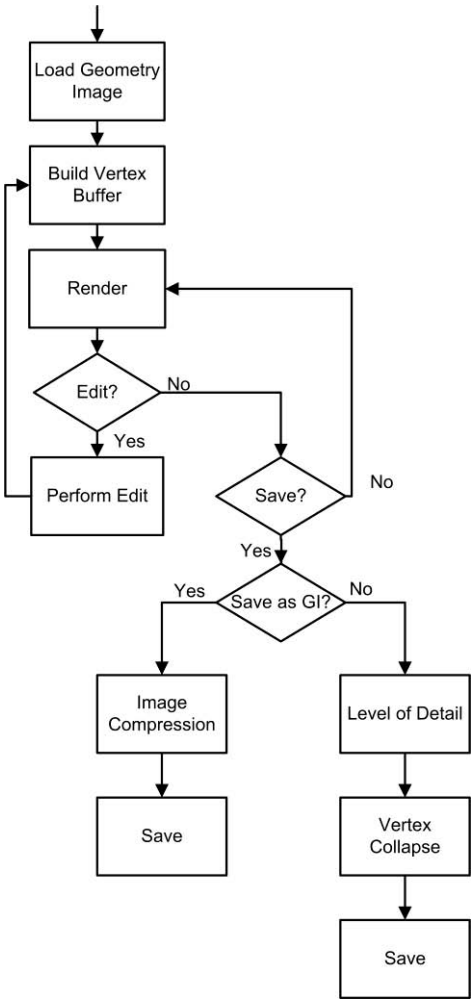

Figure 15: Framework Overview

3.1 User Interaction

The user is initially presented with a base mesh which is to be manipulated on screen. The view of the model can be rotated and scaled freely. Once the user has selected a brush style, strength and size the manipulation is performed by clicking and dragging the mouse on the surface of the mesh. During these editing actions the mesh is edited and updated on screen in real time and the user can seamlessly transition between editing different properties of the mesh such as shape, colour and other surface properties. The user simply sees the model and applies the desired changes.

The mesh editing framework takes care of hit detection on the model, surface deformation, surface integrity, loading, saving and rendering of the models. The entire user

interaction is at a high level of abstraction where the focus is not on vertex positions and texture coordinates. The interactions with the system centre around manipulating the shape, colour and surface properties with simple brush strokes.

3.2 Geometry Image

The geometry image itself is a 128 bit floating point texture format which is supported in shader model 4.0 hardware. This texture format creates 4 channels with a 32 bit float in each channel. The red, green and blue channels of the geometry image are used to hold the xyz position of the mesh in object space. The alpha channel of the image is left free to store other information about the mesh, such as mesh saliency or used as a mask for editing effects. The mesh topology is implied where the position in the geometry image is equal to the texture coordinates. Figure 17 (p.29) shows the mesh topology used where the grey grid represents the pixels of the geometry image. The black lattice overlaid represents the topology of the mesh structure which is created from the geometry image.

Due to the structure of the geometry image there is only a small percentage of vertices/texels which are considered as special cases and require attention to avoid seaming issues. The formula for calculating the percentage of special cases vertices for an n^2 geometry image is

$$\frac{400(n-1)}{n^2}$$

When the number of special cases required in [14] is compared to the standard octahedral shape proposed by [11] the results are practically the same.

Number of Polygons	Subdivision	Geometry Image
65536	1.5625	1.556396
262144	0.78125	0.779724
1048576	0.390625	0.390244
4194304	0.195313	0.195217
16777216	0.097656	0.097632

Figure 16: Comparison of Special case %

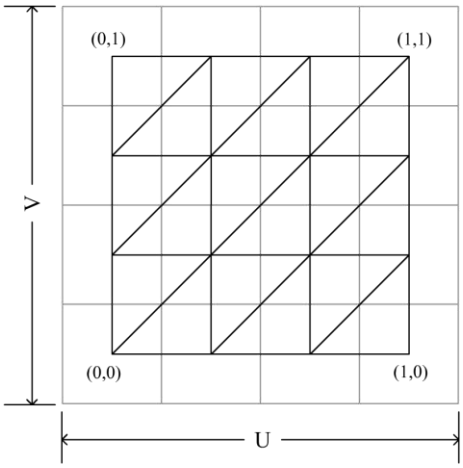

Figure 17: Mesh Topology

In this research a 32 bit per channel floating point representation was used for the geometry images. Each pixel/vertex then consumes 96 bits of memory to store the x,y,z coordinates. This provided sufficient detail to model smoothly and accurately. When using a standard 8 bit per channel format noticeable banding is visible and the values must be clamped between 0 and 1. Figure 18 compares an 8 bit geometry image against a 32 bit. The 32 bit per channel geometry image can also stretch past the range 0-1 in either direction as it is a signed format.

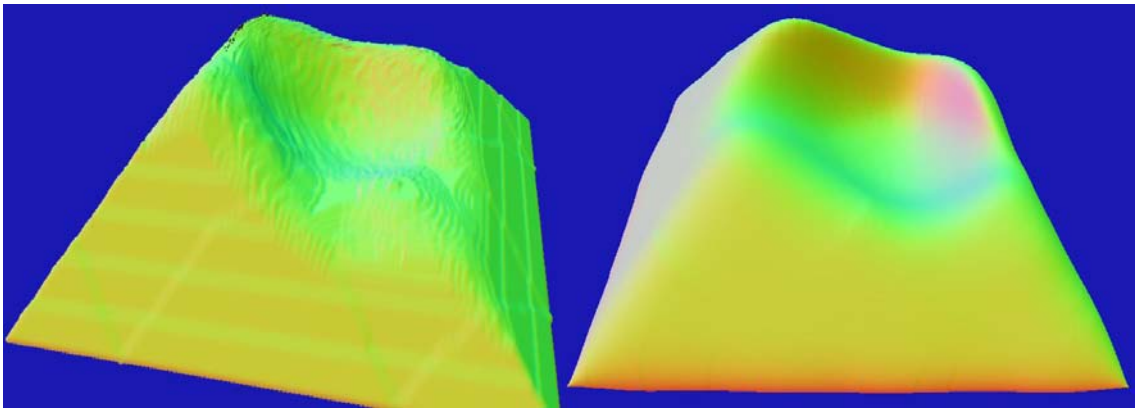

Figure 18: 8 bit vs. 32 bit

3.3 Rendering

To render a geometry image, a vertex buffer needs to be created and stored in video memory. First a dummy buffer is created which contains the texture coordinates for the vertices. The dummy duffer is run through a vertex shader and a geometry shader then it is

streamed out back into video memory. The vertex shader samples the geometry image based upon the texture coordinates provided for each vertex.

```
      VS_OUTPUT vertexShader( VS_INPUT input )
      {
            VS_OUTPUT output = (VS_OUTPUT)0;

            output.Pos = GeometryImage.SampleLevel(samNoMip,
                                          input.texCoord, 0);

            // Ensure the w coordinate is 1 for matrix
            // multiplications
            output.Pos.w = 1;

            // Pass on texture coordinates
            output.texCoord = input.texCoord;

            return output;
      }

      [maxvertexcount(3)]
      void GeometryShader( triangle VS_OUTPUT input[3], inout
TriangleStream<VS_OUTPUT> TriStream )
      {
            VS_OUTPUT output;

            [unroll]for (int i = 0; i < 3; i ++)
            {
                  output.Pos = input[i].Pos;
                  output.texCoord = input[i].texCoord;
                  TriStream.Append( output );
            }
            TriStream.RestartStrip();
      }
```

Code Listing 1: Vertex & Geometry Shader to stream out Geometry Image data

Code listing 1 shows the vertex and geometry shader code for generating a vertex buffer from the dummy texture coordinate buffer and the input geometry image. The geometry shader is a simple pass through shader. The maximum vertex output is set to 3 since no geometry is added to the mesh. Since all outputs from the geometry shader are in strip form, in this case a triangle strip, the TriStream.RestartStrip() command is used. The size of the dummy texture coordinate buffer could be reduced at the cost of speed when creating the geometry image mesh. A lower resolution could be used and the geometry shader could

tessellate the triangles and sample the geometry image for position. Also triangle strips could be used rather than triangle lists to reduce the size of the dummy texture coordinate buffer. These approaches were not used for clarity.

Once the vertex buffer has been streamed out back into video memory it can be rendered in the same way as any other mesh. When rendering vertex buffers in DirectX 10 the auto draw command can be used when the size of the buffer is unknown. This is usually because the buffer has been created on the GPU and stored using stream out. The auto draw command has a slight performance cost and because the size of the vertex buffer is known in this case there is no need to use the auto draw option for rendering.

As most rendering and editing operations require the normals, the next step is to calculate a normal map. The work in [9] stated that the normal map would need to be twice the resolution of the geometry image as it samples more detail from the original mesh. In our framework the geometry image is the original mesh and therefore the normal map remains the same size as the geometry image. A simple pixel shader is run using the geometry image as the input.

```
float offset = 1.0/GeometryImageSize;
float2 texA = input.texCoord.xy;
float2 texB = texA + float2(offset, 0);
float2 texC = texA + float2(0, offset);

texB = texWrap (texB);  // Discussed later in chapter
texC = texWrap (texC);

float4 posA = GeometryImage.Sample(samLinear, texA);
float4 posB = GeometryImage.Sample(samLinear, texB);
float4 posC = GeometryImage.Sample(samLinear, texC);

// Create the Up and Right vectors
float4 vectorA = posB – posA;
float4 vectorB = posC – posA;

float4 outNormal = cross(vectorA, vectorB);
return normalize(outNormal);
```

Code Listing 2: Code to generate an object space normal from a Geometry Image

Code listing 2 shows the pixel shader code required to generate the object space normals for a geometry image. For this a single quad is rendered on screen and the resulting image is stored as the object space normal map. Linear texture sampling is used here as the

size of the offset can be altered. Using an offset of one geometry image Texel creates the correct normals, however if the size is altered slightly then the linear sampling can be used to create softer normals. This is due to the averaging of vertex position and hence smoother edges are created.

'Normalize' and 'cross' are inbuilt HLSL intrinsic functions which normalize the input value and calculate the cross product respectively. The function 'texWrap' is used to ensure correct continuity around the seams of the geometry image. This function is further explained in detail in code listing 9 (p.41), section 3.7

3.4 Editing

To edit the geometry image a post process style of texture editing is used, where two triangles are rendered to form a quad on screen and the texture editing is done in the pixel shader. The first step is to locate the area of the geometry image to manipulate. To do this the texture coordinates of the geometry image are used. When editing, the geometry image is rendered to the back buffer outputting only the texture coordinates. The texture coordinate value is read from the back buffer at the mouse cursor position and this is used as the editing point. This process will be further explained in chapter 3.6.

Now that the editing point has been determined the geometry image is to be manipulated. The render target is changed to a temporary render target which has the same properties as the geometry image. A full screen quad is rendered to that buffer and the selected pixel shader manipulates the geometry image. Once the pass is complete the contents of the temporary render target are copied back into the geometry image texture. This is done because a resource cannot be used as both input and output at the same time. Another option would be to alternate between the two textures which would remove the need to copy the temporary render targets contents to the original geometry image. However careful management of the two textures would be required to ensure they are not incorrectly written to.

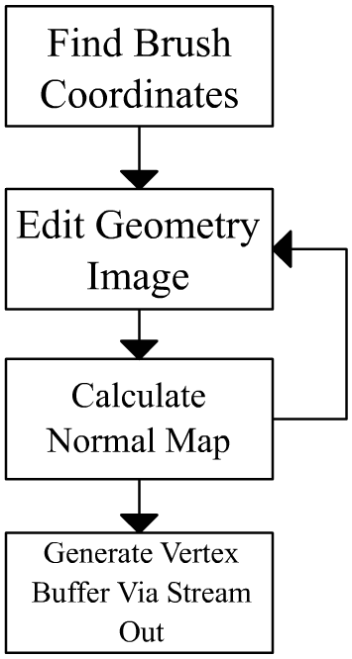

Figure 19: Overview of editing

Once the geometry image has been altered it is time to calculate the normal map. This is done in the same post processing style as the geometry image is edited; however there is no need for a temporary render target. The render target is set to the normal map and the geometry image is used as input. The normal map is then recalculated in the same way as when building the first normal map in code listing 2 (p 31).

Once the mesh has been manipulated it may require seaming to maintain surface integrity near the edges of the geometry image. This will be explained in further detail in chapter 3.7.

The previous two steps can be repeated a number of times to combine different effects. Once the different operations are completed the geometry image mesh is rebuilt. This is the same step used when rendering for the first time, and the result is put into the existing buffer for the geometry image. The resulting normal map is in object space, not tangent space like most normal maps. This means the normal map needs to be 32 bits per channel, however there is now no need to calculate tangent and bi-normal information. The rendering shader only needs to sample the surface normal from the normal map and then multiply by the current world matrix to obtain the surface normal.

3.5 Improved Detail Version

Upon analysis of the existing framework it became apparent that the normal map generation was redundant. During each editing operation the normal map for the geometry image was calculated to improve the speed of rendering. However it was the editing stages which were the bottleneck, not the rendering. The normals could be calculated from the geometry image within the pixel shader for rendering or editing. A helper function was created which takes in the texture coordinates and outputs the normal for that point. The stage for creating the normal map is no longer required, freeing video memory and increasing the speed of editing. Code listing 3 shows the helper function which is similar to code listing 2 (p31). The normalFromTexCoord function can be called during rendering or editing shaders rather than generating and sampling a normal map.

```
float3 normalFromTexCoord(float2 tex)
{
      float2 texA = texWrap(tex);
      float2 texB = texA + float2(
                              1.0/GeometryImageSize,0);
      float2 texC = texA + float2(
                              0,1.0/GeometryImageSize);

      // wrap texture coordinates B and C
      texB = texWrap(texB);
      texC = texWrap(texC);

      // Sample vertex positions
      float3 a = GeometryImage.SampleLevel(samLinear,
                                          texA, 0);
      float3 b = GeometryImage.SampleLevel(samLinear,
                                          texB, 0 );
      float3 c = GeometryImage.SampleLevel(samLinear,
                                          texC, 0 );

      // Calculate vectors
      float3 v1 = b-a;
      float3 v2 = c-a;

      float3 currentNormal = cross(v1,v2);
      return normalize(currentNormal);
}
```

Code Listing 3: Function to calculate normal based on texture coordinates

Another bottleneck during editing was creating the new vertex buffer from the geometry image. This was reduced through introducing automatic level of detail rendering during the editing stages. By creating a dummy texture coordinate buffer which samples at a lower resolution than that of the geometry image, the editing time was reduced. Due to the spatial coherency of most geometry images, the main geometric detail is preserved when viewing a lower level of detail. The high frequency data is still visible as the original geometry image is used for the surface normal calculations. With this new framework in place the geometry image resolution could be increased from 256^2 to 1024^2 with editing operations still working in real time. This gives the illusion of working with 2,093,058 polygons while only viewing 130,050 of them. The size of the geometry image could be scaled up to 4096^2 which could be rendered on a 256^2 base mesh in real time. This provides a simulated 33 million polygons of which most end up creating surface normal detail. At this resolution the editing speed became very slow, less than one frame per second. This speed is purely dependant on the speed of the GPU and will improve as faster hardware is produced. Figure 20 shows the overview of the editing framework with the normal map calculation removed and LOD rendering added.

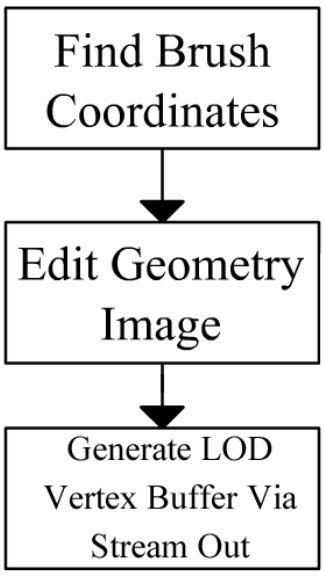

Figure 20: Overview of editing optimized framework

The additional surface normal information adds more detail to the mesh. Figure 21 (p.36) shows a comparison of 256, 1024 and 2048 sized geometry images from left to right all being rendered on a 256 sized base mesh. The detail is most visible when the specular component is high, as shown by the white highlights visible in the 2048 geometry image. The mesh editing stages can now be scaled to run on different levels of hardware.

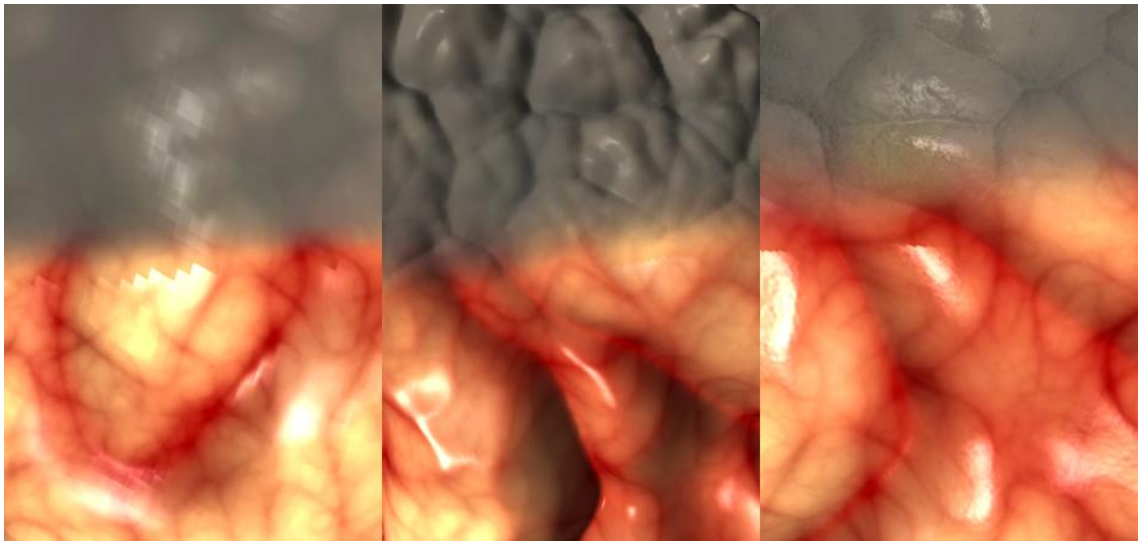

Figure 21: Comparison of geometry image sizes

3.6 Editing techniques

Editing effects can be applied globally or locally. Determining the area of effect for global effects is trivial but local effects can be applied in a variety of ways. For local effects the first step however is to locate which part of the geometry image the user wishes to manipulate. To do this the geometry image is rendered to the back buffer with the texture coordinates as the output colour. For geometry images smaller than 256^2 the standard 8 bit per channel back buffer is sufficient as each texel will map to a unique texture coordinate. However when manipulating larger geometry images a higher precision format is required for the back buffer to obtain per texel/vertex accuracy. This is because geometry image texels will be assigned the same texture coordinates over multiple vertices. Once rendered a single pixel is read from the back buffer into main memory to determine the UV coordinates of the brush on the geometry image. This is the only stage during editing that data is read back from the GPU to the CPU and being only a single pixel, the overhead is very low.

