Comparison of CPU and GPGPU performance as applied to a procedurally generated complex cave system

A project report submitted for the degree of Master of Computing in Computer Science of The Australian National University

by

Anthony Oakden
u47501940@anu.edu.au

Supervisor: Dr Eric McCreath

Research School of Computer Science
College of Engineering & Computer Science
The Australian National University

October 2015
Acknowledgement

The author wishes to thank Dr Eric McCreath for his support and encouragement throughout the duration of this project and Victoria Oakden, MBA, for her patience and support.
Abstract

In computer graphics, and in particular modern 3D computer games, a very large amount of data needs to be manipulated. These types of problem often fall into Single Instruction Multiple Data (SIMD) category and are therefore ideal candidates for a General Purpose Graphics Processing Unit (GPGPU). Although GPGPU has many advocates its application to the creation of virtual worlds is still a ripe area of research.

In this thesis GPGPU programming techniques are applied to a novel algorithm, developed by the author, for cave generation for use in virtual worlds and computer games. In its original, Central Processing Unit (CPU) bound form, the algorithm produced caves but, when used to produce caves of a reasonable visual fidelity, was too slow to be used in a commercial product. By the application of GPGPU techniques the build time for the caves can either be substantially reduced or the visual quality substantially improved.

The contribution of this project is data which compares the performance of CPU and two GPGPU techniques: the use of OpenCL to accelerate the build time of highly detailed meshes and the use of tessellation shaders in the GPU pipeline to tessellate a lower resolution mesh in real-time. A second contribution is the use of normal maps in the tessellation shade to improve render times and their combined use with triplanar shaders.
# Table of Contents

## Contents

1. Introduction .................................................................................................................. 6
    1.1 Motivation .............................................................................................................. 6
    1.2 Contribution .......................................................................................................... 7
    1.3 Project Organisation .............................................................................................. 8
    1.4 Hardware and software Specifications .................................................................. 8
2. Review of GPU hardware ............................................................................................. 9
3. The cave creation algorithm ........................................................................................ 12
    3.1 Outline .................................................................................................................. 12
    3.2 Implementation ...................................................................................................... 23
    3.3 Rendering of the cave system ................................................................................. 23
4. Acceleration using OpenCL ......................................................................................... 26
    4.1 Introduction .......................................................................................................... 26
    4.2 Implementation ...................................................................................................... 26
    4.3 Conclusion ............................................................................................................. 31
5. Optimization using Tessellation shader .................................................................... 32
    5.1 Introduction .......................................................................................................... 32
    5.2 Setting the tessellation level .................................................................................. 33
    5.3 Point Normal Triangles ......................................................................................... 36
    5.4 Adding additional detail in the tessellation shader ................................................. 37
    5.5 Implementation ...................................................................................................... 42
    5.6 Results ................................................................................................................... 43
    5.7 Conclusion ............................................................................................................. 45
6. Conclusion and further work ...................................................................................... 46
    6.1 Further Work ......................................................................................................... 47
    6.2 Comparison of hardware ....................................................................................... 47
    6.3 Comparison of different GPGPU API ..................................................................... 48
    6.4 Final Notes ............................................................................................................ 49
7. The Artefact ................................................................................................................. 49
List of Figures

Figure 3-1 Result of layout made with spheres......................................................... 13
Figure 3-2 Calculating the, radius, centre, and normal for the portal ......................... 14
Figure 3-3 Examples of cave layouts........................................................................... 15
Figure 3-4 Changing the tunnel chance parameter....................................................... 15
Figure 3-5 Projecting a vertex onto the room mesh...................................................... 17
Figure 3-6 Creating a mesh for a three portal room..................................................... 18
Figure 3-7 Output of mesh smoothing algorithm......................................................... 19
Figure 3-8 Screen shot of a cave mesh after the addition of Perlin noise .................... 20
Figure 3-9 Illustration of mesh welding........................................................................ 21
Figure 3-10 Execution times for the different stages in the cave algorithm.................. 29
Figure 3-11 Execution times plotted against number of vertices processed.................. 29
Figure 3-12 Execution time plotted against kernel complexity..................................... 30
Figure 3-13 Comparison of OpenCL execution times on different GPU..................... 31
Figure 5-1 Illustration of tearing during tessellation.................................................... 34
Figure 5-2 Diagram illustrating the results of tessellation in the render pipeline.......... 38
Figure 5-3 Comparison of rendering frame rates for different tessellation levels........ 44
Figure 5-4 Comparison of Tessellation performance on different GPU..................... 44
1 Introduction

Graphics applications often involve the processing of large amounts of data. Generally many data points, sometimes millions of three dimensional points, need to be transformed between various spatial systems and modified by various algorithms. In traditional processing this is done in sequence, by the CPU and FPU, but these types of operation lend themselves very well to SIMD [1].

In theory GPGPU provide an efficient way to parallelize these sorts of operations, because GPUs are essentially a collection of processing cores that share a large number of VPU. However in practice the advantages offered are not apparent. Data needs to be moved from main memory to GPU, processors need to be initialized, caches filled, and the data needs copying back to main memory once processing is complete. These bottlenecks reduce the effectiveness of the technique.

An alternative is to perform some of the data transformation as part of the graphics pipeline, in either the geometry or tessellation stage. This has the advantage that large amounts of data do not need to be copied back and forth between graphics and main memory, but has the disadvantage that data is recalculated every time a frame is rendered for display.

1.1 Motivation

As processing power becomes greater, the amount of data that a modern rendering engine can handle has grown enormously. This has enabled makers of virtual worlds to create realistic environments packed with detailed assets.

In the traditional development workflow the assets that make up the virtual world are created by skilled artists. Typically the artists have tools to streamline the development process but even then a large amount of effort goes into producing good quality assets. As the level of detail that can be put into each asset has increased, the workload on artists has also. The traditional techniques for making that content are slow and require high levels of skill, the cost of producing such worlds has become very expensive leading to the situation where many computer games either re-use assets multiple times or resort to creating smaller high-detailed environments.

An alternative to these traditional techniques is to procedurally generate geometry as required. Such techniques are well known and have been used successfully in many
products including computers games. However their application is usually limited to non-critical artwork rather than assets the user must interact with. Some of the problems with procedurally generated geometry are that it is usually a lower visual standard compared to that generated by an artist and can be quite slow to generate. [2]

The computer games industry requires techniques that will produce good quality content automatically in a reasonable amount of time and rival the quality produced by trained graphics artists.

1.2 Contribution

This report describes a project to investigate the use of GPGPU processing to reduce the time required to create the meshes used in a procedurally generated cave system to evaluate which technique is the most effective for generating and rendering a cave system. The cave creation algorithm, used in this project was developed by the author. The following items are the main contributions of this project.

1. Investigate the current state of technology with regards to GPGPU techniques and graphics programming
2. Determine if GPGPU programming can be successfully applied to the mesh creation process
3. Determine if real-time tessellation is a viable method for reducing the time taken to generate the meshes
4. Compare the performance of different hardware configurations
5. Building a high detailed mesh as an offline process using GPGPU for the processor intensive steps
6. Building a much lower detailed mesh and then using the tessellation shader to add additional detail in real-time
7. The combined use of tessellation shader and tri-planar projection shaders for texturing.

The comparison includes:
- An analysis of the implementation difficulty of the two techniques
- Performance analysis of mesh creation times and frame rate including timing data for various complexities of cave systems data and different hardware configurations
- Other notable advantages and disadvantages of the two techniques
1.3 Project Organisation

The project was carried out in the following phases

1. Literature survey of the use of GPGPU for procedural generation of 3D environments.

2. Replacement of the existing CPU geometry processing stage with GPGPU kernel(s) and processing operation. The fundamental nature of the algorithm did not change but the computationally intensive stages were carried out on the GPU instead of the CPU. Data was read back into main memory once the GPU process was complete. Timing comparisons were made between CPU and GPGPU approaches.

3. The existing processing was changed so that some of geometry could be processed in the graphics pipeline. Research was undertaken on into how best to accomplish this. This involved the tessellation of patch data on the GPU instead of pre-calculating all the primitives for the geometry using the CPU.

4. Timing data for all three proposed solutions was compared and conclusions drawn.

5. The programs were run on a variety of hardware configurations including different brands and combinations of CPU and GPU.

1.4 Hardware and software Specifications

The development machine was equipped with:

- i5-2400CPU@ 3.10 GHz
- 6GB RAM
- AMD Radeon HD 7800 series graphics card.
- Visual studio 2013 for development.
- Windows 7 Home Premium 64-bit

A variety of machines were used for testing including the development machine.
2 Review of GPU hardware

When used for rendering, the GPU is used to compute the results of the following render stages:

1. Vertex Specification
2. **Vertex Shader**
3. **Tessellation Control**
4. **Tessellation Evaluation**
5. **Geometry Shader**
6. Vertex Post-Processing
7. Primitive Assembly
8. Rasterization
9. **Fragment Shader**
10. Per-Sample Operations

The five stages marked in bold font are programmable.

1. The render operation begins with a buffer containing vertex information being passed to the GPU. The buffer contains an array of values each item of which corresponds to a vertex. A vertex specification dictates how the vertices are presented to the GPU. The vertex shader applies a user defined program to each vertex. Vertices can be passed straight through to the next stage but usually some sort of transformation is applied at this stage, local to screen space for example.

