**Project Final Report**

# CUDA Ray Tracer

Submitted for the *BSc* in
*Computer Science with Games Development*

May 2014

by

***Alexander Paul Rodgers***

# Table of Contents

# CUDA Ray Tracer

# 1 Introduction

## 1.1 Initial Brief

Ray tracing is a simple 3D graphics technique for generating high quality computer graphic images. Many scenes in various animated films are created based on ray tracing techniques. However, compared with pipelined-based graphics techniques, ray tracing is very computationally expensive. The aim of this project is to explore the computing capability of modern GPUs and write a ray tracer using a GPGPU technique, such as the NVIDIA CUDA SDK.

## 1.2 Context

### 1.2.1 Ray Tracing

The technique has been in common use throughout the entertainment industry by companies such as Pixar and Dreamworks, for use in realistic offline rendering in non-interactive media such as movies (Pacheco, 2008). For real-time rendering however, pipeline-based techniques are generally favored due to significant greater performance (Etheredge & Meteer, 2010). Pipeline-based graphics are advancing quickly with the aid of powerful native hardware support, yet ray tracing's  simplicity and accurate simulation of lighting physics can offer very realistic rendering without the need of additional complicated algorithms.

Referred to as an "Image-space algorithm" (Suffern, 2007, p. ix), at its simplest, ray tracing places a 2D image-plane of pixels between a virtual camera and 3D scene. The algorithm functions by following a projected line or 'ray' from the camera through each pixel until it intersects with an object. When the ray intersects, it calculates the colour based on the objects material and the scenes lighting. See Appendix A for an extrapolated ray tracing analogy.

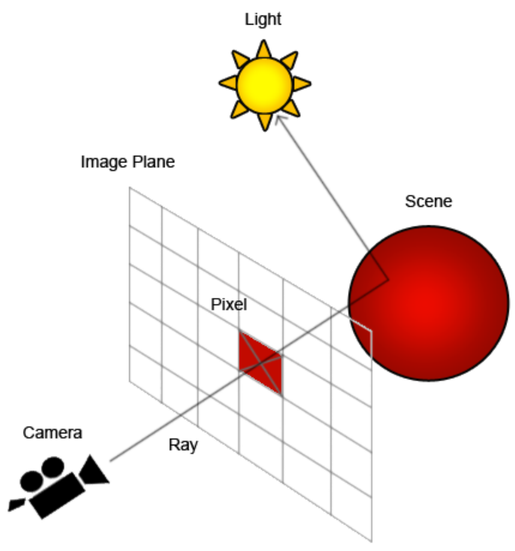


**Figure 1: Ray Tracing Scene**

The ray tracing algorithm was first presented by Appel (1968). This early algorithm allowed the shading of shapes that could be intersected with a ray, and is now specifically referred to as 'ray casting'.

Whitted (1979) later developed a recursive ray tracing algorithm that traced additional shadow, reflection and refraction rays at the point of intersection to create realistic lighting. It is this technique that forms the basis of modern ray tracing today.

### 1.2.2 General Computation on Video Cards

Modern video card parallel processing capabilities have led to an increased use for non-graphical processing over recent years, establishing the phrase 'General-Purpose computation on Graphics Processing Units' or GPGPU. The modern video cards specialised design is particularly efficient at parallel data calculations compared to a CPU. For this reason, GPGPU is now a major contributor in supercomputing and an accelerator in the innovation of many scientific fields (Trader, 2013).

Ray tracing is a natural candidate for GPGPU because of its high data throughput and parallelism. A large potential volume of ray calculations makes it computationally very expensive, therefore simultaneous processing using many smaller cores can provide substantial performance benefits when used in highly parallel routines (Jeong & Abram, n.d.).

## 1.3 Aim and Objectives

The benefits of this project will be firstly to analyse GPGPU ray tracing by comparing its offline rendering performance to that of CPU ray tracing, and secondly explore the benefits introduced via the implementation of ray marching techniques and implicit distance functions.

Four prototypes will be developed for performance and visual analysis. The first two being standard ray tracers, one utilising the CPU, the other using GPGPU techniques. The third and fourth prototypes will explore ray tracing utilising ray marching and implicit distance functions, as before with one utilising the CPU, the other with GPGPU computed rendering.

It will be written in a high performance language to optimise render times. It will be aimed at programmers with related graphics and programming knowledge that can operate it without a menu interface. Potential applications include the creation of artistic images and realistic 3D renderings.

# 2 Background Overview

## 2.1 Rendering Techniques Comparison

### 2.1.1 Pipelined-based Polygon Rendering

Pipelined-based rendering is the most popular technique used in 3D graphics today. The pipeline describes the stages along which a 3D scene is transformed into a 2D rasterized representation and then presented to the display. The pipeline is generally associated with 3D polygon rendering, a 'forward mapping' technique where an area in view is calculated and rays are sent from the vertices of each surface into the camera. Each pixel keeps track of the nearest primitive by storing a value in a 'z-buffer'. This entire process of converting a 3D scene into a 2D image is known as 'scan conversion' or 'rasterization'. Shading is then calculated afterwards by interpolating illumination values between vertices and using the 'z-buffer' for correct ordering. (Cutler, 2009)

The pipelined-based method is very fast and especially good at rendering dynamic scenes due to an object-based approach. With interactive productions such as video games, a single set of geometry data can be sent to the GPU and then rendered many times for each object instance unlike in ray tracing where generally, an entire scene must be stored in-memory and recalculated each frame (Stratton, 2008). Despite its performance benefits however, pipelined methods have no inherent capabilities to handle shadows, reflections and other visual effects that define very realistic renderings (Cutler, 2009). Instead, these effects are faked using a range of less-accurate techniques such as reflection and shadow mapping (Stratton, 2008).

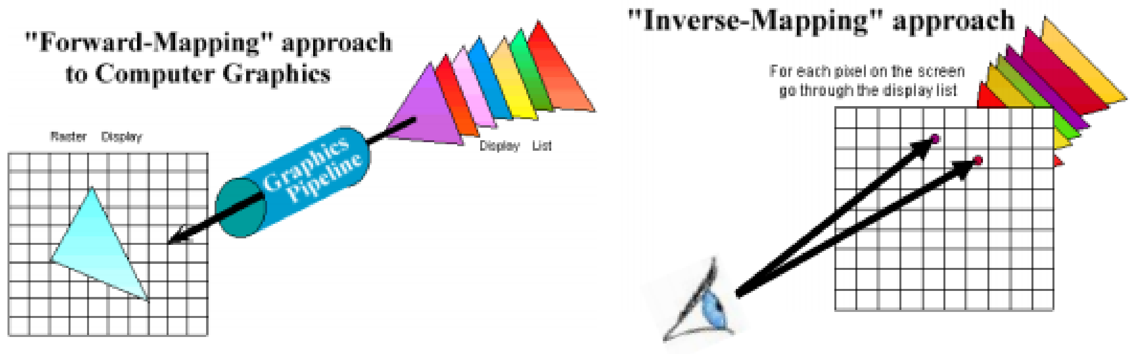

"Forward-Mapping" approach to Computer Graphics
Raster Display
Graphics Pipeline
Display List
"Inverse-Mapping" approach
For each pixel on the screen go through the display list

**Figure 2: Differing Mapping Techniques (Cutler, 2009)**

### 2.1.2 Ray Traced Rendering

In contrast ray tracing is a pixel-based 'inverse mapping' algorithm developed to accurately calculate the distribution of light throughout a scene. It allows 'physically based' rendering of environments resulting in accurate hard shadows and reflections without the need to use complex and inaccurate algorithms to fake their appearance (Howard, 2007). The simplicity of the algorithm means that additional effects such as refraction, blurring and depth-of-field can be later integrated into the base algorithm with little re-programming making it both flexible and extensible. Acceleration structures can be integrated to partition up space, greatly limiting the amount of calculations required in very complex scenes. Additionally all calculations can be performed on a CPU meaning that unless specifically utilising GPGPU capabilities as outlined in this report, a high-end GPU is not required to produce very realistic offline renderings. (Stratton, 2008)

The biggest drawback of ray tracing is performance. Although pipelined and ray tracing methods are both computationally expensive, GPU hardware has been tailor engineered towards pipelined-based rendering for years, giving it a significant advantage. Physical accuracy can also be seen as a redundant concept since with modern innovations, faking realistic effects has led often to the human eye being unable to tell the difference between an accurate ray traced scene and one mimicked using sophisticated pipelined algorithms (Stratton, 2008). Other aspects that hinder ray tracing performance include it's anti-aliasing capabilities which are detailed later in this section.

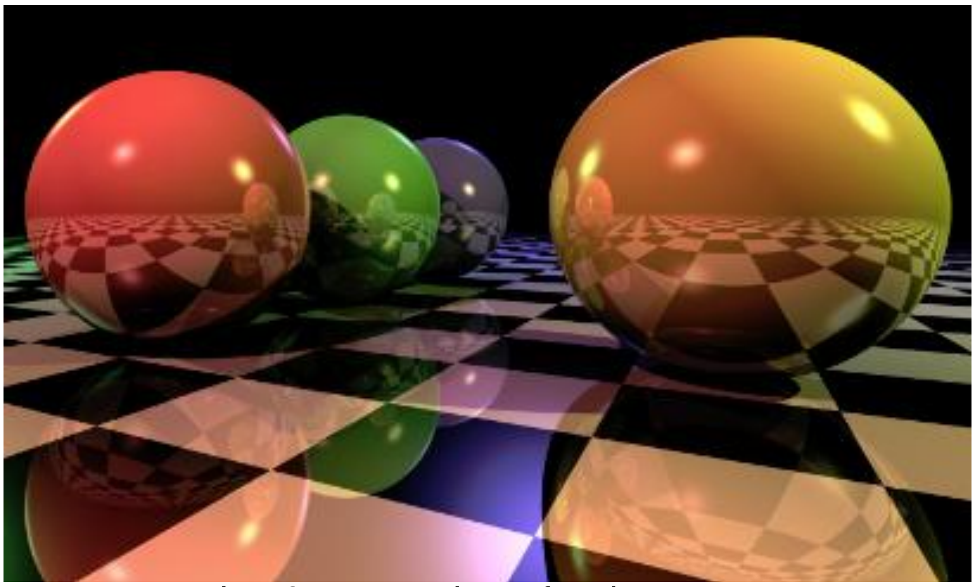


**Figure 3: A Ray traced scene from the prototype.**

## 2.2 Ray Tracing Algorithm

In Figure 4, a typical ray tracing algorithm with reflection and shadow secondary rays is presented using pseudo code:

```
01 // for each pixel
02 //       Calculate the direction between camera and pixel centre
03 //       Fire a primary ray through the pixel
04 //       Compute the nearest intersection point with an object (if any)
05 //
06 //       if the ray hit an object
07 //       {
08 //           if the object material is reflective
09 //           {
10 //               compute a secondary 'reflection' ray
11 //               if the reflection ray hit an object
12 //               {
13 //                   recurse from line 08
14 //                   add object material colour to final color
15 //               }
16 //           }
17 //
18 //           for each light source
19 //           {
20 //               compute a secondary 'shadow ray' towards light source
21 //               if 'shadow ray' doesn't intersect
22 //                   compute final colour from light and object material
23 //           }
24 //
25 //           set pixel colour to final colour
26 //       }
27 //       else set pixel colour to background colour
```

**Figure 4: Ray tracing pseudo-code**

Ray tracing is traditionally an iterative process that occurs over each pixel within an image to determine its final colour.

The first step in ray tracing requires determining the direction to fire an initial 'primary' ray from the camera (eye) position through a pixel within the 2D image plane ( See Figure 1). Once the ray is fired, linear algebra determines the closest point of intersection with an object, since objects occluded behind the closest will *generally* not be rendered, with exception to refraction discussed later in section **3.1.7**.

If the ray did not intersect any objects in the scene, the pixels colour is set to a background value. If the ray did intersect an object, the appropriate colour must then be computed using 'secondary' rays.

Lighting largely determines the resultant colour of a pixel whether directly from a light source or as reflected light from another object. Convincing reflections can be accurately created by combining different object colours via sending additional rays between objects from each intersection point. Therefore if the primary objects material is reflective, a secondary 'reflection' ray must be computed. This process can be repeated recursively until a specified depth is reached. If the reflection ray hits an object, the secondary intersected object's material colour contributes to the final pixel to simulate the reflective properties of light.
In a simple case with a single light source, a secondary 'shadow' ray is also computed and sent from the primary intersection point to the light source; if this secondary ray intersects another object then the pixel is said to be 'in shadow' and the light will not contribute to the

pixels final colour. If the 'shadow ray' *does not* intersect another object, then colour can be calculated using an illumination model such as 'Phong', by scaling the material colour and reflective properties by the diffuse, specular and ambient components of the light source. In scenes with multiple lights, 'shadow rays' must be computed for each, all contributing to the final pixel colour.

Once all pixel colours are determined and stored in a data structure, an image file can be generated or the data sent to a graphical context.

## 2.3 Aliasing Problem

Ray tracing relies on a per-pixel sampling process where each pixel is a fixed size, yet the rays that must compute the pixels each represent an infinitely thin point. This dilemma is solved by tracing a ray through the centre of each pixel and setting the colour based on that ray for the entire pixel. However, since each pixel in an image occurs at a regular interval, but their colours are determined by a single sample, this leads to a visual effect known as 'aliasing' where artifacts not representative of the scene, manifest in the final rendering. (Cooksey & Bourke, 1994)

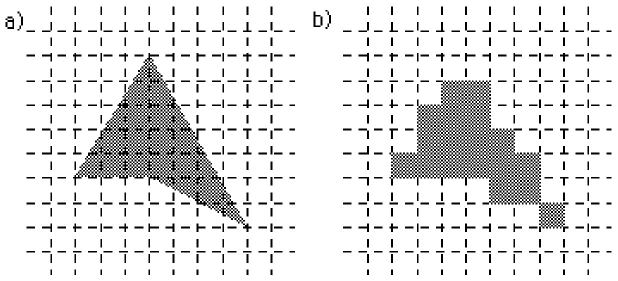


**Figure 5: The 'staircasing' aliasing effect. (Cooksey & Bourke, 1994)**

### 2.3.1 Staircasing

'Staircasing' (Figure 5) is one such manifestation that generally occurs along stark changes in intensity such as along shadows and edges. This results in a loss of image clarity and a blocky 'stepping' apparition due to each pixel being either completely shaded or not, depending on where the ray sample hits (Cooksey & Bourke, 1994)

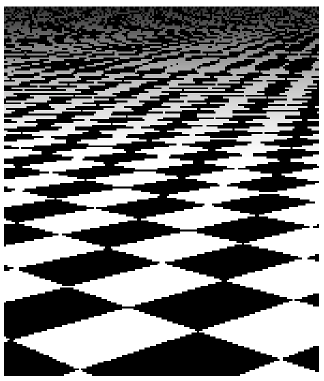


**Figure 6: Moiré interference (no AA)**

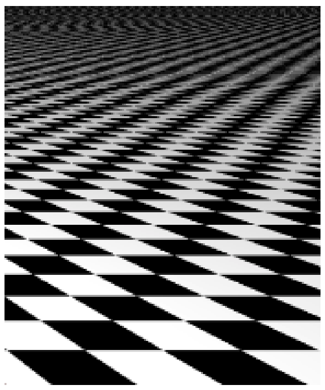


**Figure 7: Moiré interference (4x supersampling AA)**

### 2.3.2 Interference

Interference patterns are another manifestation of aliasing as demonstrated in Figure 6, a screenshot taken from the ray tracer prototype. Specifically referred to as Moiré interference (Cooksey & Bourke, 1994), the distorted effect is especially noticeable at the top of the example, caused by the undersampling of a checkered pattern at an increasingly fine frequency. This undersampling is caused by the inaccuracies of the ray tracers single point sampling method.

### 2.3.3 Supersampling Anti-aliasing

To rectify this problem there are various 'Anti-aliasing' techniques that can be applied to the ray tracing algorithm. Figure 7 demonstrates a similar scene from the prototype but with an implemented anti-aliasing technique called 'supersampling'.

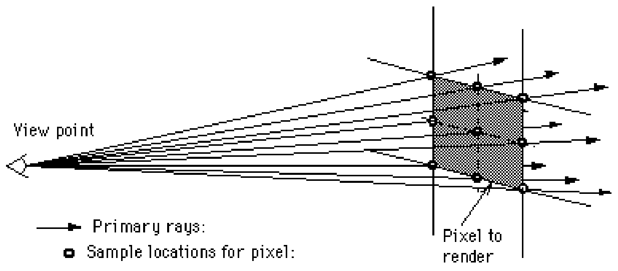


**Figure 8: Supersampling grid algorithm (Cooksey & Bourke, 1994)**

Supersampling is a simple discrete anti-aliasing method that works by taking multiple sub-samples from within a pixel and determining an averaged colour to apply to the entire pixel. Each sub-sample requires an additional ray to be cast that can be simply implemented into an existing ray tracer, yet does carry with it a large potential performance overhead by increasing the number of ray computations per pixel by $n^2$, with $n$ being the anti-aliasing strength or the √ of the total number of sub-samples per pixel e.g. '8x' anti-aliasing refers to 64 ($n^2$) sub-samples per pixel.
To find the position of each sub-sample within a pixel, a variety of supersampling configurations can be used. Figure 8 demonstrates a grid algorithm where each sub-sample is evenly distributed throughout the pixel, other potential algorithms are shown in Figure 9.

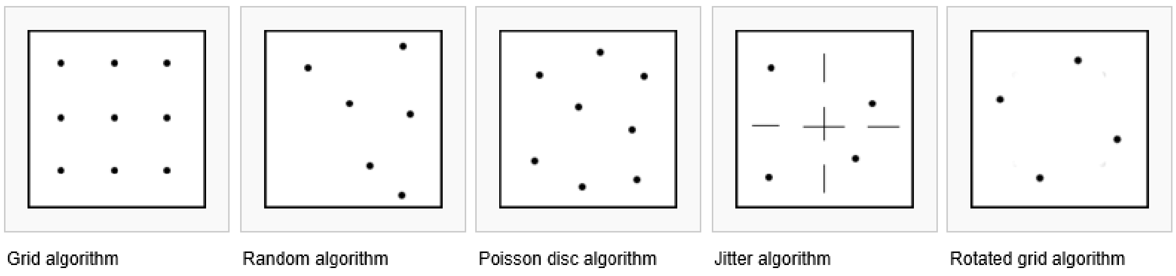


**Figure 9: Examples of Supersampling algorithms (Wikipedia, 2014)**

Once each sub-sample position has been determined, a ray is fired through each and the colour at the sample location is determined as previously described in section **2.2**. An average colour is then computed from all the sub-samples within the pixel and set as the colour for the overall pixel. The higher the number of sub-samples taken, the more accurate the sampling. It is important to note that although anti-aliasing techniques like supersampling can greatly reduces aliasing, it cannot be eliminated entirely.

## 2.4 Phong Local Illumination Model

For this project, a local surface reflection model is implemented to provide realistic illumination values for each surface point.

The Phong illumination model (also known as a reflection model) is a 'smooth shading' technique particularly appropriate for ray tracing due to its per-pixel surface normal calculations as opposed to Gouraud, an older less accurate per-vertex method. The Phong model describes how surfaces reflect light comprised of a diffuse component for rough Lambertian surfaces and a specular component for shiny surfaces. Additionally a third ambient component is factored in to represent the background level of scattered light present in a scene (Geigel, 2004).

Each surface point $(I_p)$ illumination value can be calculated using the following Phong equation (Phong, 1975):

$$I_p = k_a i_a + \sum_{m \in \, lights} \left( k_d (\hat{L}_m \cdot \hat{N}) i_{m,d} + k_s (\hat{R}_m \cdot \hat{V})^{\alpha} i_{m,s} \right)$$

Where $k_a, k_d, k_s$ are the ambient, diffuse and specular material constants respectively, $\alpha$ is the material 'shininess' constant applied to the specular component, $lights$ is the set of all light sources, $\hat{L}_m$ is a surface vector towards each light source, $\hat{N}$ is the surface normal, $\hat{R}_m$ is the perfect reflection vector and $\hat{V}$ is the surface vector towards the cameras view. Finally $i_d, i_s$ are the diffuse/specular red, green and blue (RGB) intensities for each light.

Figure 10 demonstrates the individual components that make up the Phong illumination model and the result of their combination.

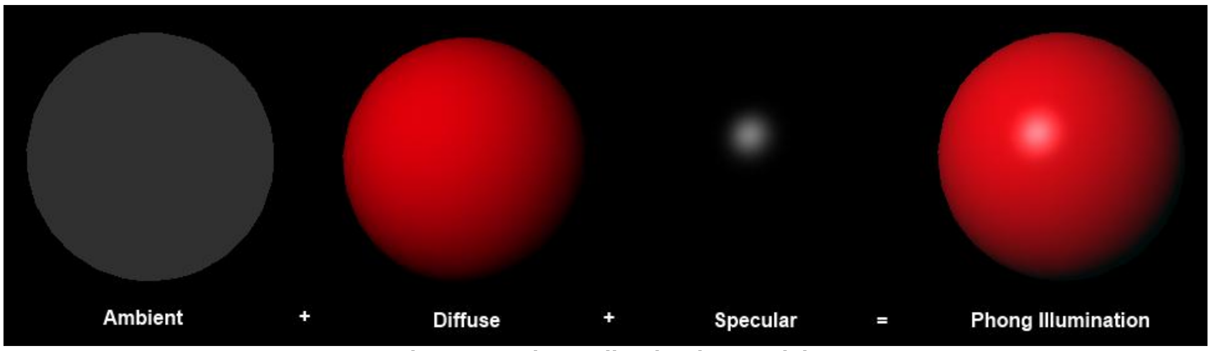


**Figure 10: Phong Illumination Model**

By applying particular constant values to material components, different textured surfaces can be simulated. Matt or rough materials can be simulated by increasing the size of specular highlights via lowering the 'shininess' exponent value, and alternatively mirror-like reflective materials can be simulated by increasing the exponent and therefore reducing the size of the specular highlights. The Specular and diffuse reflection intensity constants can also be individual adjusted to change how much light is reflected back.

## 2.5 Ray Tracing GPGPU Suitability

In a typical rendering with no spatial partitioning, a ray for each pixel in the image must be computed and tested for intersection. At a modern high-definition resolution of 1920 by 1080 pixels, this equates to 2,073,600 primary rays. Additionally, with 64 sub-sample anti-aliasing this increases yet to 132,710,400 primary rays. On top of this, there are secondary rays that must then be computed at the very least doubling the figure to well over 265,420,800 ray computations for a single frame. It is for this reason that ray tracing performance is a challenging problem to overcome even on modern hardware.

Ray tracing, by the nature of its algorithm is often referred to as 'embarrassingly parallel' (Hoberock, et al., n.d.) due to its suitability for parallel computation. As such, for this project a GPGPU variant on the initial CPU prototype has been developed (See section 4.0) to explore the performance benefits of utilising dedicated parallel processing hardware (GPU).