Once located the UV coordinates representing the brush position are sent to the pixel shader along with the brush radius and strength.

```
.x  // U coordinate of centre of brush on geometry image
.y  // V coordinate of centre of brush on geometry image
.z  // Radius of the brush
.w  // Strength of the brush
```

Code Listing 4: Structure of Brush information

There are two main ways to determine the brushes area of effect. The first is based on the UV coordinates of the geometry image and example is shown in code listing 5. This is a simple method and the result of the actual area the brush effects will vary depending on the amount of surface stretch on the geometry image. This style of brush interaction is not preferred because of the unpredictable nature of the brush on the mesh surface. In code listing 6 (p.38) the distance function refers to an intrinsic HLSL function which returns the distance between any two vectors of the same length. When sampling a texture map, the sampler state 'samNoMip' refers to sampling the geometry image with no mip-mapping and UV coordinates clamped.

```
float dist = distance(input.texCoord.xy, brushInfo.xy);
if (dist < brushInfo.z)
{
      // Do brush manipulation here
      return result;
}else{
      return GeometryImage.Sample(samNoMip,
                                  input.texCoord.xy);
}
```

Code Listing 5: Texture Coordinate based brush size calculation

The second method of brush area determination is the volume based brush which is shown in code listing 6 (p.38). This technique samples the geometry image at both the current vertex being rendered and the point at the centre of the brush. The distance in world space is compared to that of the brush radius which determines the area of effect. The result is that the underlying stretch of the geometry image does not influence the area of effect of the brush.

Any points within the sphere centred on the brush point within the brush radius will be manipulated. This means that both sides of a thin wall will be altered by the brush. This can be used to prevent self intersections but can also be a problem when then effect is unwanted, such as surface painting or smoothing. To work around this a test can be added to ensure the

surface normals are sufficiently close by comparing the dot product of the normals to a threshold value which is shown in code listing 7.

```
float3 currentPoint = GeometryImage.Sample(samNoMip,
                                    input.texCoord.xy);
float3 brushPoint = GeometryImage.Sample(samNoMip,
                                    brushInfo.xy);

float dist = distance(currentPoint, brushPoint);
if ( dist < brushInfo.z)
{
      // Do brush manipulation here
      return result;
}else{
      return currentPoint;
}
```

Code Listing 6: Volume based brush size calculation

```
float3 currentPoint = GeometryImage.Sample(samNoMip,
                                    input.texCoord.xy);
float3 brushPoint = GeometryImage.Sample(samNoMip,
                                    brushInfo.xy);

float dist = distance(currentPoint, brushPoint);
if ( dist < brushInfo.z)
{
      float3 currentNormal = normalFromTexCoord(
                                    input.texCoord.xy);
      float3 brushNormal = normalFromTexCoord(
                                          brushInfo.xy);

      float dotProduct = dot(currentNormal, brushNormal);
      if (dotProduct > threshold )
      {
            // Do brush manipulation here
            return result;
      }else{
            return currentPoint;
      }

}else{
      return currentPoint;
}
```

Code Listing 7: Volume and surface angle based brush size calculation

To create a soft edge on the brush a linear interpolation is used once the result has been found. The HLSL intrinsic function 'lerp' provides this functionality. The result from the brush calculation is linearly interpolated against the original point, based upon the distance from the brush point divided by the brush radius. Code listing 8 shows how this works to create soft edges on brush effects.

```
// once result calculated
return lerp(result, currentPoint, dist/BrushInfo.z );
```

Code Listing 8: Soft Brush linear interpolation

Another option is to use a 1D texture to encode the strength falloff of the brush. To do this the distance from the brush centre is divided by the brush radius to determine the U coordinate for the lookup texture.

3.7 Seaming

Some editing operations can cause discontinuities along the seams of the geometry image. There are two approaches to solve this problem; the first is to create special cases in all of the editing operations to ensure that the seams match. The second approach is to create a second seaming pass to ensure the seams are joined after each editing operation. The second approach avoids redundant repetition of code in all of the editing shaders.

The seaming shader simply takes the edges and shifts to the average position of itself and its counterpart on the other side of the seam. Figure 22 shows how the edges of the mesh join up, with the corners all meeting in the same place.

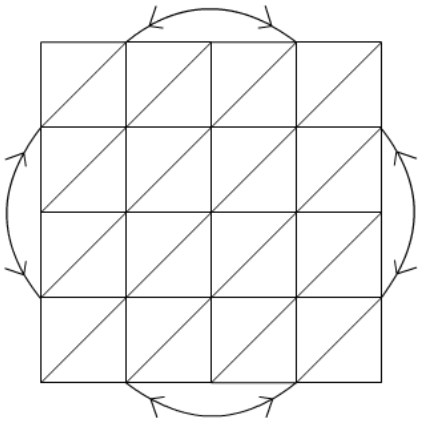

Figure 22: Showing how the seams join

Another option for seaming is to add extra geometry to stitch the seams together. This technique was not chosen however because there would be no texture map assigned to that region. One of the benefits of geometry images is a 1:1 mapping of the texture map to the geometry.

The other issue with the seams is when attempting to sample outside of the range 0-1. The existing rules which can be applied via the sampling state such as repeating or clamping the texture do not apply to the octahedral mapping used. A helper function is required which ensures that any attempts to sample the geometry image outside the range 0-1 is altered to select the correct value within the range 0-1. Figure 23 shows how the octahedron shape unfolds and is repeated over a 2D plane.

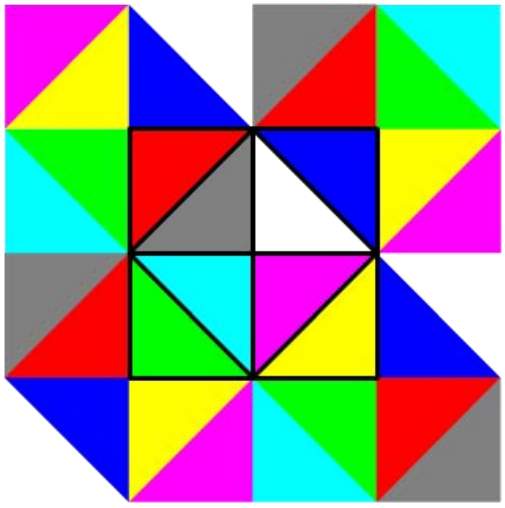

Figure 23: Wrapping

The wrapping function tests for texture coordinates which are out of bounds and wraps it back to the appropriate position within the range 0 -1. Code listing 9 (p.41) shows a section of this function, covering some of the special cases. All of the special cases are a variation of what is shown.

```
float2 texWrap( float2 input )
{
	float2 output = 0;
	if ( input.x > 1 )
	{
		if ( input.y > 1)
		{
			output = input - 1;
			return output;
		}
		if ( input.y < 0 )
		{
			output.y = 1 + input.y;
			output.x = input.x - 1;
			return output;
		}
}
// now x < 1 and between 0 - 1
output.x = 1 - ( 1 - input.x );
output.y = 1 - input.y;
return output;
}
if ( input.x < 0 )
	///
if ( input.y > 1 )
	///
if ( input.y < 0 )
	//
```

Code Listing 9: Custom Texture wrapping function

3.8 Smooth

The smoothing operation translates to a 2D image blur. This is done by sampling a selection of surrounding values and returning the average. To ensure the smooth works correctly around the edges of the geometry image the texture wrapping helper function is used. Code listing 10 (p.42) is the smoothing code, assuming that the area of affect for the brush has been calculated.

The unroll hint allows the compiler to unroll the *for* loops to increase performance. The 'samLinear' sampling state allows for linear interpolation when the texture coordinates do not match exactly to a single texel. In the above example, all samples will occur directly on a single texel. However if the spacing of the samples were changed then this would slightly alter the smoothing results.

```
int count = 0;
float4 total = 0;
float2 xoffset = float2(1.0/GeometryImageSize,0);
float2 yoffset = float2(0,1.0/GeometryImageSize);

[unroll]for (int i = -2; i <= 2; i++)
{
      [unroll]for (int j = -2; j <= 2; j++)
      {
            float2 newTex = input.texCoord.xy + i*xoffset
                                              + j*yoffset;
            newTex = texWrap(newTex);
            count++;
            total += GeometryImage.SampleLevel(samLinear,
                                              newTex,0);
      }
}
float4 result = total / count;
```

Code Listing 10: Surface Smoothing Brush

3.9 Push/Pull

The push and pull effect is crucial to manipulate a mesh. The effect can be altered in subtle ways which create a different behaviour to the brush. The standard version is to pull or push along the surface normal of the brush point. This is done by finding the surface normal of the point where the brush is and adding a fraction of that normal to the xyz value of the vertices within the brush radius. The strength of the brush is determined by how much of the brush position normal is added. This pulls or pushes the surface in a uniform direction. A 1D texture can be used to encode different fall off values for the brush or just a simple linear interpolation from full strength in the centre to no effect on the brush radius.

Another simple change is to use each vertex normal rather than the brush point normal. The effect is to balloon the current region rather than pull or push in a straight line. When used in excess, this brush can result in large amounts of surface stretch which is undesirable.

Code listing 11 shows how the push/pull brush is calculated. The pull and push brush can easily prevent self intersection which can otherwise be a difficult task. Since the brush manipulates all pixels within the brushes radius, if one side of a thin wall is pushed the opposite side of the wall will also be pushed in the same direction. When using a linear fall off based on distance from the brush point it is theoretically impossible to make the mesh self intersect. This reinforces the feel that the user is interacting with a solid object, rather than manipulating individual vertices and polygons.

```
// Once brush region identified

// For balloning brush
// float3 brushNormal = normalFromTexCoord(
                                    input.texCoord.xy);

// For straight brush
float3 brushNormal = normalFromTexCoord(BrushInfo.xy);

float4 shiftAmount = 0;
// suitable brush strengths here are
// between 0.0001 - 0.1

shiftAmount.xyz = brushNormal * BrushInfo.w;

float4 outResult = CurrentPoint + shiftAmount;
```

Code Listing 11: Normal Based Shifting Brush

3.10 Splatting

A more advanced effect is to transfer data from a texture source onto the surface of the geometry image. Using this method the geometry, diffuse and specular information can be edited in a single action by the user to transfer materials onto the surface of the mesh.

The difficulty with transferring data onto the geometry image is determining the UV coordinates to use in the source data texture. The naive approach is to simply use the existing texture coordinates of the geometry image. The advantage of this approach is that when tiling textures are used, they will join perfectly around the seams of the geometry image. However this also means that the texture transfer is bound to the stretch of the geometry image. The result is that the texture transfer is unpredictable in the way it behaves on the surface. Sometimes the transferred detail will be heavily stretched or compressed.

To solve this problem the texture was projected onto the surface of the model. The projection can originate from any position but the most useful is from the camera towards the centre of the model. The centre of the mesh is assumed to be at position 0.5,0.5,0.5 (x,y,z) based on the assumption that the mesh will be approximately contained in the bounding box with diagonals 0,0,0 to 1,1,1. Code listing 12 shows how projective texturing is used to assign texture coordinates for splatting surface properties on a geometry image.

```
// Find projective coordinates
float3 projTex = mul(currentPoint,
                              ProjectiveTextureMatrix);

// Sample map for heightmap value
float3 BrushValue =  SplattFeatureMap.Sample(
                        samLinearWrap, projTex).xxx;

// Bring heightmap to range -1 to 1
BrushValue -= 0.5;
BrushValue *= 2.0;

// Find normal direction to move in
float3 normal = normalFromTexCoord(BrushInfo.xy);

// calculate shift value
float3 shiftValue = BrushValue * normal * BrushInfo.w;

float4 outValue = currentPoint;
outValue.xyz += shiftValue;
```

Code Listing 12: Projective splatting Brush

The texture projection approach allowed the texture information to be transferred independently of the underlying texture coordinates. The drawback of this method is that it becomes difficult to surround the model completely without visible seams.

Once created the texture transfer technique can be applied to all attributes of a mesh. Figure 24 shows an example of painting to the geometry, diffuse, specular and reflective layers of a mesh in one user input. The image on the top right shows the feature map and below is the diffuse colour map. The feature map consists of the height map in the red channel, the specular values in the green channel and the reflective component in the blue channel.

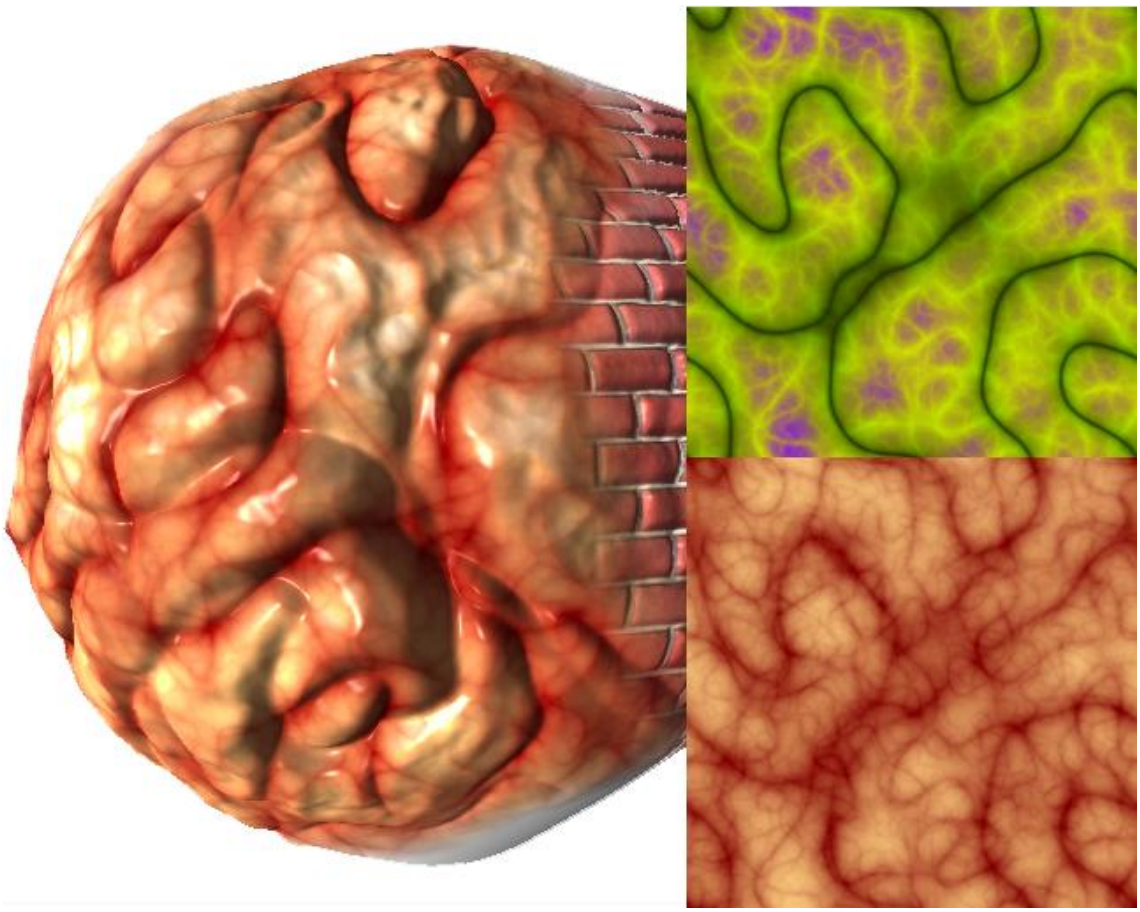

Figure 24: Multi attribute splatting

3.11 Dimple/CSG

This technique demonstrates the use of simple CSG style operations on a mesh. Here a sphere can be added or subtracted from the surface of a mesh. This technique cannot be easily expanded to full CSG style operations, but local operations are possible.

By using a dynamic flow controlled *while* loop the current vertex can be shifted until it is outside of the sphere. Code listing 13 shows an example of shifting all vertices outside of a specified sphere given the centre and radius.

The resolution variable determines the granularity of the while loop and hence controls how smooth the result is. A small value for the resolution will result in a blocky sphere but will be faster to complete. Figure 25 shows the effect of different resolutions used in the while loop. Clockwise from the top left the values are 10, 100, 500, and 1000. The radius of the sphere being added or subtracted also effects the time required to complete the operation.

```
// Once brush region identified
float3 normal = normalFromTexCoord(BrushInfo.xy);

[loop] while ( dist < BrushInfo.xy )
{
      currentPoint.xyz += normal / resolution;
      dist = distance( currentPoint, brushPoint );
}
return currentPoint;
```

Code Listing 13: CSG Brush

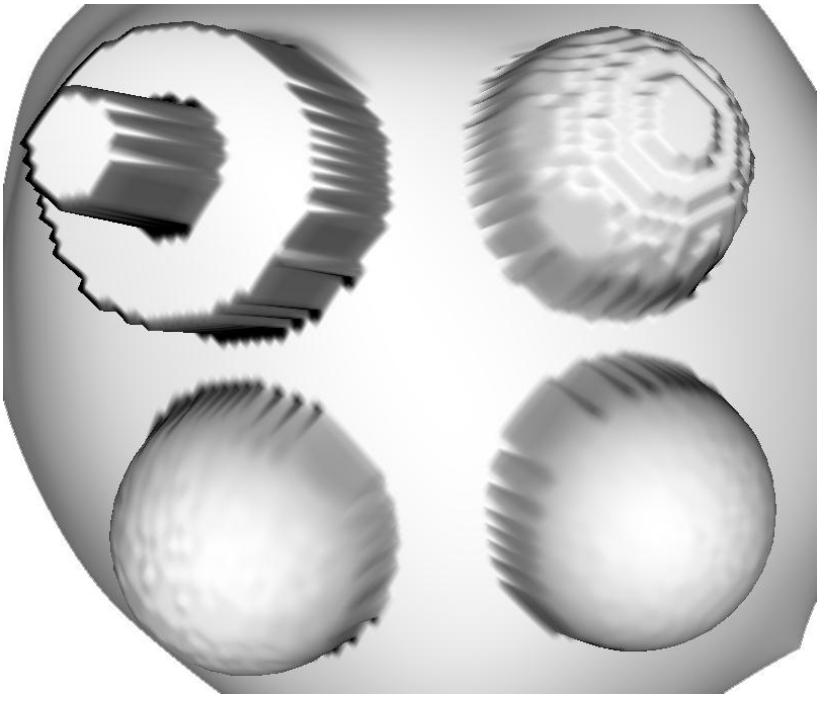

Figure 25: Different resolution for CSG operation

3.12 Summary

In this chapter the structure of the geometry image used has been detailed along with the algorithms required to render and edit the geometry image. A number of techniques combined with sample code have been provided which enable complex editing operations on large meshes in real time by utilizing the highly parallel structure of graphics hardware.