2. The **Tessellation Control Shader (TCS)** is an optional stage that allows for the subdivision of patches. This allows for real-time level of detail subdivision. Primitives can be subdivided into many smaller primitives based on whatever heuristic the programmer requires. Usually additional geometry is created when the mesh is closer to the camera so that higher detail can be added efficiently without wasting GPU time processing vertices on meshes which are too far away to be seen clearly.

3. The **Tessellation Evaluation Shader (TES)** takes the vertices which the evaluation shader has created and deforms them using a chosen algorithm. Usually some sort of interpolation is used to smooth vertices between the control points but additional noise or height map deformation can be added at this stage to give extra detail.
4. The *geometry shader* processes each of the primitives created by the previous stage. It can create additional geometry or modify the geometry created. Often it is used to create normal data or make minor changes to layout but can be used for many different applications including advanced shadow effects.

5. The *fragment shader* is invoked for each pixel that appears on the screen and is usually used for lighting effects.

When graphic processing units first appeared in the 1990s the devices were designed primarily for rendering geometry to the screen. The hardware was structured as a series of dedicated circuits designed to handle the various operations required by the *GPU* at different stages in the pipeline, transformation, animation and lighting. For example there was one piece of dedicated hardware for transforming vertices and another for lighting output pixels.

Over time the cards evolved to perform an ever increasing range of sophisticated graphics operations, such as skinned animation systems, shadow casting, normal maps, displacement maps, hardware particle systems etc. Producing specialised hardware for every operation would be prohibitively complex, instead the hardware has evolved to be more generic and programmable. Modern *GPU* have a *unified* architecture where a “pool” of general purpose processing units are pressed into service as required by the controlling hardware. In *Nvidia* cards these units are called *CUDA cores*, in *ATI* the equivalent is the *Stream Processor*. Both architectures provide a similar result but the specific functionality of the processing units varies.

Although the *GPU* was originally designed for rendering large amounts of geometry the generic nature of the processing cores means that it is now suitable for solving problems where large amounts of calculations are applied in parallel to a data set (*SIMD*). Further the large amount of vector processing hardware available on the card allows for the rapid processing of problems involving linear algebra. This application of *GPU* for general purpose programming has become known as *General Purpose Graphic Processing Unit* (*GPGPU*) programming.

Several *GPGPU* application programming interfaces *API* are available for writing code on modern *GPU* units. The Common ones are, *Compute Shaders*, *Direct Compute*, *OpenCL* and *CUDA*.

- *Compute Shaders* are a recent addition to the list of *API*. They provide for computation in the render pipeline that does not fit into the more traditional *shaders* available.
- *Direct Compute* is Microsoft’s *GPGPU API* and will only work on Windows based applications. It’s a good choice if *DirectX* is used elsewhere in the code base, e.g. for rendering purposes.

- The *OpenCL API* is provided by the Kronos group and works on a broad range of hardware and operating systems. It is the most flexible *GPGPU API* in terms of which hardware it will run on. Conversely it has a reputation for being less stable than the alternatives and the tools for debugging it are less advanced.

- *CUDA* was developed by *Nvidia* as an *API* for their hardware. It has the advantage that by only targeting *CUDA* compatible devices optimizations can be made in the code creation process. A *CUDA* implementation will generally do as well or better than an equivalent *OpenCL* implementation on *NVIDIA* hardware.

Unfortunately *CUDA* only runs on *CUDA* compatible devices and these represent less than 50% of the graphics cards on the market currently and substantially less than that of cards in current use. Therefore *CUDA* is of limited value for game developers where compatibility over a broad range of hardware is important. For a research project where the value of a project is in the analysis of data rather than in the developed software it represents a good choice though and has become very popular in academic circles where it has successfully been applied to many problems [3].

There is another choice of GPGPU API available now: C++ AMP which claims to provide a simpler GPGPU API for C++ developers. [4].
3 The cave creation algorithm

The algorithm was originally conceived, by the author, as a way to generate interesting cave systems for use in computer games. Because the caves are intended for computer game use there are some specific requirements which must be met:

- The programme must run on a wide range of currently available hardware, this is particularly tricky for PC development where there is a very large range of hardware platforms to target. It is also desirable for it to run on console and mobile platforms.
- The generated levels must be interesting to explore and provide opportunity for interesting gameplay situations.
- The code to generate the levels must execute in a reasonable amount of time because users will not wait for long periods of time for levels to appear. It would be desirable for a level to take no more than 20 seconds to build.
- The created levels must render at a reasonable frame rate. In a real application the cave is only part of the geometry that needs to be rendered. Hence the cave system, on its own, should render at approximately 100 Frames per Second (FPS) so as to leave GPU and CPU bandwidth free for the other important rendering and simulation tasks.

Although the algorithm was not developed as part of this project it is necessary to understand it before looking at the application of GPU and GPGPU techniques to it.

3.1 Outline

The challenge loosely breaks down into two components:

- Producing a layout which is interesting to explore
- Producing geometry to make the layout visually appealing.

One of the advantages of the algorithm proposed in this paper is that these two problems are largely addressed separately

3.1.1 Creating the cave layout

The creation of the cave system has two main constraints

1. It must be able to create a variety of cave systems which provide opportunity for interesting game play
2. The cave systems must be created in a reasonable amount of time

This process can be considered as a maze creation problem where the output is a graph of interconnected paths which are navigable and interesting. The structure must allow a user to navigate between all parts of the maze and should hopefully include loops.
E.g. the systems which are produced should be representable as graphs rather than tree structures. A tree structure always provides for exactly one route between any two nodes whereas a graph provides for alternate routes.

3.1.2 The chosen algorithm

The described algorithm is very simple. The cave layout is built as a series of interconnected rooms as per the Growing Tree Algorithm [5]. The main difference is that whilst the original algorithm is designed to work with a 2D grid of tiles in this implementation all rooms are different sizes and are positioned in 3D space.

All the rooms begin as spheres. Using a sphere as the starting point for each room offers several advantages:

- A sphere can conveniently be represented as a vector 4 with \( x,y,z \) the position of its centre and the \( w \) component the radius.
- Distance checks between spheres are very simple to calculate.

Figure 3-1 shows an example of a small cave system constructed from 27 spheres of various sizes.

![Figure 3-1 Result of layout made with spheres](image)

The overlap between each pair of spheres is a disc that lies on a plane whose normal is the displacement vector between the two sphere centres. Finding the parameters which define this disc is a simple application of vector maths and trigonometry.
By limiting the room shape to spheres the creation process is greatly simplified when compared to using a variety of more complex primitives such as cylinders, cones and cubes. This allows more rooms to be added in a shorter period of time and hence build very complex structures out of these simple components.

The level layout creation algorithm progresses as follows:

1. Add a single sphere of radius R at the origin to the list of rooms
2. While more spheres to add:
   a. Randomly select a room from the list of added rooms
   b. Add a new sphere to the list, which overlaps with the chosen room and give it a random displacement from the centre of the adjoining room

An additional check is required to ensure that no more than two spheres ever overlap at the same portal as this would complicate the mesh creation algorithms later. All of the portals must be simple discs on planes.

The number of rooms added directly affects the complexity of the level, the more rooms the complex the maze will likely become. Conversely the following parameters can also be changed to produce different layouts:

- **The probability of the rooms being big or small:** A cave system made with mostly small rooms will have the characteristic of a maze of narrow tunnels whereas a system made with mostly large rooms will feel more cavernous.
- **The displacement used when adding a new room to the existing system:** The new room has to overlap one other room. The relative displacement between the two rooms is determined by a vector which can be biased so as to create cave systems which are either predominantly flat, vertical or any combination. Figure 3-3 shows three such cave systems. The top is generated using a random displacement vector biased along the X axis, the lower right with a vector biased along the Y-axis and the lower left generated with vector which has no bias.

- **The likelihood that a room gets added to the most recently added room:** This variable controls the probability of longer passageways being formed. Figure 3-4 shows three alternative cave layouts generated by changing this value. From left to right, 50%, 99%, and 0% change. Note how with 0% chance the tunnels are very interconnected connected whereas with 99% chance relatively little connection occurs.

By changing these variables during the cave creation process a cave system can be created that has regions which are predominantly flat, vertical, contain small cramped passages or very large open spaces. In this way a single system can exhibit several different areas with very different characteristics. From a game perspective this is important as the user will enjoy this variety and there is the possibility of adding unique game play elements in each section.
This algorithm is relatively straightforward to implement. There are some optimizations that can be done to improve collision detection when adding rooms, such as the application of a simple spatial index, which will significantly improve the code to detect overlaps between the rooms when positioning new ones.

### 3.1.3 Adding the Geometry

Once the layout is complete, geometry must be produced for rendering and for use in the simulation. At this stage the algorithm has produced a list of spheres, which define the extents of the rooms which make up the system and a list of parameters for the portals, which define the intersection between the spheres. This needs to be used to create the mesh data used by the renderer. The format chosen for the geometry is the standard OpenGL geometry. The geometry is entirely comprised of triangles, each of which is represented by three indices from a list of vertices. A separate mesh is used for each room in the cave system. This makes rendering more efficient as a portal rendering system can be used to traverse the visible regions of the cave system and thus cull any non-visible geometry.

The algorithm used here starts by generating a sphere for each room. The resolution of the sphere is dependent on the size of the room, the bigger the room the higher resolution of the sphere. A similar mesh density across all the rooms, big and small, would be desirable as this will help to keep the final geometry looking even and avoid problems with the simulation. The higher the mesh density the more detail can be added to the walls of the cave (e.g. small ledges, crevices, irregular surfaces etc.). Conversely, higher density leads to longer mesh creation times and slower frame rates when rendering due to the quantity of geometry to process.