GPGPU ray tracing uses the same base algorithm but instead of iterating over each pixel sequentially, the image is divided up into blocks, within which a separate thread executes on each pixel concurrently. Ray tracing is a highly parallel algorithm because each of these threads and/or blocks does not need to communicate with each other, therefore each thread can be executed with independence and in any order, avoiding potential performance overheads.

By implementing a CUDA-based GPGPU ray tracer, the vast quantity of the above ray computations are broken into smaller segments and processed in parallel by more numerous, less powerful streaming multi-processors (SM). Refer to section **7.1** for a performance comparison between the CPU and GPGPU prototypes developed for this project.

## 2.6 CUDA GPGPU

CUDA (Compute Unified Device Architecture) is a parallel computing and development platform created by NVIDIA to increase parallel computing performance via the utilisation of GPU hardware (Nvidia Corporation, 2014).

As described in section **2.5**, GPUs are very efficient at processing large amounts of parallel data. This is largely due to GPU hardware design, featuring a greater transistor count dedicated to data processing rather than caching and flow control compared to that of a CPU, as demonstrated in Figure 11.

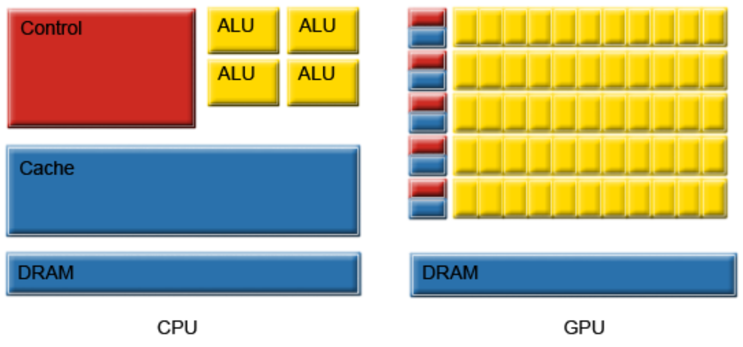


**Figure 11 : Transistors - GPU vs. CPU (Nvidia Corporation, 2013)**

### 2.6.1 Compute Capability

The CUDA platform is compatible only with NVIDIA devices with each GPU generation having a particular 'compute capability' offering various enhancements. Programs can be compiled for compatibility with a particular compute version using the NVCC compiler that accompanies the CUDA development toolkit. To date there have been three generations of CUDA devices, versions 1.x(Tesla-class), 2.x(Fermi-class) and 3.x(Kepler-class) (Wilt, 2012). Currently, the latest hardware compute version is Kepler-class 3.5. See section 6 for details on the CUDA capabilities of the hardware used in this project.

### 2.6.2 CUDA Overview

The CUDA API has three core abstractions: thread groups, shared memories and barrier synchronization, accessible as a set of language instructions. Programs are executed for many data elements in parallel. In 3D rendering, large sets of vertices, pixels and image blocks can be mapped to parallel threads. 'Kernels' are computational routines, created by the programmer to be executed in parallel by numerous different threads. Each thread executing a kernel acquires a 'thread ID'. These ID's are in a 3-component vector so that they can be assigned to 1D, 2D or 3D thread groups called 'blocks' (Nvidia Corporation, 2013), see Figure 12.

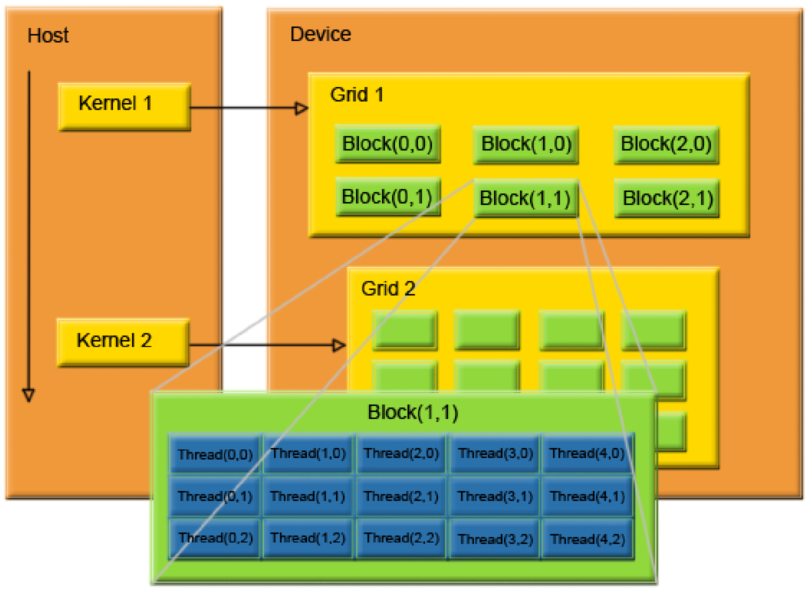


**Figure 12: Thread Blocks (Nvidia Corporation, 2013)**

Blocks are required to execute independently and rely on being able to run in any order, sequentially or parallel. Threads within blocks can share data using shared memory and cooperate via thread synchronisation, acting as a barrier where a block must wait until it can proceed. (Nvidia Corporation, 2013)

Streaming Multiprocessors (SM's) are the units responsible for executing kernels. Each SM features thousands of registers and a variety of caches (see section **2.6.3**). SM's process in units of 32 threads known as 'Warps'. Warp schedulers can change contexts quickly between threads, issuing new instructions to warps waiting for execution. In addition, each SM contains many execution cores responsible for performing integer and floating-point operations (Wilt, 2012).

Today, high-end hardware can utilise 8 SM's, each with 192 cores (Hagedorn, 2013), granting a total of 8x192 cores, dramatically higher than that of a typical CPU. Each of these cores can execute via 'Single Instruction, Multiple Thread' or SIMT developed by NVIDIA, a vector processing-based execution model similar to SIMD (Single Instruction Multiple Data) but with added functionality for conditional branching, where cores belonging to the same group perform the same instruction using multiple threads (Wolfe, 2010).

### 2.6.3 GPU Memory Hierarchy

Memory management is an important aspect of CUDA programming due to performance between the various memory caches. Therefore, managing where data is being accessed from has a significant effect on performance.

It is optimal to reduce the amount of data transfer occurring between the CPU(host) and GPU(device), and further to use cached memory where possible over device memory (Barros, 2009).

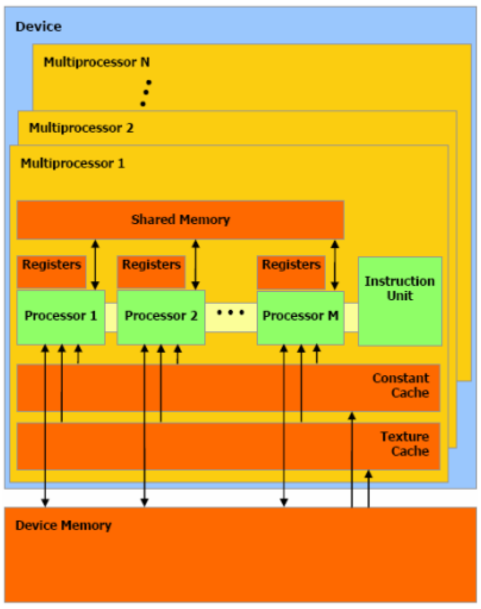


**Figure 13: Multiprocessor Architecture (Nvidia Corporation, 2013)**

The following is a summary of the main memory types available on the GPU (Zahran, 2012):

- 'Register' memory is the fastest available but is limited in size and availability. It is bound to a single threads lifetime.
- 'Shared' memory is extremely fast and highly parallel due to being accessible by every thread in a block, for the duration of that blocks lifetime.
- 'Global' device memory is usually implemented as DRAM and has high latency and limited bandwidth making it far slower than other memory types. It is however the largest source of memory on the GPU and is accessible from any thread or block

with access to a pointer to associated data. Data from the CPU host is generally first allocated and copied to global device memory prior to computation by a kernel.

- 'Constant' memory is a read-only cache of global memory and as a result has shorter latency and higher bandwidth.

# 3 Ray Tracing Principles

To implement a ray tracer there are several key components required, consisting of rays, a camera , an image-plane and shapes to represent a scene such as spheres and a plane.

### 3.1.1 Defining a Ray

Within the context of ray tracing, a ray is an infinite straight line defined by the following parametric equation:

$$\boldsymbol{p} = \boldsymbol{o} + t\boldsymbol{D}$$

where $\boldsymbol{p}$ is an arbitrary point along the line, $\boldsymbol{o}$ is the lines origin, $\boldsymbol{D}$ is a direction vector and $t$ is a coefficient of $\boldsymbol{D}$ determining distance. Because the line is infinite, $t$ describes where relative to the origin a position on the line is. Therefore $t = \boldsymbol{0}$ at the ray origin and
$t > 0,\ t < 0,$ for an arbitrary point along the line.

### 3.1.2 The Camera

The camera's local coordinate system is established as an orthonormal basis by calculating three perpendicular unit vectors $(\hat{\boldsymbol{u}}, \hat{\boldsymbol{v}}, \hat{\boldsymbol{w}})$ in world space to represent its axis.

Assuming the camera looks down the negative z-axis, $\hat{\boldsymbol{w}}$ represents its facing direction (See Appendix B for details). $\boldsymbol{u}$ can be calculated as the camera 'right' direction via $\hat{\boldsymbol{w}} \times \hat{\boldsymbol{y}}$ with $\hat{\boldsymbol{y}}$ defined as an 'up' direction representing the worlds y-axis. Finally $\hat{\boldsymbol{v}}$ is calculated as the camera 'up' direction via $\hat{\boldsymbol{w}} \times \hat{\boldsymbol{u}}$**.**

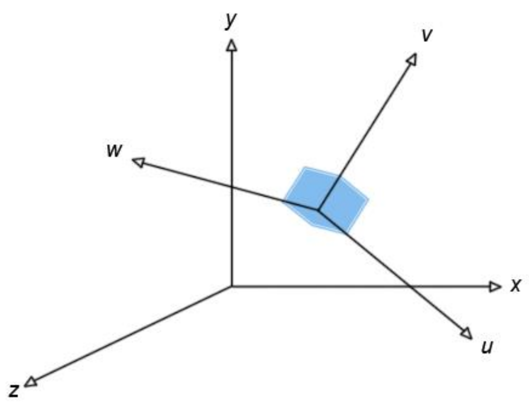


**Figure 14: Orthonormal basis (Suffern, 2007, p. 41)**

### 3.1.3 The Image Plane

Using the following technique (Scratchapixel, 2012), Image plane pixels coordinates are mapped to the camera's field of view(FOV) and finally into world space to calculate each ray's direction (See Appendix C for details).

An initial pixels raster coordinates are first converted into 2D normalized device coordinates(NDC) where pixel point $a(x,y)$ and $\{0 \le x \le 1\}$, $\{0 \le y \le 1\}$:

$$a = \left(\frac{x + 0.5}{ImageWidth}, \frac{y + 0.5}{ImageHeight}\right)$$

Since the image plane will be centred to the camera, it reasons that the cameras axis be aligned along the image plane centre where pixel $b(x,y)$ and $\{-1 \le x \le 1\}$, $\{-1 \le y \le 1\}$ :

$$b = \left(2a_x - 1, 1 - 2a_y\right)$$

$b$ can then be converted to camera coordinate space:

$$c_x = b_x tan\left(\frac{fov}{2}\right)A$$
$$c_y = b_y tan\left(\frac{fov}{2}\right)$$

where $fov$ is a FOV angle and $'A'$ is the image aspect ratio. This scales the pixels normalised coordinates based on distance from the FOV's centre. $'A'$ adjusts $c_x$ for non-square images.

Finally it is converted to world space by scaling $c$ along the camera's orthonormal basis where the pixel's ray direction $D$ can be calculated**:**

$$\boldsymbol{D} = \boldsymbol{w} + (c_x\boldsymbol{u}) + (c_y\boldsymbol{v})$$

### 3.1.4 Ray - Plane Intersections:

A plane can be defined using the implicit equation:

$$\boldsymbol{p} \cdot \boldsymbol{n} = -\boldsymbol{d}$$

where $\boldsymbol{p}$ is an arbitrary position on the plane, $\boldsymbol{n}$ is the normal determining the planes orientation and $\boldsymbol{d}$ is the distance from origin.

If the ray is not parallel (see Appendix D for details), an intersection must occur and so must determine its distance along the ray. The following technique (Suffern, 2007) can be used to determine this distance:

Taking the equations for both a ray and plane, the ray equation can be substituted into the plane equation for $\boldsymbol{p}$:

$$(\boldsymbol{o} + t\boldsymbol{D}) \cdot \boldsymbol{n} = -\boldsymbol{d}$$

Solving for $t$ finds the distance along a ray from its origin that an intersection has occurred with a plane:

$$t = \frac{-\boldsymbol{d} - \boldsymbol{o} \cdot \boldsymbol{n}}{\boldsymbol{D} \cdot \boldsymbol{n}}$$

It is then known that if $t > 0$ the intersection has occurred within the scene and also its distance from the rays origin.

### 3.1.5 Ray - Sphere Intersections:

A sphere can be defined using the implicit equation:

$$(\boldsymbol{p} - \boldsymbol{c}) \cdot (\boldsymbol{p} - \boldsymbol{c}) - r^2 = 0$$

where $\boldsymbol{p}$ is a point on the surface of the sphere, $\boldsymbol{c}$ is the centre of the sphere and $r$ is the radius.

The following technique (Olano, 2002) describes calculating the value of $t$ upon intersection with a sphere.

Substituting the ray equation into the sphere equation for $\boldsymbol{p}$ looks as follows:

$$(\boldsymbol{o} + t\boldsymbol{d} - \boldsymbol{c}) \cdot (\boldsymbol{o} + t\boldsymbol{d} - \boldsymbol{c}) - r^2 = 0$$

This equation can then be expanded into the quadratic form $at^2 + bt + c = 0$ for $t$:

$$\boldsymbol{d} \cdot \boldsymbol{d}t^2 + \boldsymbol{2d} \cdot (\boldsymbol{o} - \boldsymbol{c})t + (\boldsymbol{o} - \boldsymbol{c}) \cdot (\boldsymbol{o} - \boldsymbol{c}) - r^2 = 0$$

The quadratic formula can be used to calculate $\boldsymbol{t}$ and thus the distance the ray has travelled to intersect with the sphere (See Appendix E for details):

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

### 3.1.6 Reflection Rays:

A secondary reflection ray described in section **2.2** can be calculated using the following vector reflection equation:

$$\boldsymbol{R} = 2(\boldsymbol{N} \cdot \boldsymbol{L})\boldsymbol{N} - \boldsymbol{L}$$

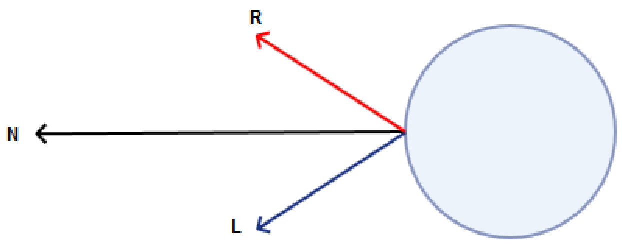


**Figure 15: Reflection Vector**

Where $\boldsymbol{R}$ is the reflected ray direction, $\boldsymbol{L}$ is the inverse incoming light direction and $\boldsymbol{N}$ is the surface normal at the point of intersection.

Firstly the projection of $\boldsymbol{L}$ in the direction of $\boldsymbol{N}$ **is** performed via $2(\boldsymbol{N} \cdot \boldsymbol{L})\boldsymbol{N}$**,** the required reflected vector is then determined by subtracting $\boldsymbol{L}$.

### 3.1.7 Refraction Rays

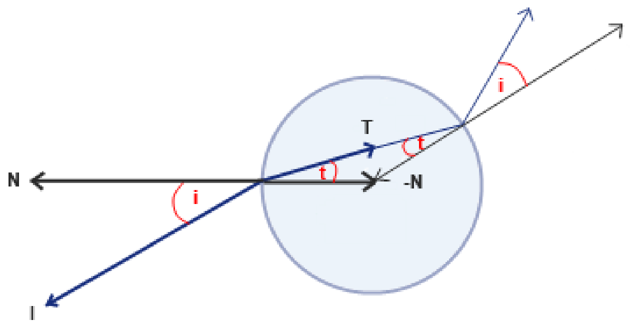


**Figure 16: Refraction Transmission**

Secondary refraction rays can be cast upon intersection with transparent objects to simulate materials such as glass and water. The following equation is used to calculate the direction vector of the transmission ray based on the direction of the incident ray, a refractive index and the surface normal:

$$\hat{\boldsymbol{T}} = \left(\eta_r(\hat{\boldsymbol{N}} \cdot \hat{\boldsymbol{I}}) - \sqrt{1 - \eta_r^2\left(1 - (\hat{\boldsymbol{N}} \cdot \hat{\boldsymbol{I}})^2\right)}\right)\hat{\boldsymbol{N}} - \eta_r\hat{\boldsymbol{I}}$$

where $\hat{\boldsymbol{T}}$ is the transmission ray direction, $\hat{\boldsymbol{N}}$ the normal, $\hat{\boldsymbol{I}}$ the ray of incidence's negative direction and $\eta_r$ the refractive index of the material.

The refractive index with accordance to Snell's Law, determines the amount of light that is refracted when entering a material and is defined as a ratio of the speed of light in a vacuum to the speed inside the material.

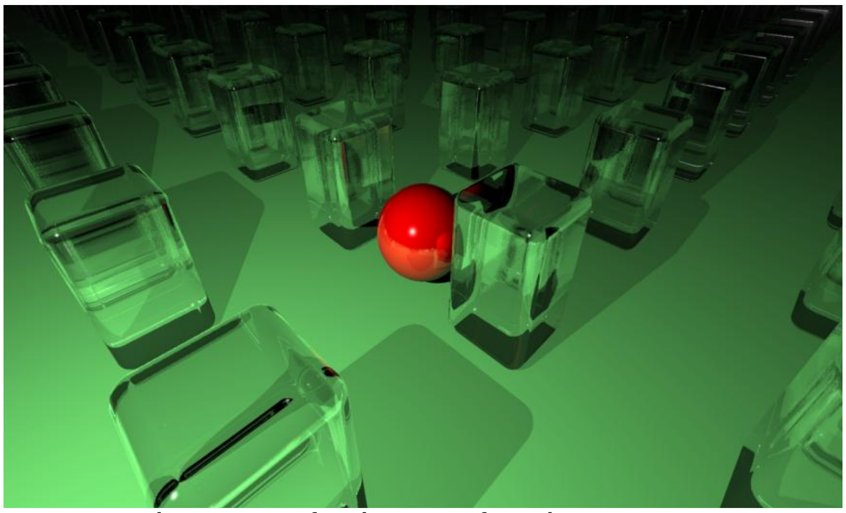


**Figure 17: A refraction scene from the prototype.**

# 4 Geometric Modeling Techniques:

## 4.1 Polygonal Mesh

Ray tracing relies on calculating the intersection of rays with geometry to successfully render a scene. Most commonly, a polygonal mesh can be used comprised of many vertices, usually forming tessellated triangle faces. This technique is particular dominant throughout pipeline-based rendering and the creative industries for a number of reasons:

Meshes allow flexible construction of specific shapes and models by artists, allowing direct per-vertex manipulation during creation through the use of specially designed software packages such as 'Maya' and 'ZBrush'. These software packages are particularly highly used in the games and special effects industries (Ingram, 2010). Once constructed, the resultant models vertices positions can be stored in a data structure and later loaded into a program or game for rendering. Once loaded, shader techniques used in pipeline rendering API's such as OpenGL and DirectX can then operate on each vertices and alter the model to enhance the visual effect of the model further or create special effects such as distortion and displacement. Additionally, meshes can be created from a variety of alternate means such as laser and image scanning devices when absolute precision is needed. In general, the larger the number of triangles comprising the mesh, the greater the realism.

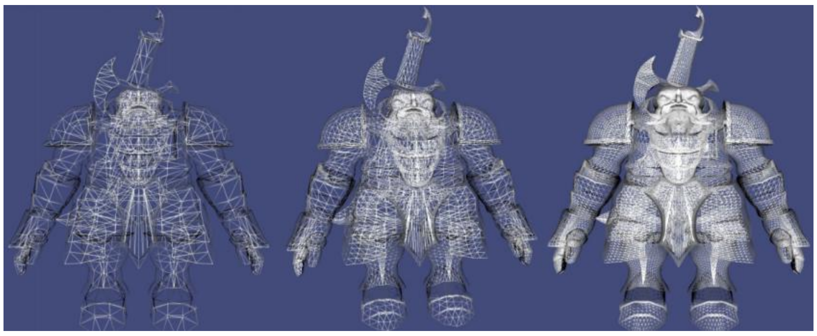
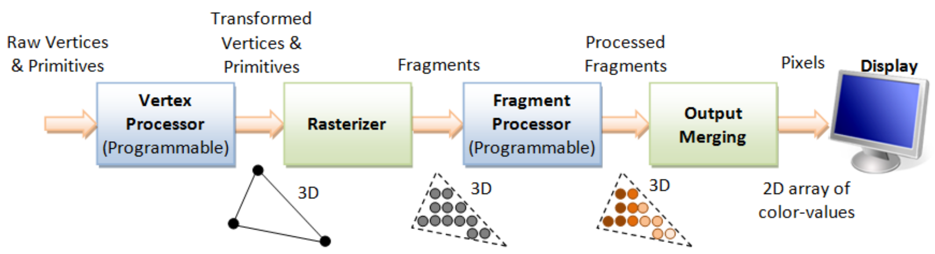


**Figure 18: Example mesh of a model (Li, 2013)**



Raw Vertices & Primitives
Transformed Vertices & Primitives
Fragments
Processed Fragments
Pixels
Display
Vertex Processor (Programmable)
Rasterizer
Fragment Processor (Programmable)
Output Merging
3D
3D
3D
2D array of color-values

**Figure 19: (Hock-Chua, 2012)**

Another advantage is owed to highly developed hardware support offered by modern graphics cards and pipeline API's, which are designed specifically to handle mesh rendering (OpenGL.org, 2013) allowing efficient computation of models by breaking down complex calculations into smaller simpler tasks based on the properties of polygonal faces (Li, 2013). The importance of meshes in modern API is highlighted by the prominence of the vertex 'shader' in pipeline architecture, such as can be observed in Figure 19, featuring a basic diagram of the OpenGL pipeline.

Although meshes are common to pipeline rendering, the same ray intersection principles highlighted previously for simple algebraic geometry can also be applied to allow ray tracing of mesh models. The method presented below looks at calculating the intersection with a triangle, by treating it as a subset of a plane (Marschner, 2003).