The rendering and editing sections of the proposed framework has shown to be flexible and easy to add new brushes as new pixel shader effects. The framework has also proven to be scalable through the use of level of detail rendering by altering the size of the vertex buffer in use while the geometry image remains the same resolution. These results are shown in chapter 6.

Chapter 4 - Polygon Count Reduction for Automatic Level of Detail

Reducing the polygon count of a mesh is very important to increase the speed of rendering. The proposed geometry image painting framework, discussed in this chapter, by its nature has a very high polygon count. While a model can be rendered while editing, a large scene of full resolution geometry image models will not render at interactive speeds on any consumer hardware available today. So for general interactive rendering the geometry image needs to be converted into a vertex buffer with far fewer polygons while retaining the original shape. The benefit of the high polygon count of the original geometry image is that an object space normal map can be generated on the GPU and stored for use on the reduced polygon mesh.

As identified in Figure 11 (p.23) a very low detail mesh can be used when the model is far away from the viewer, and hence occupying a small region of the screen. Rendering speed is critical for real time rendering so the polygon count needs to be reduced whenever possible to maintain low rendering time. When a model consumes only a small region of the screen it can be replaced with a model with less detail. What is needed is a system to quickly generate different levels of detail for a mesh in real time as the model changes distance from the viewer.

4.1 Mesh Based

The initial work on Level of Detail (LOD) generation was a CPU based piecewise polygonal mesh based approach. This is the typical approach used by many LOD algorithms [54]. The first step is to define the mesh structure to be used in memory as this will affect the speed and memory requirements of any operations on the mesh. Additional data about the mesh can be stored such as neighbour information which can increase speed in some algorithms. However additional data may need to be recalculated each time the mesh is altered. Creating different levels of detail requires that the mesh is constantly changing, so a structure which held only the required information was chosen.

The mesh structure was a standard array based index of triangles, which pointed to arrays of vertices, texture coordinates and normal data. One important design decision was whether or not to allow duplicate entries in the vertex arrays. If no duplicates were permitted then adding vertices, normals or texture coordinates could become a slow process as the entire array would need to be scanned to determine if it already exists. Knowing that there are no duplicate values makes operations such as finding neighbours and deleting faces faster.

```
Structure Triangle
      int Vertex1, int Vertex2, int Vertex3;
      int TextureCoord1, int TextureCoord2, int
                                          TextureCoord3;
      int Normal1, int Normal2, int Normal3;

Structure Vertex
      float x, float y, float z;

Structure TextureCoordinate
      float u, float v;

Structure Mesh
Array of Vertex VertexArray
Array of TextureCoordinate TextureCoordinateArray
Array of Vertex NormalArray
Array of Triangle TriangleArray
```

Code Listing 14: Mesh Structure

Due to having a fixed topology the structure of the mesh is known so it is suitable for local pattern matching techniques to simplify the geometry. To do this a face is selected and tested to ensure the dot product of the surrounding normals is within limits. If the surface is flat enough and the topology matches then the simplification can proceed. Using the techniques described in Chapter 2.12 for polygon reduction through pattern matching, geometry images were tested. The results for pattern matching were highly dependent on the starting location of the algorithm on the mesh. Figure 26 (p.50) shows different iterations of the square based polygon reduction. The top row shows the best case scenario on one side of the mesh, while the bottom row shows the other side, were the pattern matching did not work so well.

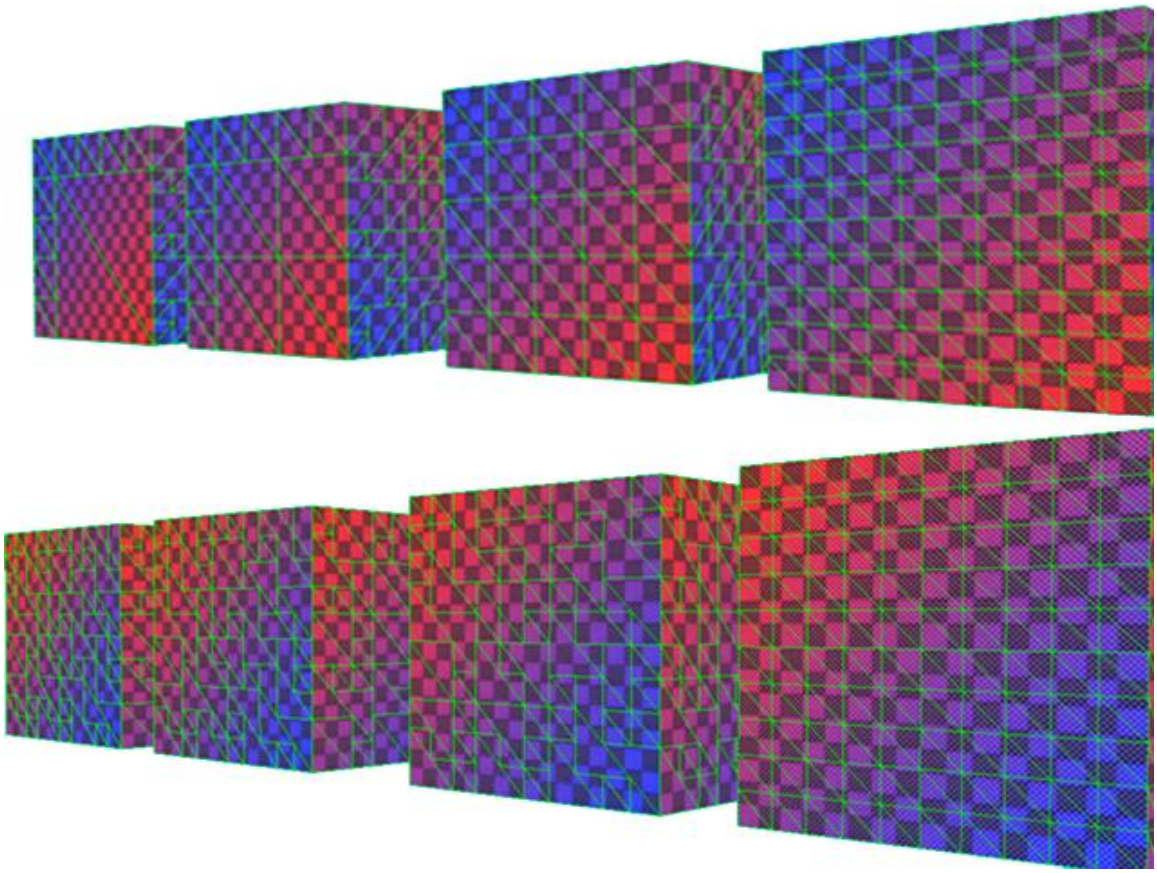

Figure 26: Square based reduction

The vertex collapse was also tested and was found to be far more effective, especially in the regions of the mesh which were associated middle of the edges of the geometry image. These regions would tend to have a high polygon count in a small area. An example of this is shown in Figure 27 (p.51), where we can see multiple vertices folded around a point. In Figure 28 (p.51), we can see the result of a vertex collapse pass, which removes the extra vertices and reduces the polygon count.

This technique can create very different surface normals however this is easily solved. The original geometry image is used to calculate the object space normals rather than the face normals from the reduced mesh.

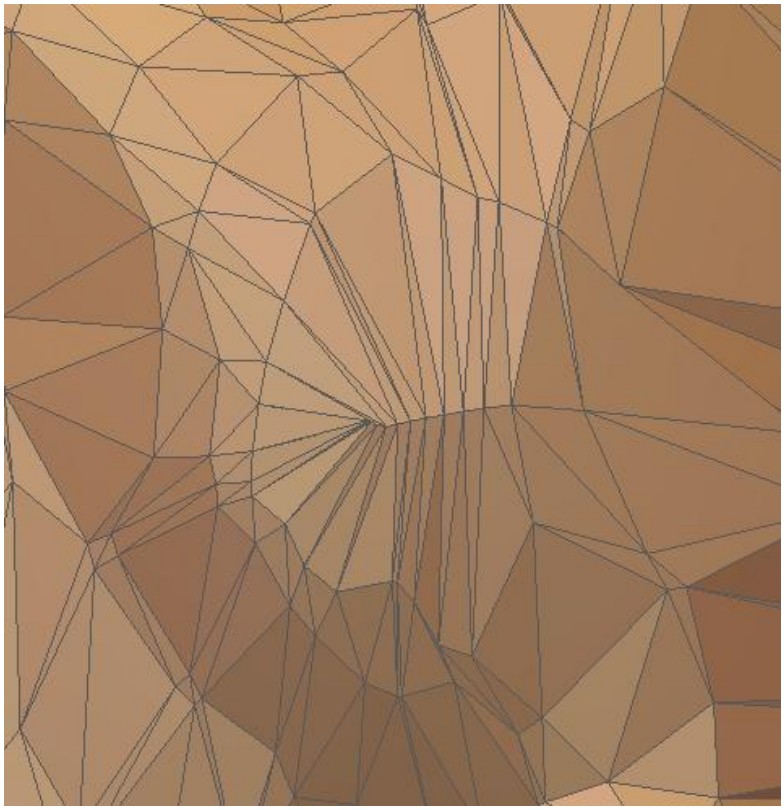

Figure 27: Mesh before Vertex Collapse

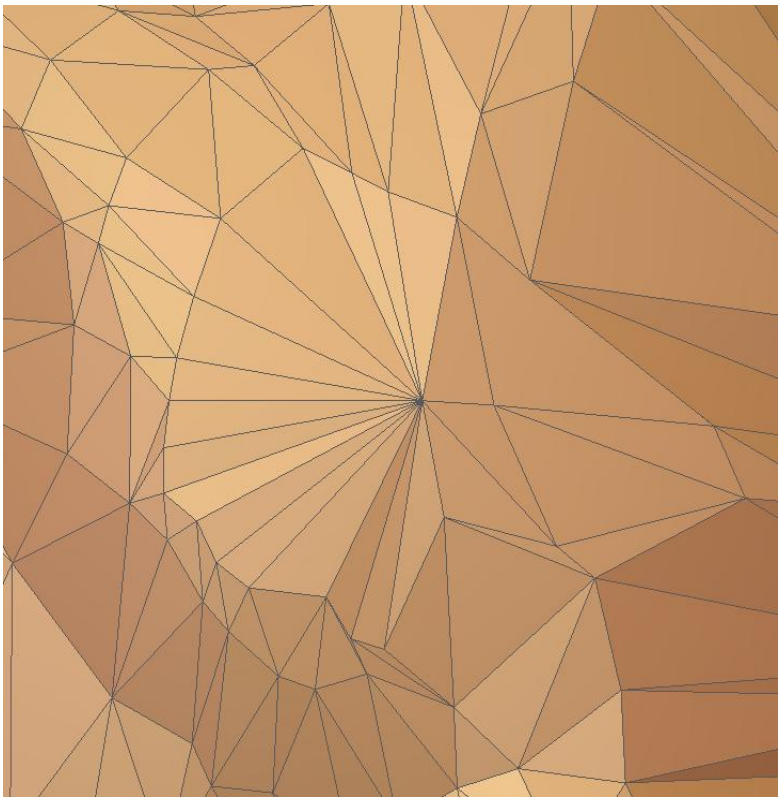

Figure 28: Mesh after Vertex Collapse

4.2 Geometry Shader

Another option for generating different LOD for a mesh was to investigate the geometry shader. The ability to create, alter and discard polygons is a key requirement for generating different LOD meshes and the geometry shader can perform this task. Using the stream out to feed the result back into video memory would permit the geometry shader to iteratively simplify a mesh entirely on the graphics hardware.

The first step is to determine which polygons are suitable for editing. This was accomplished through the use of adjacency information provided to the geometry shader. Figure 3 (p.3) shows the layout of a triangle and the surrounding adjacency information. Once calculated for a mesh the adjacency information can be made available from within the geometry shader. The adjacency information can be calculated using a built in DirectX 10 function. To determine if a face is suitable for removal or simplification, the dot product between the current surface normal and the neighbouring surface normals is evaluated. If the dot product is below the threshold for all neighbours then it is suitable for culling. The dot product result for two vectors ranges between -1.0 and 1.0. Where the dot product is 1 the surface normals are equal, conversely when they are parallel but opposite the result is -1.0. A threshold value should be positive and close to one e.g. 0.9. The lower the threshold value the more aggressive the mesh reduction will be and more polygons will be remove. However if the threshold is set too low the original shape of the mesh will be lost.

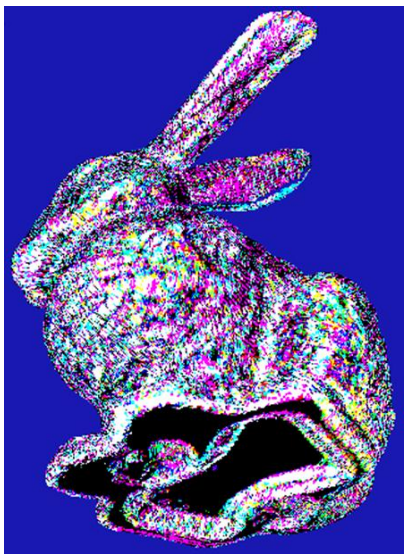

Figure 29: Polygon Culling Suitability

Figure 29 (p.52) shows the suitability of polygons colour coded, where black means suitable, colours mean suitable in some directions and white means curved in all directions. Black polygons on the edge of a black area are to be edited to join the other side while black polygons surrounded in black polygons are to be removed.

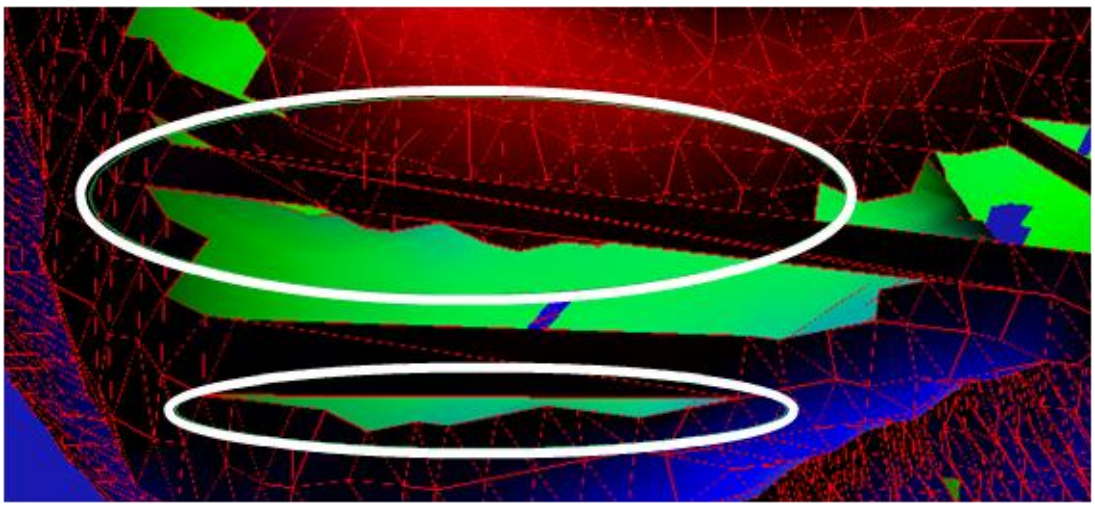

Figure 30: Overlap and Tear

To simplify a mesh the geometry shader first identify if the current polygon is suitable for editing or culling. The suitability is based upon the dot product of the surface normal with neighbouring polygons. If unsuitable, the current face is passed through the shader without change and the shader exits early. If the polygon was found to be at the edge of an area of suitable polygons, it would be stretched to cover the gap and new texture coordinates assigned to the stretched end. If a polygon was found to be inside an area of suitable polygons then it would be discarded.

The problem with this technique was the tears and overlaps that form due to edges not matching perfectly. Figure 30 shows both of these errors, the top white oval shows an overlap caused by the strip crossing other polygons and the bottom oval shows a tear where a line of suitable polygons borders a line of polygons with curvature. It became apparent that this approach would not be suitable, as the problem could not be split and run in parallel. As each reduction takes place the original data needs to be altered.