A new instance of a sphere is created for the room to be processed. The following part of the algorithm removes vertices and triangles that are on the wrong side of the portal and those on the right side are projected onto a smooth surface to give the appearance of continuous cave walls.

### 3.1.4 Calculating the mesh for rooms with two portals

For rooms which have two portals connecting them to other rooms the process is relatively straightforward. The geometry created is a tube with random features added to it.
Once all the vertices are processed they are projected onto the rim of the two portals using the following steps:

- Calculate the shortest distance of each vertex to all of the portal planes in the room by application of the dot product to the vertex coordinate the portal normal. If the result is negative then the point is on the “wrong side” of the portal and needs to be removed later.
- Scale the portal normal by the previously calculated distance of the vertex to the portal plane
- Add that to the vertex. This is the point on the portal plane which is closest to the original vertex.
- Subtract this from the centre of the portal and normalize the result. This produces a vector pointing from the centre of the portal to the projected vert.
- Scale this vector by the radius of the portal and add it to the centre of the portal. This is the point on the portal disc which is closest to the original point.

At this point the algorithm has calculated a point on each portal which is closest to the vertex. Using these points, and the portal normals it is possible to create a hermite spline. The point can be projected onto the spline using the distance of each point from each control point to work out the ratio of the point along the spline. This operation is repeated for every vertex in the original sphere. Figure 3-5 illustrates the process.

![Figure 3-5 Projecting a vertex onto the room mesh](image)

If this was the only operation performed then the result would be a completely smooth set of tunnels perfectly circular in cross section. For a cave system the tunnels need to have irregularities along their length. This is done by means of Perlin noise. [6][7]

The reader may be questioning the use of the donor sphere suggesting instead synthesise the mesh for the tube completely from scratch. The process sounds simple: create a set of
evenly distributed points around the rim of each portal, join them with *hermite* splines, step along the splines at suitable interval to generate vertices and then algorithmically calculate the indexes for the required triangles. This would work in the case of a simple tube, however there are some advantages to this technique. This algorithm generates a mesh with a reasonably even density of triangles on the surface. If the mesh was created using the algorithm above and the tube is distorted then a much higher density of triangles will occur around the inside than outside. There are algorithms that can be used to correct for that inconsistency but the code is quite complex and time consuming. Further this algorithm can be adapted to work with rooms containing more than two portals, which is a much harder problem to solve using other techniques.

### 3.1.5 Generating a mesh for rooms with three or more portals

The process of creating surfaces for rooms which have more than two portals is slightly more complex but the algorithm used is an extension of the one described above. Rather than create a single tube between the two portals multiple splines are created between every combination of portals. Figure 3-6 illustrates how multiple splines are used when creating a mesh for a room with three portals.

![Figure 3-6 Creating a mesh for a three portal room](image)

Each vertex is project onto each spline in turn. The distance of the projected vertex to the original vertex is summed and stored in a weight. The various weights are used to calculate the final position of the vertex using a weighted average. Thus a spline that is close to the original vertex will have a high influence on its final position whereas one that is far away will have little effect. This technique produces a reasonably smooth manifold providing a pleasant blend between none overlapping portals without resorting to very complex mathematics. Figure 3-7 shows part of a section of a cave system generated using the algorithm. The cave system shown is generated with no Z component hence it is flat and perpendicular to the Z axis. The lighting artefacts, which
are visible, are caused by normal mapping in the fragment shader and not from the geometry.

![Figure 3-7 Output of mesh smoothing algorithm](image)

### 3.1.6 Perlin Noise

The algorithm as it stands generates smooth meshes. To make the rooms look like caves some sort of pseudo random deformation needs to be applied. *Perlin* noise is an algorithm ideally suited to this type of application.

*Perlin* noise has the advantage that it can be used to produce several octaves of noise at different amplitudes thus allowing large features (bumps and crevices) in the system and smaller more lumps too. Pleasing results can be obtained this way and various octave frequencies and attenuations can all be adjusted to create different effects (very jagged rough rocks such as granite through to smoothly curved rocks such as sandstone). With a judicious use of look up tables and other optimisations the algorithm is reasonably quick to compute.

The portals need to be irregular in shape. The easiest way to achieve this is to use *Perlin* noise to alter the effective radius of the portal when creating the control points for
the splines. The coordinates of the vertex being deforming is used as the parameters to derive the Perlin noise. Perlin noise returns a single floating point value which, can be scaled appropriately and added to the radius of portal in the final step of the vertex projection code. This produces a series of splines with start and end control points that are displaced somewhat from the portal outer disc.

The vertex coordinates are in world space so vertices on either side of the portal which are close to one another will generate similar Perlin noise values (vertices in two separate rooms which share the same world space coordinate will generate identical values from Perlin noise). In this way the manifold created by this technique will be random within each of the rooms, but at the point where two rooms join in a portal the same random values will be generated and a smooth join between the rooms can be achieved (some stitching of vertices is still required though to remove all the seams as discussed later).

For the larger rooms an additional lower frequency Perlin octave, with higher amplitude, is added and scaled based on how far away the point being sampled is from the portal. In this way the portals themselves are not affected by this low frequency noise, important otherwise unsightly seams would result between the rooms, but there are interesting features towards the centre of the rooms. Figure 3-8 demonstrates the result of adding Perlin noise to the cave mesh as seen from inside the mesh.

Figure 3-8 Screen shot of a cave mesh after the addition of Perlin noise
3.1.7 Joining the room meshes

Once all the vertices are created the triangles on the wrong side of the portal need to be removed. As part of the previous stage the vertex that were moved to the rim were flagged as this helps identify which triangles need removing. If a triangle is made up entirely of vertices on the rim then it needs removing from the list, conversely if only one or two vertices are on the rim then it is part of the list of triangles which make up the interface between one room and the next. At this stage unused vertices can also be removed.

The algorithm has now generated a list of cave rooms with geometry. These can be used to render the cave system but there will be obvious seams between the rooms. The reason for this is that although all the points exist on the same 3D manifold the points on two meshes on opposite sides of a portal do not exist in exactly the same place. To get rid of the seams the vertices of the two meshes need to be stitched together. Figure 3-9 shows an example of a joint between two cave rooms, before and after welding. The red line is the joint.

**Figure 3-9 Illustration of mesh welding**

Stitching vertices on two arbitrary meshes together is usually complex however in this case quite a bit of information is known about the vertices, which makes the process easier. Only vertices that were moved to the rim of a portal need to be stitched and these were flagged in the previous stage. Also these vertices lie on the plane of the portal and are distributed evenly around the edge of the disc that defines the portal. A transform matrix representing the rotation of the portal can be used to project the vertices from world space onto the plane of the portal. The vertices will all have a Z value of 0 after the projection and the problem becomes 2D instead of 3D. The points are then converted from Cartesian to polar coordinate system and the length of the vector can be ignored (it is nominally the same for all points). Only the angle of the polar coordinate is required to identify each point, effectively reducing the problem from two to one dimension. This can easily be solved by sorting the points on both sides of the portal using the angles. It is then straightforward to determine which points are closest to each other and either move or insert new points to remove the gaps between the meshes. The advantage of this
algorithm is that it’s fairly simple and runs in (O)n time where n is the number of vertices at the interface between two rooms.

Vertex normals are created at this stage and smoothed along the join between the meshes. This is done by creating per primitive normal then summing normal at each member vertex. Once all the primitives in all the cave rooms are processed the normal for each vertex is summed included those welded across the portal, then all the normal are normalized. This produces continuously smooth normal across all primitive edges including those at portal boundaries.

3.1.8 Marching Cube

It is worth describing another more traditional algorithm used extensively for creating mesh data from a cloud of points - Marching Cubes [8]. In this algorithm a volume is space is divided up into a three dimension grid of boxes. The algorithm begins by examining each of the boxes in turn to see which of its vertices are inside or outside the solid region of space. The algorithm uses this information to pick the geometry for the box that best suits the combination of edges.

Once all the boxes have been examined the algorithm has generated a mesh that defines the surface of the volume expressed by the function. Whilst this algorithm works well it is quite slow and as resolution is increased so becomes very slow to execute. Also in this case it would be difficult to derive a function which defines the surface from the portal data.

Compared to marching cubes the process described in this thesis for creating the meshes may seem more complex but it offers several advantages:

- There a high degree of control over the layout of our cave system and control over the percentage of narrow, wide, long short passages we want.
- The density of the geometry used to create the mesh can be controlled accurately and is reasonably even.
- It is possible to add additional features into the caves, such as flat floors, domed ceilings, stalactites and stalagmites, with some additional functions.
- The cave system is constructed from a series of rooms that can be rendered efficiently using the portal technique described earlier for occlusion.
- The system can easily be further adapted to generate new rooms on the fly as part of a separate processing thread, opening the way to unlimited sized caves.
3.2 Implementation

The original implementation for the cave system was undertaken using the Unity3D game creation framework [9] with all of the program code written in C#. The framework uses the Mono framework, which is an open source implementation of Microsoft’s .NET and provides a set of tools including C# compiler and common language runtime. Importantly it provides a very rich set of abstract data types (containers) and Unity3D provides an extensive collection of linear algebra functions. This greatly simplifies the creation of complex algorithms and allows for much faster prototyping of code. This was the main reason for the choice of this framework over C++ when developing the algorithm.

The final algorithm was demonstrated running in Unity. Cave creation times vary depending on the complexity and size of the cave but a cave system with 100 rooms and approximately 300,000 vertices in it took approximately 40 seconds to build.