There are two major steps to the process, firstly a ray intersection is calculated on the triangle's plane. It must then be determined if this lies inside the triangle bounds and therefore intersects.

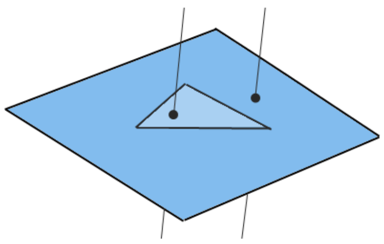


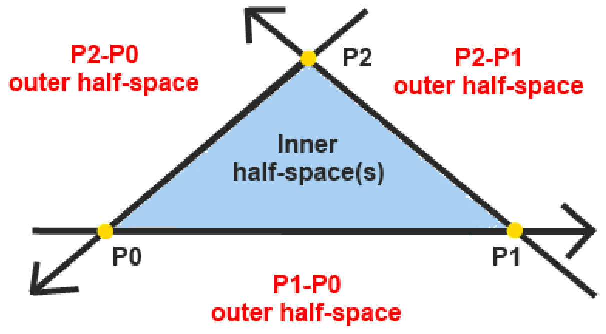
**Figure 20: Triangle subset of a plane and intersecting rays**

The initial intersection is calculated using the implicit equation of a plane as described in section **3.1.4,** but with the triangle vertices as the position on the plane and the normal derived from the cross product of two triangle edges:

$$\boldsymbol{n} = (\boldsymbol{p}_1 - \boldsymbol{p}_0) \times (\boldsymbol{p}_2 - \boldsymbol{p}_0)$$

Where $(\boldsymbol{p}_1 - \boldsymbol{p}_0)$ and $(\boldsymbol{p}_2 - \boldsymbol{p}_0)$ represent the edges of the triangle, and the order is such that $\boldsymbol{n}$ points towards the front side of the triangle, determined as the counter-clockwise ordered vertices.

It must then be determined if the intersection point lies within the triangle. This can be achieved by describing the triangle as the intersection of three lines along the edges, where either side of each line is a half-space (inner and outer). If the intersection lies in the correct half-space of all three lines then it must be inside the triangle.



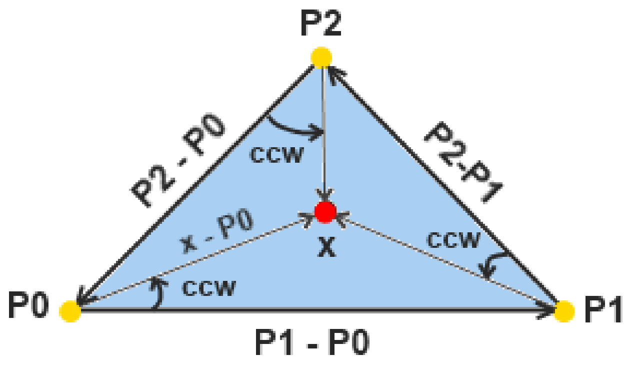


**Figure 21: Top - Triangle described as three half-spaces. Bottom - Calculating a point inside.**

If $\boldsymbol{p}$ is a vertex point of the triangle, vector $\boldsymbol{v}$ is a line along the edge such as $\boldsymbol{p_1} - \boldsymbol{p_0}$, and $\boldsymbol{x}$ is the intersection point, it can be determined if $\boldsymbol{x}$ lies inside by defining a vector $\boldsymbol{x} - \boldsymbol{p}$ and calculating if this is counter-clockwise to $\boldsymbol{v}$**,** i.e. it is to the left of the edge.

This can be determined by taking the cross-product of the two vectors $\boldsymbol{v}$ and $\boldsymbol{x} - \boldsymbol{p}$ , and then calculating its dot product with the normal as defined earlier. This is because we know that if one vector is counter-clockwise to another as per the arrangement of the triangle vertices, the cross-product will point to the front as proved by the normal calculation earlier. Therefore if the cross-product of $\boldsymbol{v}$ and $\boldsymbol{x} - \boldsymbol{p}$ is facing the same way as the normal, we know that $\boldsymbol{x} - \boldsymbol{p}$ must be counter-clockwise to $\boldsymbol{v}$. If the resultant dot products for each edge are positive, then we know that $\boldsymbol{x}$ lies inside the triangle. The above can be reduced into the following formula:

$$(\boldsymbol{p_1} - \boldsymbol{p_0}) \times (\boldsymbol{x} - \boldsymbol{p_0}) \cdot \boldsymbol{n} \geq 0$$
$$(\boldsymbol{p_2} - \boldsymbol{p_1}) \times (\boldsymbol{x} - \boldsymbol{p_1}) \cdot \boldsymbol{n} \geq 0$$
$$(\boldsymbol{p_0} - \boldsymbol{p_2}) \times (\boldsymbol{x} - \boldsymbol{p_2}) \cdot \boldsymbol{n} \geq 0$$

## 4.2 Meshless Implicit Modeling

An alternate to polygonal mesh modeling particularly suited for ray tracing is offered via the use of mathematical implicit functions to represent geometric objects. Implicit geometry contrasts greatly with explicit methods such as meshes by not defining models through precise vertex positioning, but instead by computing a function for each ray to determine if it hits a surface based on its geometric properties.

### 4.2.1 Implicit Functions

A geometric object can be represented by the following implicit function:

$$H = F(x, y, z)$$

Where the function can represent a solid shape if $F(x, y, z) > 0$ or if $F(x, y, z) < 0$ and alternatively a 'surface' if $F(x, y, z) = 0$ (See Figure 22).

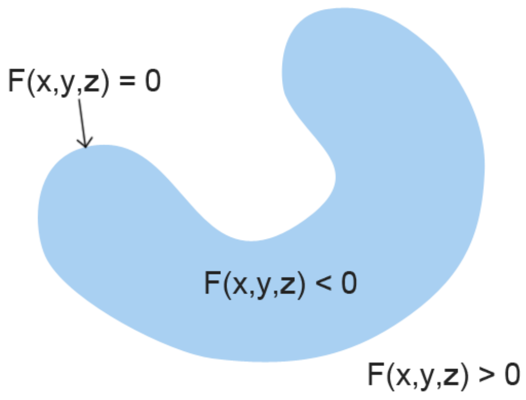


**Figure 22**

An example is outlined in section **3.1.4** where such an implicit equation for a plane is described: $(\boldsymbol{p} - \boldsymbol{a}) \cdot \boldsymbol{n} = \boldsymbol{0}$**,** which in its component form can be expressed more generally as the implicit function:

$$\boldsymbol{F(x, y, z) = Ax + By + Cz + d}$$

where $\boldsymbol{ABC}$ are the components of the normal and $\boldsymbol{d = -a \cdot n}$**.**

### 4.2.2 Signed Distance Functions (SDF)

Implicit surfaces are particular useful because of their potential for use in procedural geometry and their blending capabilities. However, rendering implicit surfaces in real-time can be difficult due to the required incremental searching of surface positions. In answer, techniques such as 'marching cubes' and 'ray marching' have become popular to optimise the process by approximating surface points via the use of signed distance functions to speed up rendering (Reiner, et al., 2011).

Implicit geometric surfaces can be approximated simply through the use of distance functions by calculating the distance from a given point to a surface rather than attempting to determine it exactly.

The following describes the mathematical definition of a signed distance function (Reiner, et al., 2011):

If $\Omega^-$ and $\Omega^+$ are respectively the regions in and outside of an implicit surface and $\partial\Omega$ is the set of points comprising the surface, the signed distance function $\phi(\boldsymbol{x})$ can be defined as the following:

$$\phi(\boldsymbol{x}) = \begin{cases} min(|\boldsymbol{x} - \boldsymbol{x}_I|) & \text{if } \boldsymbol{x} \in \Omega^+ \\ 0 & \text{if } \boldsymbol{x} \in \partial\Omega \\ -min(|\boldsymbol{x} - \boldsymbol{x}_I|) & \text{if } \boldsymbol{x} \in \Omega^- \end{cases}$$

returning the distance from $\boldsymbol{x}$ to the closest surface point, being negative if inside, positive if outside, or zero if at the surface. It is therefore a subset of implicit functions, described in section **4.2.1**.

### 4.2.3 Ray Marching with Distance Functions

Ray tracing though suitable for simple implicit surfaces, has difficulty with complex surfaces as the computational requirements for calculating the exact surface position increases. Ray marching, a technique proposed by Hart, et al., (1989) can be implemented into ray tracing for use with implicit surface modeling by utilising SDF to speed rendering performance and therefore allowing the rendering of complex implicit geometry without the use of meshes.

Ray marching works by firing rays into the scene as normal, however instead of calculating an exact intersection point, iterative steps are taken along the ray until it enters a surface, determined when the SDF returns negative. Once inside, the march is taken back to the last positive position and the step size is decreased until a threshold value is reached. The initial size of the step along the ray can be adjusted to suit performance or accuracy, where too large a step can lead to missing geometry or inaccurate surface approximation and conversely too small a step may impact performance by increasing the number of distance function calculations.

#### *4.2.3.1 Sphere Tracing*

A solution to the step size problem is a technique known as 'sphere tracing' as demonstrated in Figure 23. Sphere tracing works by instead of taking regular interval steps along a ray, it attempts to take a shortcut by calculating the distance to the closest object using distance fields and therefore the maximum distance it can travel along the ray before encountering a surface. The 'unbounding' spheres as they are also known, are volumetric representations of the space centred on a single sampling point along the ray at each step. The region inside an unbounding sphere is guaranteed to be free of any surface allowing advancement to the boundary of the sphere. This results in a large reduction of steps along each ray, but does suffer an increased amount of bounding sphere checks as it nears a surface edge. This is particularly evident with tightly packed geometry or when nearing object boundaries as observed by Quilez (2008) in his 'Slisesix' rendering. Additionally, since sphere tracing knows the distance to the closest surface, it effectively removes the need to check for entering an object and retracing steps back unless absolute precision is required. Instead, a low threshold value can be used to determine when the surface intersection is satisfied.

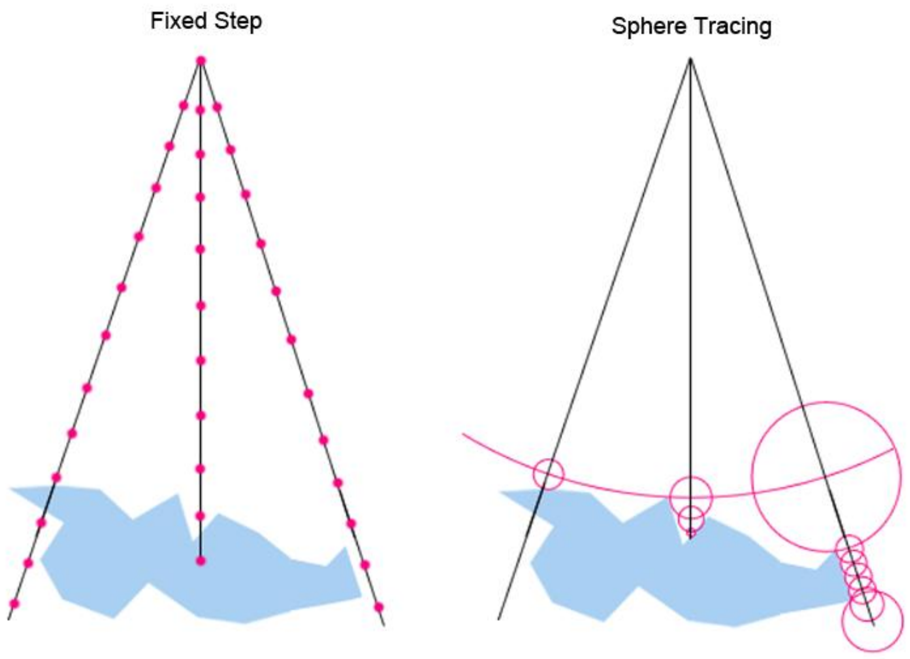


**Figure 23: Fixed step and sphere traced ray marching comparison.**

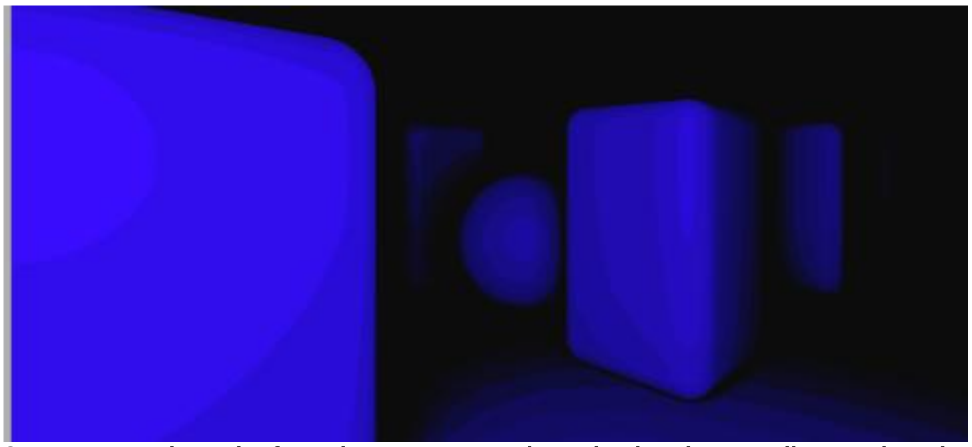


**Figure 24: A ray march render from the prototype, coloured using the step distance into the scene (lighter shades are nearer then darker shades).**

### 4.2.4 Normal Calculation

Normal's are the perpendicular vector or line for a given position on a 3D surface or a 2D curves tangent line. They are important for many aspects of 3D rendering including shading, lighting and intersection (demonstrated in section **4.1)**.
Modeling techniques like mesh rendering can explicitly define the normal of each vertex comprising the mesh. Ray marching with distance functions to form complex surfaces however, requires dynamic computation of the normal for each surface point. This is particular evident with certain implicit geometry such as 3D fractals that feature undefinable surfaces.

#### *4.2.4.1 Gradient Computation*

One effective method to calculate the normal is through approximation by comparing a point to its surroundings. This can be achieved using the gradient of a point on the surface (Hart, et al., 1989):

$$N_x = D_x + \varepsilon, y, z - D_x - \varepsilon, y, z$$
$$N_x = D_x + x, \varepsilon, z - D_x - x, \varepsilon, z$$
$$N_x = D_x + x, y, \varepsilon - D_x - x, y, \varepsilon$$

where $D_{x,y,z}$ in this case represents a distance function of the point on the surface, and $\varepsilon$ determines the direction in which to sample.

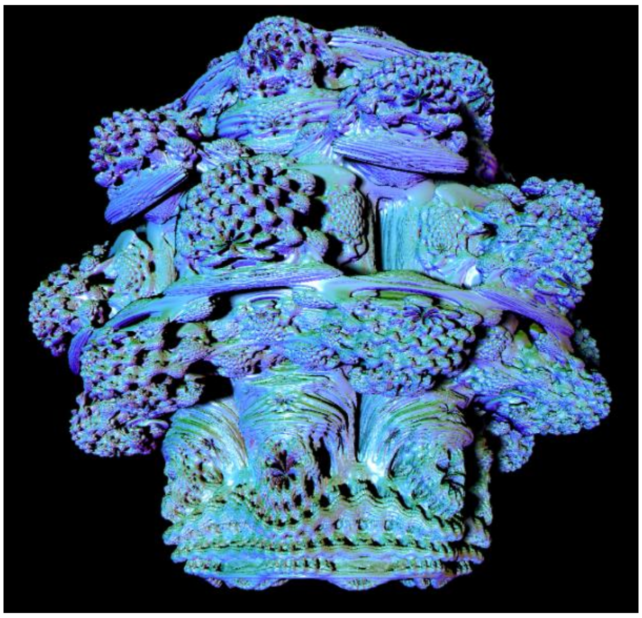


**Figure 25: A Mandelbulb Fractal ray marched using the prototype.**

In Figure 25, a Mandelbulb fractal (implicit surface) has been rendered and shaded using the gradient computation method to calculate the normals. Colour has been applied by scaling RGB values with the components of the normal for each surface point.

### 4.2.5 Soft Shadows

As discussed in section **2.2,** shadows are calculated in a simulated fashion to the real world, where a line of sight is traced from the point of intersection to see if there is unobstructed view to a light source. If obstructed, it is shadowed, else it is not.

The problem with this traditional ray tracing approach is that it creates 'hard' shadows due to the binary decision of whether there is shadow or not. In the real world, light sources are not a single point in size, so some positions on a surface will see more of the light source then others, resulting in a 'penumbra' as demonstrated in Figure 26.

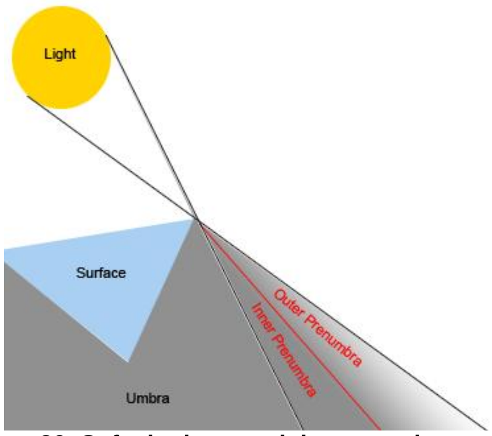


**Figure 26: Soft shadows and the penumbra region.**

The size of the penumbra region is associated with its distance to the obstructing surface resulting in shadows appearing softer when further from the object and much sharper when nearer. One point of note however is that due to the fact that only the outer half of the penumbra can be calculated, the inner half cannot make it to the light source due to the occluding surface, as observed in Figure 26. Therefore the inner penumbra region will always be in shadow. Despite this shortcoming, the resultant penumbra shadows are far more realistic than traditional hard shadows (See Figure 27).

As pointed out by Quilez (2010), soft shadows are computationally cheap to do in ray marching because multiple points are already sampled by the stepping algorithm towards the shade point along the ray. We can use this opportunity to test how close a ray comes to hitting an object on its way to the shade point, applying a penumbra value to it based on its distance, and then using the darkest value of all the penumbras (the minimum).

The penumbra value can represent the surface size of the light source, and by increasing its size the shadows will appear softer and conversely decreasing its size will harden the shadows.

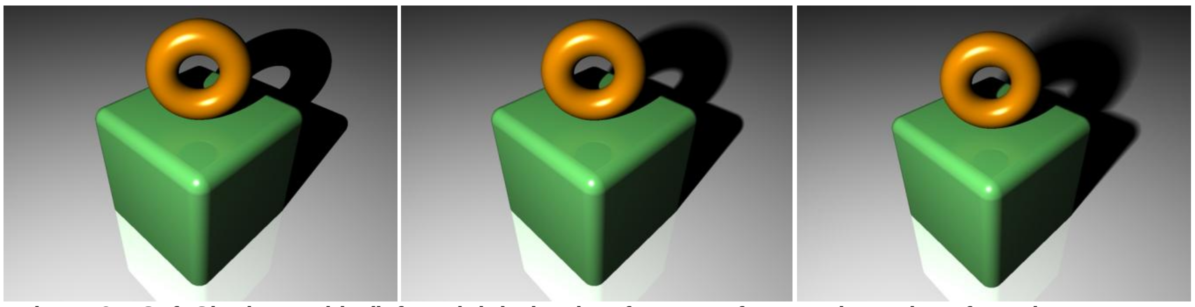


**Figure 27: Soft Shadows with, (left to right) - hard, soft, very soft penumbra values from the prototype.**

A noticeable observation is that the hard shadows feature significant aliasing due to the hard edged contrast as discussed in section **2.3**. The penumbra effect on the soft shadows removes the aliasing and results in anti-aliased edges at almost no extra computational cost.

### 4.2.6 Transformations

Hart (1994) describes that implicit surfaces can be transformed by first using an inverse transformation of the surface positions domain and then computing the distance function with the transformed point. The transformation can be represented as the following:

$$f\left(T^{-1}(\boldsymbol{x})\right) = 0$$

where $T(\boldsymbol{x})$ is a transformation and $f(\boldsymbol{x})$ defines the implicit surface.

It is pointed out that certain transformation can lead to the loss of distance information because the transformation may result in a distance function no longer providing an accurate value representing any object. In these cases, an extra adjustment of the domain must be performed to compensate.

There are a number of transformation that can safely be performed without such a loss called 'isometries', examples of which include translations, reflections and rotations. An isometry is a transformation that preserves the distance value which Hart represents as:

$$d\left(\boldsymbol{x}, \boldsymbol{I} \circ f^{-1}(0)\right) = d(\boldsymbol{I}^{-1}(\boldsymbol{x}), f^{-1}(0))$$

where $\boldsymbol{I}$ is an isometry transformation and $f^{-1}(0)$ an implicit surface.

Scaling represented as $\boldsymbol{S}(\boldsymbol{x}) = s\boldsymbol{x}$ on the other hand does not preserve the distance. Hart shows that its inverse $\boldsymbol{S}^{-1}$ is $1/s$ and therefore the accurate distance to a scaled surface can be shows as:

$$d(\boldsymbol{x}, \boldsymbol{S}(f^{-1}(0))) = sd(\boldsymbol{S}^{-1}(\boldsymbol{x}), f^{-1}(0))$$

Thus to scale an implicit surface, $\boldsymbol{S}^{-1}$ should first be applied to the domain, and then the resultant distance be multiplied by the scale $s$.

### 4.2.7 Combining Distance Fields

Distance fields can consist of one or more distance functions to construct complex scenes of implicit geometry via ray marching. A powerful trait of distance functions is evident when rather than defining implicit surfaces individually and computing the distance functions separately; distance fields can instead be constructed and combined procedurally to allow implicit surfaces to emerge (Quilez, 2008). This approach allows the deforming, twisting and repetition of space by operating on the 3D surface position along the ray, also referred to as the 'domain'.