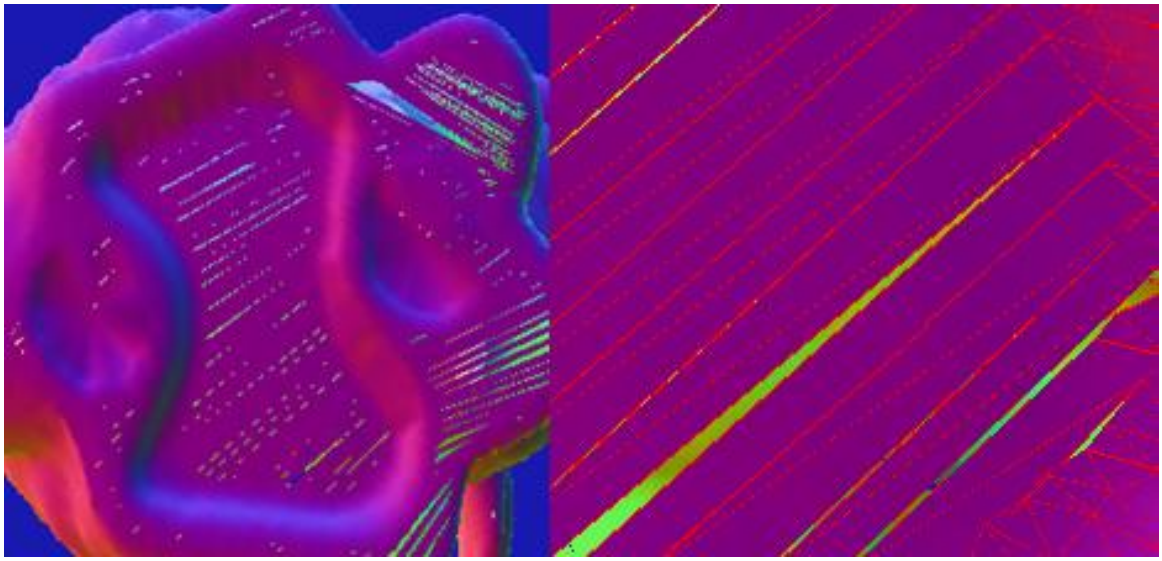

Figure 31: Geometry Shader Decimation

4.3 Geometry Image Based

It was mentioned in [9] that creating different Levels of Detail could be done through a simple mip mapping of the geometry image. In their work the size of the geometry image $(2^n + 1) \times (2^n + 1)$ meant that there was no need to manipulate the seams after the geometry image had been resized. To resize the image they used simple sub sampling which provided good results.

One of the advantages of the geometry image is that the saliency of the mesh can be calculated very easily on the GPU. A pass of the geometry image can add to the alpha channel the saliency of the mesh. For our LOD saliency metric we used the change in surface normal sampled at different distances. This meant that not only small changes in the surface normal are identified but also larger curves, this was similar to the technique used in [29]. There are a range of other factors that could be taken into account such as ambient occlusion, texture map density and polygon size. A number of different approaches were taken when sampling the geometry image for resizing. Among these were average, point sampling, salient weighted average and salient point sampling. From these techniques the salient based point sampling provided the best results. The salient point sampling simply selected the pixel with the highest saliency from those which were to be down sampled.

When resizing the geometry image to arbitrary sizes holes could form along the seams of the geometry image. To solve this, a seaming pass was added after the resize had taken place. To join the seams, the edges were averaged with their counterpart.

Seam carving was a technique introduced by [53] which allowed for the dynamic resizing of images while maintaining the important aspects of the scene. The algorithm worked by pre-calculating the paths of least change through an image, then as the image was resized these paths would be removed.

This concept seemed to lend itself to the dynamic resizing of geometry images. The initial work was to cut single rows and columns from the image, rather than a path. It became immediately apparent that the implicit texture coordinates no longer functioned when the geometry image was resized. To solve this, the original texture coordinates were paired with the xyz values of the geometry image as it was resized.

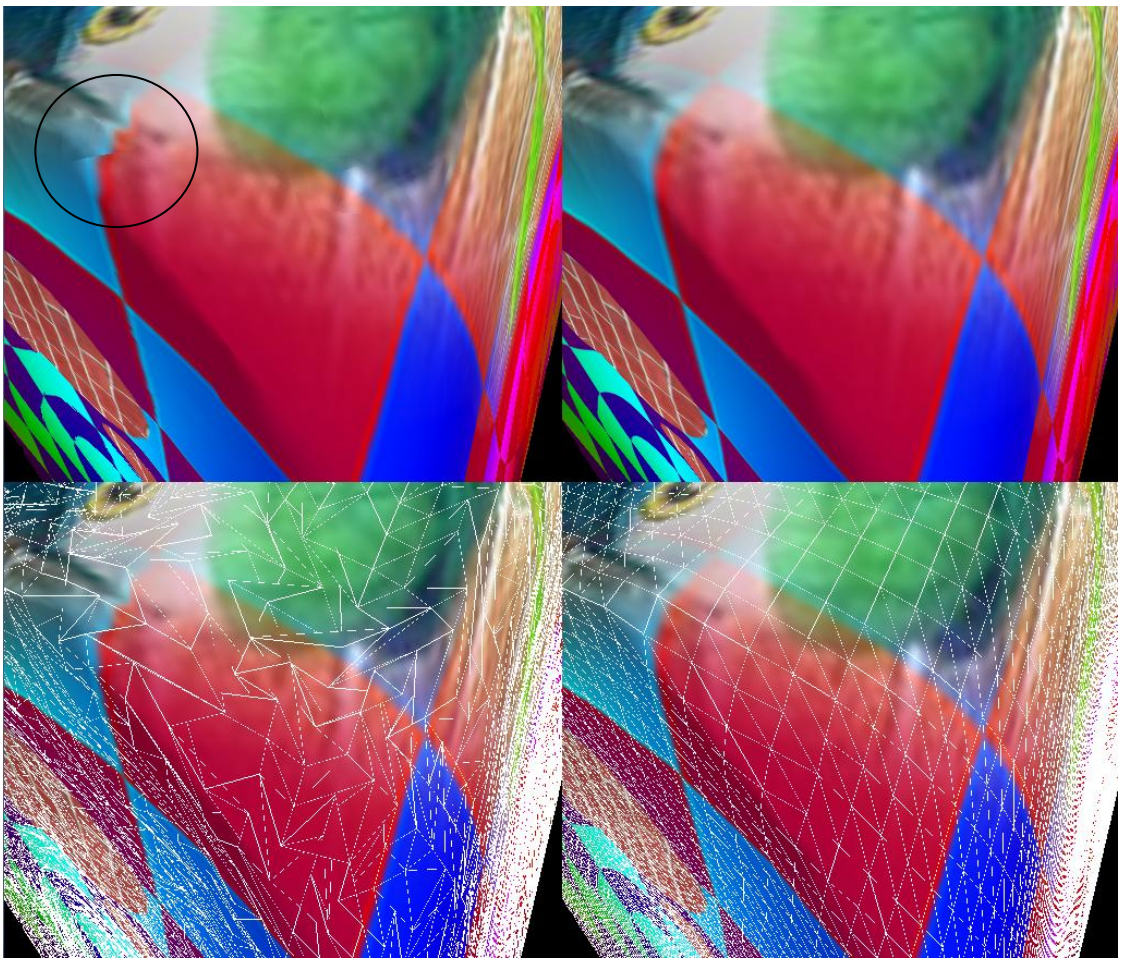

Figure 32: Seam carving approach

There was a problem with this and the quality of the mesh would quickly degrade with polygons overlapping heavily, Figure 32 (p.55). In most cases the result was visually acceptable however artefacts would begin to appear. At the top left of the top left image discontinuities can be seen in the diffuse map. The overlapping of polygons also meant that there were redundant polygons in the mesh, which were mostly or entirely covered by other faces.

Another approach was to simply remove a pixel from each row which had the lowest saliency value. This was also affected by the same problems and overlapping polygons began to form. However as we can see in Figure 33 the smooth areas have a polygon reduction while the areas of curvature are left alone.

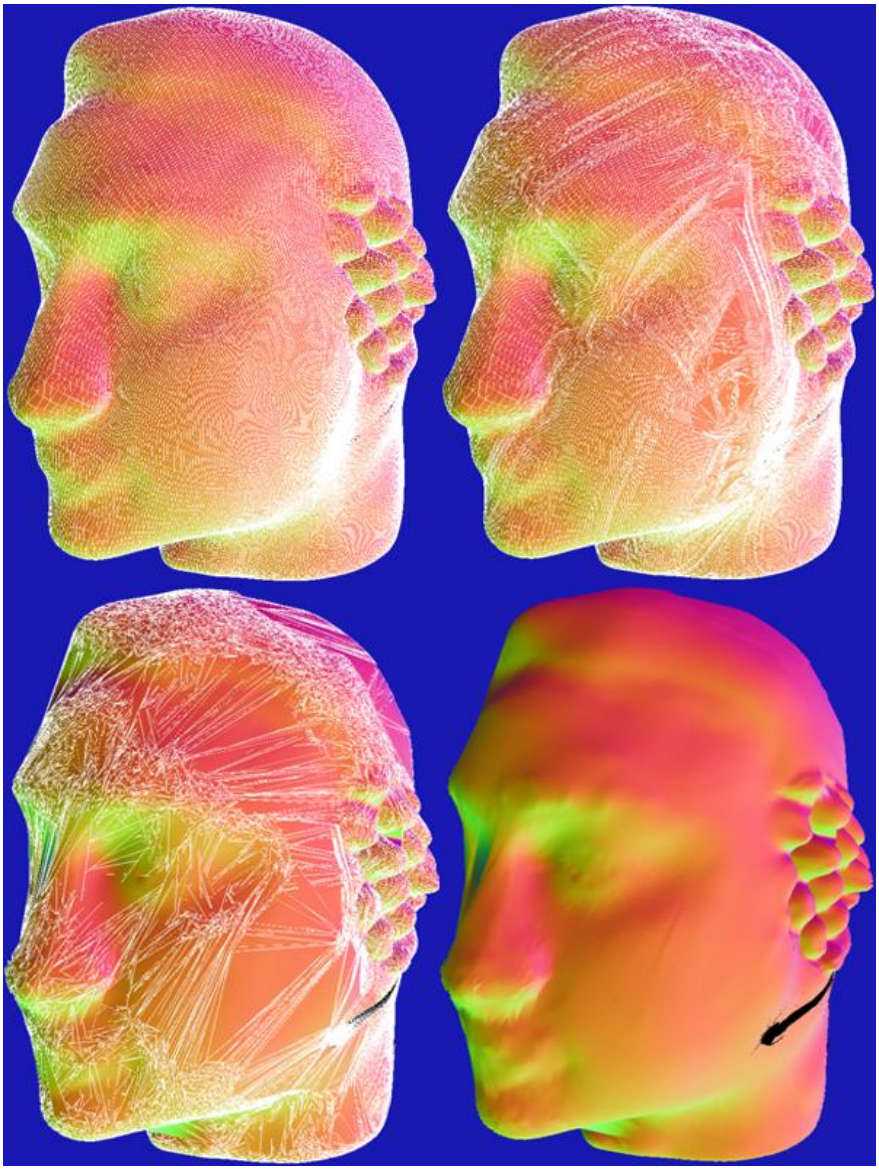

Figure 33: Comparison of single pixel removal

4.4 Final Method

The solution which gave the best balance of speed, quality and polygon count was to combine the geometry image resizing with a CPU based vertex collapse. First the geometry image resizing based upon salient point sampling is done on the mesh to get the base mesh for that level of detail. Because there is a low polygon count in the mesh, the vertex collapse algorithm can then be run on the mesh in a small amount of time.

Other options which were not investigated were the use of the GPU to perform the vertex collapse. A possible method for this would be to assign vertices/texels the same value in a pixel shader if they are sufficiently close. Then any triangle in the geometry shader which two vertices shared the same values could be removed.

4.5 Summary

In this chapter the importance of level of detail rendering was explained. A number of techniques for automatic LOD were explored and compared. The result is a combination of GPU and CPU work to create models with low polygon count and good representation of the original model. There is room for further research in this area and one of these options is detailed.

Chapter 5 - Compression of Geometry Images for storage

Geometry images should be good candidates for compression due to the spatial coherency that is found in most genus-0 models. This results in images which are suitable for traditional photographic image compression techniques. This is a highly researched field and the aim is only to prove that image compression techniques can provide gains over traditional mesh connectivity compression. The work in [13] covered the use of wavelets for geometry image compression so other methods are explored in this research.

In addition to image based methods two alternative techniques were investigated which exploit spatial coherency. One was modelling change with Bezier curves and the other delta encoding combined with custom bitpacking.

The file size of a 256^2 geometry image with 32 bits per channel is 768KB and when stored in DirectX 10's texture format where the alpha channel must be included the size is 1024KB. When using a 1024^2 geometry image the size becomes 12MB which is too large for storage and transmission of a model.

It is preferred to keep the model in its original geometry image format for storage and transmission as opposed to conversion to a reduced level of detail polygonal mesh and associated normal map. If the normal map is going to be stored then the original geometry image should be used instead as this will allow for later editing or creation of the full detail mesh for extreme close up or offline rendering. Since the creation of lower LOD meshes is simple and fast there is also no need to pre calculate them, this can be done at start-up or even during runtime.

Compression on standard geometry images should obtain better results than what would be found with the geometry image from [14] due to the spatial coherency.

5.1 Existing file formats

From the field of High Dynamic Range imaging (HDR) there are a number of file formats available which are designed to store and compress high precision image data, much

like that found in a geometry image. HDR imaging is concerned with capturing and storing light information which exceeds the standard 8 bit per channel colour range available on computer monitors. There are three established HDR storage formats, HDR, TIFF and EXR as well as the new PHOTO HD which is in beta.

OpenEXR is an open source format developed by Industrial Light and Magic and is designed to compress HDR film images. A custom 16 bit floating point data type is used which is the same as NVIDIA's half type found in its shader language Cg [24]. Full 32 bit per channel data is supported and a new 24 bit format developed by Pixar is also supported. Compression options are a wavelet based method, zip, run length encoding, Pixar's PIX24 and B44. Additional information can also be imbedded into the file such as custom layers with different formats.

When integrating the OpenEXR format with the test implementation there were multiple compatibility problems between the projects, so all file conversion and compression was done through Photoshop. The supplied plugin was rather limited so the third party plugin ProEXR [23] was used. When quantising to the 16 bit floating point format the geometry of the model remained visually the same. However the surface normal information became significantly altered and artefacts became visible. When using either environment mapping or when the diffuse light was low on a smooth surface the normal artefacts become apparent. This can be seen in Figure 34 (p.60) where the smooth transition between light and dark is replaced by noise in the surface normals.

The TIFF file format can store images with 32 bit IEEE floating point data per channel much like the DirectX R32G32B32A32 format but with the option to remove the alpha channel. Just like the DDS format however compression does not work well due to the noise in the mantissa of the floating point data. But large areas of the geometry images used consist of smooth gradients so traditional ZIP compression is still of use and can compress at around 30%.

The HDR format as shown in Figure 35 (p.60) is unsuitable for the task as it does not contain enough precision and quantises far too heavily. The HDR format stores 32 bits per pixel, 8 bits for each RGB channel and then 8 bits for an exponent. The data is then run length encoded and usually achieves around 25% compression [25]. The provided example is a 1024^2 geometry image which is 12,288KB compressed down to 1,572KB. The HDR representation is more suited to HDR images and not for compression of geometry images.

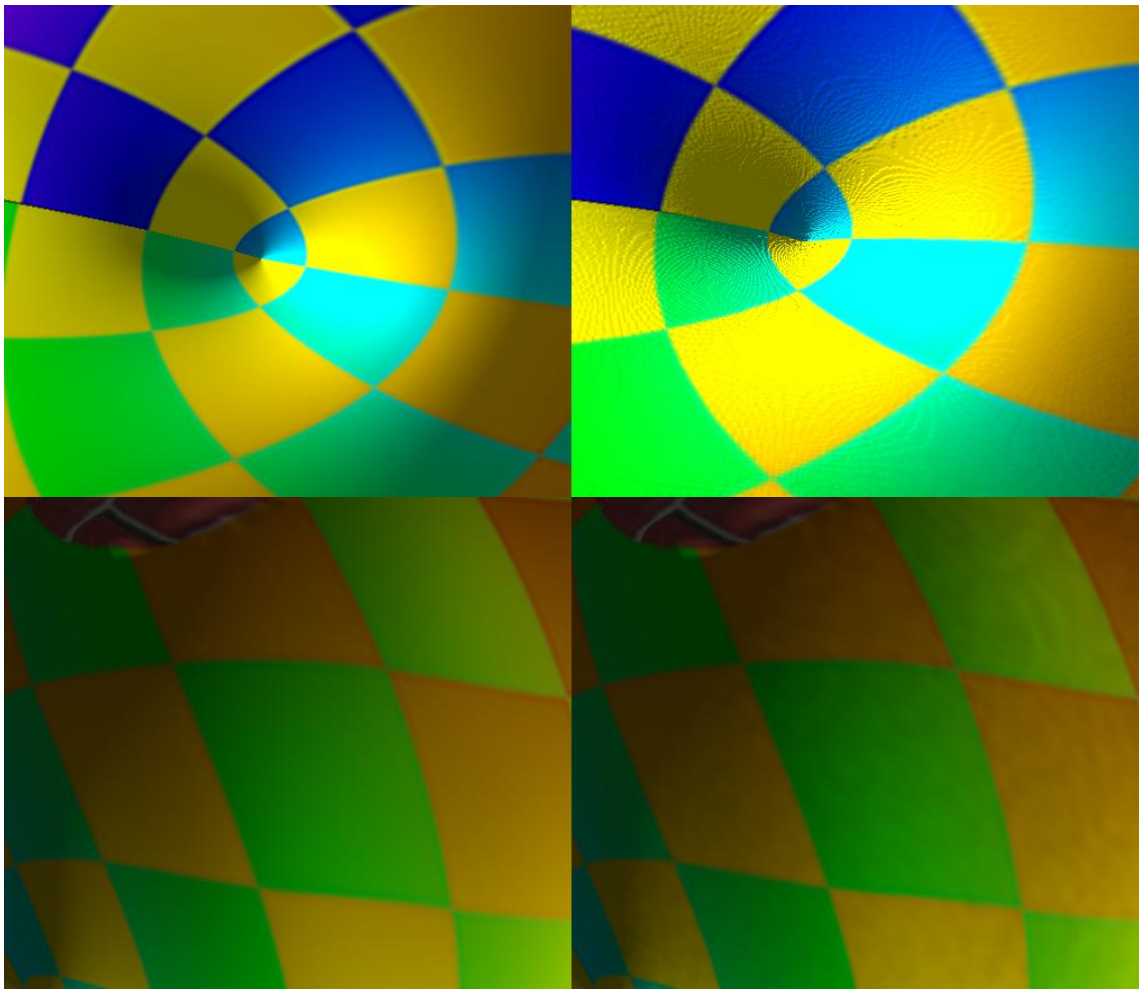

Figure 34: Effect of 16bit compression on Normals

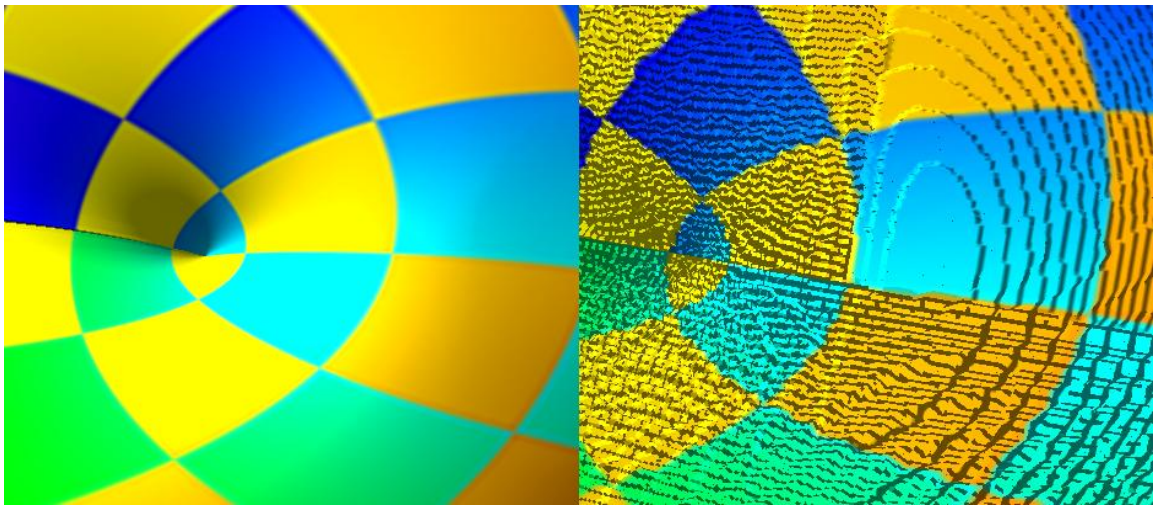

Figure 35: Comparison of HDR image format

Photo HD is a new image format from Microsoft which is currently in beta. This image format has been submitted and accepted to the joint photographic experts group and will become JPEG XR where the XR stands for extended ranges. This is yet to be created however so photo HD is used instead, which is provided as a royalty free image compression method. There are Photoshop plug-ins available which were used for the compression and decompression of the geometry images. Photo HD can be lossless (Figure 36) or lossy (Figure 37) with a range of compression options available. Up to 32 bit floating point per channel precision can be used. The lossless compression causes small visual artefacts on the surface which are only visible on reflective surfaces or when the diffuse light is at a grazing angle, this is shown in the inset of Figure 36.