The later implementation and the one whose performance is analysed in depth in this report, is written in C++ from scratch. The main reason for this switch was that earlier versions of Unity3D did not support GPGPU development. Later versions offer some support for GPGPU but it was felt that to really understand the process involved in GPGPU development it would be easier to start from scratch.

Once the algorithm was rewritten in C++ testing was undertaken. It was discovered that total execution time for a 100 room cave system on the CPU was 9 seconds.

3.3 Rendering of the cave system

3.3.1 Render Pipeline

The renderer used for this project is a classic deferred renderer. No special optimisations are made. All of the lighting calculations are carried out in screen space.

3.3.2 Texturing

The vertex data for the cave mesh does not include texture coordinates therefore traditional texturing techniques will not work. Whilst it would be possible to create texture coordinates it is tricky to do this in such a way that the coordinates map evenly to the texture because the mesh is based on a sphere which cannot be easily unwrapped to a plane. Texturing of the mesh is more easily performed using a technique known as
triplanar projection. There are several good article giving clear descriptions of the technique [10].

With a slight modification it is possible to extend the algorithm to apply a different texture to the walls, ceiling and floor. This can be used to give the cave extra realism with dirt floor, rocky walls and mossy ceilings. With further modification it is possible to blend multiple textures together to get effects such as rock seams or different rock types in outcrops. It is even possible to blend in different specular and gloss masks at different points in the cave system thus further differentiating specific areas of the map.

Finally we mention normal mapping. This is a very effective technique that adds lighting detail to the scene. The technique works by warping the surface normals by a normal map, usually in the fragment shader. The technique requires normals but also tangents and bi-tangents to the UV coordinate system used to map the textures. Whilst normals are passed through with the vertices, tangents and bi-tangents are not as these must be aligned with the texture axis, which is not possible in this case as there are no texture coordinates.

The tangents and bi-tangents can be easily synthesised in the vertex shader. By finding the cross product of the normals with a vector pointing along one of the axis (say 1,0,0) a suitable tangent can be found. By taking the cross product of this tangent with the original normal the bi-tangent is found. These can then be passed through the render pipeline down to the fragment shader and used in the usual way to create the tangent, bi-tangent, normal matrix for use when transforming a normal map from texture to world space. [11]

3.3.3 The choice of premade or procedurally generated texture

It is possible to procedurally generate effective textures and normal maps for use in rendering. Using suitable algorithms it is possible, for example, to create wood grain, concrete, stone and plastics. [12]

However investigation reveals that the quality of the result is highly dependent on the time taken to generate the texture. To generate a realistic rock texture really requires an offline process. It is not practical to create effective pseudo random textures that simulate natural phenomena in the render pipeline. Instead most of the known real-time techniques generate pseudo random textures through a combination of blend and distortion of existing textures. This is the approach adopted in the cave rendering algorithm. Textures are loaded when the cave system is generated, then used for
rendering. The source of the textures is currently from a file but could equally well be generated as an offline process and stored for later use.

### 3.3.4 Hidden detail culling using portal rendering

A very useful by-product of the cave creation algorithm described is that it naturally breaks the cave system down into a set of rooms connected by portals. The geometry for each room is completely contained within the original sphere’s volume. The portal connecting the room to its neighbour lies on a plane and the aperture is within the radius of the centre point of the portal that was originally found when creating the layout. Hence the contents of an adjacent room can only be visible if the portal to it is also visible. This provides the opportunity to use a simple but powerful, visibility determination algorithm known as portal rendering. Usually for this technique to work an artist or level designer has to manually place portals in the level. The cave creation algorithm creates the portals automatically as a bi-product of creating the geometry.

The portal rendering algorithm briefly works as follows:

- Identify the room the camera is currently in push it onto the list of rooms-to-render.
- Repeat whilst there are rooms in the list of rooms-to-render
  - Pop the first room of the list and render it.
  - Project the portals that are part of the room into clip space (clip space is initially the size of the screen but is reduced each time the algorithm traverses a portal to the extent of the portal)
  - For portals that overlaps visible space, add the room which it connects to into the list of rooms-to-rendered

As rendering continues through each portal the size of the clip space reduces. Less objects are rendered in each room and the chances of new portals intersecting with the clipping area are reduced. The technique is most effective in room layouts where portals are not aligned in a particular direction so that line of sight is limited.
4 Acceleration using OpenCL

4.1 Introduction

The steps involved in creating the cave system are:
1. Create the layout using spheres
2. Find the portals between the spheres
3. Create donor spheres
4. Process the sphere meshes to create the cave rooms
5. Stitch the meshes together
6. Render the cave system

In the cave creation algorithm described in the previous section it is the fourth step, the processing the donor sphere to form the cave rooms, which is the most intensive from a processor perspective and also lends itself best to a GPGPU approach. The routine can be written where each vertex is processed completely independently to the others, this is referred to in parallel processing jargon as \textit{embarassingly parallel}. It is therefore possible to process each vert on a separate processing thread. In GPGPU terms a processing kernel can be used to do this.

The simplest way to do this is to pass all the vertex data for the spheres to the GPU, deform the verts using the described algorithm and then read the data back into main memory so the remaining steps can be performed. This does mean that data is being moved between main memory and GPU memory but the bandwidth of the data-bus is very high so there should not be too much latency as a result of this. It is theoretically possible to do all the processing on the GPU and leave the data there ready for rendering. This approach is problematic though because the data is very large in a typical cave system and the problem of building it on the fly, buffering, streaming etc, is beyond the scope of this project.

4.2 Implementation

Before coding can begin a suitable \textit{GPGPU API} needs to be chosen. As this project will eventually be used in a virtual world, possibly computer game, compatibility over a large range of hardware is important so \textit{CUDA} is a poor choice as it only works with NVidia hardware. Compute shaders are more suited to calculations within the render pipeline and in this case we want to process the vertices as a separate process and store the mesh for future use. This leaves Direct Compute and OpenCL. OpenCL is more versatile in terms of target operating system and also works well with OpenGL, which is the API used
for rendering in the engine within which the framework will need to work. A second advantage is that OpenCL code looks a lot like C but with some syntactic sugar to make vector processing easier. The author has substantial experience of working in C and C++ so the OpenCL API already had familiar elements to it compared to Direct Compute or CUDA. For these reasons OpenCL was chosen as the GPGPU API for this project.

The OpenCL development process begins by rewriting the C++ code to be as close as possible to OpenCL kernel. OpenCL kernels are self-contained processing units and therefore cannot make use of pointers to structures outside of the function. A first step then is to ensure that all variables that the kernel needs to work with are passed into it in a suitable way. In OpenCL this is achieved by means of the `clSetKernelArg` function. As many variables as required can be passed into an OpenCL kernel although it is easier to group them into buffer if more than a few are needed. In this case we pass in the following two buffer:

**meshStructBuffer** – An array of structures that represent the vertices making up the donor sphere. When the processing kernel is called it will work on each of these in parallel. On entry the vertices will be on the surface of the sphere. On exit they will have been deformed into the final room mesh with vertices which need have been moved to the rim of a portal flagged appropriately.

**portalBuffer** – An array of structures which represent the portals which define the interface between this room and its neighbours. Each portal structure contains the portal centre, radius and normal. For simplicity we limit the maximum number of portals to a predefined value (10 in our example cave) so we can pass this in as a static array and handle it as such in the kernel. The main difference between the meshstructbuffer and the portalbuffer is the allocation of memory for them. The meshstructbuffer is dynamic, each kernel needs its own copy of the mesh and our code will generate as many instances of the kernel as possible up to the number of vertices in the buffer. The portalbuffer on the other hand is a constant array and sits in shared memory.

The following constants are also passed to the kernel:

**Integer**: **PortalCount** – The number of portals in the array

**Vector4**: **Room Centre** – The centre of the room in world space, a vector 4 is used to avoid alignment errors on the GPU.

The development process was very straightforward. The code had already been thoroughly debugged in C before porting to OpenCL. This meant that the lack of good debugging software in OpenCL was not a huge problem although there were still some
issues. In general it was found that the safest approach to programming was to complete the process piecemeal, with a thorough test of each new section before continuing. Whilst it is not possible to set break points and view the contents of variables during the execution of the kernel it is possible to write test variables into the return buffer so that the values of intermediate results can be checked.

The biggest issues encountered stemmed from the way in which address spaces are handled in OpenCL and how this differs from C. Each kernel in OpenCL must run completely independently of the others. The use of global variables is not allowed. To allow all kernels to access the same data it must be passed into the kernel in the header arguments and these need to go into the correct disjoint name space. In this case the array of mesh structures are placed in the __global disjoint space so that all kernels are read and write to independently. The portal structures are in __constant space so they can be read from. In C and CPP the use of global and constant is also common but with different results from OpenCL. Constant is usually merely a compiler optimisation but in the case of OpenCL it causes the program to place the variable in an entirely different part of memory. GPU have a more complex memory structure than a traditional CPU which allows for different clock speeds when reading and writing. It is important, from an optimization perspective, to ensure variables go into the correct space.

The OpenCL kernel expects all data to be 64bit aligned and it is important to ensure that this is the case. One particular crash bug was encountered as a result of this and proved difficult to fix. It was caused by placing Vector 3 floating point numbers into the structure that was sent to the kernel. On the CPU side the data was packed on 32 bit boundaries but on the GPU side it was assumed to be 64 bit aligned.