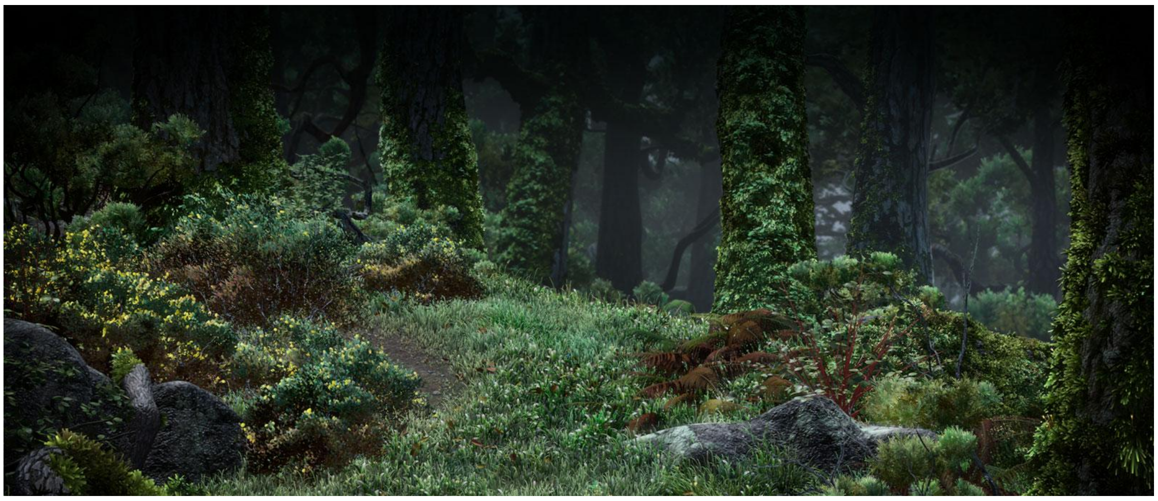


**Figure 28: Procedurally generated scene from Disney movie 'Brave' using procedural implicit rendering techniques (Disney, 2012)**

The 2012 Disney movie 'Brave' is a recent prominent example of the potential of procedural geometry, described in an interview (Robertson, 2012) as "painting with code" by its developer Quilez. He describes the need to use meshless procedural geometry in large outdoor scenes in order for it to look more natural. An example of this is how the amount of clover in the moss foliage was calculated by determining the normal of the surface, where those facing sunlight would generate an increased amount.

#### *4.2.7.1 Distance Operations*

Distance fields as discussed can be used to construct detailed scenes consisting of complex shapes from simple 'building block' primitives. Rather than operating on an entire model or mathematically defined shape, manipulation of the domain space inhabited by an implicit surface can produce many different effects.

Three of the most common 'domain operations' are now described and expressed mathematically according to Hart (1994).

#### 4.2.7.1.1 Union

Union is the staple domain operation required for ray marching. By comparing the relative distance from the ray sample position with nearby implicit surfaces, it returns the closest or 'minimum' term. This operation has two useful applications. Firstly and most importantly it is required to ensure the closest object is rendered to the image plane, effectively occluding out of sight surfaces. Secondly, it can be used as shown in Figure 29 to effectively 'hard' blend two objects together.

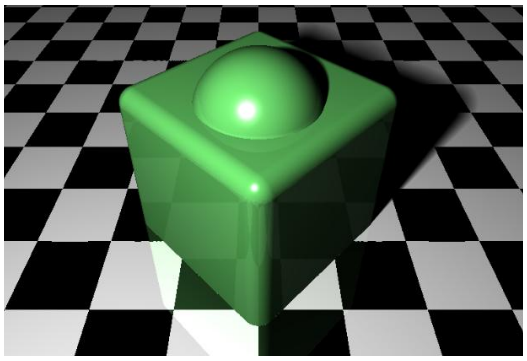


**Figure 29: A union between a cube and a sphere in the prototype.**

According to Hart, the union operation can be represented as the following:

$$d(\boldsymbol{x}, A \cup B) = min f_A(\boldsymbol{x}), f_B(\boldsymbol{x})$$

where $f_A$, $f_B$ are the signed distance functions of sets A and B, representing two implicit surfaces.

#### 4.2.7.1.2 Intersection

The intersection of two implicit surfaces can be achieved simply by returning the maximum or greater distance from two distance fields, resulting in a new surface where both meet.

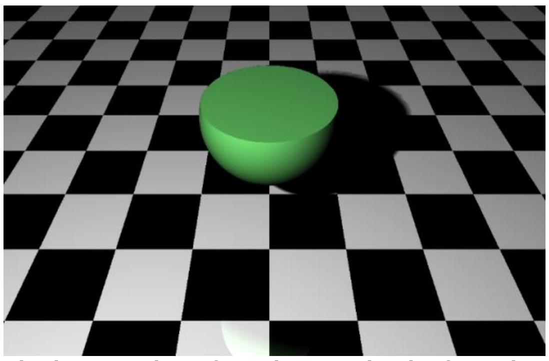


**Figure 30: The intersection of a sphere and cube from the prototype.**

Hart expresses intersection as follows:

$$d(\boldsymbol{x}, A \cap B) \geq max f_A(\boldsymbol{x}), f_B(\boldsymbol{x})$$

#### 4.2.7.1.3 Subtraction

The subtraction operation can be used in the opposite fashion to the union operation. By inverting one of the distance fields and then calculating an intersection between the two.

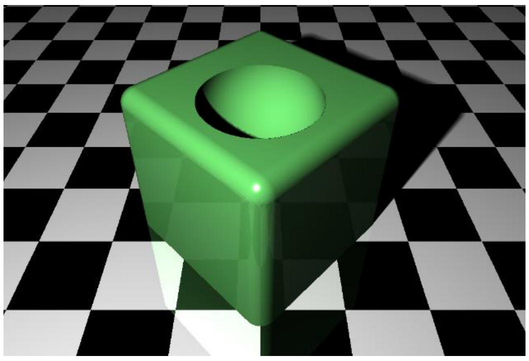


**Figure 31: A sphere is subtracted from a cube in the prototype.**

Based on Harts intersection expression, the subtraction operation differs only in the negated distance of one of the surfaces as shown in the following:

$$d(\boldsymbol{x}, A \cap B) \geq max f_A(\boldsymbol{x}), -f_B(\boldsymbol{x})$$

#### 4.2.7.1.4 Blending

Quilez (2013) describes in code, the implementation of a smooth minimum function to replace the basic union operation that results in discontinuities in the joined shape. The smooth minimum function requires the same objective as union, to return the closest surface, yet to do so without the discontinuities caused by a standard minimum function. The blend shown in Figure 32 is carried out with approximate shape preservation, by applying an exponential smoothing function to the two surface distance fields and using a constant parameter that adjusts the overall range of the blend.

$$smin = \frac{-\log\left(\exp(-kf_A(\boldsymbol{x})) + \exp\left(-kf_B(\boldsymbol{x})\right)\right)}{k}$$

where $\log$ is the natural logarithm, and $-k$ creates a decreasing function and controls blending range. The two surfaces are combined (union) and blended exponentially with diminishing influence as point $\boldsymbol{x}$ becomes further away from the surface.

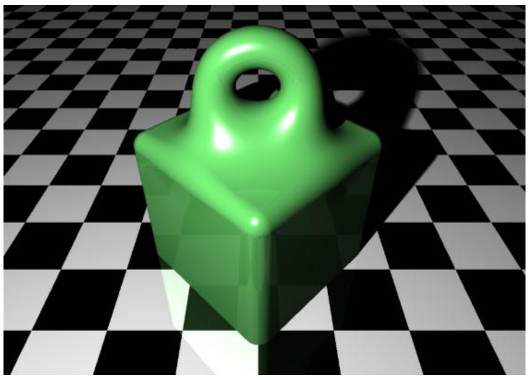


**Figure 32: A blended torus and cube from the prototype.**

#### 4.2.7.1.5 Domain Repetition

A potent example of domain based operations are no more evident than with infinite repetition. By simply applying a modulo operation to the domain position it can effectively map an endlessly repeating implicit surface across the entire space as shown in Figure 33.

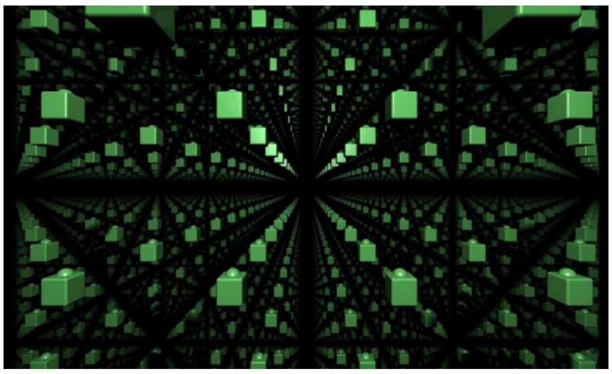


**Figure 33: Infinite domain repetition in the prototype.**

# 5 CUDA Principles

As detailed in the background section in **2.6,** the GPGPU implementation aspect of this project will utilise NVIDIA's CUDA framework.

The key implementation aspects of CUDA related to the project will now be detailed.

## 5.1 Kernels

Kernels are the workhorse functions that CUDA execute to compute data on the GPU device. The programmer defines the kernel in similar fashion to that of a C function, but with the addition of a '*__global__*' keyword to declare it as resident on the GPU, yet callable from the host. Kernels also require a 'launch configuration' which determines the number of parallel executions of the kernel that will be performed based on the number of allocated thread blocks i.e. each block effectively executes a copy of the kernel, using the following syntax:

```
kernel<<<2,1>>>();
```

where in this simple example, two 1D blocks would be specified to launch the kernel at run-time.

To differentiate between each instance of the kernel, a built-in CUDA variable can be called upon at any time to determine the block ID of the current execution as shown here:

```
int x = blockIdx.x;
int y = blockIdx.y;
```

For problems with two dimensional domains such as image rendering or matrix math's, 2D blocks can be defined to allow appropriate indexing for the problem at hand as shown above by the 'y' variable. The CUDA '*dim3*' type representing a 3D tuple, can be used to define multi-dimensional blocks in similar form to declaring a 2D vector type. Currently, despite '*dim3*' type being in 3D tuple form, CUDA only supports 2D grids, with the third tuple automatically being assigned to '1' (Sanders & Kandrot, 2010).

Once a handle has been established on the current block execution, traditional CPU SIMD operations like a one hundred iteration 'for loop' of the form:

*for(int i = 0; i < 100; ++i)*

is equivalent in SIMT fashion to defining a 1D block size of one hundred, executing the kernel and then retrieving the block ID as follows:

*kernel<<<100,1>>>();*
*(Inside the kernel)      int i = blockIdx.x;*

the large difference of course being that the latter SIMT example would execute in parallel rather than iteratively.

## 5.2 Device Functions

The kernel is the entry point that enables computation on the GPU and other functions defined on the device can be then called from within the kernel to assist. These functions require the '__*device*__' prefix in order to tell the NVCC compiler that it is to be executed on the GPU. Host functions cannot be directly called from a kernel however, in the case of custom classes this can be worked around by prefixing functions with both '__*host*__' and '__*device*__' to inform the compiler to make two copies that can be executed either by the host or device.

## 5.3 CUDA Thread Mapping

Clearly for this project, due to the requirement of producing an offline rendered image, the problem domain is that of two dimensions. As mentioned in section **2.2,** the ultimate goal of the ray tracing algorithm is the construction of an image determined by the storing of pixel colour information. This presents two primary requirements to make use of CUDA's parallel functionally during rendering. Firstly, to break down the problem into many blocks of threads that can be independently executed over the entire problem space, and secondly to retrieve the pixel data from GPU device memory back onto the host to construct the image.

Fortunately due to ray tracing being a highly parallel task, branch divergence aside, there is next to no fundamental changes required to the original ray tracing algorithm outlined in section **2.2,** for CUDA implementation. The greater challenge comes from the implementation of the CUDA API, syntax difference and fine-grained compatibility nuances that makes porting from a CPU ray tracing implementation to CUDA a non-trivial task.

The first challenge of breaking the problem down, can be addressed as hinted at earlier in section **5.1**, by defining the appropriate kernel launch configuration and 'Grid' layout.

A grid, being simply the '*dim3*' two dimensional layout of the number of blocks that are to execute a kernel (see Figure 34).

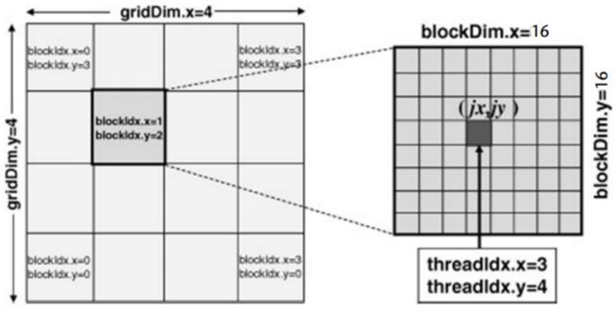


**Figure 34: Visualisation of grid / block layout (Zachmann, 2013)**

## 5.3.1 Thread Block and Grid Dimensions

The size of the image that is required to be processed is an important factor to establishing the size of the problem area. To make things easier to calculate, the size of the grid can be defined by the following method in CUDA:

*dim3 grid(IMAGE_WIDTH / block.x, IMAGE_HEIGHT / block.y);*

this effectively creates the grid of the appropriate dimensions by dividing the image space into the appropriate number of thread blocks. The larger the thread block dimensions, the fewer blocks will be needed to comprise the grid and conversely, the smaller the thread blocks, the more that will be required.

Before establishing the grid dimensions, the block dimensions need to also be defined. To optimise performance the dimensions should be multiples of 32 in accordance with the CUDA predefined warp size (Nvidia Corporation, 2013) where up to 32 threads can be executed at any one time per block. A 2D block can be defined in CUDA as follows:

*dim3 block(16,16);*

defining a block of 16x16 dimensions and thus 256 threads, where groups of 32 threads will be executed as 'warps' by the SM. The aspect ratio of the image is also an important element in the decision when determining the block size since any overlap of the grid compared to the image dimensions will be a inefficient, and any insufficiently sized grid will result in missing parts of the image.

Figure 34 shows a representation of how the image space can be divided up into regions, where the individual threads in each block represent the pixels. To index the threads for use in processing each pixel, the thread ID can be accessed via the block ID built-in variable as follows:

*int x= blockIdx.x * blockDim.x + threadID.x;*
*int y= blockIdx.y * blockDim.y + threadID.y;*

### 5.3.2 Device Memory Allocation and Retrieval

#### 5.3.2.1 *Retrieval*

The second requirement is that once processing is finished and the data has been constructed, it must be retrieved from the device memory back to the host. This can be handled in a number of ways in CUDA depending on your implementation and how your kernel(s) interact.

To retrieve any data back from the device  a pointer should be passed to the kernel upon launch. The memory for the pointer is first pre-allocated on the device before the kernels execution as the following example shows where 'd_data' is of type '*int*':

*cudaMalloc(&d_data, sizeof(int));*

upon completion of the executing kernel, the passed in pointer will act as a handle to the area of memory on the device where the required data is stored. It is then copied from device memory back to host memory with the following command:

*cudaMemcpy(h_data, d_data, sizeof(int), cudaMemcpyDeviceToHost);*

where 'h_data' is an '*int*' variable in host memory and the destination to where the device 'd_data' will be copied.

#### 5.3.2.2 *Allocating Global Memory*

There are a number of ways that variables can be allocated on the device for processing as per section **2.6.3**. Global memory is useful because it can be used to input and output data to and from the device, additionally it can be accessed across different kernels. Global memory can be declared statically on the device using the '*__device__*' prefix on a globally scoped variable and retrieved using '*cudaMemcpyFromSymbol*'. It can also be dynamically allocated at runtime as per section **5.3.2.1** using 'cudaMalloc' and retrieved using '*cudaMemcpy*'. Lastly, global memory can be allocated inside a kernel dynamically using C-style '*new*' and '*malloc*' keywords, however with this method, the memory cannot be accessed via the host.

Data can also be copied into device global memory from the host using the following method:

*cudaMemcpy(d_data, &h_data, sizeof(int), cudaMemcpyHostToDevice);*

where 'd_data' is a pointer pre-allocated using 'cudaMalloc' to the area of memory on the device where 'h_data' should be copied. 'd_data' can then be passed to the appropriate kernel for computation.

#### *5.3.2.3 Allocating Constant Memory*

Constant memory in CUDA relates to a limited area of memory that doesn't change over the duration of the kernel. It features a faster read time due to it being stored on the device read-only cache as described in section **2.6.3**. To declare a constant variable it must be done at global scope using the '__*constant*__' prefix. '*cudaMemcpyFromSymbol*' can be used to copy data directly into constant memory from the host. Additionally on newer devices, kernel parameters are stored in constant memory when passed as a 'const' pointer to the kernel.

#### *5.3.2.4 Allocating Shared Memory*

Shared memory resides on-chip and so is far faster than global memory. Additionally any data stored in shared memory is accessible to all threads in a block making it useful for cooperative parallel algorithms and data caches (Harris, 2013). Shared memory is declared inside the kernel using the '__*shared*__' prefix.

#### *5.3.2.5 Allocating Thread Register Memory*

For compute capability 2.1 devices as used for this project, each thread is limited to 63 registers. Register memory is very fast and is allocated by default when declaring variables on device without the aforementioned prefixes. If however the limit of registers is reached, variables may instead be assigned from local memory, adversely impacting performance.

## 5.4 CUDA Structure and File Compilation

### 5.4.1 File Compilation

Prior to the CUDA SDK 5.0 there was no support for separate compilation of CUDA code files. While host code could still be separated, all source files containing CUDA code had to reside in a single file referred to as 'whole program compilation' (Nvidia Corporation, 2014). Newer versions of CUDA do support separate file compilation, but is disabled by default, requiring manual activation. Whether this support is something that should be enabled is one example of CUDA specific behavior that can have a significant impact on how the program is designed.

To use any custom C++ class behavior for a CUDA project; classes can each be kept in separate '.h' header files without issue as long as the class definitions are also defined along with the prototypes, ensuring also that the said header file is 'included' in any CUDA source files that require access to this class type.

### 5.4.2 CUDA Project Structure Example

An example method of structuring a CUDA and C++ hybrid solution such as found in this projects, is now described.

The initial 'main' entry point function can be placed in a '.cu' CUDA source file that despite containing C++ source, the NVCC compiler will still compile without issue.

Inside 'main' the CUDA program can be initialised, with the defining of any device memory allocations, grid and block dimensions followed by the calling of the kernel and launch configuration. After this, any transfer of data back from the device to host memory can be done, followed lastly by freeing up any allocated memory on the device and terminating the program.

The CUDA kernels and device functions can be kept in a separate single '.cu' file, recognized by the NVCC compiler as a CUDA source file. Due to the kernel definitions being located in another source file to that of 'main', a '.h' header file will be required for the declaration of each kernel prototype. This is also a good location to store any global '*structs*' or values that may be needed by both '.cu' files.

A traditional C++ or C# programming approach to static or global variables would be defining them as 'constant' in a source file or header. With CUDA, this is not always appropriate since anything defined on the GPU such as a '*struct*' or a custom class with a non-empty constructor, cannot be statically instantiated in global memory like on the host (Nvidia Corporation, 2013). In general, particularly for simple global integer and float variables, it is an option to use compiler ''*#define*' statements instead. Indeed, these have been used effectively in the project for things such as setting scene configurations, math constants and also some mid-function compiler statements for performance related optimisations.

After the kernel prototypes have been successfully declared in the header file and included, any required device functions that are to be executed by the GPU can be defined in the '.cu' source file with the kernels. Since the device functions will be called only from the kernels and from each other within the same file, there is no need to declare any function prototypes in the header file unless they are also required to be executed from the host in the 'main' source file, in which case the functions can be prefixed with both '*__host__*' and '*__device__*' as outlined in section **5.2**.

# 6 Technical Implementation

As outlined in the projects aims and objectives in section **1.2.3,** the intention as the project developed was to implement four separate prototypes to showcase both the performance of ray tracing on the CPU compared to GPGPU, and also two different ray tracing techniques, namely traditional ray tracing and ray marching with distance functions.

All four of these prototypes have been successfully implemented using the research and theory detailed in the prior 'Background Overview', 'Ray Tracing Principles', 'CUDA Principles' and 'Meshless Implicit Modeling' sections. This section will describe the implementation of each prototype.

## 6.1 Development Environment

It was decided that the prototypes be developed using C++ for several reasons, firstly C++ is a widely used industry standard for 3D graphics and importantly, also compatible with CUDA. Secondly, it is highly modular, allowing the prototypes to be developed as lean as possible with only required bare-bone libraries. Graphical API's such as OpenGL/DirectX were not used, instead all rendering is performed using from scratch code developed for this project. Since the project would aimed at a standard modern Windows environment, Visual Studio 2012 was the natural choice at the time of starting the project due personal familiarity, wide support with CUDA and excellent functionality. The project was developed on a Windows 8 64-bit system.

### 6.1.1 GLM 0.9.4.4 - Open GL Mathematics

The GLM math's library is included only with the two later ray marching prototypes for a number of reasons. Firstly while developing the project and carrying out research it was helpful for vector syntax to be identical to that of OpenGL shader language (GLSL) due to nearly all research material on the subject being presented in such form. Despite developing my own vector math utility header, it lacked a number of matrix math functions that GLM offers already. These could have been integrated manually with enough time but the decision was made to prioritize development goal progress over the creation of unimportant utility libraries that would duplicate functionality that is already available. The GLM library is very light weight and modular and has no implications to the overall achievement of the project. This is the only third party library used in any of the prototypes excepting of course the CUDA SDK.

### 6.1.2 CUDA Software Development Toolkit 5.5

The two CUDA prototypes have the necessary inclusion of the NIVIDA CUDA libraries in order to program the NVIDIA GPU. Specifically, only the 'cuda_runtime.h' and 'device_launch_parameters.h' headers were used.

The toolkit features the necessary plug-ins for Visual Studio 2012 which allows using the IDE GUI to adjust settings such as the compute capability version that the CUDA source files should be compiled for and altering the NVIDIA GPU debugging options. The SDK also installs the 'NSight' debugging tool which assists in troubleshooting errors and behavior when inside the kernel at run-time.

## 6.2 Hardware Environment

The project was developed on a Dell XPS-17 laptop of a moderate performance specification at the time of the projects inception. A summary of the hardware specifications and GPU CUDA capabilities used for the project can be found in Appendix I.

The prototype performance review should factor in that the GPU chip on the laptop was a mobile version, and thus optimised for low energy consumption and thermal output rather than performance like an equivalent desktop NVIDIA graphics card.

## 6.3 Prototypes Overview

This section is in five parts. The first is a succinct note on the terminology that will be used to refer to each of the prototypes (**6.3.1**). The second covers the shared standard ray tracing algorithm used by prototypes 1 and 2 (**6.3.3**) followed by a description of the CPU ray tracing prototype (**6.3.4**). The third is a section on general CPU to CUDA migration issues (**6.3.5**) followed by a description of the CUDA ray tracing prototype (**6.3.6**). The fourth part covers the description of the implemented ray marching algorithm (**6.3.7)** and lastly the fifth part details the last two ray marching prototypes both featuring the algorithm (**6.3.8-9**). Any class or design behavior that is common across multiple prototypes will be addressed as it is first raised, thus the later prototype descriptions will exclude any prior mentioned classes.

### 6.3.1 A note on terminology

The terms ‘ray tracing’ and ‘ray marching’ are not mutually exclusive and by utilising ray marching, ray tracing is still potentially being carried out. Ray tracing's most defining attribute is that of the computation of 'secondary' rays for reflections, refractions and shadows, which sets it apart from 'ray casting', a technique that concerns itself only with the primary rays. Ray marching can be considered a subset of ray tracing through which implicit surfaces can be computed via the use of distance functions.