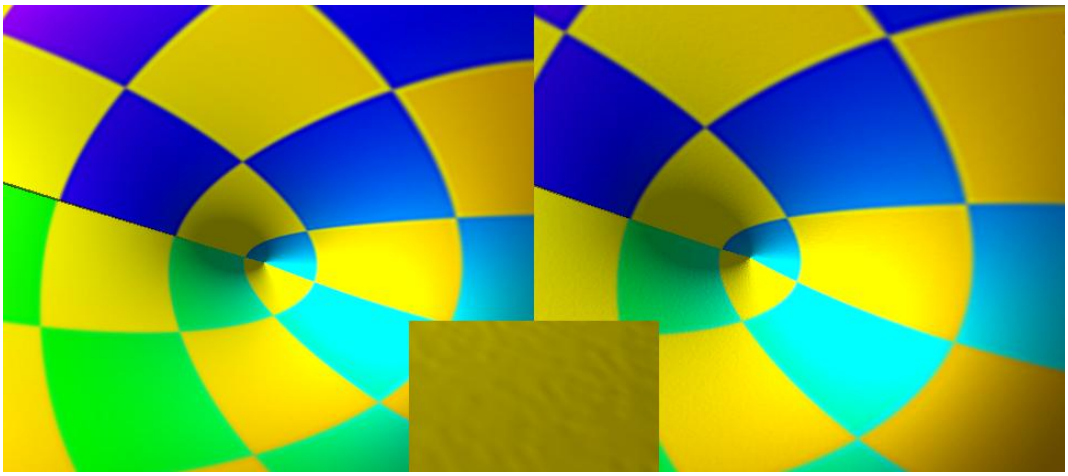

Figure 36: Lossless Photo HD compression

The compression is very good; the original 1024^2 geometry image of file size 12,288KB was used. When using lossless compression the resulting file size was 1,959KB and the lossy compression became 705KB.

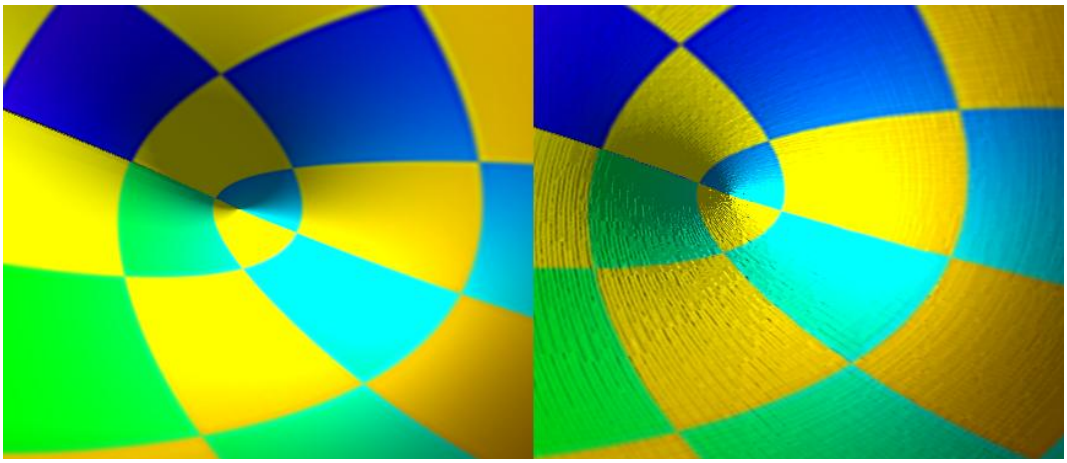

Figure 37: Lossy Photo HD compression

5.2 NVIDIA and ATI compression tools

Image compression tools from both ATI and NVIDIA were tested. Both image compressors could handle 32 bit floating point data, however the compression schemes were lossy and based upon the human visual system. The result is that the compression is not even across all channels and the mesh reproduced from the compressed geometry image was full of artefacts. Compression which is based upon the human visual system preserves the luminance channel and compresses the chromatic layer. This is because the human visual system can more easily detect changes in brightness than colour.

NVIDIA's texture tools were designed to use the GPU to accelerate the speed of compression and decompression. The results from both image compressors were the same, the surface wave artefact propagated across the mesh.

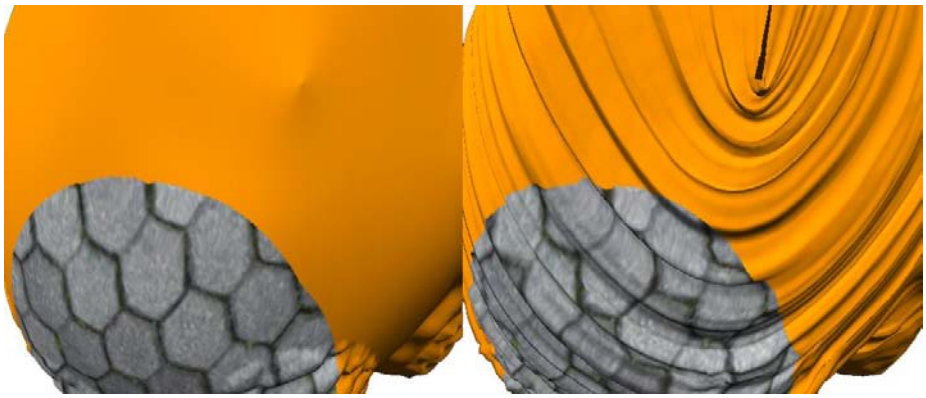

Figure 38: ATI compressinator results

5.3 DCT/ floating point jpeg

The commonly used jpeg image format is based upon using a discrete cosine transform followed by quantization and compression. This lossy compression method however is designed for use on integer data specifically 8 bit integers. For this project a floating point JPEG implementation was created based upon the code found in the libjlm and JLM project from SourceForge. The implementation was altered to work on single precision floating point data rather than double precision. Also the compression scheme was changed to simple WinRar format as well as some bugs removed.

While JPEG is no longer state of the art in image compression it was used in this project due to quantisation issues with the other HDR formats tested. This compression technique was implemented specifically to prove the principle that quality and file size is a scalable trade off. With the floating point JPEG system artefacts quickly become apparent at low bit rates. In particular blocking and fringing artefacts occur in areas of high frequency. The JPEG compression system was not designed for lossless encoding which explains why the file sizes for high quality compression are larger than the entropy encoded raw data.

The results suggest that more modern image compression techniques that are not block based such as wavelets could be ideal for further research. This confirms the research of [13] which applies wavelet compression to geometry images.

In Figure 39 (p.64) the results clearly show that much smaller files can be generated at the cost of quality. Artefacts that are created through image compression techniques on geometry images are localised similar to compressing normal photographic images.

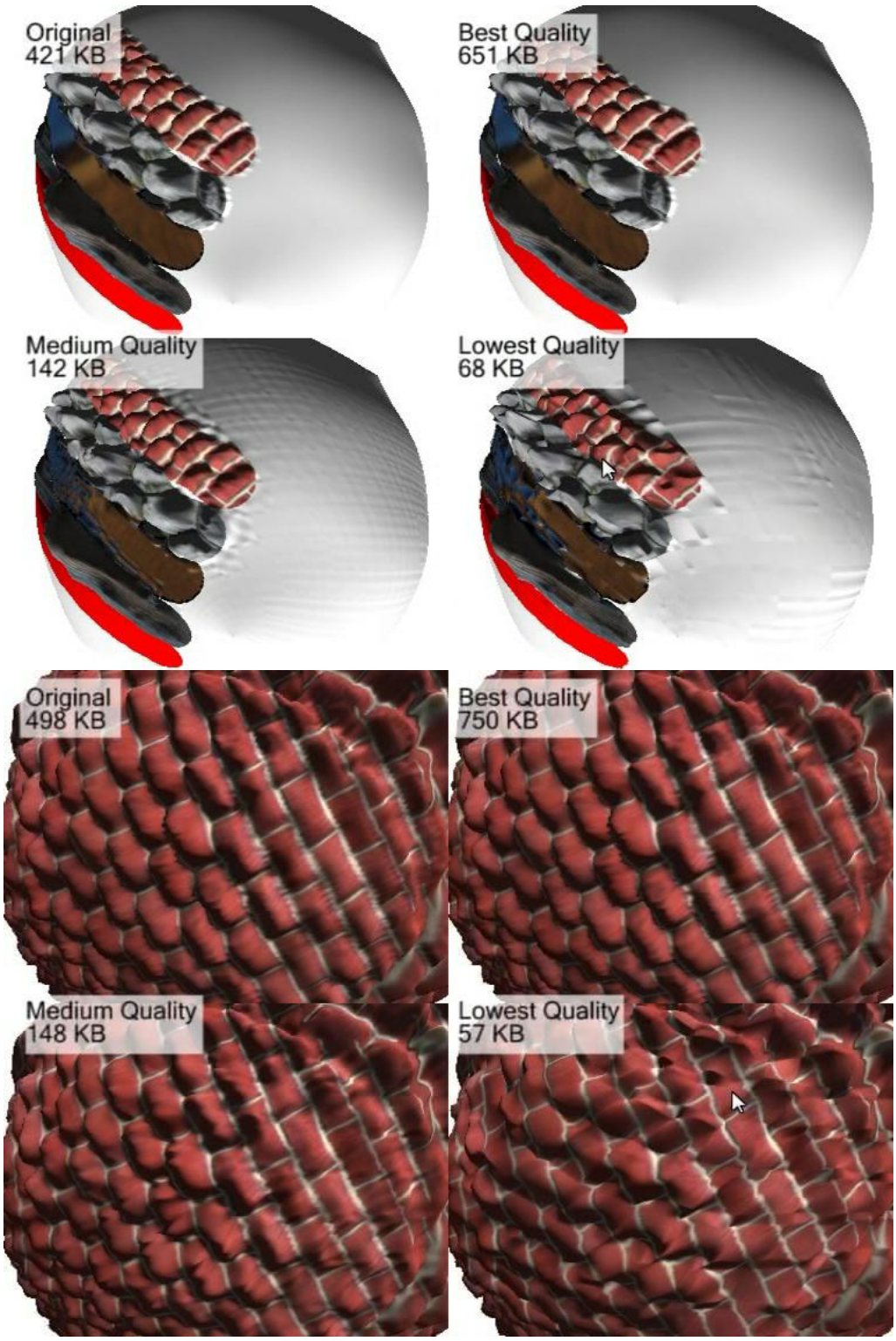

Figure 39: Floating point JPEG compression

5.4 Bezier curve compression

To take advantage of the smooth gradients found in the majority of geometry images a Bezier curve fitting technique was applied. This was an existing algorithm developed by staff at Unlimited Realities and was originally used to compress animation data which provided a suitable lossy compression of approximately 100:1. The animation data was a series of curves measured over time, and so was possibly suitable for geometry image compression.

To compress the data it is broken up into sections and a Bezier curve fitted to that section. The result is that a few floating point values which determine the Bezier curve are used to replace a long string of floating point absolute values. The number of sections to which the curve is broken into determines the final quality and file size of the compression.

The Bezier compression technique was applied to both the original geometry image and also the delta encoding of the geometry image and both provided poor results. The problem was due to no correlation between rows which resulted in a wave effect forming across the surface of the mesh. Figure 40 shows the wave effect on the left and the original image on the right. When a single compressed row is compared against the original such as in Figure 41 (p.66), there is only a small difference between them, however the error between any row and its neighbours can be large and cause the wave artefact. This algorithm could be expanded into two dimensions, however the number of patches and maintaining coherency between them would be of even greater difficulty.

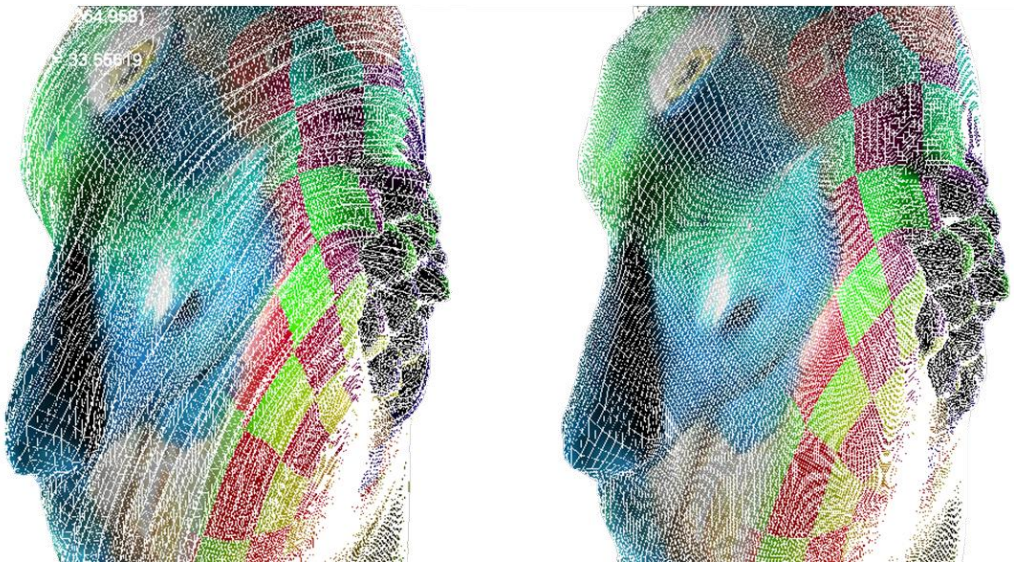

Figure 40: Bezier Curve Compression

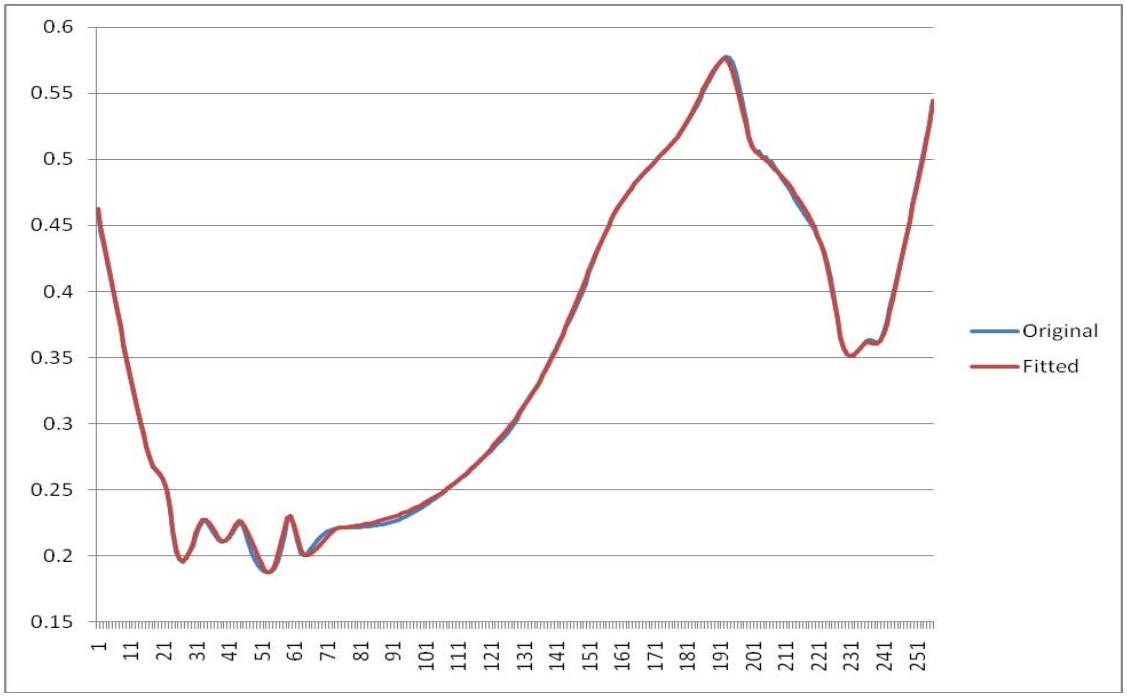

Figure 41: Bezier Curve against Original Data

5.5 Custom byte packing

To take advantage of the spatial coherency of geometry images a new encoding was created. Because the change between vertices/texels is relatively small this change could be encoded as an 8 bit offset from a 32 bit floating point value. The number of regions which the geometry image is broken up into will determine the quality and size of the compression.