4.2.1 Performance Results

Cave build process execution time analysis

The build times for the various stages in the cave creation process where analysed and the results shown in Table 3.9

Times for each of the five major steps in the process are shown for comparison. Only the smooth mesh time is available for both OpenCL and CPU versions. The results show that OpenCL represents a significant advantage over the CPU for the mesh smoothing operation. Before the optimization mesh smoothing is the most significant cost in terms of processing time. After the optimization it is the least significant. Creating the normals represents the next most significant execution time and this would probably also benefit from conversion to GPGPU processing.
Comparison of CPU and GPGPU performance as applied to a novel cave generation algorithm

Figure 3-10 Execution times for the different stages in the cave algorithm.

**Vertex buffer size comparison test**

In the next experiment the time taken to process an individual buffer of verts is compared as the number of vertices increases. This was measured by creating a variety of vertex buffer sizes and running them multiple times in OpenCL to obtain an average execution time. The graph in figure 3-10 shows how execution time increases linearly with the number of vertices. Each test run was repeated 1000 times.

Figure 3-11 Execution times plotted against number of vertices processed.

Note how with only 10 verts execution per kernel is still 1.6 ms. The 1.6Ms is the time taken to set up the kernel regardless of the amount of data to be processed.

**Function complexity comparison test**

The next experiment was designed to measure execution time compared to complexity. The OpenCL kernel was modified so that immediately after starting the Perlin noise function ran and a result returned. A fixed vertex buffer size, in this 64,000 verts was
passed into the kernel and timing data recorded for an increasing number of Perlin noise iterations. Figure 3-11 shows the Perlin noise iterations plotted against execution time.

![Comparison of execution time against kernel complexity](image.png)

**Figure 3-12 Execution time plotted against kernel complexity**

As can be seen the graph is reasonably linear. With a buffer size of 64,000 verts, even when there is no computations the processing time for all of the verts is 1.6ms.

The conclusion drawn from these results is that regardless of how many vertices are being processed or how complex the function applied to each of them is, the execution time is at least 1.6ms. This is accounted for as the set up time for the kernel. The practical consequence of this is that wherever possible it is better to process a larger number of vertices in a single operation rather than smaller numbers in multiple operations. For example in the case of the cave system there are between 1000 and 64000 vertices in each room and 160 rooms in the test case. Processing time could theoretically be saved if all the mesh data was concatenated into a single buffer and processed in a single call. In the case of the cave system this is possible but not trivial due to the fact each room must reference different portal data. In addition the saving in time would be small - maximum of 1.6ms * 160 = .26 seconds. In other applications this might be a significant saving.

**Hardware Comparison test**

The time taken to create a medium size mesh on a variety of hardware configurations was measured. The results are compared in Figure 3-12. As can be seen even relatively old hardware offers a substantial advantage for OpenCL. The GTX 260, which was released in 2008, offers a four times improvement in speed even compared to a modern I7 CPU.

When comparing Nvidia and ATI cards of a similar specification it was noted that the ATI cards offered the greater advantage.
4.2.2 Kernel build and link time

Finally it should be noted that unlike C code, which is compiled and linked into an exe and then distributed, OpenCL code must be compiled and linked into an executable kernel on the target hardware. The time taken for the compilation is dependent on the target hardware and the complexity of the code. In the case of the test hardware, and the source code used in this project, approximately 10 seconds was required to compile and link. Note that this is a one-off cost and once the kernel is built it can be run as many times as required. If this build time is significant then it is possible to retrieve the compiled binary and store it for future use. In most cases this is unlikely to be an issue as the user will most likely not notice the compilation time during the loading of the application.

4.3 Conclusion

The results obtained from this experiment clearly demonstrate that for SIMD style parallel operations GPGPU offers significant performance gains and can be applied to practical programming problems. Development using OpenCL was straightforward and the resulting code ran well on all the hardware tested.
5 Optimization using Tessellation shader

5.1 Introduction

In this section the use of GPU hardware for real-time tessellation is explored. The emphasis is switched away from using GPU for offline techniques to real-time generation of geometry. The method used is referred to as Hardware Tessellation. Support for real-time tessellation of patches in the render pipeline has improved substantially in the past few years. The following paper provides a good introduction to the techniques involved and their applications. [13]

The theory is to generate a much lower resolution mesh the cave creation process (approximately 20% of the vertices), but use real-time tessellation to add detail as the geometry is rendered. Thus the visual quality, when rendered, is the same or better than the original high resolution mesh version.

From the perspective of rendering the advantages of generating a lower resolution mesh are:

- The cave creation algorithm will execute much faster
- The algorithm requires significantly less memory
- The resultant index and vertex buffers are much smaller so less data-bus bandwidth is required to copy the data to the video card
- Detail can be added back into the mesh in real-time based on suitable heuristics so theoretically much higher levels of detail can be achieved in the final render, compared to using a single high resolution mesh, in comparable render frame rates.

Additionally, lower resolution versions of the mesh are required for other aspects of the virtual world including:

- Physical simulation – Physics engines allow for collision detection between arbitrary meshes, as long as they are static, but complexity of the geometry is usually limited and performance improves with simpler meshes.
- Control of None Player Characters NPC – Navigational meshes [14] and Dijkstra or A* path finding [15]. These provide a neat solution for pathfinding around complex environments and this data can be built procedurally from the cave mesh. However building a navmesh, and the resultant connection graph, from a highly detail mesh is computationally very intensive and the time taken to find a shortest path using A* or Dijkstra is also heavily dependent on the complexity of the graph.
Therefore a low resolution version of the mesh needs to be created. Processing time and memory can be saved if the same mesh can be used for both rendering and simulation purposes.

5.1.1 Chosen API

OpenGL was chosen as the API to access the render pipeline. The function of the tessellation shader is to take an existing primitive, called a patch in OpenGL terminology, and subdivided in a number of smaller primitives. The amount of subdivision and the way that the created primitives are be manipulated is completely programmable. Unlike other shaders there are two separate programs involved in producing a tessellation shader:

- **Tessellation Control Shader (TCS)** - sets up the tessellation and controls how much subdivision occurs. It also specifies what control point information is passed into the next stage.
- **Tessellation Evaluation Shader (TES)** – receives each control point from the TCS, calculates its position in whatever spatial system is appropriate and any other parameters required, e.g. normal, UV’s etc. and outputs vertices for the next stage in the render pipeline.

OpenGL can manipulate two types of patch; triangles and quadrangles. The process is very similar for either with minor differences in the process. For this project triangular patches were used because the cave algorithm is more efficient with triangular primitives and passing them through to the render pipe line is trivial. Some of what follows though is specific for triangular patches and will not work with quadrangles.

5.2 Setting the tessellation level

The main function of the TCS is to subdivide the patch into smaller primitives. Subdivision level for each primitive is set using internal and external levels. The internal level controls how many control points will be added to the inside of the primitive. The external level controls how many control points will be added to each of the edges of the primitive.

Internal control is for the entire primitive but the external control is individually controllable for each of the edges. The reason for this has to do with dynamic level of detail (LOD) and the problem of tearing in the tessellated mesh.
5.2.1 Dynamic LOD in tessellation

In the simplest implementation the primitive could be subdivided into a fixed number of smaller primitives. Thus a low resolution mesh could be passed to the GPU, tessellated and smoothed in real-time to give a more pleasing visual effect. The tessellation shader is able to alter the level of subdivision at render time on a per patch basis. This allows the level of tessellation of a patch to be controlled by how much screen area it occupies. For example, for optimum visual quality and performance a patch covering a large amount of screen area should be subdivided to a higher extent than one which is smaller. Whilst all the patches could be subdivided to a high level, regardless of where they are, this would waste GPU power because primitives would be produced which would be very small on the screen. The technique of subdividing based on a real-time heuristic is known as dynamic Level of Detail (LOD) and is one of the major advantages that the tessellation stage offers in a modern GPU.

5.2.1.1.1 Tearing in the tessellation shader

The naïve implementation for dynamic LOD would be to simply calculate how far the patch is from the camera and adjust the level of subdivision on a per patch basis. However this causes rendering artefacts known as tearing. Figure 5.1 is a screen shot of the cave system showing tearing in the mesh. The white lines between the triangles are tearing.

![Figure 5-1 Illustration of tearing during tessellation](image)

Tearing occurs when the level of tessellation on adjacent patches is different and is the reason why the TCS allows for separate control of internal level and independently for
the edges. To avoid tearing the tessellation level for each individual edge is calculated based on an algorithm using the screen space coordinates of its two vertices. Thus the tessellation level for two adjacent edges which are in different patches will be the same and there will be no tearing between the adjacent patches. Once we have calculated the tessellation level for the three edges we can calculate the level for the centre.

The simplest way to calculate the tessellation level is to use the distance to camera heuristic. In a deferred renderer, where the points are already in camera space by the time the TCS is invoked, we simply find the magnitude of the point vector. This can then use this as a divisor to calculate the tessellation level, with a simple clamp to stop the value getting very small or very big.

This works OK if the mesh density is reasonably even. If the mesh is uneven though, e.g. it contains a mixture or small and large primitives, then better results are obtained by projecting the edges to the screen and subdividing based on how much screen area is taken. This is known as the “sphere diameter in clip space” heuristic [16] and is the recommended method.

One of the advantages of the mesh creation algorithm discussed in this paper is that it creates a mesh with fairly even density of primitives so for our application distance to camera is satisfactory as a heuristic and is cheaper to compute than the alternative mentioned above.