For ease, and the purpose of this report, the 'ray tracing prototypes' will refer to the two standard ray tracer prototypes featuring no ray marching and algebraically defined spheres and planes. The 'ray marching prototypes' therefore will refer to the other two prototypes utilising ray marching and distance functions.

### 6.3.2 Ray Tracing Prototypes

Both the CPU and CUDA ray tracing prototypes have the same level of functionality and use the same ray tracing algorithm for fairness of comparison. Despite this, various code design differences do exist between prototypes to accommodate the migration of the algorithm over to CUDA.

### 6.3.3 Ray Tracing Prototype Features (CPU and CUDA)

Both prototypes are capable of the following functionality:

- Rendering a ray traced 3D scene using algebraically defined implicit geometry (spheres, plane) to Bitmap.
- Outputs Bitmap file with image resolution to desired size and aspect .
- Phong-based local surface illumination.
- Supersampling anti-aliasing to desired amount of sub-samples.
- Supports multiple light sources, both 'point' and 'directional' lights.
- Inter-object light scattering (Reflections) to desired depth.
- Accurate hard shadows.
- Positional camera.

#### *6.3.3.1 Implemented Algorithm*

The implemented ray tracing algorithm for both prototypes is presented in pseudo code:

```
01 // for each pixel
02 // {
03 //     for each AA sub-sample
04 //       {
05 //           calculate the direction between camera and pixel sample
06 //           fire a primary ray through the sample
07 //           compute nearest intersection point with an object
08 //
09 //           if the ray hit an object
10 //           {
11 //               update sample colour with ambient term
12 //               if the object material is reflective
13 //               {
14 //                   compute a secondary 'reflection' ray
15 //                   if the reflection ray hit an object
16 //                   {
17 //                       recurse from line 12
18 //                       add object colour to sample colour
19 //                   }
20 //               }
21 //
22 //               for each light source
23 //               {
24 //                   if light is a 'point light'
25 //                       compute light attenuation
26 //                   compute a secondary 'shadow ray' towards light
27 //                   if 'shadow ray' doesn't intersect & attenuation>0
28 //                       update sample colour with diffuse & specular
29 //
30 //               }
31 //               set sample colour to final colour
32 //           }
33 //           else set sample colour to background colour
34 //       }
35 //
36 //       for each sub-sample of this pixel
37 //           compute the average colour
38 //       set the pixel colour to averaged colour
39 // }
40 //
41 // output pixel buffer to bitmap image.
```

**Figure 35: Ray tracing algorithm pseudo code**

### 6.3.4 CPU Ray Tracer - Prototype 1

#### *6.3.4.1 Class Diagram and Design*

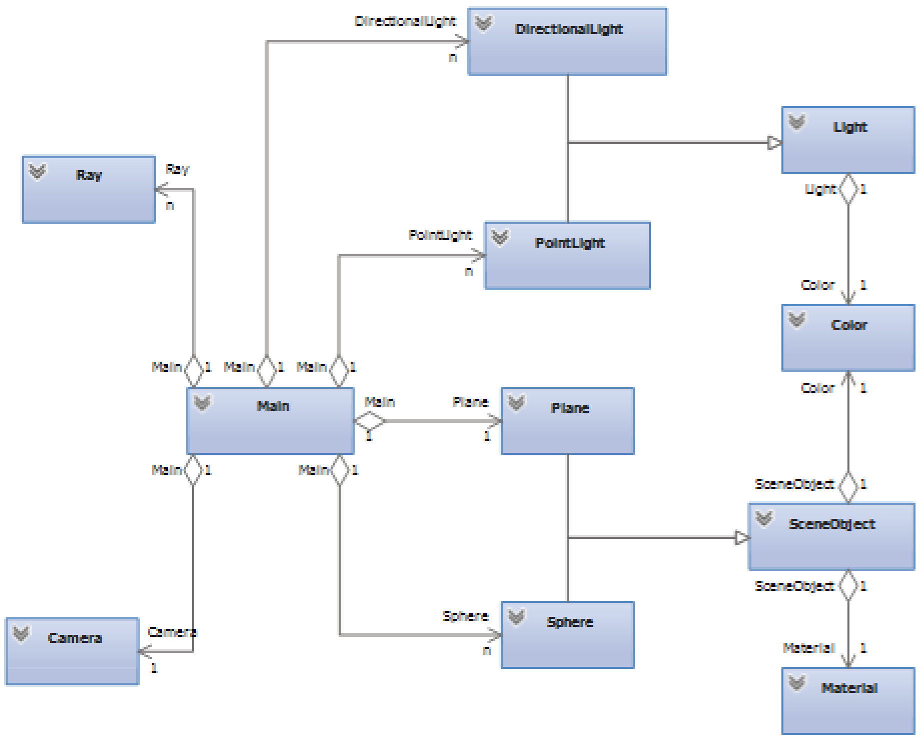


**Figure 36: UML Class diagram for CPU ray tracer prototype.**

As can be seen from Figure 36, the CPU ray tracer prototype has been implemented with a heavily object orientated design which makes construction of scenes much more manageable and reduces code repetition. Inheritance has been used to allow abstraction for handling multiple scene objects and lights via the use of vector list containers. Not shown on the diagram is the 'Vec' class, a vector type that has been developed for use with the ray tracing prototypes. Nearly all classes instantiate 'Vec' in some form and so has been left from the diagram for clarity.

#### *6.3.4.2 'Main.cpp'*

The main ray tracing algorithm as detailed in section **6.3.3.1** is executed here in the entry point for the program and should be consulted for a detailed 'step-by-step' of the programs procedure. 'Main' instantiates the majority of classes in the program, intitialises the camera and scene primitives, initiates the performance timers and ends with the creation of the Bitmap image file by encoding the computed colour buffer. Many constant values are located at the top for configuring the program e.g. adjusting anti-aliasing depth, recursion limit for reflected rays, field of view angle and image dimensions.

#### *6.3.4.3 'Vec' Class*

The 'Vec' class as mentioned prior is a vector type consisting of three float components '(x,y,z)' in standard 3D tuple form. The class contains various helper functions as well as commonly used operator overloads. The helper functions consist of accessors for each vector component and dot product, magnitude, cross product, normalise and invert functions. The operator overloads consist of addition, subtraction, the unary minus operator and also scalar multiplication for both left and right hand-sides.

#### *6.3.4.4 'Camera' Class*

This is a simple convenience class and contains four 3D vector members used to derive the correct direction for firing the primary rays into the scene. Alongside the members are also the appropriate accessors. The camera determines how all the ray and lighting computations in 3D space are to be mapped onto the image plane for the rendering of the image. It is the origin of all primary rays that are 'cast' into the scene.

#### *6.3.4.5 'Ray' Class*

Again, a simple convenience class used to define a ray consisting of just two 3D vectors and getters used to store the rays origin and direction vector.

#### *6.3.4.6 'Color' Class*

This class has a 3D tuple layout similar to 'Vec' each representing the three 'RGB' colour values used to create the Bitmap image. As well as featuring accessors and mutators for each component it also has some basic arithmetic helper functions used for adding, multiplying, scalar multiplying and averaging.

#### *6.3.4.7 'Material' Class*

Here, a material is defined for use with the Phong illumination model. The class contains the ambient, diffuse, specular, specular exponent, reflectivity and transparency properties for successfully lighting the scene. The material is instantiated inside each primitive.

#### *6.3.4.8 'Light' Class*

The 'Light' class acts are the 'base' for the two different lighting types available in the ray tracing prototypes, 'directional' and 'point'. Any common members between the types are added here to avoid code duplication and importantly the class aids abstraction allowing both lighting types to be treated generically inside vector list containers. A useful 'boolean' accessor function is also added here to help identify the lighting type. This is added because only point lights require attenuation calculations and thus it prevents unnecessary calculations to boost performance inside the recursive loop.

#### *6.3.4.9 'DirectionalLight' and 'PointLight' Classes*

These classes are mainly for abstraction with most of the behavior implemented either inside 'Light' or the 'Main.cpp' source file for lighting calculations. 'DirectionalLight' differs from 'PointLight' because it has no lighting radius or attenuation. The directional light represents a 'Sun' type light source with infinitely far reaching light rays. 'PointLight' represents a smaller light source with a limited radius.

#### *6.3.4.10 'SceneObject' Class*

An important 'base' class for 'Sphere' and 'Plane', the two geometric primitive types used in the ray tracer prototypes. It features a material member and function prototypes for accessing the colour, normal and calculating an intersection.

#### *6.3.4.11 'Sphere' and 'Plane' Classes*

The primitive classes are used to instantiate instances of geometric surfaces for rays to intersect against as derived from the implicit equations of a sphere and plane as detailed in section **3.0.** The most notable members are the intersection functions that take a ray parameter and return the solution to the implicit equation, thus determining an intersection.

### 6.3.5 CUDA Migration Considerations

There are a number of necessary steps required by both CUDA prototypes when migrating over the functionality of the CPU prototypes that will be listed here. All of which applies to both CUDA implementations (prototypes 2 and 4).

The CUDA prototypes in both scenarios were developed after the equivalent CPU prototypes, which were used as templates to develop from. Significant time was spent researching and learning the different principles behind developing with the CUDA API, especially with regards to kernel launch configuration, thread ID mapping, device/host differentiation and GPU memory management. After assessing CUDA's capabilities, it was apparent there would be several limitations that would require change to the initial prototype template.

#### *6.3.5.1 Polymorphism*

The polymorphism based design that is relied on 'in prototype 1' does not function in CUDA. As a result, inheritance is removed from several classes. Specifically for the CUDA ray tracer, new classes, 'SceneObjects' and 'SceneLights' are present containing pointers for the primitives and light types which can be accessed as a type specific array. This means moving some logic into the kernel to compensate for the lack of abstraction. Namely, copying in each scene primitive and light from the host, into device memory and passing them as array pointers. The ray marcher prototypes use no inheritance or polymorphism and handle primitive creation differently, as described later in section **6.3.8.3**.

#### *6.3.5.2 Thread ID Mapping*

The removal of the iterative pixel '*for*' loop that processes the colour for each part of the image is required to take advantage of the SIMT CUDA behavior described in section **5.1**. Block threads are instead mapped to every pixel and processed in parallel. Once the pixel coordinate is derived it can be mapped from a 2D element (block thread) onto a 1D structure (pixel buffer) after colour computation by using the following equation:

$$p = y * W + x$$

Where $p$ is the current pixel in the 1D buffer, $y$ and $x$ are the 2D representation of the current pixel, and $W$ is the image width.

### *6.3.5.3 Function Prototype Declarations*

Thirdly, for CUDA to be able to compile custom classes in separate header files each class function prototype required the addition of the '__device__ __host__' double prefix. Additionally for the 'Vec' class, the operator overloads are global and have to be forced inline via '__forceinline__' to prevent multiple definition errors when 'Vec' is included.

### *6.3.5.4 Recursion Alternative*

CUDA capability 2.0 and later devices do support recursion for device kernels, however after researching alternative approaches (Tsiodras, 2011) to handling the secondary rays, an alternative simple method using C++ template meta-programming is implemented as outlined in a simple example in Figure 38:

```
template <int depth>
__device__ Color getColorAt( /*...*/)
{
 // .....

 // Replaces recursion
 Color reflectionIntersectionColor = getColorAt<depth+1>(/*...*/);

 // ..... calc final color

 return finalColor;
}

template <>
__device__ Color getColorAt<MAX_RAY_DEPTH>(/*...*/)
{
 return Color(0,0,0);
}
```

**Figure 37: Template recursion alternative code**

In Figure 38, the first kernel "getColorAt" is initially called with a template argument corresponding to the initial starting depth of zero, then during a reflection ray calculation where a recursive call would usually be made, the kernel is called again but with a "<depth+1>" argument, and thus calling a new copy of the function. This is then repeated until the second template prototype at the bottom of Figure 38 is triggered when the "depth" value reaches the depth limit, ending the recursion.

### 6.3.6 CUDA Ray Tracer - Prototype 2

#### *6.3.6.1 Class Diagram and Design*

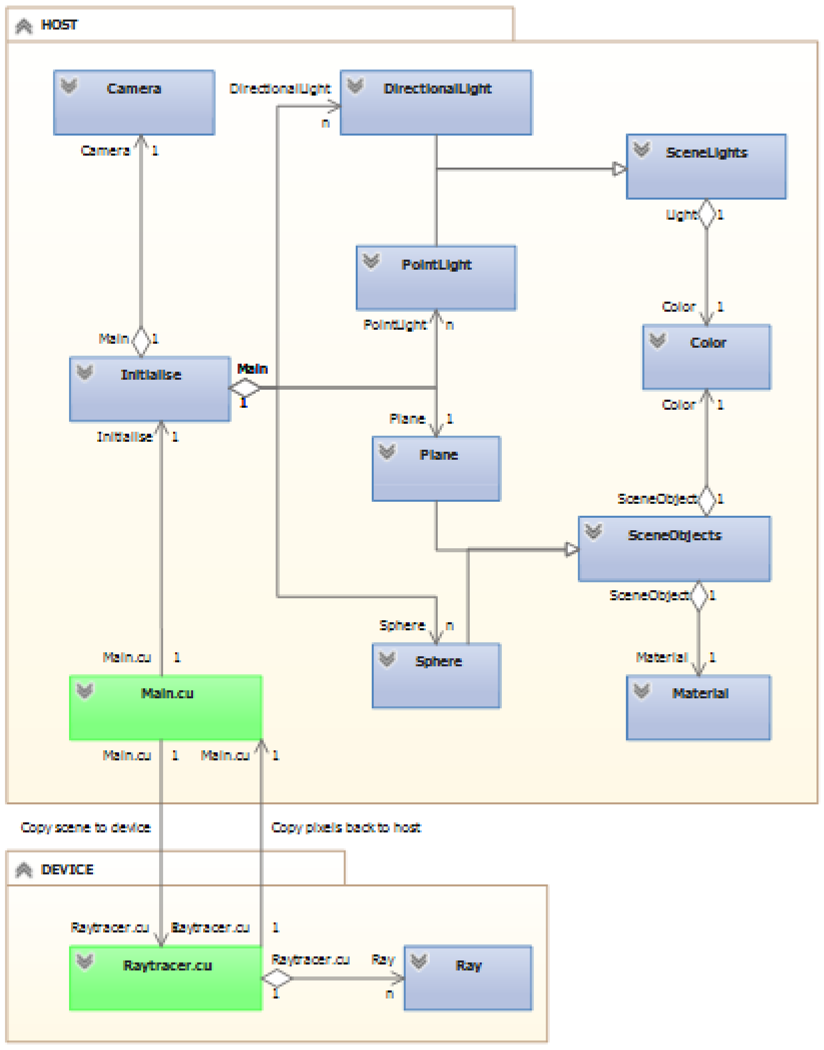


**Figure 38: Class diagram for CUDA ray tracer prototype.**

The CUDA ray tracer prototype features many of the same class types as 'prototype 1'. The class diagram in Figure 39, highlights the two classes containing CUDA specific code in green. The classes are contained either in the 'host' or 'device' to demonstrate where they are instantiated in the program. Much of the same object orientated design has been replicated however due to limitations with CUDA, some changes have been made for either compatibility or ease. One change is the addition of the 'Intialise' class used to instantiate the scene but also to store the program configuration settings that were previously in 'Main' in prototype 1.

#### *6.3.6.2 'Main.cu'*

The entry point to the program, 'Main' is responsible for calling 'Initialise' to instantiate the scene objects, allocate and copy them into device memory and then launch the 'raytracer' kernel. It also starts the performance timer, handles the transfer of pixel data from the device back to the host and creates the Bitmap image file.

#### *6.3.6.3 'Initialise' Class*

This class is called from within 'Main' and importantly is responsible for instantiating all scenes objects such as the primitives, lights and initialising the camera. This is done on the host and then copied into device global memory for access by the kernel. In order to quickly arrange different scenes, the 'InitialiseScene' is a 'template' function that takes an integer to identify which scene should be created. A scene is selected via a compiler 'define' statement at the top of the class which then feeds into the function call from 'Main'. Additionally, all program configuration settings such as image resolution, anti-aliasing depth etc. have been relocated into here as compiler 'define' statements for use with CUDA as described in **5.4.2**.

#### *6.3.6.4 'Raytracer.cu'*

This is the CUDA source file that contains the ray tracing kernel that executes the algorithm outlined in **6.3.3.1**. All CUDA migration steps have been implemented as mentioned in **6.3.5**, with particular note to the change in the way secondary ray recursion is now handled using template meta-programming to allow the GPU to instantiate new copies of the 'getColorAt' function recursively to a desired maximum depth. As per CUDA SIMT, each thread block will execute a copy of the kernel and compute the image pixel data, storing it in the memory location defined by the pointer passed into the kernel upon launch. The threads are able to independently execute the program with no inter-thread communication needed. After the end of the kernel, the threads are synchronized in 'Main.cu', waiting for the last thread to finish before stopping the performance timer.

### 6.3.7 Ray Marching Prototype Features (CPU and CUDA)

Both ray marching prototypes are capable of the following functionality:

- Ray traced 3D scene via sphere traced ray marching techniques.
- Implicitly defined complex geometry via the use of distance functions and domain ops
- Ray marching adjustable to number of desired steps and surface epsilon (threshold).
- Surface Distance operations: Union, intersection and subtraction.
- Domain operations: soft minimum blend and infinite domain repetition.
- Outputs Bitmap file with image resolution to desired size and aspect.
- Inter-object light scattering (reflections and refraction) to desired depth.
- Refractive index adjustments simulate refractive material e.g. hollow, solid glass.
- Accurate hard and soft shadows with adjustable penumbra scaling.
- Phong-based local surface illumination.
- Supersampling anti-aliasing to desired amount of sub-samples.
- Adjustable reflection and refraction depth value (default 5 recursions).
- Supports single 'point' light with adjustable attenuation.

The implemented ray marching algorithm for both prototypes is now presented (Figure 41)

```
01 // for each pixel
02 // {
03 //       for each AA sub-pixel
04 //       {
05 //           calculate direction between camera and pixel sample
06 //           fire a primary ray through sample and march
07 //
08 //           for each step along the ray
09 //           {
10 //               get current step position
11 //               compute distance to nearest surface (distance field)
12 //               set next step size to distance of nearest surface
13 //               if distance < intersection threshold (surface hit)
14 //               {
15 //                   Calculate surface normal and illumination vectors
16 //
17 //                   fire a new ray towards light source and march
18 //                   for each step along the shadow ray
19 //                   {
20 //                       get current step position
21 //                       compute distance to nearest surface
22 //                       set next step size to nearest surface distance
23 //                       if distance < intersection threshold (occluded)
24 //                       {
25 //                           return a shadow value based on transparency
26 //                       }
27 //                   }
28 //                   else return a calculated penumbra value
29 //
30 //                   compute surface material lighting
31 //
32 //                   if reflective {cast reflection ray (recurse line6)}
33 //
34 //                   if surface transparent
35 //                   {
36 //                       calculate refractive index
37 //                       cast refraction ray (recurse to line 6)
38 //                   }
39 //
40 //                   compute lighting attenuation
41 //                   compute colour from light, material, shadow values
42 //                   break loop
43 //               }
44 //           }
45 //           return colour
46 //       }
47 //       for each sub-pixel sample get average colour and assign
48 // }
49 // output the pixel buffer to a bitmap image
```

**Figure 39: Ray marching algorithm pseudo-code.**

### 6.3.8 CPU Ray Marcher - Prototype 3

#### *6.3.8.1 Class Diagram and Design*

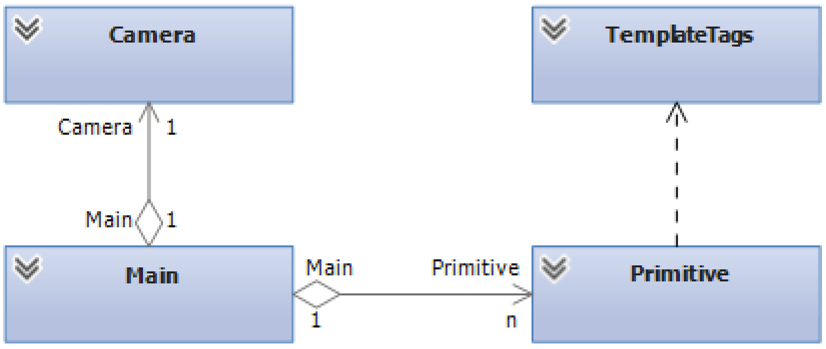


**Figure 40: UML Class diagram for CPU ray marching prototype.**

As can be observed from Figure 42, much of the object orientated design has been stripped out for the last two prototypes. This decision was taken a several reasons. Firstly, object orientated design, while convenient has performance implications due to additional instantiation, accessor and mutator overheads. Additionally, certain roles such as 'Color' and 'Ray' were effectively redundant and have been replaced using vectors. The prototype has been developed to be as lightweight and flexible as possible.

#### *6.3.8.2 'Main.cpp'*

'Main' is the entry point for the program and also where the core ray marching algorithm is executed as per **6.3.7**. Caching objects globally and accessing them statically via reference avoids any instantiations during the ray march loop which could impact performance. This is particularly evident with matrices instantiations. Much of the distance function behavior is encapsulated in a 'Primitive' type object that represent the various shapes that have been implemented. Inside 'Main' is the primary distance field function where combined fields can be added and returned via the union operation. The same template meta-programming implementation for the recursive secondary rays is also utilised here. Additionally, helper functions for normal and shadow calculations are present.

#### *6.3.8.3 'Primitive' Class*

To improve scene creation, a wrapper for handling the creation of primitives has been developed. Inside are the constructors for various primitives. Additionally, it includes accessors and mutators for data values as required. Typically, distance fields return a distance float representing the distance to the surface, however in order to access additional information about a surface such as its material type, the distance functions in 'Main.cpp' return a reference to a 'Primitive' object. 'Primitive' also includes all the distance functions and matrix transforms such as translate and rotate as well as a scaling op. The constructors assign the primitives position in the scene and overloads carry out shape specific initialisation.

Template functions are implemented for each primitive shape, whose class type is located in the 'TemplateTags' class. The tags have no functionality and are used simply as an ID mechanism that can be passed into a '*instance*.computeDistance<*tag*>()' function call, after which the primitive transforms the domain position by its matrices and then calls the correct distance function, storing the distance value for later access. Implementing this, considerably reduced the amount of code located inside the ray march functions inside 'Main.cpp'.