The compression starts by creating a MIP map of the geometry image, the resolution of which determines the number of regions to split the geometry image. The mip map is in a 32 bit per channel floating point format. Then for each region the largest and smallest delta value from the value stored in the mip map is recorded. These values are stored in a separate binary file and are 32 bit floating point. Finally the delta values are encoded by finding the difference from the mip map value for that region of the geometry image. The delta values are then scaled by the minimum and maximum values such that they lie between 0 and 255. The scaled value is then stored as an unsigned char format in the standard 8 bit per channel RGB image.

The three files can then be grouped together in a single compressed file. The small mip map and minimum maximum value files will not compress very well, however the 8 bit per channel offset image will compress well. Figure 42 shows the original geometry image on the left, the mip mapped floating point image (magnified 1600%) and the 8 bit per channel offset image.

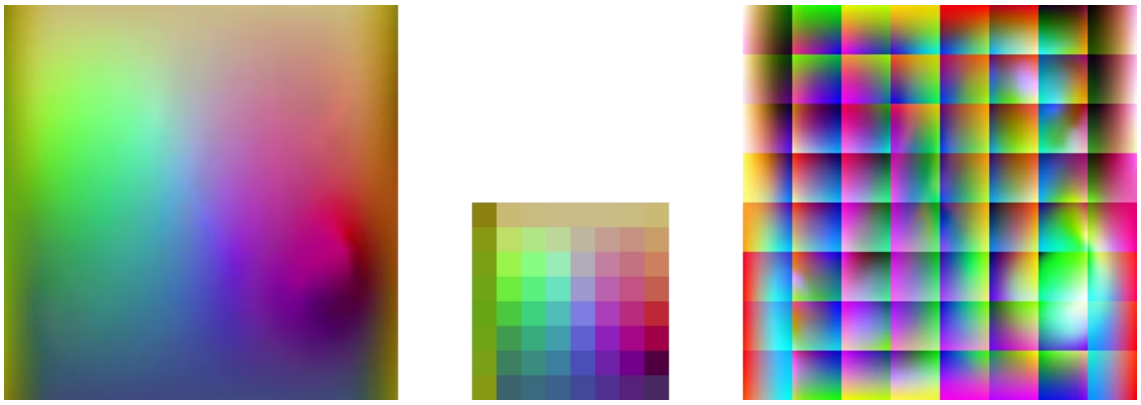

Figure 42: Custom mip map based storage format

To decompress the file each region is constructed by reading the base position from the small mip map and scaling the offset by the minimum and maximum value.

Mesh	Regions	File Size	Mesh	Regions	File Size
Bricks	4x4	91	Face	4x4	53
	16*16	148		16x16	120
	64x64	291		64x64	275
	Original	498		Original	592
Mirror Ball	4x4	36	Mixture	4x4	42
	16x16	82		16x16	98
	64x64	250		64x64	260
	Original	506		Original	422

Figure 43: Comparison of mip map resolution with file size

The resolution of the small mip map determines the final file size and image quality. Figure 43 shows the effect of compression at different levels on a range of models. Models which exhibit a smooth surface compress the best, most noticeably the mirror ball model which is similar to a sphere.

The effect of the compression is only seen on smooth surfaces either when they are reflecting or when the diffuse light is at a grazing angle. The best and worst case scenario for this compression method can be seen as a rough surface with lots of different diffuse information and a smooth mirror ball respectively. As we can see in Figure 44 there is no noticeable difference in the meshes. All of the examples in this section follow the following guideline, top left is the original mesh, top right has 64x64 regions, bottom left has 16x16 regions and bottom right has 4x4 regions.

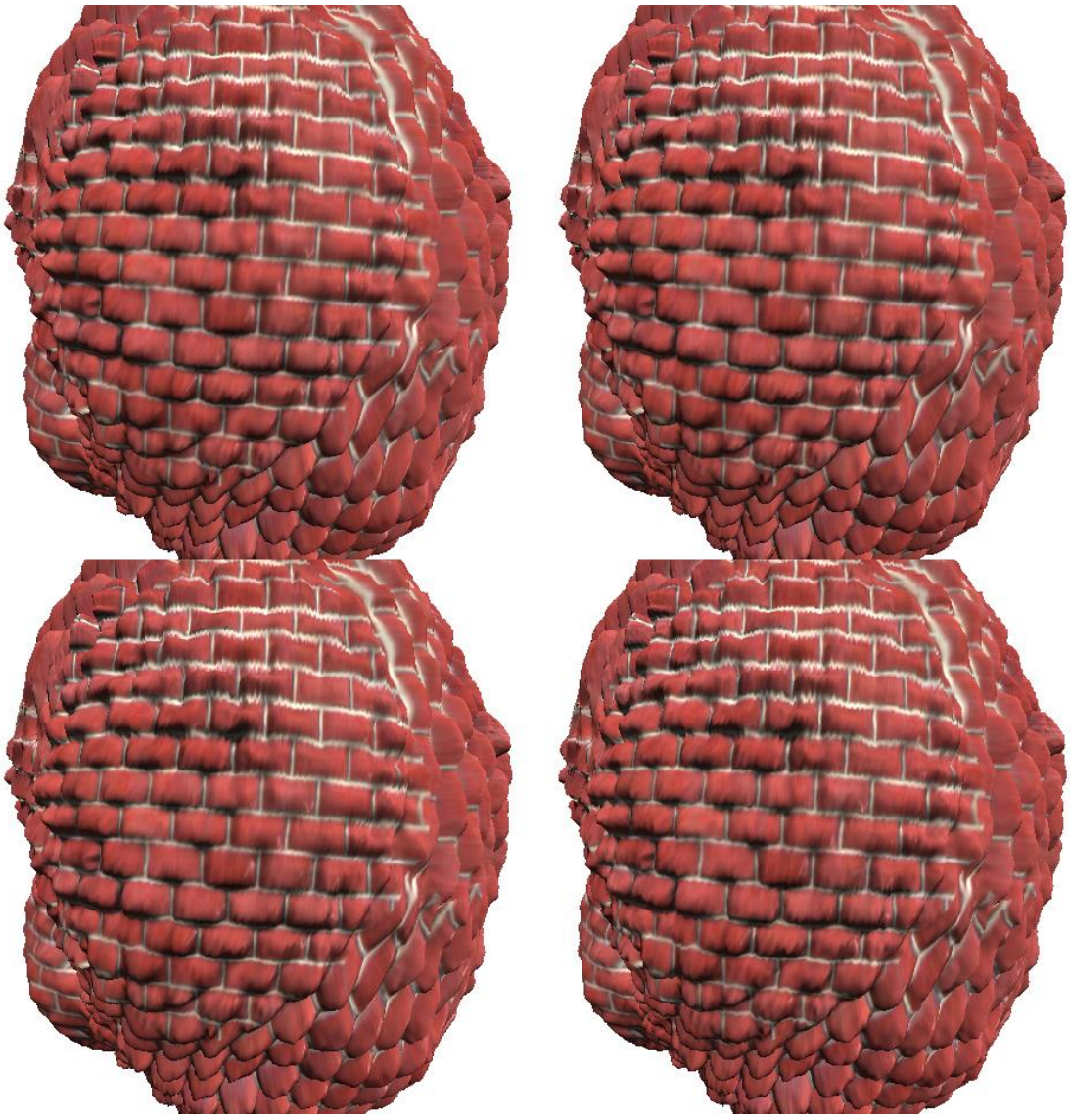

Figure 44: Compression on ball of bricks

When we compare an up-close view of a region with high detail we can begin to see differences as shown in Figure 45 (p.69) with the 4x4 region compression. However these differences are not visible from most levels of detail and can be considered acceptable. The worst case scenario is shown in Figure 47 (p.71) where the surface is smooth and

reflective/shiny. The smooth surface is hard to replicate over large areas using an 8 bit quantization of the change. While geometrically the mesh looks the same, the difference is noticeable in the surface normals which are generated as the mesh is loaded. When the diffuse light is at a low angle and the light at the surface is transitioning between light and dark, rather than a smooth transition, blocky artefacts begin to appear.

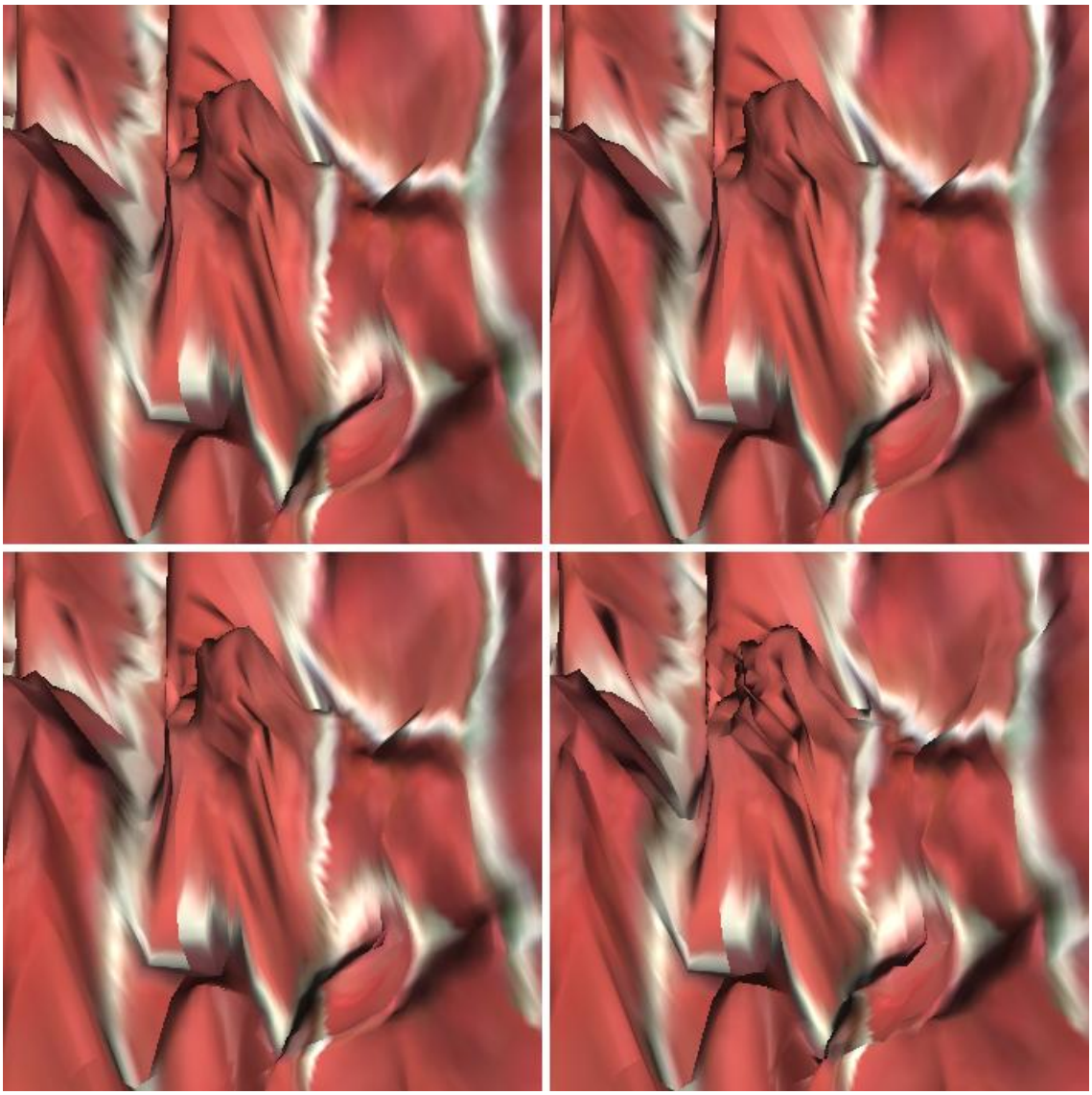

Figure 45: Close up of lossy Compression

The other time the artefacts appear is when the specular component is high or environment mapping is applied. The perfect example of this is a mirror ball model as shown in Figure 46 (p.70). The quality of the environment mapping quickly deteriorates as compression increases. The 64x64 region compression has approximately half the file size compared to the original with only slight errors in the surface normals.

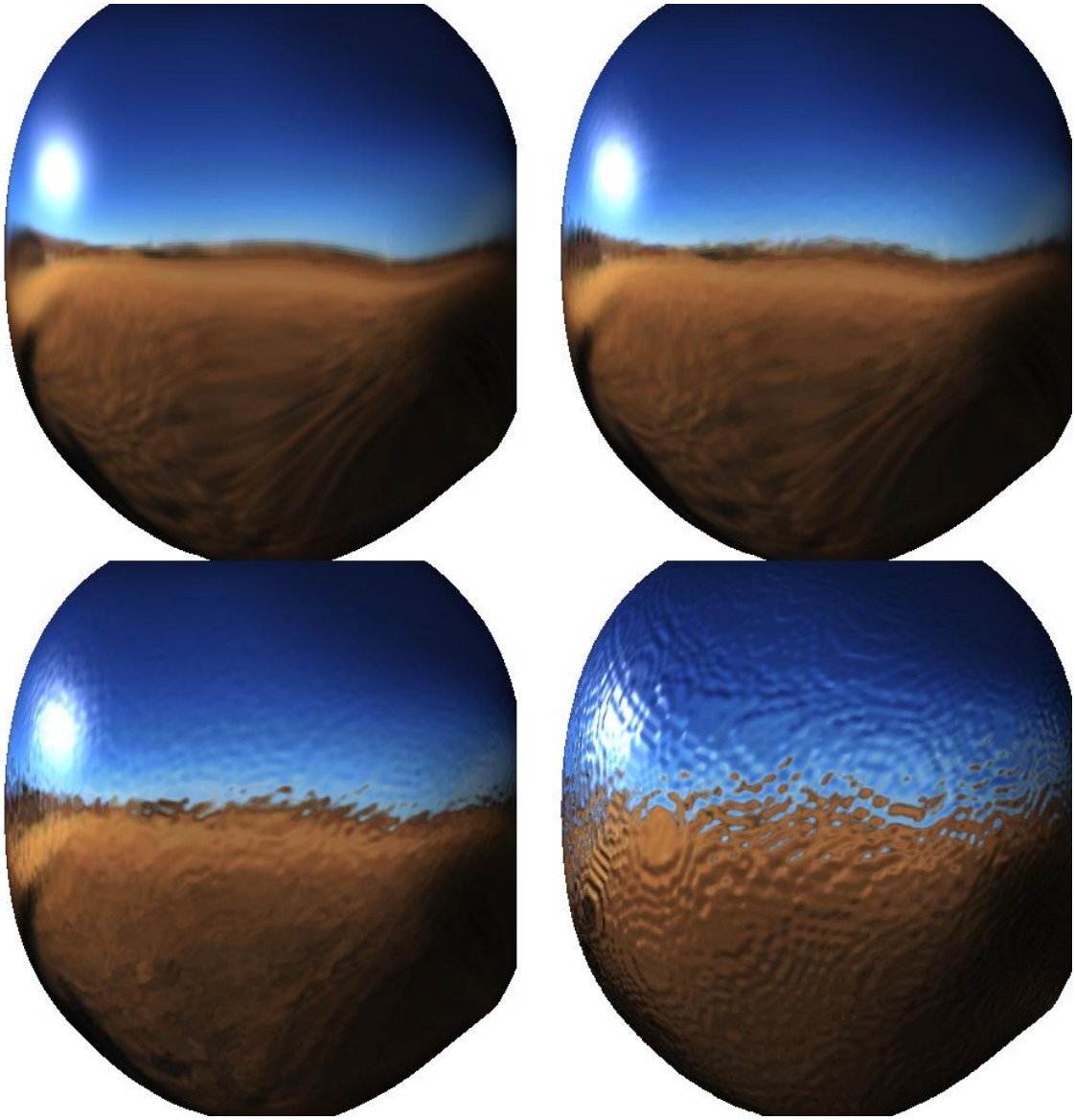

Figure 46: Compression on Smooth Surface

This shows that this lossy compression technique is suited best to models without smooth surfaces with flat regions of colour. Figure 47 (p.71) shows the worst case lighting scenario for a model with different surfaces applied. The red plastic and mirror sections of the model show large amounts of compression artefacts, while the wood, stone and brick sections look identical to the uncompressed version.

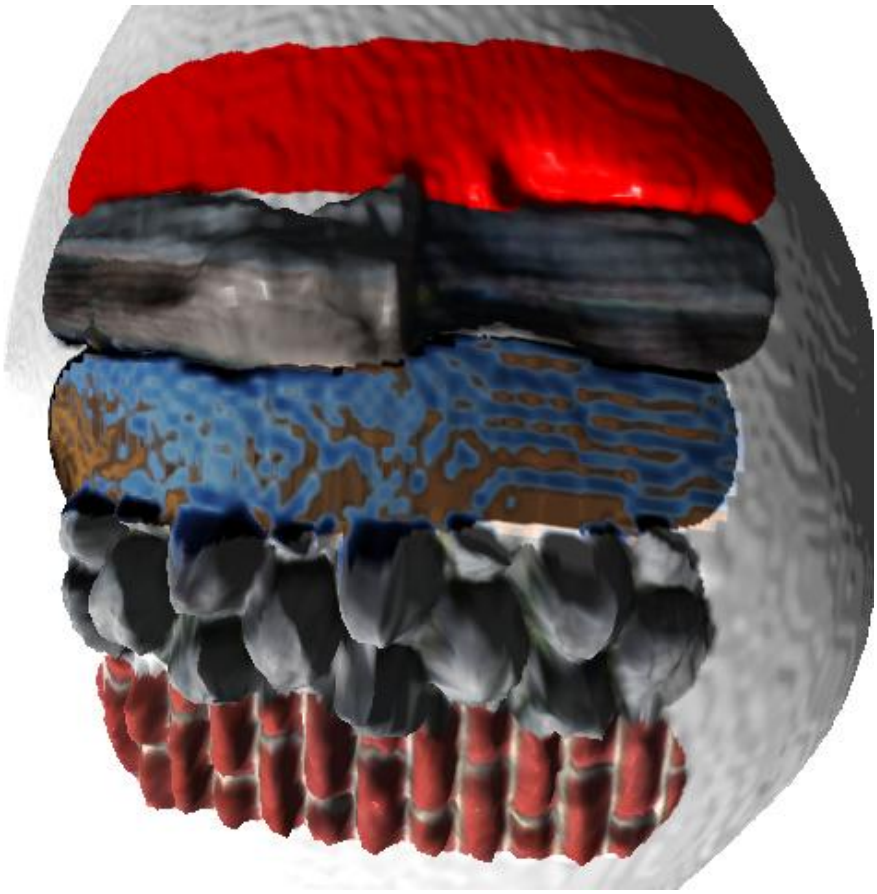

Figure 47: Worst case lighting for different surface

5.6 Summary

Based on existing image compression research [13] and experimentation, traditional image compression techniques have been proven to work on geometry images. Geometry images for most genus-0 3d envelopes can be described as slow changing gradients in two dimensions. However traditional image compression quantisation on low frequency gradients results in stepping artefacts. This is particularly noticeable where a mirrored texture is applied to the object.