During experimentation it was found that the naïve heuristic used to calculate the LOD, dividing by the distance to camera, gave poor results because it does not give adequate control as to what distance tessellation starts at or how much tessellation occurs close to the camera and there is a very steep fall off in tessellation levels immediately close to the camera which produces obvious temporal artefacts when the camera is moved.

A better heuristic would provide separate control of the tessellation distance range, the maximum tessellation level and provide a smoother transition between high and low tessellation levels. After some experimentation the following function was developed and provides significantly better visual quality at higher frame rates:

\[ \text{Tessellation Level} = \text{MaxTessellation} \times \cos \left( \frac{D}{\text{MaxDistance}} \times \frac{\pi}{2} \right) \]

Where D is the distance from the point being tested to the camera, clamped to the range 0 to MaxDistance. This formula provides the desired level of control and ensures that the maximum change in LOD occurs away from the camera.
5.2.2 Smoothing the patch

The previous stage was responsible for subdividing our patch. In the next stage the previously generated control points are projected onto the final surface.

5.3 Point Normal Triangles

The technique used here for generating a smooth surface is point normal (PN) Triangles [17]

Each of the vertices, passed into the render pipeline, has a position and a normal associated with it. A triangular patch is made up of three vertices. The TCS breaks the patch down into a number of smaller triangles and new control points are generated and passed through to the TES. The TES then takes the new control points that make up the triangles, calculates the position of the interpolated vertices and then projects them onto a curved surface defined by the points and normal.

5.3.1 Barycentric Coordinates

When using triangular patches the TCS uses the barycentric coordinate system to specify the position of the new control point relative to the original triangular patch [18]. This system uses area coordinates triples, in the case of triangles, to specify the location of a point relative to the vertices of the triangle. Each of the values in the triple sums to one, each specifying the normalised distance to a vertex.

The use of this coordinate system makes the calculation of the vertex and normal for each control point trivial. The each vertex or normal is multiplied by the corresponding barycentric component and the results summed. The following equations are used for this:

\[
V_{out} = (P_{V1} \cdot B_1) + (P_{V2} \cdot B_2) + (P_{V3} \cdot B_3)
\]

\[
N_{out} = (P_{N1} \cdot B_1) + (P_{N2} \cdot B_2) + (P_{N3} \cdot B_3)
\]

If linear interpolation is used with the original points, normal and barycentric- coordinates then new vertices can easily be created on the plane of the original patch. The next step is to deform that mesh to conform to a smooth surface controlled in some way by the point normal in the original patch. Two techniques were investigated.

5.3.2 Bezier Triangles

There is an excellent tutorial on the use of Bezier triangles in the tessellation shader on line as part of the OpenGL tutorials. [19]
Bezier triangles are a subtype of Bezier Surface. The algorithm is a simple extension of the Bezier line. Each edge of the triangle is a Bezier line, which uses the two control points at end (normals and vertices). The points in the centre of the triangle are interpolated from these. Division of a patch surface basically consists of the usual Bezier transforms to find the control points along the edges followed by addition transform (using the newly calculated control points and the normal of the central control points provided by the TCS) to find required points in the centre of the triangle.

For tessellation these points are projected onto the original patch in order to calculate their final coordinates. The normals have already been calculated as described earlier.

There are many OpenGL implementations for Bezier patches using tessellation shaders available for reference.

5.3.3 Phong interpolation

Phong interpolation is described as a “geometric version of Phong normal interpolation”. [20]

Normals are projected and interpolated on tangent planes to the vertices of the original patch. The resultant projections are interpolated again to find the final points displacement from the original patch plane. Those displacement are then used to calculate the x,y,z of the interpolated point.

5.3.4 Interpolation Implementation

Both algorithms were implemented. Visual appearance of the two approaches is very similar; Phong gives a slightly more rounded appearance than Bezier and offers a very slightly improved framerate in the experiments. In direct comparison of scenes Bezier was approximately 2% slower than Phong and gave subjectively more pleasing results. For that reason it was used for the project.

5.4 Adding additional detail in the tessellation shader

In many computer graphics applications, such as rendering of human heads or other smooth objects, the smooth interpolation of NP triangles is all that is required. In the case of the cave system though a smooth blend between points is not always desirable, as rocks are often jagged and have small imperfections. These are generated using higher octave Perlin noise in the high resolution version of the mesh. When the lower resolution mesh is created these high frequencies will be aliased out. Hence there is no benefit in
adding the higher octaves to the Perlin noise function when creating the lower resolution mesh and this is a further time saving, when only three octaves of noise are generated instead of four. However the noise must be added back in before rendering the mesh and this needs to be done after the tessellation of the patch in the tessellation shader. The dynamic LOD offers the advantage that the high frequencies can be added to geometry which is close to the camera. The appearance of very rough looking edges can be added to the meshes in the foreground.

Figure 5-2 demonstrates how the same mesh can be passed into the render pipeline but appears differently when rendered using tessellation. The left most render has no tessellation, the middle uses Phong interpolation to smooth the mesh, the right most uses a height map to add extra detail to the mesh

![Figure 5-2 Diagram illustrating the results of tessellation in the render pipeline](image)

Two techniques can be used to introduce this high frequency noise:
- Random Noise Displacement
- Displacement mapping using a displacement map.

In both these cases the simplest way to deform the mesh is to displace each control point, which is generated by the tessellation shade, along its normal by a suitable displacement. In this way the underlying shape of the surface remains the same, and folds in the mesh are avoided. This fits neatly into the existing PN triangle process for both Bezier and Phong interpolation. An additional step is added after the interpolation where the random displacement is multiplied by the interpolated normal and the result added to the interpolated point.

The amount of displacement can be generated in a number of ways and these are discussed next.

### 5.4.1 Sampling from a 3D texture

The simplest way to give the effect of noise in a shader is to sample from a texture. Although strictly speaking this is not a true random function if the texture is big enough and the sample rate suitably chosen the effect is close to random. The shader can use the
texture sample function to return the value of the noise at a coordinate. If the sample-rate is below the texel size then bilinear filtering will perform a simple interpolation between neighbouring values in the texture and provide a smoothed version of the random noise.

In practice the X,Y,Z of the sample point is taken in world space, scaled in some way, and used it to sample from a 3D texture. 3D textures are very memory intensive so the texture size is limited to 128 * 128 *128 * 8 bits. Generating the random noise texture is performed on the CPU. Additional octaves of noise can be created by sampling at different frequencies as discussed earlier.

Unfortunately the bilinear filtering that the shader sampler performs is not usually good enough to give the quality of results required. The sampling produces ugly artefacts and edges where the blending is rough. There are other techniques that can be used. GLSL textureGather() allows for the retrieval of four neighbouring texels from a 2D texture, eight from a 3D texture, in a single pass and these can be interpolated to give a better result.

The final technique will generate noise but frame rate is quite slow and there are other problems such as how to find the surface normal.

5.4.2 Perlin Noise

It is possible to implement Perlin noise using the same algorithm as used for OpenCL. This was tested but performance was very poor. Frame rates of 4 FPS were initially recorded. Further investigation revealed that one part of the implementation in particular was to blame, reading the values from the permutation look up table. This was replaced with a texture and a texture sampler. Performance improved substantially to around 30FPS. Further research is recommended to understand why this occurs. Using Perlin noise in the tessellation shader is an interesting option however there are other ways to generate displacement with similar or better appearance and performance.

5.4.3 Using a displacement map

The simplest technique, and the one used in many of the tessellation examples seen in the research, is to use a premade displacement texture. The technique is similar to the random noise texture technique mentioned previously but with the important difference that the texture coordinates of the point being created are used to index into a premade texture containing the displacement. Usually the texture is created as part of the art process. Most modern art packages allow the artist to work with a very high resolution
model, with millions of vertices, and automatically generate a lower resolution model for use in real-time rendering. The high resolution detail, which is lost in the down sampling process, is exported in displacement and normal maps, which can be used to add the detail back in during rendering. Texture coordinates can be synthesised using triplanar projection as previously discussed. It is possible to combine several height-map textures (sampled at different frequencies) to simulate different types of surface roughness. [21]

This is the fastest way to generate displacement and provides good results as long as there is a displacement texture available and the user is not completely wedded to the idea of procedural generation of everything. As previously noted, random generation of good quality textures in real-time is not feasible on current generation hardware. Premade textures can be obtained which are matched with normal maps and displacement maps.

5.4.4 Calculating the normal

The above techniques all work to a certain extent but there is a problem. The interpolated normal will no longer be correct after the points have been displaced. Without recalculating, the normal, lighting and the triplanar texturing in the fragment shader will not look correct. There are four main techniques that can be implemented:

- Synthesizing normals in the Geometry Shader
- Synthesizing normals using three point interpolation in the TES
- Derivative of the noise function
- Displacement map and normal map

Each technique has its strengths and weaknesses and where tested during development with the following results

5.4.5 Creating normals in the Geometry Shader

The geometry shader is a programmable stage in the render pipeline that occurs after tessellation and before the fragment shader. Like the TCS it has access to an entire primitive, conversely unlike the TCS it can be used to radically change the structure of the geometry that is being processed, to the extent of completely removing and adding new points.

In this application the fact that the geometry shader has access to all three points in each triangle the TES outputs is very useful. For each triangle a normal vector, to its
surface, is calculated by finding the cross product of two edge vectors and normalising the result. This is written out by the geometry shader for each vertex in the primitive.