### 6.3.9 CUDA Ray Marcher - Prototype 4

#### *6.3.9.1 Class Diagram and Design*

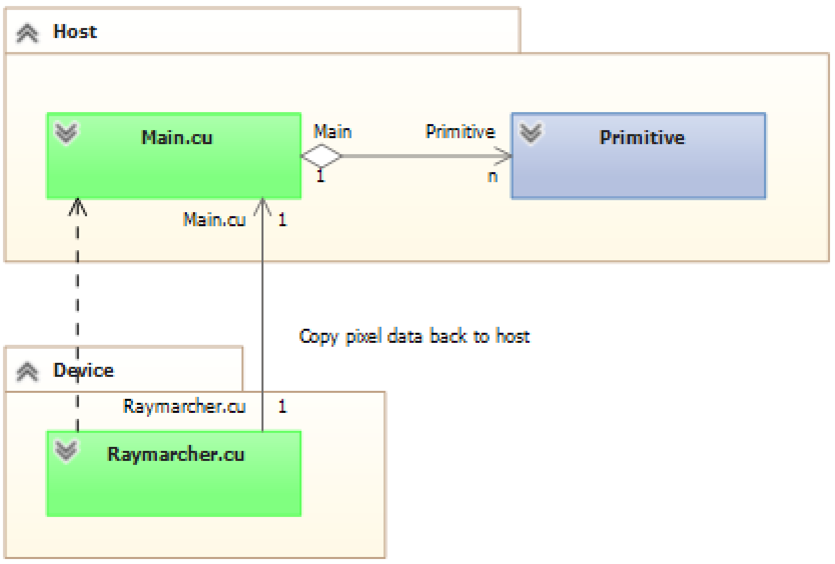


**Figure 41: UML Class diagram for the CUDA ray marcher prototype.**

Figure 44 demonstrates the compactness of the CUDA ray marcher prototype. The camera class has been removed in favor of using vectors and the 'TemplateTags' class has been merged into the 'Primitive' class. Consult section **6.3.5** for details on the general changes implemented for migration of the ray marcher to CUDA.

#### *6.3.9.2 'Main.cu'*

In similar fashion to the CUDA ray tracer prototype, 'Main.cu' is the entry point to the program and is responsible for intialising the kernel launch configuration and executing the kernel. Unlike with the ray tracer CUDA prototype, the scene is not intialised on the host and copied into device memory. This is because of the way in which ray marching generates the implicit surfaces procedurally within the ray march loop. 'Main.cpp' is again responsible for the copying of image data back from the device and the creation of the Bitmap file.

#### *6.3.9.3 'Raymarcher.cu'*

This is where the core functionality of the ray marching program is located and where the algorithm outlined in **6.3.7** is executed inside the kernel. Scene management is an important change for the CUDA ray marcher due to the procedural nature of the ray marching algorithm. As per the CPU ray marcher the 'Primitive' class is implemented to make creation of implicit surfaces and lighting management more streamlined, with the distance functions returning a primitive object instead of a float. To avoid unnecessarily large amounts of parallel instantiations within each ray step, the scene is initialised once per thread inside fast registry memory. This memory is local to the thread and ends with its lifespan. To aid performance, the primitive is instantiated outside the march loop and is passed by reference to the distance fields and functions inside. Global variables are not a simple option for this design since many threads would be contending over the same memory locations causing undefined behavior. The per thread configuration avoids this pitfall.

## 6.4 Prototype Rendering Times

See Appendix J for details on how the below timings were benchmarked and Appendix K for images of the scenes used for benchmarking:

### 6.4.1 Ray tracer Prototype Render Times:

| Rendering Settings | CPU Render Time | CUDA Render Time | CUDA Performance Gain Factor | CUDA Block Count / threads per block |
|---|---|---|---|---|
| Scene 1, 640x480, no AA | 0.91s | 0.22s | 4.14 | 2400 / 128 |
| Scene 1, 640x480, 4x AA | 12.5s | 3.66s | 3.42 | 2400 / 128 |
| Scene 1, 640x480, 8x AA | 50.52s | 14.54s | 3.47 | 2400 / 128 |
| Scene 1, 1024x768, no AA | 2.125s | 0.51s | 4.17 | 6144 / 128 |
| Scene 1, 1024x768, 4x AA | 31.92s | 8.55s | 3.73 | 6144 / 128 |
| Scene 1, 1024x768, 8x AA | 130.25s | 34.81s | 3.74 | 6144 / 128 |
| Scene 1, 1920x1080, no AA | 4.53s | 1.22s | 3.71 | 16200 / 128 |
| Scene 1, 1920x1080, 4x AA | 72.53s | 19.09s | 3.79 | 16200 / 128 |
| Scene 1, 1920x1080, 8x AA | 291.35s | 76.58s | 3.8 | 16200 / 128 |
| Scene 2, 640x480, no AA | 1.06s | 0.43s | 2.46 | 2400 / 128 |
| Scene 2, 640x480, 4x AA | 16.48s | 6.82s | 2.41 | 2400 / 128 |
| Scene 2, 640x480, 8x AA | 71.98s | 27.53s | 2.61 | 2400 / 128 |
| Scene 2, 1024x768, no AA | 2.72s | 1.00s | 2.72 | 6144 / 128 |
| Scene 2, 1024x768, 4x AA | 42.89s | 16.37s | 2.62 | 6144 / 128 |
| Scene 2, 1024x768, 8x AA | 166.89s | 66.02s | 2.52 | 6144 / 128 |
| Scene 2, 1920x1080, no AA | 6.41s | 2.28s | 2.81 | 16200 / 128 |
| Scene 2, 1920x1080, 4x AA | 99.79s | 37.33s | 2.67 | 16200 / 128 |
| Scene 2, 1920x1080, 8x AA | 436.43s | 151.28s | 2.88 | 16200 / 128 |
| | | **Average**: | **3.2** | |

### 6.4.2 Ray Marcher Prototype Render Times:

| Rendering Settings | CPU Render Time | CUDA Render Time | CUDA Performance Gain Factor | CUDA Block Count / threads per block |
|---|---|---|---|---|
| Scene 3, 640x480 | 3.24s | 0.46s | 7.04 | 2400 / 128 |
| Scene 4, 640x480 | 4.92s | 0.69s | 7.13 | 2400 / 128 |
| Scene 5, 640x480 | 9.88s | 8.09s | 1.22 | 2400 / 128 |
| Scene 6, 640x480 | 13.3s | 3.48s | 3.82 | 2400 / 128 |
| Scene 3, 1024x768 | 8.9s | 0.98s | 9.08 | 6144 / 128 |
| Scene 4, 1024x768 | 12.73s | 1.43s | 8.90 | 6144 / 128 |
| Scene 5, 1024x768 | 24.3s | 17.48s | 1.39 | 6144 / 128 |
| Scene 6, 1024x768 | 33.77s | 6.81s | 4.96 | 6144 / 128 |
| Scene 3, 1920x1080 | 20.88s | 1.94s | 10.76 | 16200 / 128 |
| Scene 4, 1920x1080 | 33.62s | 3.43s | 9.8 | 16200 / 128 |
| Scene 5, 1920x1080 | 62.34s | 39.8s | 1.57 | 16200 / 128 |
| Scene 6, 1920x1080 | 73.55s | 11.16s | 6.59 | 16200 / 128 |
| | | **Average:** | **6** | |

## 6.5 Project Task List

The project task list (see Appendix G), is a table of tasks required for not only the implementation of the project outlined in this report, but also associated research and documentation. It is broken into five overall phases marked by prototypes stages as detailed in the table. The Task ID's correspond to the ID's in the project Time Plan (Appendix H).

## 6.6 Project Time Plan

A time plan has been constructed (see Appendix H) based on the allocated tasks in the Task List (Appendix G).

## 6.7 Prototype Renderings

See Appendix K for some images produced by the prototypes.

# 7 Critical Evaluation

## 7.1 Project Achievements and Analysis

The project goals outlined in section **1.2.3** set out  to produce four prototypes to research the performance benefits of using a GPGPU technique to implement a ray tracer and additionally investigate the functionality of ray marching and distance functions.

All objectives were fulfilled in the project, this in itself alongside the research and renderings included in this report are one aspect of the projects achievements. The rendering times of each prototype detailed in section **6.4** can now be analysed. The renderings shown in Appendix K on the other hand show explicitly the visual benefits to be gained from the implementation of ray marching.

The times will be examined in three parts, firstly the findings of the two ray tracing prototypes, secondly the two ray marching prototypes and thirdly a final conclusion.

### 7.1.1 Ray tracing Prototypes (1 & 2) Performance

The first table in **6.4.2** demonstrates the significant impact that the anti-aliasing measures implemented have on the performance of the algorithm, ultimately supporting an established limitation of ray tracing being aliasing. Specifically, in the 1920 by 1080 resolution rendering of scene 2; the 8 times anti-aliasing increases rendering duration by 68 times compared to that of the same scene with no anti-aliasing. The cause of this is detailed in section **2.5**.
The 'CUDA Performance Gain Factor' column in the table shows how many times faster the CUDA rendering times were compared to the CPU. On average CUDA is 3.2 times faster than the CPU at ray tracing the scenes, with each scene performance gain being pretty consistent at each resolution. One interesting observation in that the performance gain for scene 2 is consistently less than that of scene 1. Scene 2 is a more complex scene and demonstrates that CUDA is faster at computing simpler tasks in relation to the CPU than it is computing more complex ones. This is likely caused by the fact that each SM on the GPU although comprised of many more cores than the CPU, does not individually have the computational might of the CPU, therefore performance gain is less prominent with more 'fine-grained' or complicated calculations. Indeed, this supports the GPGPU architecture outlined in section **2.6**, that is to say, CUDA is especially good at many numerous simple calculations.

As expected, the render times increase linearly as the resolution size increases, again this confirms expected behavior in that the larger renderings require many times more ray computations.

### 7.1.2 Ray marching Prototypes (3 & 4) Performance

As mentioned in Appendix J, no anti-aliasing timings were carried out for the ray marching prototypes for reasons described. Instead four new scenes were rendered showing a variety of the ray marching capabilities and complexity. Although it was perhaps a foregone conclusion that CUDA would outperform the CPU based on the prior results, the margin by which it did was surprising, especially given the increased complexity of the ray marching algorithm.

On average, the CUDA ray marcher was 6 times faster at rendering than the CPU ray marcher, with an exceptional example of being 10 times faster in scene 3 at 1920 by 1080 resolution (1.94 seconds compared to 20.88 seconds). The likely explanation for such a high increase compared to the standard ray tracer is probably due to stepping or 'marching' aspect of the algorithm which when performed iteratively by the CPU loses out considerably compared to the parallel computations along multiple rays done by CUDA.

Yet again it is observed that CUDA's greatest performance gain occurs on the simplest scene of the test (scene 3), emphasizing its particular benefits for simple parallel arithmetic, particularly if less branching and conditional statements are involved.

There is one anomaly that can be observed in the rendering times related to scene 5 specifically. The gain factor for this scene is significantly smaller than that of the other scenes, approximately only 1.5 times faster. The explanation for this lies in the large presence of refractive material in the scene, namely the solid glass columns. CUDA appears less effective with large amounts of conditional branching, where additional secondary refraction rays are calculated up to 5 levels of recursion. Clearly with light scattering all around the scene to such depth; the branching behavior of SIMT mentioned in section **2.6.2** has some difficulties.

### 7.1.3 Results Conclusion

In conclusion the benefits offered by CUDA are far in excess of what was expected before the start of the project and with a more refined implementation its clear even greater gains could be made. Although there are some key scenarios identified where CUDA seems less effective, in all tests, its performance was superior to the CPU. It is clear that the potential for GPGPU for use in ray tracing is great, especially if a more synergized approached detailed later in section **7.2.2,** could be implemented taking advantage of both specialised pieces of hardware.

The conclusion on the use of ray marching for rendering complex implicit geometry is remarkable, with highly impressive rendering possible in very small programs as little as 4KB, as shown by Quilez (2008). As described in section **4.2.7**, the applications for procedural rendering with distance functions opens up almost limitless potential, especially for natural or organic looking environments. The renderings created by the two ray marching prototypes in this project (see Appendix K), though limited, already demonstrate far more implicit surface complexity then is as easily possible without the technique. The power of the domain based transformations and blending operations particularly emphasize this potential.

## 7.2 Further Development

Despite achieving the majority of what could have feasibly been implemented in the time frame; there are some areas in which improvements could be made with knowledge gained.

### 7.2.1 CUDA Single Kernel Design

Both CUDA prototypes are implemented using a single kernel where the ray tracing and ray marching algorithms are executed. The problem with this design is the Windows 'Timeout detection and recovery level (TDR)' causing the device driver to time out and crash the program after a certain duration if the rendering takes too long (Wilt, 2012). This is because traditionally the GPU unlike the CPU is not designed to be computing large programs, instead it is designed to run very small, quickly executed programs for computing graphics. GPGPU architecture such as CUDA changes that by effectively treating the GPU as a CPU. With a single kernel design, the program doesn't terminate until the rendering is complete. By using a multi-kernel design where each kernel is smaller in size, passing computed data onto another, it would help prevent the driver time out issue.

### 7.2.2 CUDA Prototype - Host and Device Synergy

This project has focused primarily on creating a full rendering program residing on either the host or the device. To get the best performance, making use of the strengths of both host and device hardware by spreading out the rendering load across both could have been a good design decision. CUDA is very fast, particular at small quick computations that can be parallelized. The CPU is very good at logical branching and calculating non-parallel tasks (section **2.6**).

### 7.2.3 Object Orientated Design

In ways symptomatic of the use of a single kernel and the lack of efficient synergy between both hardware; enhanced use of object orientated design principles would have been a natural benefit making the handling of scene primitives and scene creation a quicker process. As it stands, object orientated design has been implemented well on the CPU ray tracer prototype but the CUDA prototypes could not effectively take advantage of what it offered under the current design.

### 7.2.4 Real-time Rendering

As well as further CUDA optimisations, space partitioning structures could be implemented in the future to significantly reduce the number of ray computations and achieve real-time performance. This would be alongside a window context where the pixels could be copied via 'blitting' directly to the screen.

## 7.3 Personal Reflection

The project has been a very enlightening experience for me in relation to graphics rendering and research. The concepts of implicit rendering at first seemed confusing and abstract, but through extensive research and implementation I have gained an excellent grasp of their use and potential. Indeed this subject will likely stay very close to me in years to come as ray marching with distance functions in particular, is highly satisfying and enjoyable to work with, and the pursuit of perfecting the techniques is something that will likely take time and experience.

Due to the later introduction of ray marching to the project; I produced four prototypes from scratch and as a result undertook considerable extra work. Their development took a large amount of time, having to simultaneously grasp the many principles involved not only in ray tracing and ray marching but CUDA as well, none of which I had any experience with prior to this project. CUDA though powerful was very cumbersome and in ways limiting at times until I became more familiar with it. Again, CUDA had a steep learning curve to understand the principles behind its parallel computation as well as its terminology and syntax.

Overall I view the project as very successful and it has surpassed what I initially envisaged would be achieved, particular the quality of the renderings. At the time of writing this report, the GPGPU and ray marching subjects are very much at the forefront of graphics industry research, with far fewer learning material available when compared to pipelined based rendering. As a result, much of my research consisted of academic papers and so consequently was a difficult topic to pursue, especially at undergraduate level. I would say this has increased my sense of accomplishment further.

**Refer to Appendix K, for renderings produced during this project.**

# 8 References

Appel, A., 1968. *Some techniques for shading machine renderings of solids.* [Online]
Available at: http://graphics.stanford.edu/courses/Appel.pdf
[Accessed 17 10 2013].
Barros, K., 2009. *Massively Parallel Computing with Processors and CUDA.* [Online]
Available at: http://physics.bu.edu/~kbarros/talks/bu_30_1_09/bu_cuda_09.pdf
[Accessed 19 January 2014].
Cooksey, C. & Bourke, P., 1994. *Antialiasing and Raytracing.* [Online]
Available at: http://paulbourke.net/miscellaneous/aliasing/
[Accessed 19 January 2014].
Cutler, B., 2009. *The Traditional Graphics Pipeline.* [Online]
Available at:
http://www.cs.rpi.edu/~cutler/classes/advancedgraphics/S09/lectures/15_Graphics_Pipeline.pdf
[Accessed 19 01 2014].
Disney, 2012. *Brave.* [Online]
Available at: http://www.disneyme.com/brave/images/forest.jpg
[Accessed 25 April 2014].
Etheredge, C. E. & Meteer, O., 2010. *Recent developments in ray tracing for video games.* [Online]
Available at: https://www.inter-actief.utwente.nl/studiereis/pixel/files/indepth/EtheredgeMeteer.pdf
[Accessed 23 October 2013].
Geigel, J., 2004. *So You Want to Write a Ray Tracer?.* [Online]
Available at: http://www.cs.rit.edu/~jmg/courses/cgII/20041/slides/raytrace-assn-3.pdf
[Accessed 19 January 2014].
Hagedorn, H., 2013. *EVGA GeForce GTX 770 SC review - Graphics architecture.* [Online]
Available at:
http://www.guru3d.com/articles_pages/evga_geforce_gtx_770_sc_review,4.html
[Accessed 19 January 2014].
Harris, M., 2013. *Using Shared Memory in CUDA C/C++.* [Online]
Available at: http://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/
[Accessed 25 April 2014].
Hart, J. C., 1994. *Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces.* [Online]
Available at: http://mathinfo.univ-reims.fr/IMG/pdf/hart94sphere.pdf
[Accessed 25 April 2014].
Hart, J. C., Sandin, D. J. & Kauffman, L. H., 1989. *Ray Tracing Deterministic 3-D Fractals.* [Online]
Available at: http://graphics.cs.illinois.edu/sites/graphics.dev.engr.illinois.edu/files/rtqjs.pdf
[Accessed 18 April 2014].
Hart, J. C., Sandin, D. J. & Kauffman, L. H., 1989. *Ray Tracing Deterministic 3-D Fractals.* [Online]
Available at: http://graphics.cs.illinois.edu/sites/graphics.dev.engr.illinois.edu/files/rtqjs.pdf
[Accessed 25 April 2014].
Hoberock, J., Crane, K. & Hart, J. C., n.d. *Fast GPU Ray Tracing of Dynamic MEshes using Geometry Images.* [Online]
Available at:
http://www.cs.columbia.edu/~keenan/Projects/RayTracingGeometryImages/paper.pdf
[Accessed 19 January 2014].
Hock-Chua, C., 2012. *3D Graphics with OpenGL: Basic Theory.* [Online]
Available at:
http://www.ntu.edu.sg/home/ehchua/programming/opengl/CG_BasicsTheory.html
[Accessed 25 April 2014].

Howard, J., 2007. *Real Time Ray-Tracing: The End of Rasterization?.* [Online]
Available at: https://blogs.intel.com/intellabs/2007/10/10/real_time_raytracing_the_end_o/
[Accessed 19 January 2014].
Ingram, S., 2010. *CS 563 Advanced Topics in Computer Graphics: Triangle Meshes.* [Online]
Available at:
http://web.cs.wpi.edu/~emmanuel/courses/cs563/S10/talks/wk9_p2_scott_Tri_Meshes.pdf
[Accessed 25 April 2014].
Jeong, B. & Abram, G., n.d. *8 Things You Should Know About GPGPU Technology - Q&A with TACC Research Scientists.* [Online]
Available at: http://www.tacc.utexas.edu/documents/13601/88790/8Things.pdf
[Accessed 18 10 2013].
Li, D. Q., 2013. *3D Shape Modelling Using Polygonal Meshes,* s.l.: The University of Hull.
Li, Q., 2007. *Smooth Piecewise Polynomial Blending Operations for Implicit Shapes.* [Online]
Available at: http://hyperfun.org/FHF_Log/LiQ_PolyBlend_CGF07.pdf
[Accessed 30 April 2014].
Marschner, S., 2003. *CS465 Notes: Simple Ray-Triangle Intersection - Cornell University.* [Online]
Available at: http://www.cs.cornell.edu/courses/cs465/2003fa/homeworks/raytri.pdf
[Accessed 25 April 2014].
Nvidia Corporation, 2013. *CUDA C Programming Guide.* [Online]
Available at: http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[Accessed 25 October 2013].
Nvidia Corporation, 2014. *CUDA Home Page.* [Online]
Available at: http://www.nvidia.com/object/cuda_home_new.html
[Accessed 19 January 2014].
Nvidia Corporation, 2014. *NVIDIA CUDA Compiler Driver NVCC.* [Online]
Available at: http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/#axzz2zcsH1z4B
[Accessed 25 April 2014].
Olano, M., 2002. *UMBC - CMSC 435/634: Introduction to Computer Graphics.* [Online]
Available at: http://www.csee.umbc.edu/~olano/435f02/ray-sphere.html
[Accessed 15 October 2013].
OpenGL.org, 2013. *Vertex Specification.* [Online]
Available at: https://www.opengl.org/wiki/Vertex_Specification
[Accessed 25 April 2014].
Pacheco, H., 2008. *Ray Tracing in Industry: An up-to-date review of industrial ray tracing applications.* [Online]
Available at: http://paginas.fe.up.pt/~aas/pub/Aulas/DiCG/HugoPacheco.pdf
[Accessed 23 October 2013].
Pacheco, H., 2008. *Ray Tracing in Industry: An up-to-date review of industrial ray tracing applications.* [Online]
Available at: http://paginas.fe.up.pt/~aas/pub/Aulas/DiCG/HugoPacheco.pdf
[Accessed 23 October 2013].
Phong, B. T., 1975. *Illumination for Computer Generated Pictures.* [Online]
Available at: http://www.cs.northwestern.edu/~ago820/cs395/Papers/Phong_1975.pdf
[Accessed 19 January 2014].
Pollard, N., 2004. *Ray Tracing.* [Online]
Available at: http://graphics.cs.cmu.edu/nsp/course/15-462/Spring04/slides/13-ray.pdf
[Accessed 19 January 2014].
Quilez, I., 2008. *Rendering Worlds with Two Triangles with raytracing on the GPU in 4096 bytes.* [Online]
Available at: http://www.iquilezles.org/www/material/nvscene2008/rwwtt.pdf
[Accessed 25 April 2014].