In summary choices in common image compression techniques designed to target the human visual system have not proved to be ideal for compressing geometry images. However the principal of localised errors and scalable compression ratios has been demonstrated. It is recommended that future research be directed towards wavelet based techniques. These are not block based and can be tuned to support a linear 'visual model.

Chapter 6 - Results

The result of the proposed framework is that separate problems in the domain of 3d model rendering, storage and manipulation have been bought together to a unified system and solved.

For editing, a high level approach to geometry and surface detail editing has been created which allows users to work at a level above vertices and texture coordinates. Texture coordinate allocation which is a difficult and unsolved problem for arbitrary meshes is handled through the implicit texture coordinate system associated with geometry images. New techniques for editing can be created through writing new pixel shader programs. By utilizing the GPU editing models of 2 million polygons becomes possible in real time.

When rendering automatic level of detail generation becomes a simple process by utilizing the structure of the geometry image combined with vertex collapse techniques. The automatic level of detail generation presented in this framework allows for real time generation of differing levels of detail.

The storage of geometry image has been investigated and new compression techniques proposed. The results have shown that existing image compression techniques can be applied to mesh data in the form of geometry images. However the current techniques need adjusting to work with floating point data and compress in a uniform colour space.

6.1 Visual Results

The pull and push brushes give the user the freedom to stretch out or compress areas of the model while keeping its integrity, Figure 48 (p.73). The user cannot punch holes or tear the surface of the model by design; this gives a consistent feel of modelling with a soft solid. The below example is run on a 256^2 geometry image and runs smoothly in real time. The pull and push is a basic tool which is crucial to this modelling style.

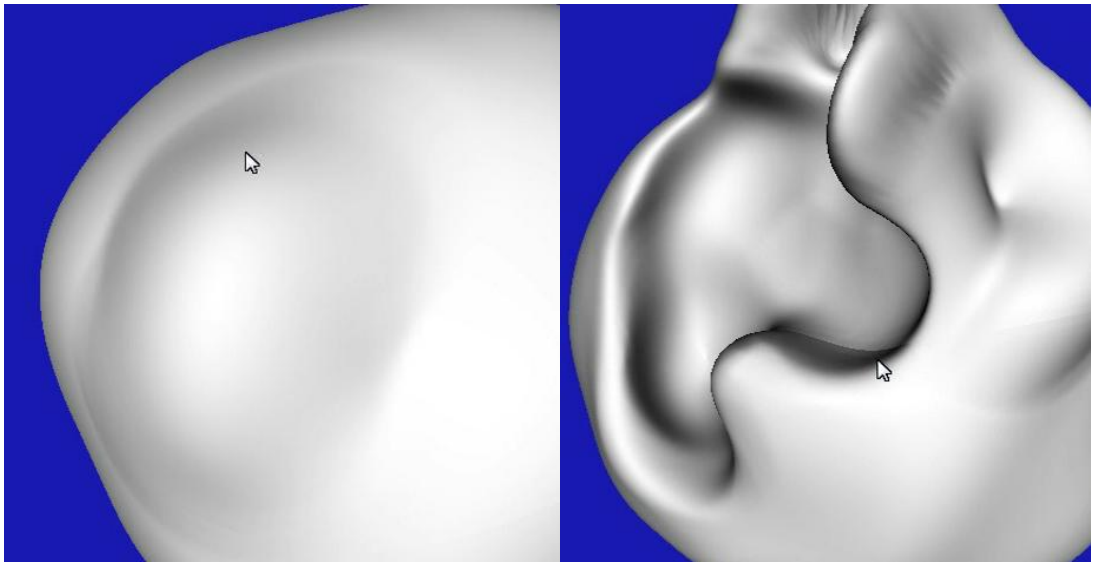

Figure 48: Before and After using Pull/Push brush

A practical example of the push brush is shown in Figure 49 where an existing model is altered by pushing the nose inwards to create a different looking model. These small changes can quickly alter a generic base model into the required shape.

Figure 49: Face manipulation

The advanced splatting technique allows for large amounts of surface detail to be transferred onto the mesh while preserving the underlying shape of the mesh. As shown in Figure 50 (p.74) different surfaces have been transferred onto the mesh.

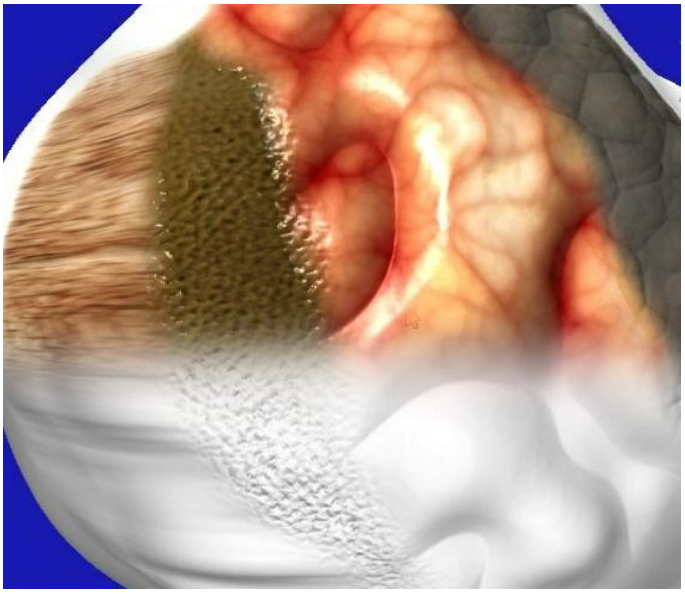

Figure 50: High detail splatting

The rendering in Figure 51 shows a 256^2 base mesh with a 1024^2 geometry image being used to calculate the surface normals. This technique of editing a lower LOD mesh while preserving the fine detail provides an effect way to create a highly detailed mesh in real time.

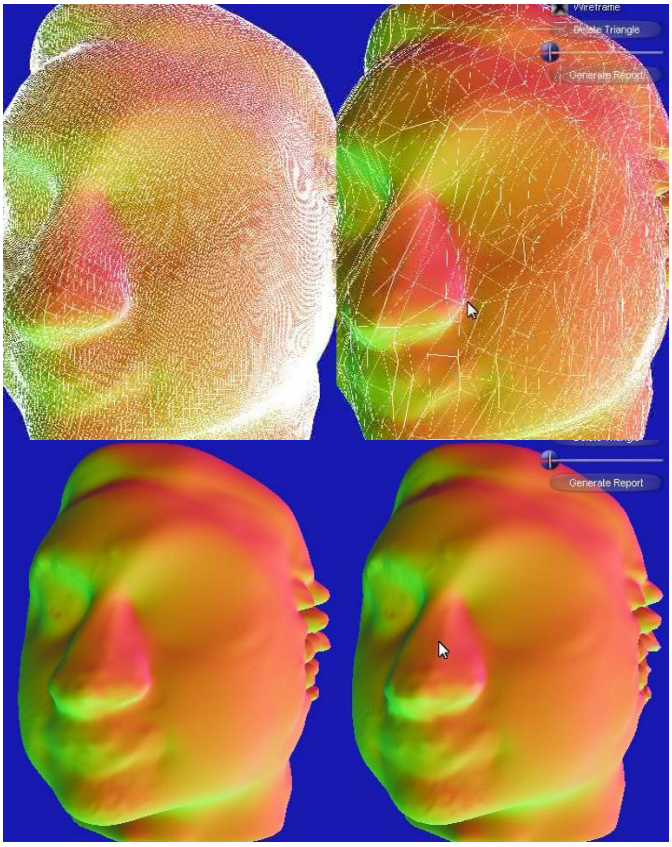

Figure 51: Automatic LOD Comparison

The automatic LOD generation as demonstrated in Figure 51 (p.74) is the final technique used and is a combination of salient GPU based sub sampling and CPU based vertex collapse. This technique runs in real time and provides an excellent representation of the original model with a reduced polygon count. This is mostly attributed to the full sized geometry image in video memory which retains full surface normal information. The geometry image used for normal calculations can also be sub sampled if low video memory requirements are imposed.

6.2 Numeric Results

The results shown in this section are taken from the accompanying DVD. It is recommended to view the DVD demonstration to fully appreciate the results of this project. In this section some performance metrics are given. The machine used for performance testing had the following specifications:

Intel Dual Core 2.66 Ghz	2GB RAM
Nvidia 8600GT	Windows Vista

For the purposes of testing the same geometry images and surface textures were used. The different resolutions were created through sub sampling 4096^2 images. The application was run at a resolution of 1280x1024. For viewing the model, the same view angle and distance was used to ensure the pixel shading stage remained constant during testing. Level of detail rendering was used during the rendering stage, fixed to a 256^2 base mesh. The editing metric was based on texture splatting to all surfaces of the geometry image and running the seaming pass. This represents the worst case scenario for editing a geometry image.

The results show that the rendering pipeline remains relatively constant which is due to the level of detail rendering. The sharp drop in editing frame rates shows the effect of increasing the size of the geometry image.

Map Size			Frame rate per second	
Geometry	Diffuse	Other	Rendering	Editing
256	512	512	118	69
1024	512	512	115	37
2048	512	512	103	4
256	1024	1024	115	61
1024	1024	1024	110	33
2048	1024	1024	105	4
256	2048	2048	110	37
1024	2048	2048	105	25
2048	2048	2048	105	3

Figure 52: Rendering and Editing Frame rates

6.3 Other Implementations

To investigate the suitability of the proposed framework for older hardware test implementations were created. The values given to represent performance in this section are only rough figures given to provide an estimate of the differences between different methods. This is because there are far too many variables in comparing these test implementations. To reduce the pixel shader overhead, the pixel shader was set to the dot product of the surface normal and eye vector. This provides good information about the surface of the mesh and is a very short shader.

The first was a CPU based implementation using DirectX 10, in which the geometry image is stored in main memory, and a vertex buffer is sent to the GPU each time the image changes. The rendering rate was similar to that of the GPU based framework however the performance dropped dramatically during editing. This drop in rendering speed could be due to two factors, resending the vertex buffer to video memory or the editing of the geometry image. To test the effect of sending the vertex buffer to the GPU upon editing a test was created which simply shifted a single pixel in a fixed direction. This editing operation is the smallest that can be done to reduce the bottleneck of the CPU editing. The results were only marginally slower than the GPU based version, showing that sending 256x256x3 polygons to the graphics card is not a significant bottleneck.

The next step was to test more complex editing operations to confirm the CPU bottleneck. A smoothing operation was created which worked on a fixed area in the geometry image. The fixed size on the geometry image translates to varying sizes on the actual mesh, so is not suitable for editing. The performance of this brush was reduced which begins to show the CPU bottleneck.

To replicate the brush operations found in the GPU based version, the brush needs to affect every vertex/texel within the radius of the brush. This means the CPU version must scan over each pixel in the geometry and test its suitability before applying the brush operation. When this was created the speed of the editing dropped to 3-4 frames per second. This highlights the GPU's ability to split the task of editing geometry images and run it on different stream processors. An approximate 10x increase in speed can be gained from utilizing the GPU for geometry image manipulation.

Next an OpenGL version was created to compare SM3.0 and SM4.0 hardware. The test program simply loaded the geometry image into video memory and used the vertex texture fetch operation to sample the geometry image in the vertex shader. The input mesh was simply a list of texture coordinates and dummy vertex positions and the model would be made by the vertex shader each frame. The shader model 4.0 card (8600GT) would render this at around 16 frames per second. On the older 7800GT which is shader model 3.0 the frame rate was around 1-2 frames per minute. This is due to the vertex texture fetch being very expensive on shader model 3.0 hardware.

To compare the OpenGL and DirectX 10 versions of the geometry image renderer, the stream out functionality was removed from the GPU manipulation framework. The DirectX 10 version would render the geometry image at approximately 500 frames per second which is a significant improvement over the 16 frames per second found in the OpenGL implementation. The cause for the significant difference of the OpenGL and DirectX 10 implementation is the utilization of load management, where a vertex shader heavy effect has more stream processors allocated to the vertex shader.

6.4 Converting arbitrary meshes to Geometry Images

Converting arbitrary piecewise polygonal meshes into geometry images is a difficult task. The problem is automatically creating a parameterization for the mesh with low stretch. This has been discussed by [9] which cover creating cuts in the surface of the image, to allow the mesh to be parameterized to a 2D plane with low amounts of stretch. Additional concepts can be taken from [21] to parameterize a mesh based upon surface signal criteria. The proposed implementation was based upon a zero genus mesh so the octahedral parameterization as demonstrated in [11] is used.

The geometry images used in this project were either created within the geometry image editor or parameterized by hand before converting into a geometry image. To convert an arbitrary genus-0 mesh into a geometry image the texture tools found in HEXAGON 2 were used. A spherical parameterization was applied, however this was never ideal and manual tweaking was required. To parameterize the model the texture coordinates were manipulated such that the entire 0-1 range was covered in both the U and V axes. A cross cut was made on the surface to help the parameterization. The centre of the cross becomes the corners of the geometry image while the ends become the middle of the sides. Once the parameterization was created a separate program was implemented to turn an OBJ mesh into a geometry image. This was done by rendering the mesh, however the texture coordinates were set as the screen position to render to. The position was then rendered as the RGB colour. The rendering was done into a 32bit per channel buffer and then saved into a texture file. The resolution of the geometry image could be altered by changing the resolution of the rendering.

Chapter 7 - Conclusions and Future Direction

This project has provided research into many fields surrounding 3D model manipulation, compression and level of detail generation and combined them into a single coherent framework. The resulting modelling tool allows for intuitive editing of a 3D mesh in a similar fashion to moulding a lump of clay. A high level of abstraction away from low level polygon focused modelling is presented to the user through simple operations such as push/pull and smoothing.

By utilizing features found on modern graphics hardware high levels of detail and interactivity can be achieved. The system is scalable to run on budget modern cards and can be up scaled as faster hardware is released. The proposed system significantly outperforms CPU based modelling systems in terms of speed and level of detail as shown in the results. A modelling system based on this framework will allow users to create highly detailed models in a fully interactive and responsive environment. Users can also visualise the model at different levels of detail interactively to determine how models will look in traditional real time rendering systems.

In this project the entire mesh manipulation process has been shifted onto dedicated graphics hardware. This frees the CPU for other tasks and more importantly utilizes the highly parallel structure of graphics hardware to reach new levels of performance. As more powerful graphics hardware is released this framework will be able to scale up to allow for even more detail in large models while real time performance is maintained. The framework can also be scaled down to work on lower performance graphics hardware with DirectX 10 compatibility.

When modelling within the provided framework, the user can edit with the level of detail found in offline rendering packages, and visualise how the final product will look at varying levels of detail designed for real time rendering. The tools designed are simple to use and new tools can be easily developed through writing pixel shaders.

The through the use of geometry images, the difficult task of texture coordinate assignment which traditionally requires expert users, is now handled automatically, allowing novice users to paint surface properties directly onto a mesh.

This research combines many different fields of research into a single coherent framework which handles mesh storage, compression, manipulation and level of detail generation.

7.1 Limitations of Geometry Images

There are certain limitations with the proposed framework which are inherent to geometry image. The most notable is that the genus of the model must remain the same. The genus of a model refers to the shape of the surface, the amount of holes or loops that exist across the surface. A sphere for example is genus-0 while a torus is genus-1. The initial design specification was to allow modelling with a single solid lump, a genus-0 surface with no surface holes. The proposed framework could easily be extended to allow for other types of model such as planes and tori. A plane would require no seaming calculations and a torus would have a surface mapping as shown in Figure 52. To ensure an even parameterization a geometry image with different width and height can be used. However no solution to convert a geometry image between differing levels of genus is proposed.

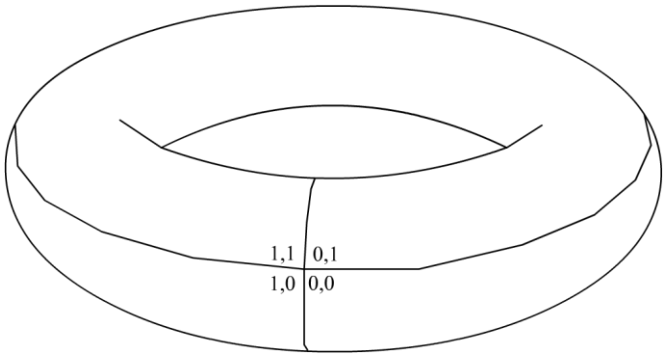

Figure 52: Torus geometry image mapping

Another issue is that of excessive stretch when manipulating the geometry image. When a large amount of pull is applied to a small area, the density of polygons is reduced. With the lower polygon density the texture Texel density is also reduced which lowers the effective resolution of the diffuse and other layers at that point. An example of that is shown in Figure 53 (p.81). Also with a reduced polygon density the ability to paint fine detail geometry information is lost. A wireframe view of an over stretched section of a geometry image is shown in Figure 54 (p.81). A simple solution to the problem is to increase the

resolution of the geometry image, however this is not ideal. One possible solution would be to apply a relaxation algorithm to the geometry such that clusters of small polygons would be spread and enlarged while large polygons would do the opposite. The difficulty would be to retain the original shape of the mesh and to take care to reassign the values of the diffuse maps etc such that the original surface properties are not shifted. This task is left as future research.