This technique generates a single normal for each triangle. This gives an appearance of “facets” in the rendered image. There is no way to blend the normal of adjacent triangles in the geometry shader but in the case of rocks the faceted appearance that it creates looks good. Rocks are usually quite jagged and this technique produces a convincing effect. Unfortunately the technique is rather expensive in terms of GPU time because the geometry shader is quite slow [22]. Adding the geometry shader into the pipeline to create the normal reduced the framerate from 40FPS to 32FPS in the displacement map example.

5.4.6 Three Point Sampling in the TES

The second approach is to synthesise a point normal in the TES. The TES also does not have access to neighbouring points in the patch so those cannot be used to generate the normal. Instead two additional sample points are creating by adding small deltas onto the sample point. The delta are projected along the tangent and bi-tangent to the original normal. The distance chosen for the delta needs to be small enough that high frequency noise is not aliased out but big enough to avoid floating point errors; (.01) works well. The calculated deltas are then added to the centre point and whatever technique used for displacement is used again with the new points. The two new displacement and the original displacement can then be used to calculate the normal by the usual technique of cross product. The resultant normal is passed through to the next stage.

This technique works but requires that the displacement technique described earlier be repeated two additional times. This further exacerbates the performance issues previously described and makes Perlin noise impractical for real-time applications.

5.4.7 Derivative of the noise function

It is possible to calculate the derivative of the Perlin noise function. Quilez et al. [23] describes this process and it is possible to calculate the normal from the derivative. This technique was not experimented with as the time taken to generate Perlin noise in the tessellation shader was already too slow to be practical and the additional step of generating the normal would have slowed things down further. However this sounds like a promising area of research.
5.4.8 Displacement map and normal map

The easiest solution to the problem of the normal is to use a precalculated normal map in conjunction with a displacement map. If the displacement map is used for displacement of the points and there is a corresponding normal map then this should provide us with the modified normals. The data from the normal map will need transforming to match any scaling which is applied to the displacement map.

5.5 Implementation

A low resolution version of the cave system was generated. This has approximately 20% of the geometry data compared to the high resolution version. If the resolution of the mesh is reduced further then visual artefacts appear along the joints between the rooms after welding of the meshes. Creating a lower resolution mesh data for the cave systems is trivial, simply start with lower resolution donor spheres.

All of the shader code was produced in Visual Studio in separate files. Visual studio has no in-built support to help with writing shader code, however there are plug-ins, which streamline the development process to a certain extent. Nvidia provide Nsight, which is quite a good tool as long as the developer is working on a windows machine and the video card being used is an Nvidia card. [24]

Shader code is harder to write than normal C or C++ code because of the poor debugging environment and requirement to compile the shaders every time the application is run. As with OpenCL the best approach is to take things one step at a time and thoroughly debug each stage before moving onto the next.

The render pipeline for the cave geometry looks like this:

- **Vertex Shader**: We transform all the verts into view space
- **TCS**: Tessellation level is calculated based on distance to camera of the patch and control data created dependant for the Phong interpolation created
- **TES**: Phong interpolation is completed, Triplanar projection is used to index into displacement map to add noise
- **Fragment Shader**: The final stage of the pipeline - the fragment shader, remains similar to that used in the none-tessellation version. There is one important difference though which is caused by the triplanar texturing shader. As previously noted when the points are displaced by the height-map the interpolated normal also need recalculating, this is done by using the normals from the normal map which corresponds to the displacement map. If the interpolated normals are used for the triplanar texturing in the fragment shader
then unpleasant stretching can be seen in the texture because the normals do not match the surface normals for the displaced primitives. This was partially solved by using the deformed vertex and normal from the normal map to control the blending. Some stretching is still evident, possibly because the displacement map scale is not quite correct for the normal, but it is less evident.

Three further refinements were added to the displacement code:

- To add more character to the levels the displacement was biased so that it mostly affects surfaces on the underside of rock outcrops. To do this the Y component of surface normal is taken in world space, biased so that it is in the range 0 to 1, before being used to scale the displacement. This gives the impression of stalactites on the underside of the ledges and lumps and cracks on the walls.

- The displacement was scaled using a further pseudo random algorithm based on the point’s world space. This was done to add further variety to the world with some surfaces having a rougher surface than others.

- The displacement value was passed through into the next stage of the render pipeline and used to control tinting and alpha levels in the fragment shader. In this way the tips of the bumps were almost white and shinier giving the effect of stalactites.

5.6 Results

For the first experiment a fixed tessellation level was used. The resolution of the cave mesh, passed into the render pipeline, was reduced by approximately 25% by creating the mesh using lower resolution spheres. Tessellation was used to increase the resolution of the mesh to the previous level and timing data recorded. Figure 5.3 compares various levels of tessellation with render frame rate. The high resolution mesh renders with a frame rate of ~53FPS, When the resolution is reduced the framerate improves slightly to ~55FPS. After tessellation is incorporated into the render pipeline the frame rate drops to ~ 44 FPS.

Therefore we can conclude that passing a lower resolution mesh into the render pipeline and tessellating in real-time to improve the level of detail is less efficient than passing in a higher resolution mesh. As the level of tessellation increases, the frame rate goes down.
However there are other reasons for using tessellation as previously noted. *Dynamic LOD* offers the chance to get higher levels of detail in the mesh for reasonable frame rates. The last two entries in Figure 5-3 demonstrate this. The performance of *Dynamic LOD* was used and set up to generate a tessellation level of 50 at the camera and the fall off in *LOD* adjusted so that it was not noticeable. In this configuration frame rate was 45. Finally, the geometry shader was added into the pipeline and frame rate dropped again to 40 FPS.

The code was run on a variety of hardware and the results shown in Figure 5-3. As can be seen there is a significant improvement in build time for the geometry but frame rates are significantly worse when using tessellation.
combination of displacement maps and normal maps is the cheapest solution in terms of GPU usage but also the least visually attractive.

5.6.1 Geometry Shader

When using the geometry shader to create normals, visual quality improved substantially, edges were much crisper and lighting more convincing compared to the use of the normal from the normal map. For rocks, which are by their nature jagged, the faceted look of the resultant geometry is very convincing.

Unfortunately the results show that using the geometry shader resulted in a significant reduction in frame rate. For the cards tested frame rate dropped to between 60% and 80% of non-tessellated levels. When comparing similar cards by ATI and NVIDIA we note that Nvidia cards outperform ATI in this regard.

5.7 Conclusion

This chapter focussed on the use of tessellation to improve the visual quality of a low resolution mesh. Phong and Bezier interpolation were described and an implementation explored. The results show that Phong tessellation has a marginal advantage in terms of render speed and quality.

Techniques were then investigated for adding high frequency random noise into the geometry to simulate cracks and minor irregularities in the rock surface. Several techniques were explored. Perlin noise proved too slow to implement to a reasonable standard. Displacement mapping, using a pre-made displacement map, worked well and gave good results particularly when combined with a suitable LOD heuristic.

The use of the geometry shader to generate normals gives the best visual results for a rocky cavern but is quite expensive in terms of processing time. Multi sampling of the height map might give better results but that has not yet been implemented.

Using the normals from the normal map to control the lighting and to control the texture sampling in the triplanar shader does work somewhat but is not as attractive as the result from the geometry shader. It is not known at this stage if the reduction in visible quality is an inevitable result of using the normal map for this purpose or if other refinements to the algorithm might address the problem somewhat. Clearly a lot more research needs to be undertaken in this area. In particular there is a complex interaction between the tessellation and the triplanar shader that merits further investigation.
6 Conclusion and further work

This report began by describing a novel technique to create caves systems which was invented by the author. The project went onto to describe ways the original algorithm could be improved by the use of GPGPU acceleration. The final project been shown to generate cave systems which satisfy the requirements described at the start of the report:

- **Interesting to explore:** the interconnected nature of the cave systems, the structure which is used and the semi-random nature, provide a great variety of spaces to explore
- **Attractive to look at:** The combined use of mesh smoothing algorithms, Perlin noise and texturing techniques produce an attractive world to explore
- **Fast to generate:** The use of OpenCL in the creation process greatly reduced the time taken to perform the mesh smoothing operation, which was the major bottleneck in the original algorithm. Other parts of the algorithm are now slowing down the creation process and would benefit from further work.
- **Render at a reasonable FPS:** Frame rates on modern hardware are generally good. With tessellation turned on the frame rate drops somewhat but is still within an acceptable range. Once the portal rendering system is added into the render process there should be at least a doubling of the frame rate. This requires further work

*OpenCL* is a very effective way to accelerate SIMD problems. In this project a 2000% improvement in performance on top end machines was recorded. Performance varies based on the hardware but it is always advantageous to use OpenGL

Tessellation is harder to quantify. Modern GPU are highly optimized for processing geometry. Very complex mesh data can be passed to the GPU and through the render pipeline very efficiently. Using the tessellation shader to “fill in the missing verts” does not improve performance significantly in our experiments. Various lighting and fragment shader effects are available to improve the visual appearance of our rendered meshes as is.

Conversely what tessellation provides is the ability to add large amounts of interesting detail at the render stage and this does add to the visual quality of the scene. Adding that amount of detail into the mesh would require very large amounts of mesh data.
6.1 Further Work

6.1.1 Investigation of algorithm performance compared to marching cubes

For a thorough test marching cubes would need to be implemented on both CPU and GPGPU and applied to the cave creation algorithm. The performance of the two algorithms would need to be compared.

6.1.2 Noise generation in the Tessellation Shader

The implementation of Perlin noise tested was too slow to be viable in the engine. Unfortunately time constraints prevented further research but there are possibilities which are worth pursuing. In particular the use of 3D textures for look up tables should be explored.