Quilez, I., 2010. *Free penumbra shadows for raymarching distance fields.* [Online]
Available at: http://www.iquilezles.org/www/articles/rmshadows/rmshadows.htm
[Accessed 25 April 2014].
Quilez, I., 2013. *Smooth Minimum.* [Online]
Available at: http://www.iquilezles.org/www/articles/smin/smin.htm
[Accessed 25 April 2014].
Reiner, T., Muckl, G. & Dachsbacher, C., 2011. *Interactive Modeling of Implicit Surfaces using a Direct Visualization Approach.* [Online]
Available at: http://cg.ivd.kit.edu/publications/p2011/IntModelingSDF/IntModelingSDF.pdf
[Accessed 25 April 2014].
Robertson, B., 2012. *The Royal Treatment - Computer Graphics World.* [Online]
Available at: http://www.cgw.com/Publications/CGW/2012/Volume-35-Issue-4-June-July-2012/The-Royal-Treatment.aspx
[Accessed 25 April 2014].
Sanders, J. & Kandrot, E., 2010. *CUDA By Example: An Introduction to General-purpose GPU Programming.* [Online]
Available at: http://developer.download.nvidia.com/books/cuda-by-example/cuda-by-example-sample.pdf
[Accessed 25 April 2014].
Scratchapixel, 2012. *Building Primary Rays and Rendering an Image.* [Online]
Available at: http://www.scratchapixel.com/lessons/3d-basic-lessons/lesson-6-rays-cameras-and-images/building-primary-rays-and-rendering-an-image/
[Accessed 28 October 2013].
Stratton, J., 2008. *State of Ray Tracing (in Games).* [Online]
Available at: http://www.cs.utah.edu/~jstratto/state_of_ray_tracing/
[Accessed 19 01 2014].
Suffern, K. G., 2007. *Ray Tracing from the Ground Up.* Massachusetts: A K Peters Ltd.
Trader, T., 2013. *GPUs Power One-Third of Top Russian Supercomputers.* [Online]
Available at: http://www.hpcwire.com/hpcwire/2013-10-02/gpus_power_one-third_of_top_russian_supercomputers.html
[Accessed 18 10 2013].
Tsiodras, T., 2011. *Renderer 2.x - Porting to CUDA.* [Online]
Available at: http://users.softlab.ntua.gr/~ttsiod/cudarenderer-BVH.html
[Accessed 19 January 2014].
Whitted, T., 1979. *An Improved Illumination Model for Shaded Display.* [Online]
Available at: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.156.1534
[Accessed 17 10 2013].
Wikipedia, 2014. *Supersampling.* [Online]
Available at: http://en.wikipedia.org/wiki/Supersampling
[Accessed 19 January 2014].
Wilt, N., 2012. *Streaming Multiprocessors.* [Online]
Available at:
http://www.cudahandbook.com/uploads/Chapter_8._Streaming_Multiprocessors.pdf
[Accessed 19 January 2014].
Wilt, N., 2012. *The CUDA Handbook.* s.l.:Pearson Education.
Wolfe, M., 2010. *Understanding the CUDA Data Parallel Threading Model.* [Online]
Available at: http://www.pgroup.com/lit/articles/insider/v2n1a5.htm
[Accessed 19 January 2014].
Zachmann, G., 2013. *Thread Layouts for 2D Computational Problems.* [Online]
Available at: http://cgvr.cs.uni-bremen.de/teaching/mpar_13/folien/02%20-%20Intro%20to%20CUDA-2.pdf
[Accessed 25 April 2014].
Zahran, M., 2012. *Graphics Processing Units (GPUs): Architecture and Programming - Lecture 6: CUDA Memories.* [Online]

Available at: http://cs.nyu.edu/courses/spring12/CSCI-GA.3033-012/lecture6.pdf
[Accessed 19 January 2014].

# 9 Appendices

## 9.1 Appendix A - A Ray Tracing Analogy

In nature, the human eye perceives light from sources like the Sun. When light hits a surface, some light is reflected off into the eye. It is from this reflected light, that the eye perceives the world. Ray tracing simulates this process to create realistically illuminated environments. In nature however, there are an infinite number of light rays; performing countless light ray calculations on a computer would be impossible and inefficient, since only a tiny percentage of light rays ever actually enter an observers eye. For this reason, ray tracing casts the light backwards from the eye instead of from the light source. Upon light hitting an object, another secondary ray is then fired towards the light source. If this secondary ray hits another object, then the original object must therefore be in shadow. It is through this process, operated on for each pixel, that an images colours are determined.

## 9.2 Appendix B - The Cameras Facing Direction

Calculating the $\boldsymbol{w}$ axis (facing direction) of the camera:

$$\boldsymbol{w} = \boldsymbol{l} - \boldsymbol{c}$$

Where $\boldsymbol{l}$ is a defined 'look at' position, $\boldsymbol{c}$ is the camera position.

## 9.3 Appendix C - Pixel Space Conversion

The conversion process of pixel raster coordinates to world space for calculation of ray directions to be sent into the scene moves through the following spaces:

$$\textbf{Raster} \rightarrow \textbf{NDC} \rightarrow \textbf{Camera} \rightarrow \textbf{World}$$

## 9.4 Appendix D - Parallel Ray-Plane

It can be determined that no intersection has occurred if a ray is parallel to a plane by considering the dot product of the planes normal and the rays direction. Since a planes normal is always perpendicular to a parallel ray, if the result is zero there is no intersection.

## 9.5 Appendix E - Ray-Sphere Intersection Roots

Depending on the value of the quadratic formula's discriminant there is potential for zero, one or two roots; if the discriminant is negative the ray has missed the sphere, if it is equal to one then the ray has intersected the sphere tangentially and if positive the ray has intersected twice, upon entering and leaving the sphere.

## 9.6 Appendix F - Risk Analysis Table

| Risk | Description | Action | Chance | Severity |
|---|---|---|---|---|
| Hardware failure | Catastrophic failure of computer hardware used for developing the project, including data loss of source code and documentation. | Backups to SVN repository, internet cloud storage and physical media | Low | Low |
| Complexity | Potential complex theory, mathematics or GPGPU theory could disrupt implementation if insufficient or incorrect research is performed. | Extensive research from multiple sources (electronic/books, academic material). Regular meetings with project supervisor. | Low | Medium |
| Time | Incorrect planning of tasks or overestimation of available time. Inefficient working pattern. | Production of project time plan, kept updated. Allocation of regular days for project work. Working in a suitable environment with no distractions. | Low | Medium |

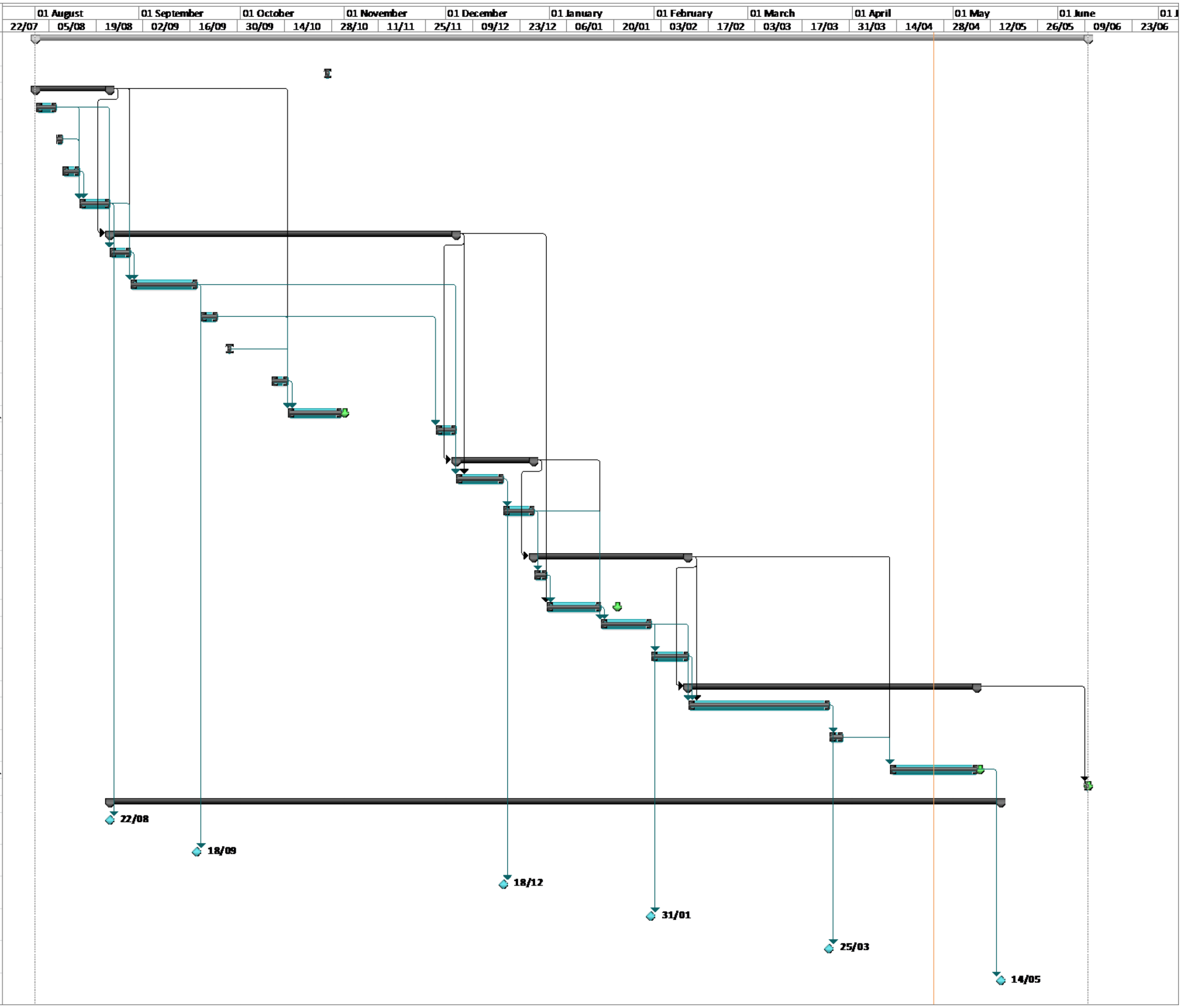
## 9.7 Appendix G - Project Task List

| ID | Task / Phase | Description | Milestone | Overall Phase |
|---|---|---|---|---|
| **3** | Ray Tracing Technical Research: Phase 1 | Research into basic principles of ray tracing theory and mathematics. Specifically, camera, image plane and intersections. | | 1 |
| **4** | Development Technology Research | Researching appropriate tools, software and technology to create a ray tracer e.g. IDE, language. | | 1 |
| **5** | Implementation Technology Research | Investigate the appropriate type of ray tracer to develop e.g. build from scratch or use OpenGL, CPU-based or Shader-based program. | | 1 |
| **6** | Ray Tracer Prototypes: Phase 1 | Program that can render colours to a Bitmap file, send rays into scene and intersect with a sphere and plane. | Y | 1 |
| **8** | Ray Tracing Technical Research: Phase 2 | Research into intermediate principles of ray tracing. Specifically, secondary rays (reflections and shadows), light sources (point , directional), materials and anti-aliasing. | | 2 |
| **9** | Ray Tracer Prototypes: Phase 2 | Program can render a scene to a bitmap with realistic lighting, reflections, shadows and multiple lights. Anti-aliasing implemented. Objects have different material properties. | Y | 2 |
| **10** | GPGPU Technical Research: Phase 1 | Basic Research into theory behind parallel computing and the hardware inside modern GPUs. | | 2 |
| **11** | GPGPU Applications Research | Investigating the background of GPGPU and client applications. Studying uses and benefits and future applications. | | 2 |
| **12** | Initial Report Writing Research | Research techniques, structure and contents of writing the initial report. | | 2 |
| **13** | Initial Report | Production of initial report for ongoing project | | 2 |
| **14** | GPGPU Technical Research: Phase 2 | Research how to implement prototype using CUDA. Understand how to develop a CUDA program. | | 2 |
| **16** | Ray Tracer Prototypes: Phase 3 | Create a second ray tracer prototype using CUDA. Compare and analyse rendering speeds versus the | Y | 3 |

| | | | | |
|---|---|---|---|---|
| | | CPU ray tracer. | | |
| **17** | Ray Marching Technical Research: Phase 1 | Research Ray marching techniques and distance functions and how it can be used to improve ray tracing. | | 3 |
| **19** | Interim Report Writing Research | Research techniques, structure and contents of writing the interim report. | | 4 |
| **20** | Interim Report | Production of Interim report for ongoing project | | 4 |
| **21** | Ray Marching Prototypes: Phase 1 | Develop a CPU ray marcher prototype and implement refraction for rendering of transparent surfaces. | Y | 4 |
| **22** | Algorithm Optimisation Research | Research and analyse different algorithms and enhance render times for both CUDA and CPU prototypes. | | 4 |
| **24** | Ray Marching Prototypes: Phase 2 | Implement a CUDA ray marcher prototype. | Y | 5 |
| **25** | Final Report Writing Research | Research techniques, structure and contents of writing the final report. | | 5 |
| **26** | Final Report | Production of final report, concluding the project. | Y | 6 |
| **27** | Prototype Demonstration | Present finished prototype. | | 6 |

## 9.8 Appendix H - Project Time Plan

| ID | ⓘ | ID | Task Name | Duration | Start | Finish | Deadline |
|---|---|---|---|---|---|---|---|
| 0 | ✓ | **0** | **Time Plane** | **223 days** | **Thu 01/08/13** | **Mon 09/06/14** | **NA** |
| 1 | ✓ | 1 | | 1 day | Sat 26/10/13 | Sat 26/10/13 | NA |
| 2 | ✓ | **2** | **Phase 1** | **16 days** | **Thu 01/08/13** | **Thu 22/08/13** | **NA** |
| 3 | ✓ | 3 | Ray Tracing Technical Research: Phase 1 | 4 days | Thu 01/08/13 | Tue 06/08/13 | NA |
| 4 | ✓ | 4 | Development Technology Research | 2 days | Wed 07/08/13 | Thu 08/08/13 | NA |
| 5 | ✓ | 5 | Implementation Technology Research | 3 days | Fri 09/08/13 | Tue 13/08/13 | NA |
| 6 | ✓ | 6 | Ray Tracer Prototypes: Phase 1 | 7 days | Wed 14/08/13 | Thu 22/08/13 | NA |
| 7 | ✓ | **7** | **Phase 2** | **73 days** | **Fri 23/08/13** | **Tue 03/12/13** | **NA** |
| 8 | ✓ | 8 | Ray Tracing Technical Research: Phase 2 | 4 days | Fri 23/08/13 | Wed 28/08/13 | NA |
| 9 | ✓ | 9 | Ray Tracer Prototypes: Phase 2 | 14 days | Thu 29/08/13 | Tue 17/09/13 | NA |
| 10 | ✓ | 10 | GPGPU Technical Research: Phase 1 | 3 days | Thu 19/09/13 | Mon 23/09/13 | NA |
| 11 | ✓ | 11 | GPGPU Applications Research | 1 day | Fri 27/09/13 | Fri 27/09/13 | NA |
| 12 | ✓ | 12 | Initial Report Writing Research | 3 days | Thu 10/10/13 | Mon 14/10/13 | NA |
| 13 | ✓ | 13 | Initial Report | 12 days | Tue 15/10/13 | Wed 30/10/13 | Thu 31/10/... |
| 14 | ✓ | 14 | GPGPU Technical Research: Phase 2 | 4 days | Thu 28/11/13 | Tue 03/12/13 | NA |
| 15 | ✓ | **15** | **Phase 3** | **17 days** | **Wed 04/12/1:** | **Thu 26/12/13** | **NA** |
| 16 | ✓ | 16 | Ray Tracer Prototypes: Phase 3 | 10 days | Wed 04/12/13 | Tue 17/12/13 | NA |
| 17 | ✓ | 17 | Ray Marching Technical Research: Phase 1 | 7 days | Wed 18/12/13 | Thu 26/12/13 | NA |
| 18 | ✓ | **18** | **Phase 4** | **32 days** | **Fri 27/12/13** | **Mon 10/02/14** | **NA** |
| 19 | ✓ | 19 | Interim Report Writing Research | 2 days | Fri 27/12/13 | Mon 30/12/13 | NA |
| 20 | ✓ | 20 | Interim Report | 12 days | Tue 31/12/13 | Wed 15/01/14 | Mon 20/01... |
| 21 | ✓ | 21 | Ray Marching Prototypes: Phase 1 | 11 days | Thu 16/01/14 | Thu 30/01/14 | NA |
| 22 | ✓ | 22 | Algorithm Optimisation | 7 days | Fri 31/01/14 | Mon 10/02/14 | NA |
| 23 | ✓ | **23** | **Phase 5** | **62 days** | **Tue 11/02/14** | **Wed 07/05/14** | **NA** |
| 24 | ✓ | 24 | Ray Marching Prototypes: Phase 2 | 30 days | Tue 11/02/14 | Mon 24/03/14 | NA |
| 25 | ✓ | 25 | Final Report Writing Research | 4 days | Tue 25/03/14 | Fri 28/03/14 | NA |
| 26 | ✓ | 26 | Final Report | 19 days | Sat 12/04/14 | Wed 07/05/14 | Thu 08/05/... |
| 27 | ✓ | 27 | Demonstration | 1 day | Mon 09/06/14 | Mon 09/06/14 | Mon 09/06... |
| 28 | ✓ | **28** | **Milestones** | **189 days** | **Thu 22/08/13** | **Wed 14/05/14** | **NA** |
| 29 | ✓ | 29 | Milestone: Prototype Phase 1 | 0 days | Thu 22/08/13 | Thu 22/08/13 | NA |
| 30 | ✓ | 30 | Milestone: Prototype Phase 2 | 0 days | Wed 18/09/13 | Wed 18/09/13 | NA |
| 31 | ✓ | 31 | Milestone: Prototype Phase 3 | 0 days | Wed 18/12/13 | Wed 18/12/13 | NA |
| 32 | ✓ | 32 | Milestone: Prototype Phase 4 | 0 days | Fri 31/01/14 | Fri 31/01/14 | NA |
| 33 | ✓ | 33 | Milestone: Prototype Phase 5 | 0 days | Tue 25/03/14 | Tue 25/03/14 | NA |
| 34 | ✓ | 34 | Milestone: Final Report | 5 days | Thu 08/05/14 | Wed 14/05/14 | NA |



## 9.9 Appendix I - Hardware Specifications

The prototypes have been developed on a Dell XPS17 laptop with the following specification:

- Intel i7-2630QM 2.00GHz CPU
- 6GB DDR3 (1333MHz) RAM
- NVIDIA GeForce GT555m GPU

The CUDA capabilities for the 'GeForce GT555m' GPU used during development and for benchmarking are as follows:

| | |
|---|---|
| CUDA driver version | 6.0 |
| Compute capability | 2.1 |
| Total multiprocessors (SM's) | 3 |
| CUDA Cores per multiprocessor | 48 |
| Warp Size (threads per unit) | 32 |
| Max threads per multiprocessor | 1536 |
| Max threads per block | 1024 |
| Max dimension size of a thread block | (1024,1024,64) |
| Max dimension size of a grid | (65535,65535,65535) |

## 9.10 Appendix J - Prototype Performance Testing

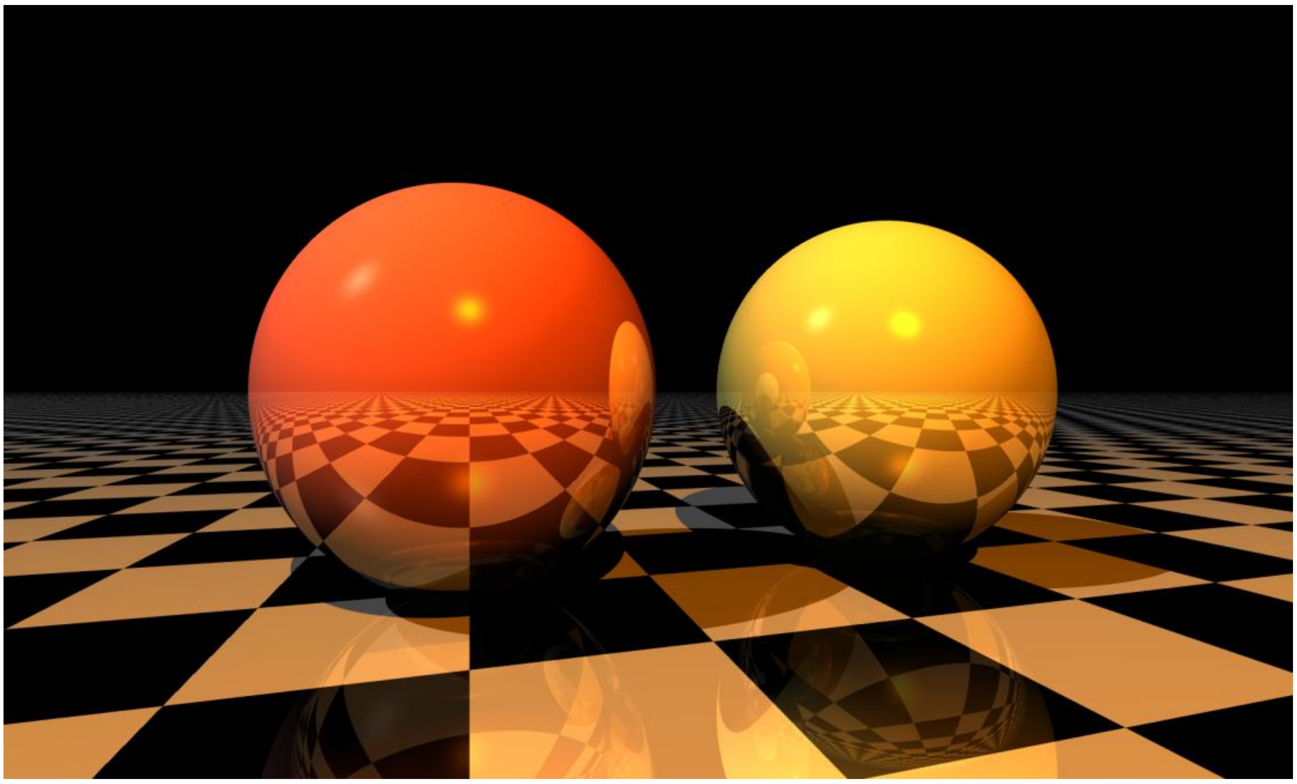
The rendering times were calculated using "clock_t" from the 'C' library, a measurement of CPU clocks representing the overall process running time. For the CUDA prototypes, a manual device synchronisation call was made after kernel execution to ensure the time was accurate to the last thread finishing. Additionally for the CUDA prototypes  have included the time taken to copy the data back from the device to the host. For all prototypes, the time taken to generate the bitmap image has been excluded.