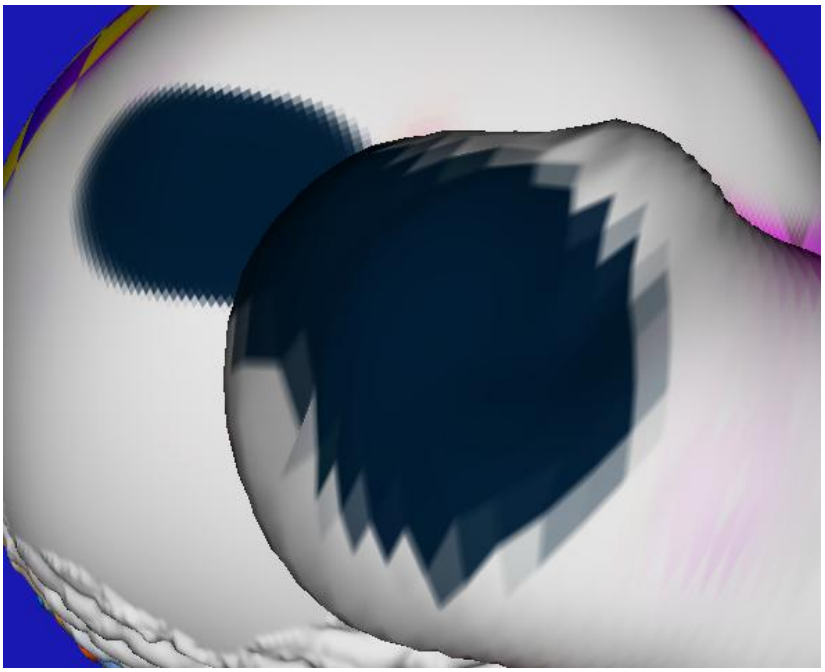

Figure 53: Effect of Stretch on Diffuse Map

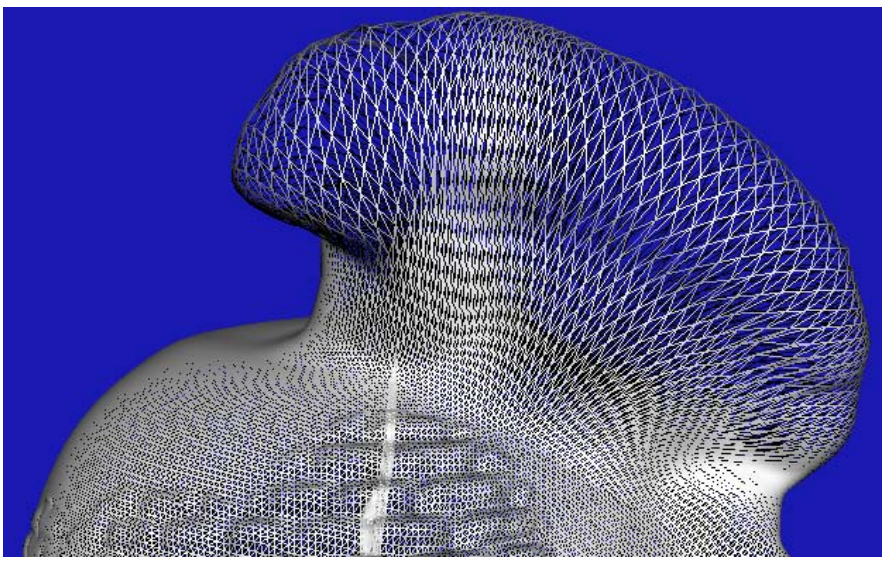

Figure 54: Shows Effect of Large Stretch

Another drawback with geometry images, is that a common technique from piecewise polygonal meshes cannot be used. The technique is to assign the same texture coordinates to different parts of the mesh, so that data from the diffuse map can be reused. This technique is typical when the symmetry of a model is high. If a model has two eyes for example, different vertices will reference the same place in the texture map and reuse the diffuse data. On the flip side of this geometry images do not suffer from mipmapping problems. When two sections of a diffuse map are close together when a mipmap is generated, the colour from one may bleed into the other. The result is slight edges visible on the final model. Because geometry images match across the surface, this error can only occur around the seams of the geometry image. This problem would be simple to correct by implementing a custom mipmapping routine which will tile correctly for the attribute layers.

7.2 Future Directions

The implementation created is limited to modelling only zero genus models, however it could be easily altered to allow for 1-genus models e.g. torus. A suitable mapping is required and then the seaming and texture lookup functions need to be altered to resemble the new mapping used. However no research has been done into converting a geometry image between different levels of genus and is left as a future research possibility.

Due to the large breadth of this project some areas remain suitable for future research for those wishing to further expand this work. Animation was left out from this work, however conceptually could be added easily. Additional research could be done in the area of floating point data compression. Automatic level of detail generation could be further improved to provide GPU based vertex collapsing.

A system for seaming multiple geometry images could be created to allow even more complex models to be manipulated. The main focus would be maintaining coherent seams between the geometry images without excessive CPU interference.

Due to the flexibility of programming brushes as pixel shaders further research can still be done to create new brush effects beyond those presented in this research.

The compression techniques found in this project have proven good, however due to the somewhat predictable nature of geometry images, there could be more research done in this area.

There were a large number of possible extensions to the proposed mesh editing framework that due to time constraints could not be investigated. Animation was not a goal of this project however consideration towards animation was. With this is mind animation was not implemented however as shown in the attribute painting system, this would become trivial. Different textures could be used to paint the bone weighting onto the mesh. The animation could be played in real time as a preview and paused when the painting operation is taking place. The bone weighting textures would compress heavily as the majority of each image would be a zero value.

Another area which was not implemented was exporting meshes from the proposed framework for use in other rendering systems. This would require the export of tangent space normal maps and mesh structures. The exporting of mesh structures simply requires a reverse building of the mesh on the CPU and exporting it. The conversion of an object space normal map to tangent space would require some work however it should be relatively straight forward.

Investigation into a more complex mesh editing system was not done due to both time constraints and the goal was to create a simple editing system. Additional research could be put into a multi-resolution system, where geometry images are layered on top of each other. This would allow layers to be stacked to create different effects over the whole model. The concept of stitching multiple geometry images is another option. By joining four 1024^2 geometry images together and only editing a 1024 area of interest real time editing could be maintained when working with upwards of 8 million polygons. If four 4096 geometry images were joined together over 134 million polygons could be manipulated. Level of detail rendering would be crucial for this technique and management of the editing window would be critical.

The proposed framework could also be used in other applications apart from modelling. Real-time simulations could utilize this framework for soft body deformation based upon physics calculations. For example when two objects collide, one could dent the surface of the other and texture transfer could occur such as two cars colliding.

This thesis has been successful in creating a framework for the real time editing, rendering and storage of 3d mesh data. The proposed framework utilized modern advancements in graphics hardware and provides a simple interaction metaphor to allow novice users to edit 3d mesh data. This thesis has researched many areas of 3d mesh storage, rendering and editing and combined this research to create a unified framework which was designed with all research areas in mind. The editing framework can work with large numbers of polygons in real time and will scale up as more powerful graphics hardware is released.

References

[1] Rusinkiewicz, S., and Levoy, M. 'QSplat: a multiresolution point rendering system for large meshes', *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, p.343-352, July 2000

[2] Mueller, K., Möller, T., andCrawfis, R., 'Splatting without the blur', *Proceedings of the conference on Visualization '99: celebrating ten years*, San Francisco, California, United States, p.363-370, October 1999.

[3] Lorensen, W.E., and Cline, H.E, 'Marching cubes: A high resolution 3D surface construction algorithm', *ACM SIGGRAPH Computer Graphics*, v.21 n.4, p.163-169, July 1987

[4] Kaufman, A., 'Efficient algorithms for 3d scan-conversion of parametric curves, surfaces, and volumes',*Proceedings of ACM SIGGRAPH 1987*, USA, p.171–179, July 1987.

[5] Kaufman, A., and Shimony, E., '3d scan-conversion algorithms for voxel-based graphics', *In Proceedings of ACM Workshop on Interactive 3D Graphics*, Chapel Hill, NC, USA, p.45–76, October 1986.

[6] Guskov, I., Vidimče, K., Sweldens, W., and Schröder, P., 'Normal meshes', *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, p.95-102, July 2000

[7] Khodakovsky, A., and Guskov, I., 'Normal Mesh Compression', *Geometric Modeling for Scientific Visualization*, G. Brunnett, B. Hamann, and H. M¨uller, Eds. Springer Verlag, 2002.

[8] Friedel, I., Schröder, P., and Khodakovsky, A., 'Variational normal meshes', *ACM Transactions on Graphics (TOG)*, v.23 n.4, p.1061-1073, October 2004

[9] Gu, X., Gortler, S.J., and Hoppe, H., 'Geometry images', *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, San Antonio, Texas, July 23-26, 2002.

[10] Briceño, H.M., Sander, P.C., McMillan, L., Gortler, S., and Hoppe, H., 'Geometry videos: a new representation for 3D animations', *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, San Diego, California, July 26-27, 2003.

[11] Losasso, F., Hoppe, H., Schaefer, S., and Warren, J., 'Smooth geometry images', *Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, Aachen, Germany, June 23-25, 2003.

[12] Lai, Y.K., Hu, S.M., Gu, D.X., and Martin, R.R, 'Geometric texture synthesis and transfer via geometry images', *Proceedings of the 2005 ACM symposium on Solid and physical modeling*, Cambridge, Massachusetts, p.15-26, June 13-15, 2005.

[13] Hoppe, H. and Praun, E., 'Shape compression using spherical geometry images', MINGLE 2003 Workshop, *Advances in Multiresolution for Geometric Modelling*, N. Dodgson, M. Floater, M. Sabin (eds.), Springer-Verlag, p27-46, 2005.

[14] Ritschel, T., Botsch, M., and Müller, S., 'Multiresolution GPU Mesh Painting', *EUROGRAPHICS*, 2006.

[15] Beneš, B. and Villanueva, N. G. 'GI-COLLIDE: collision detection with geometry images', In *Proceedings of the 21st Spring Conference on Computer Graphics* (Budmerice, Slovakia, May 12 - 14, 2005), SCCG '05, ACM, New York, NY, p.95-102, 2005.

[16] Carr, N.A, Hoberock, J., Crane, K., and Hart, J.C., 'Fast GPU ray tracing of dynamic meshes using geometry images', *Proceedings of the 2006 conference on Graphics interface*, Quebec, Canada, June 07-09, 2006.

[17] Lee, A., Moreton, H., and Hoppe, H., 'Displaced subdivision surfaces', *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, p.85-94, July 2000

[18] Hoppe, H., 'Progressive meshes', *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, p.99-108, August 1996

[19] Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P., 'Ray tracing on programmable graphics hardware', *ACM SIGGRAPH 2005 Courses* (Los Angeles, California, July 31 - August 04, 2005). J. Fujii, Ed. SIGGRAPH '05. ACM, New York, NY, p268, 2005.

[20] Chan, S.L., and Purisima, E. O., 'A new tetrahedral tessellation scheme for isosurface generation', *Computers & Graphics*, Feb. 1998.

[21] Sander, P.V., Gortler, S.J., Snyder, J., and Hoppe, H., 'Signal-specialized parametrization', *Proceedings of the 13th Eurographics workshop on Rendering*, Pisa, Italy, June 26-28, 2002.

[22] Lefebvre, S., and Hoppe, H., 'Perfect spatial hashing', *ACM SIGGRAPH 2006 Papers*, Boston, Massachusetts, July 30-August 03, 2006.

[23] http://www.fnordware.com/ProEXR/

[24] http://developer.nvidia.com/object/cg_toolkit.html

[25] Reinhard, E., Ward, G., Pattanaik, S., and Debevec, P., 'High Dynamic Range Imaging Acquisition, Display and Images-based Lightin', Morgan Kaufmann, 2006

[26] Meagher, D., 'Octree encoding: A new technique for the representation, manipulation and display of arbitrary three dimensional objects by computer', Rensselaer Polytechnic Institute, Troy, N.Y., 1980.

[27] NVIDIA SDK 10 Code samples, Smoke, http://developer.download.nvidia.com/SDK/10/direct3d/samples.html#Smoke

[28] Sander, P.V., Wood, Z.J., Gortler, S. J., Snyder, J., and Hoppe, H., 'Multi-chart geometry images', *Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, Aachen, Germany, June 23-25, 2003.

[29] Lee, C.H, Varshney, A., and Jacobs, D.W., 'Mesh saliency', *ACM Transactions on Graphics (TOG)*, v.24 n.3, July 2005.

[30] Hoppe, H., 'Progressive meshes', *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, p.99-108, August 1996.

[31] Rantam M., Inui, M., Kimura, F., and Mäntylä, M., 'Cut and paste based modeling with boundary features', *Proceedings on the second ACM symposium on Solid modeling and applications*, Montreal, Quebec, Canada, p.303-312, May 19-21, 1993.

[32] Kho, Y., and Garland, M., 'Sketching mesh deformations', *Proceedings of the 2005 symposium on Interactive 3D graphics and games*, Washington, District of Columbia, April 03-06, 2005.

[33] Shiue, L.J, Goel, V., and Peters, J., 'Mesh mutation in programmable graphics hardware', *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, San Diego, California, July 26-27, 2003.

[34] Catmull, E., and Clark, J., 'Recursively generated B-spline surfaces on arbitrary topological meshes', *Seminal Graphics: Poineering Efforts that Shaped the Field ACM*, New York, NY, p.183-188, 1998.

[35] 3D Studio Max, http://www.autodesk.com/3dsmax

[36] Wavefront OBJ specification, http://www.csit.fsu.edu/~burkardt/txt/obj_format.txt

[37] Huang, X., Li, S., and Wang, G., 'A GPU based interactive modeling approach to designing fine level features', *Proceedings of Graphics interface 2007* (Montreal, Canada, May 28 - 30, 2007). GI '07, vol. 234. ACM, New York, NY, p.305-311, 2007.

[38] Yu, J., and Chuang, J., 'Consistent Mesh Parameterizations and Its Application in Mesh Morphing', http://cggmwww.csie.nctu.edu.tw/research/YC03.pdf, 2003.

[39] Praun, E., Sweldens, W., and Schröder, P., 'Consistent mesh parameterizations', *Proceedings of the 28th Annual Conference on Computer Graphics and interactive Techniques SIGGRAPH '01. ACM*, New York, NY, p.179-184, 2001

[40] Goldfeather, J., and Hultquist, J., 'Fast Constructive Solid Geometry Display in the Pixel-Powers Graphics System', *Computer Graphics*, Vol. 20, No. 4, p.107-116, August 1986.

[41] Kelley, M., Gould, K., Pease, B., Winner, S., and Yen, A., 'Hardware accelerated rendering of CSG and transparency', *Proceedings of the 21st Annual Conference on Computer Graphics and interactive Techniques SIGGRAPH '94. ACM*, New York, NY, p.177-184, 1994.

[42] Stewart, N., Leach, G., and John, S., 'An improved z-buffer CSG rendering algorithm', *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (Lisbon,

Portugal, August 31 - September 01, 1998). S. N. Spencer, Ed. HWWS '98. ACM, New York, NY, p.25-30, 1998.

[43] Frisken, S.F., Perry, R.N., Rockwood, A.P., and Jones, T.R., 'Adaptively sampled distance fields: a general representation of shape for computer graphics', *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, p.249-254, July 2000

[44] Pope, J., Frisken, S.F., Perry, R.N., 'Dynamic Meshing Using Adaptively Sampled Distance Fields', *ACM SIGGRAPH*, August 2001

[45] Figueiredo, L.H., Velho, L., and Oliveira, J.B., 'Revisiting Adaptively Sampled Distance Fields', *XIV Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI'01)*, p. 0377, 2001.

[46] Rossignac, J., 'Edgebreaker: Connectivity Compression for Triangle Meshes', *IEEE Transactions on Visualization and Computer Graphics*, v.5 n.1, p.47-61, January 1999.

[47] Deering, M., 'Geometry compression', *Proceedings of the 22nd Annual Conference on Computer Graphics and interactive Techniques,* S. G. Mair and R. Cook, Eds. SIGGRAPH '95, ACM, New York, NY, p.13-20, 1995.

[48] Isenburg, M., Lindstrom, P., and Snoeyink, J., 'Lossless compression of floating-point geometry', *CAD'3D*, 2004.

[49] Touma, C., and Gotsman, C., 'Triangle mesh compression', *Graphics Interface 98 Conference Proceedings*, p.26–34, 1998.

[50] Isenburg, M., Alliez, P., 'Compressing polygon mesh geometry with parallelogram prediction', *Proceedings of the conference on Visualization '02*, Boston, Massachusetts, October 27-November 01, 2002.

[51] Hexagon 2, http://www.daz3d.com/i.x/software/hexagon

[52] Zbrush 3.1, http://www.pixologic.com/zbrush

[53] Avidan, S. and Shamir, A., 'Seam carving for content-aware image resizing', *ACM SIGGRAPH 2007 Papers* (San Diego, California, August 05 - 09, 2007). SIGGRAPH '07. ACM, New York, NY, p.10, 2007.

[54] Luebke, D., Watson, B., Cohen, J.D., Reddy, M., and Varshney, A., 'Level of Detail for 3D Graphics', *Elsevier Science Inc.*, New York, NY, 2002.

Bibliography

Ward, G., 'High Dynamic Range Image Encodings', www.anyhere.com/gward/hdrenc/Encodings.pdf, 2007.

Blythe, D., 'The Direct3D 10 System', http://download.microsoft.com/download/f/2/d/f2d5ee2c-b7ba-4cd0-9686-b6508b5479a1/Direct3D10_web.pdf, 2007

Botsch, M., Pauly, M., Kobbelt, L., Alliez, P., Levy, B., Bischoff, S., and Rössl, C., 'Geometric Modeling Based on Polygonal Meshes', *ACM SIGGRAPH 2007 Course Notes,* 2007

Khayam, S.A., 'The Discrete Cosine Transform (DCT): Theory and Application1', Department of Electrical & Computer Engineering, Michigan State University, 2003.

Collins, G., and Hilton, A., 'Mesh Decimation for Displacement Mapping', *Eurographics 2002*, Short Papers, 2002.

Alliez, P., and Gotsman, C., 'Recent Advances in Compression of 3D Meshes', http://www.ee.bilkent.edu.tr/~signal/defevent/html/abstract/a1239.htm, 2007

Gotsman, C., Gu, X., and Sheffer, A., 'Fundamentals of spherical parameterization for 3D meshes', *ACM SIGGRAPH 2003 Papers (San Diego, California, July 27 - 31, 2003). SIGGRAPH '03. ACM*, New York, NY, p.358-363 2003.

Foley, J.D., Dam, A., Feiner, S.K., and Hughes, J.F., 'Computer Graphics: Principles and Practice second edition in C', Addison-Wesley Publishing Company, 1997.