Generating the normal for the surface proved a problem. Current techniques are to either generate it using the geometry shader or to sample three times in the tessellation shader at close points to generate a micro triangle and use basic linear algebra to calculate the normal to it. The use of the geometry shader is slow and we would expect that sampling addition times will also be slow. The advanced Perlin noise function referenced in the report mentions that it returns the three derivatives to the Perlin noise. It may be possible to use these to create the normal to the surface. Further research is required. [25]

6.2 Comparison of hardware

The GPU used in the development was exchanged for six different GPU cards and frame rate, mesh build times where compared for both tessellation and none tessellation.

As expected FPS improves as the processing power of the GPU improves as does the speed advantage offered by OpenCL. Interestingly even on a relatively old card- Nvidia GTX 260 - OpenCL offers a substantial advantage over the CPU – about 500%. On the later cards that jumps to ~2000%.

To draw any sort of comparison between cards there needs to be a way of quantifying them. The simplest is to rate them on cost. With this metric ATI cards always outperform Nvidia in terms of OpenCL performance and vice-versa for tessellation frame rate. For example the GTX650 and HD7800 can be considered roughly equivalent in terms of market placement. Unfortunately we did not have a GTX650 to test on but a tests were
conducted on a GTS250 which is similar to the GTX650. In those tests Radeon offers significantly quicker build times with OpenCL compared to the Nvidia card but the Nvidia card renders the tessellated scene quite a bit faster.

It’s interesting to hypothesise the reasons for this. Nvidia have put considerable effort into improving tessellation performance on their GPU with significant research and dedicated hardware added to the new cards. ATI appear less interested in tessellation. It is not as clear why OpenCL should perform so much better on ATI. One explanation lies in the fact that Nvidia promote their own API for GPGPU programming in the form of CUDA but ATI rely on OpenCL and Direct Compute. So whilst it is probable that there is no reason why OpenCL could not perform just as well on Nvidia as Radeon hardware there is less incentive for Nvidia to produce highly optimised versions of it, instead their effort presumably goes into optimizing CUDA. An interesting exercise would be to rewrite the OpenCL code in CUDA and compare performance. One would expect it to outperform OpenCL.

The oldest card (GTX260) failed to render the tessellated version of the cave system, as the shader did not link correctly. No further tests were conducted as to why but some research did not uncover any systemic problem with this card and tessellation so the crash could probably be solved. However frame rate without tessellation was already down at 18 FPS so it seems unlikely that such old cards would ever give reasonable performance.

Further investigation of GPU hardware is needed to better understand where the bottle necks are and how to improve performance

6.3 Comparison of different GPGPU API

As noted earlier there is a choice of API available for GPGPU programming. OpenCL was chosen for this project and this has provided very promising results. It would be interesting to compare the performance of OpenCL with other API though. In particular one would expect to see a performance improvement if CUDA was used because of the tighter coupling between API and hardware. In that case NVIDIA cards would have to be used for testing purposes but for some applications this would not be a problem, for example in commercial installations where the type of hardware used can be dictated by the developer. This interests the author because although it seems reasonable to expect that CUDA will outperform OpenCL on CUDA enabled devices, little quantifiable data is available to back up hardware manufacturers claims in this regard.
A comparison of Direct Compute, Compute Shaders and C++ AMP would also be very interesting. Comparison of the ease of use of the API and available tools for debugging is also very important. In the authors experience having effective development tools and good documentation makes a significant difference to a programmer's productivity with a poor development environment going a long way to cancelling out theoretical performance advantages that sophisticated hardware may offer.

6.4 Final Notes

In the authors opinion this has been a successful project which has applied several modern techniques to a practical problem. The results are very encouraging and demonstrate that it is possible to build interesting and attractive environments for use in computer games and other applications.

This project has only addressed the issue of generating environments, but similar techniques can be applied to other elements of a virtual world including the generation of mesh and texture data for vegetation and animal life. The technology can also be applied to creation of suitable behaviour simulations for the animals and plants.

Many interesting questions have been raised which required further research to answer.

7 The Artefact

The game engine used for this project is a deferred renderer. The majority of the code was not written for this project but is required to run the samples.

The following sample code is submitted:

- Example OpenCL loading framework
- OpenCL Kernel
- Phong Shader code - including vertex, TCS, TES, Fragment
- Bezier Shader code - including vertex, TCS, TES, Fragment
- Phong Shader Code with Geometry shader for normal creation - including vertex, TCS, TES, Geometry, Fragment
- Vertex shaders for the above code
- An executables demonstrating the key elements of this project:
  - CaveRenderer.exe
- Two Batch files to simplify launching the project:
  - CaveNoTessellation.exe
  - CaveWithTessellation.exe
8 References


9. Unity3D. Unity 3D. Retrieved from Unity 3D: http://unity3d.com/5?gclid=CjwKEAjwnf2wBRCF3sOp6oTtnjYSJAANOfheT_EN6D eu8KoyUX9NwcmCW1oArpdFSPeX1tgfY9ghixoCcnDw_wcB


23. Quilez, I. *advanced Perlin noise*. Retrieved from iquilezles.org:
   http://www.iquilezles.org/www/articles/morenoise/morenoise.htm


9 Appendix 1: Study Contract

INDEPENDENT STUDY CONTRACT

Note: Enrolment is subject to approval by the projects co-ordinator

SECTION A (Students and Supervisors)

UnitID: ______U4750194_________
Surname: Oakden_________ First Names: Tony_________
Project Supervisor (may be external): ________________
COURSE SUPERVISOR (e.RCS academic): Dr Eric McCreath_________
COURSE CODE, TITLE AND UNIT: COMP6470 Special Topics in Computing

Semester: [ ] S1 [ ] S2 Year: 2015

Project Title:
Comparison of CPU and GPGPU performance as applied to procedurally generating complex cave systems

Learning Objectives:
The student would gain a good understanding of GPGPU software development. With a focus on looking at performance and scale relating to computer graphics and procedural generation. Also it is expected that the student would gain general skills relating to: writing a report, and giving a seminar.

Project Description:
Graphics applications often involve the processing of large amounts of data. In general many data points, sometimes millions of three dimensional points, need to be transformed. In traditional processing this is done in sequence, by the CPU and GPU, but these types of operation lend themselves very well to SIMD (Single Instruction Multiple Data) type parallelization. In theory General Purpose Graphics Processing Units (GPGPU) provide an efficient way to parallelize these sorts of operations, because GPUs are essentially a collection of processing cores which share a large number of Vector Processing Units (VPU). However in practice the advantages offered are not clear cut. Data needs to be moved from main memory to GPU, processors need to be initialized, caches filled, and the data needs copying back to main memory once processing is complete. These bottlenecks reduce the effectiveness of the technique.
An alternative is to perform some of the data transformation as part of the graphics pipeline,

Research School of Computer Science
Jan-12

Form updated
in either the geometry or tessellation stage. This has the advantage that large amounts of data
do not need to be copied back and forth between graphics and main memory, but has the
disadvantage that data is recalculated every time a frame is rendered for display. A third
approach is to adopt some sort of compromise where some data is pre-calculated and the
results cached in memory but other is calculated on the fly in the graphics pipeline.
The students have developed an original algorithm which allows three dimensional cave
systems to be generated procedurally and rendered in real time. The algorithm offers several
key advantages over existing approaches and is reasonably efficient. However the nature of
the calculations would seem to lend themselves to the GPGPU approach. In this assignment
the student will investigate the use of GPGPU processing to reduce the time required to create
the meshes used in the cave system.
The investigation will be carried out in three phases:
Phase 1 - Background:
  • Literature survey of the use of GPGPU for procedural generation of 3D environments.
Phase 2 – Implementation:
  • Replacement of the existing CPU geometry processing stage with GPGPU kernel(s) and
    processing operation. The fundamental nature of the algorithm will not change but the
    computationally intensive stages will be carried out on the GPU instead of the CPU data and
    read back to main memory once complete.
  • The existing processing will be changed so that some of the processing of geometry can be
    performed as part of the graphics pipeline. Research will need to be carried out into how best
    to accomplish this but it will probably involve the tessellation of patch data on the GPU
    instead of pre-calculating all the primitives for the geometry using the CPU.
  • Combination of the above two processes so that the GPGPU is used to pre-calculate patch
    data and then the tessellation shader is used to compute the final geometry during the render
    stage.
Phase 3 – Analysis and evaluation:
  • Timing comparisons will be made between CPU and GPGPU based approaches. This
    performance would be discussed in comparison to current procedural generation approaches
    as reviewed in the literature survey. Also the balance between pre-calculation and the
    calculation during rendering will be evaluated.

ASSESSMENT (as per course’s project rules web page, with the differences noted below):

<table>
<thead>
<tr>
<th>Assessed project components:</th>
<th>% of mark</th>
<th>Due date</th>
<th>Evaluated by:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report: name style:</td>
<td>45%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(e.g. research report, software description...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Artefact: name kind:</td>
<td>45%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(e.g. software, user interface, robot...)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Presentation:</td>
<td>10%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MEETING DATES (IF KNOWN):

Research School of Computer Science
Jun-12
STUDENT DECLARATION: I agree to fulfil the above defined contract:

Signature

Date

SECTION B (Supervisor):

I am willing to supervise and support this project. I have checked the student's academic record and believe this student can complete the project.

Signature

Date

REQUIRED DEPARTMENT RESOURCES:

none

SECTION C (Course coordinator approval)

Signature

Date

SECTION D (Projects coordinator approval)

Signature

Date