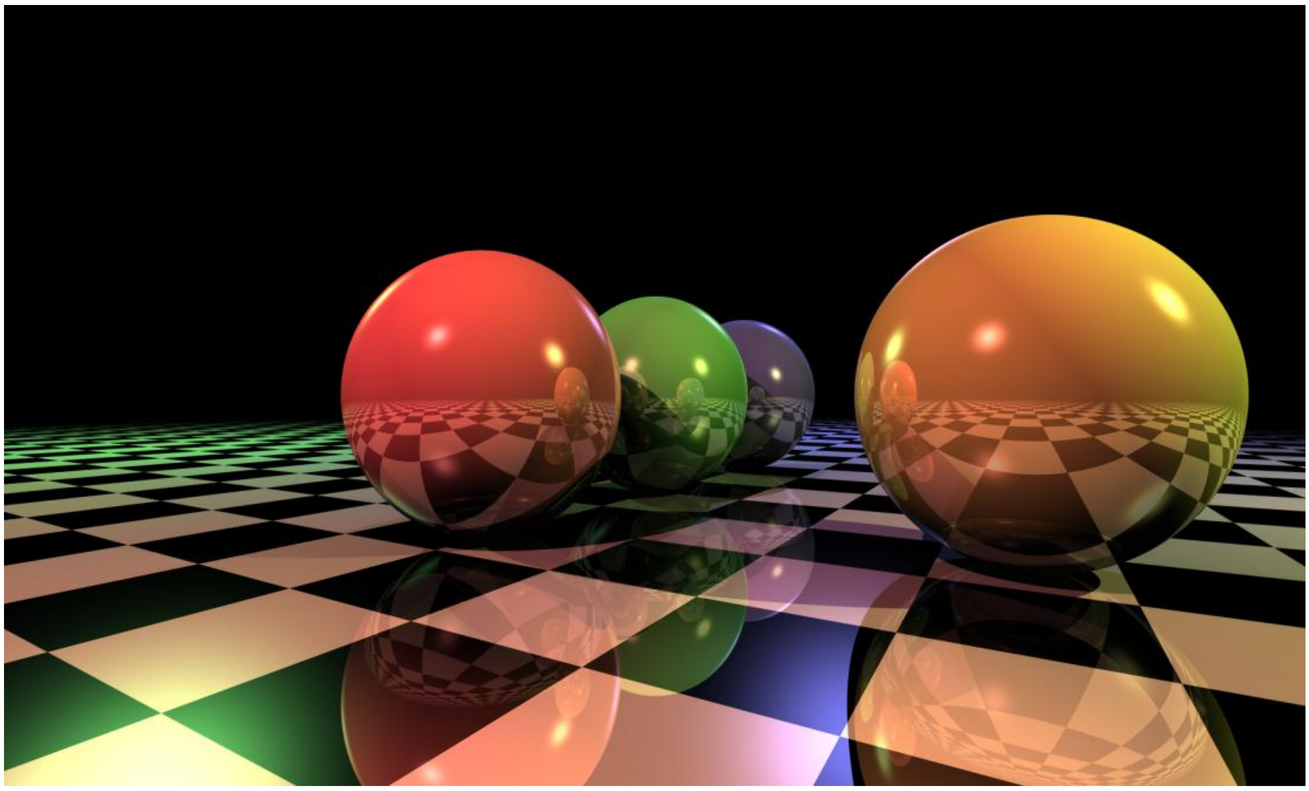
For the ray tracer prototypes, two scenes (1 and 2 from Appendix K) were used for benchmarking, representing a simple and more complex scene respectively. Both scenes feature reflections and shadows. The simple scene features a point light, a directional light, two spheres and a plane. The more complex scene features four point lights, 4 spheres and a plane.

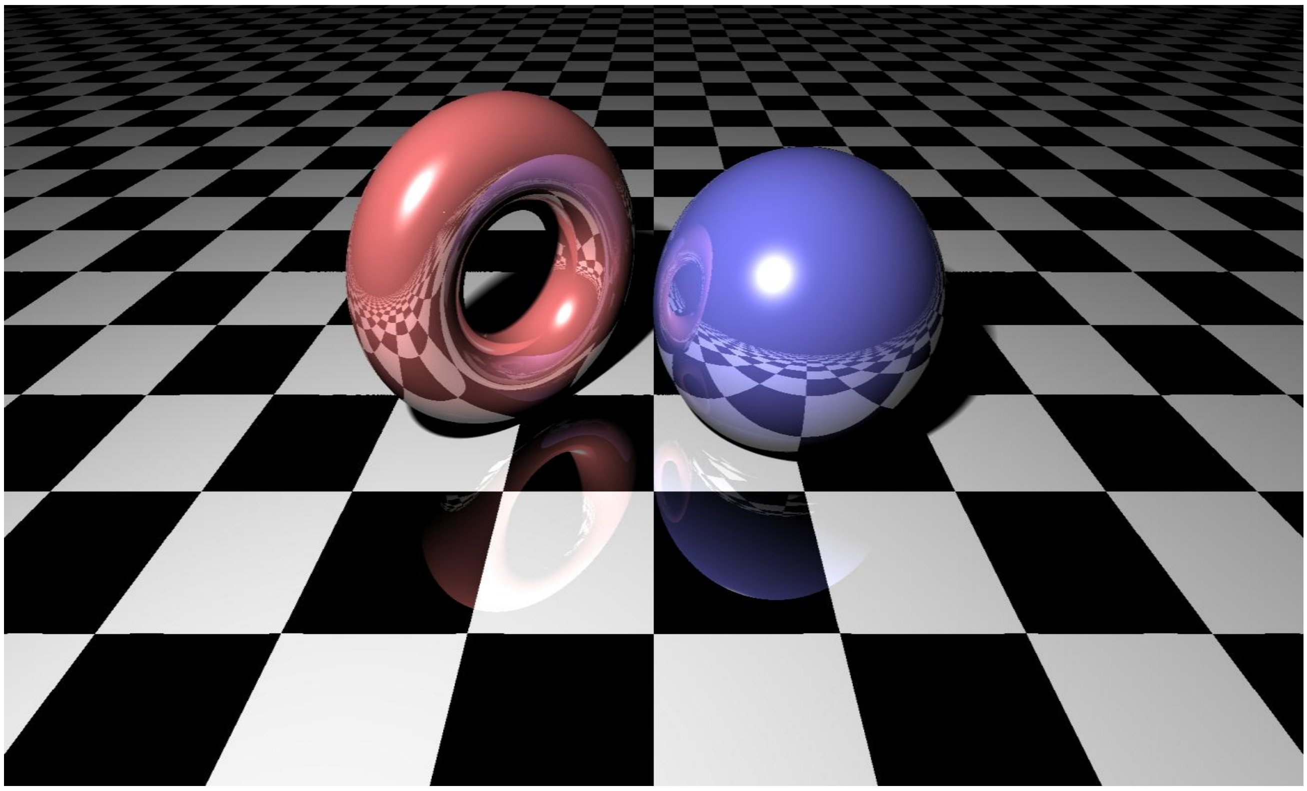
For the ray marcher prototypes, no anti-aliasing tests were carried out due having already being observed to have significant impact of the render times and deeming this to not be the focus for testing, and noting that the soft penumbra shadow implementations in the ray marchers effectively anti-aliased parts of the image anyway. To compensate for the lack of test variety, more scenes have been included (3, 4, 5 and 6 from Appendix K) to showcase ray marching capability.

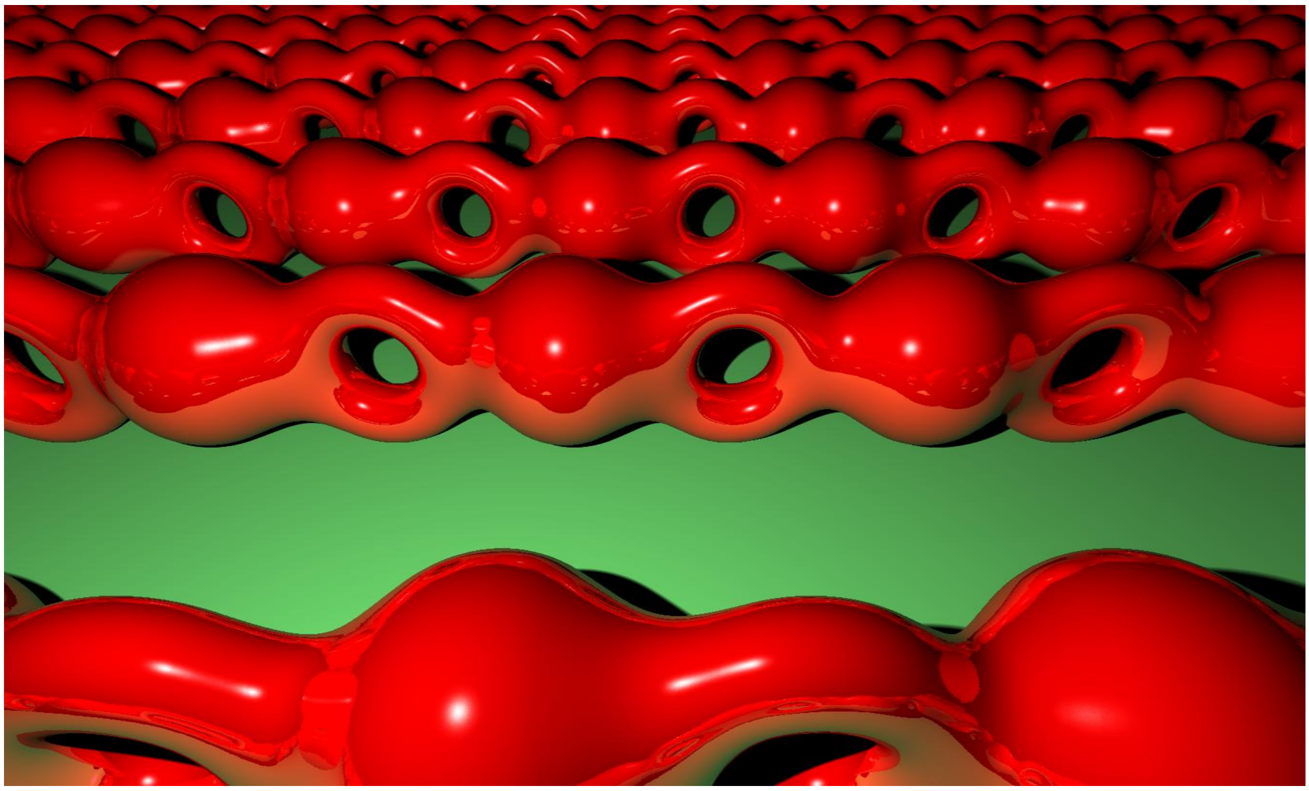
Scene 3 features a simple torus and sphere on a plane. It features full reflections, a point light and soft shadows. Scene 4 uses a combination of distance fields operations including soft minimum blend and domain repetition on just a single instance of a torus and sphere, featuring reflections and soft shadows. Scene 5 is a scene utilising ray refraction, reflections and soft shadows. The solid glass columns use domain repetition. Scene 6 is a rendering of a Mandelbulb fractal and plane with soft shadows.

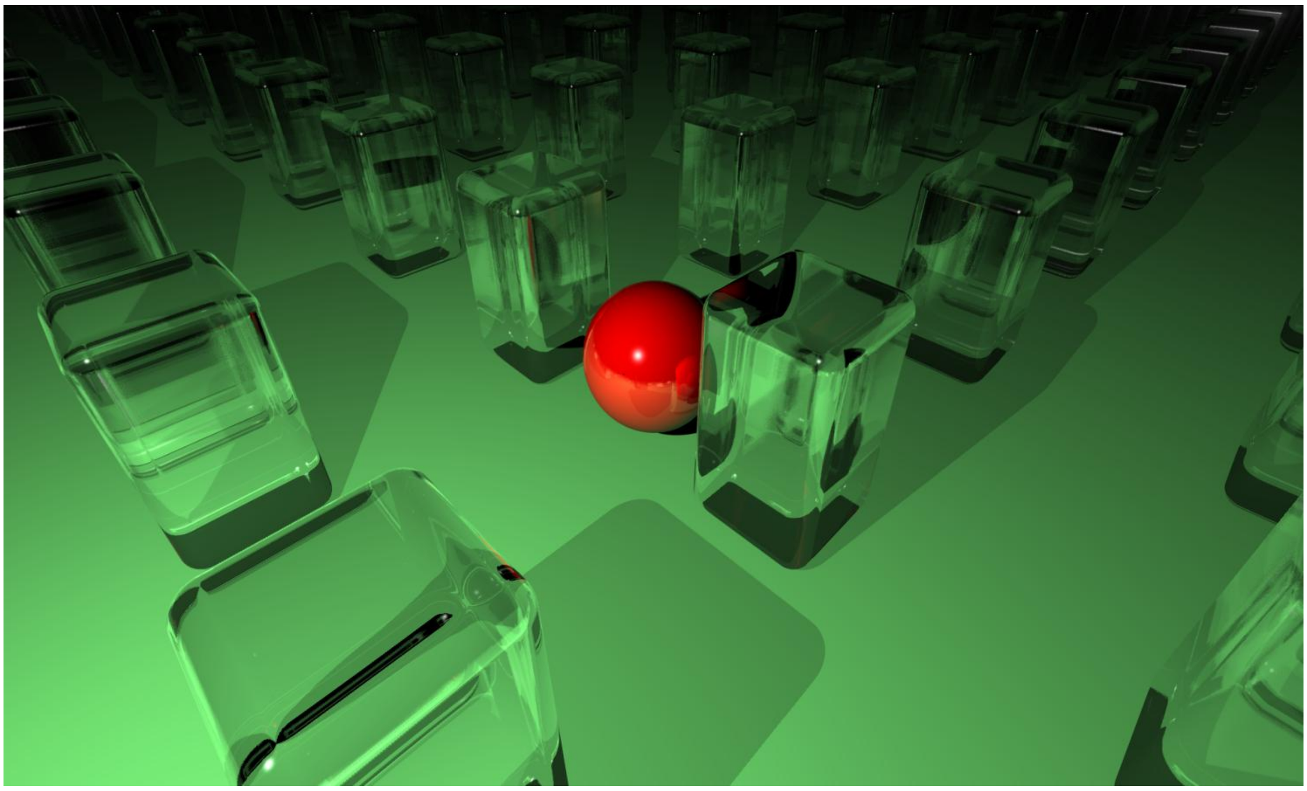
## 9.11 Appendix K - Prototype Renderings

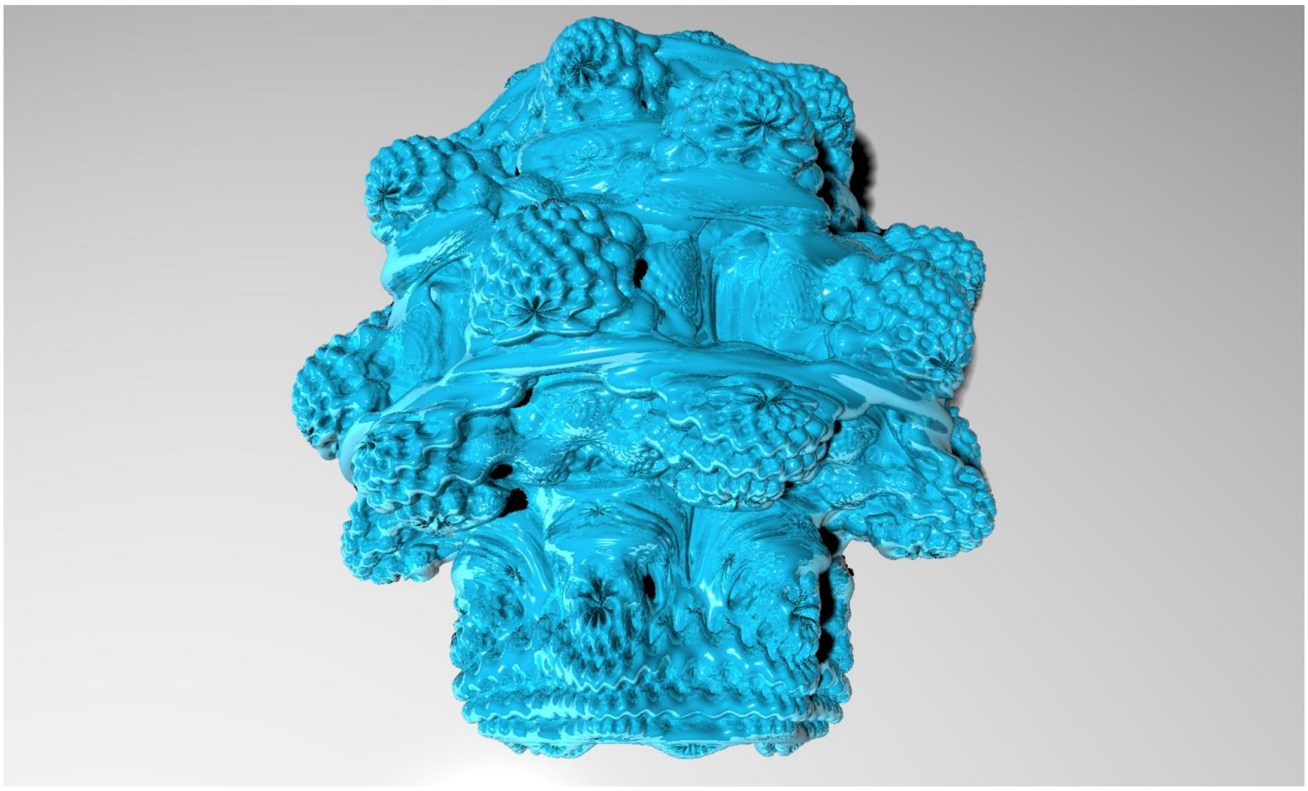

**Scene 1 ray traced with 2 lights, 2 spheres, 8xAA.**

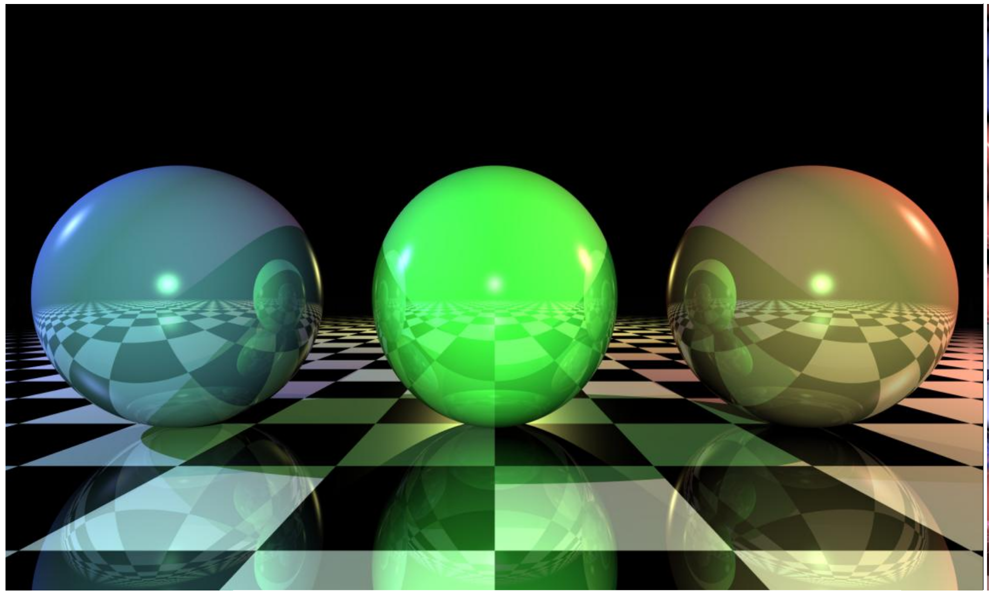

**Scene 2 ray traced with 4 lights, 4 spheres, 8xAA.**


**Scene 3 ray marched with torus and sphere, soft shadows.**

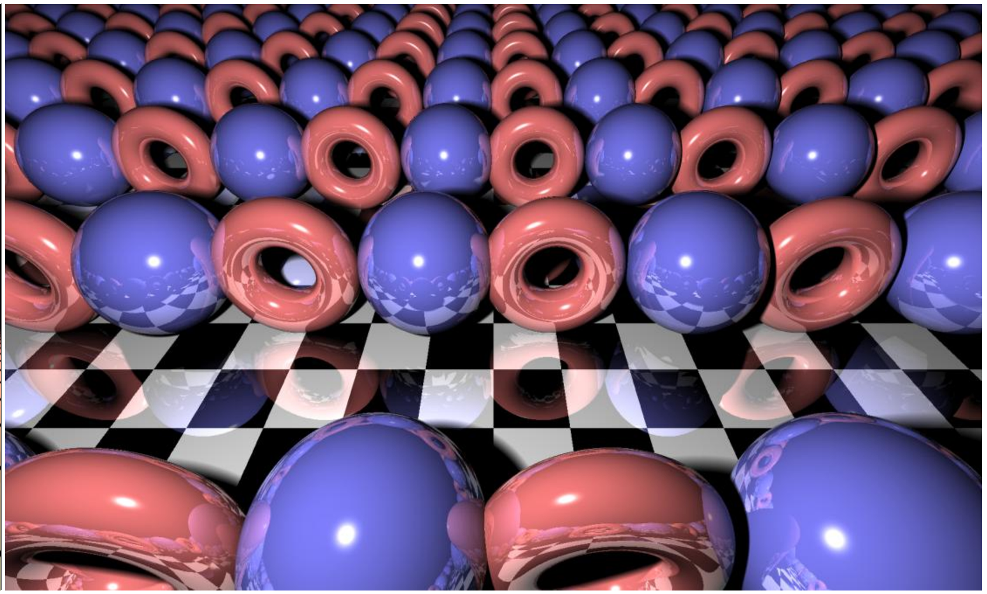

**Scene 4 ray marched with torus and sphere, smooth blended and repeated with soft shadows.**


**Scene 5 ray marched with rounded solid glass cubes repeated , with a single sphere.**

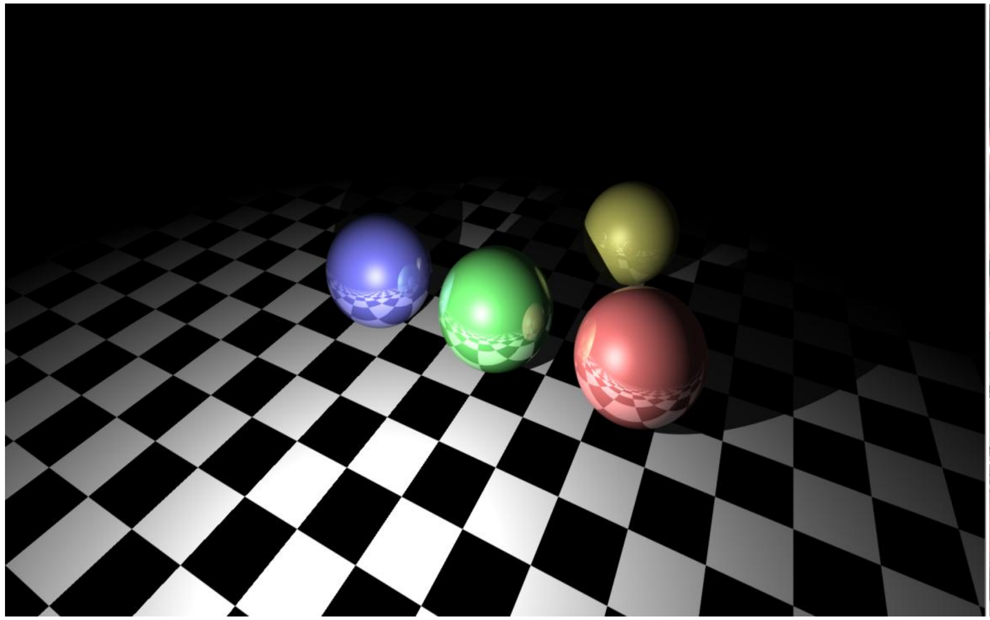

**Scene 6 ray marched with Mandelbulb fractal distance function.**



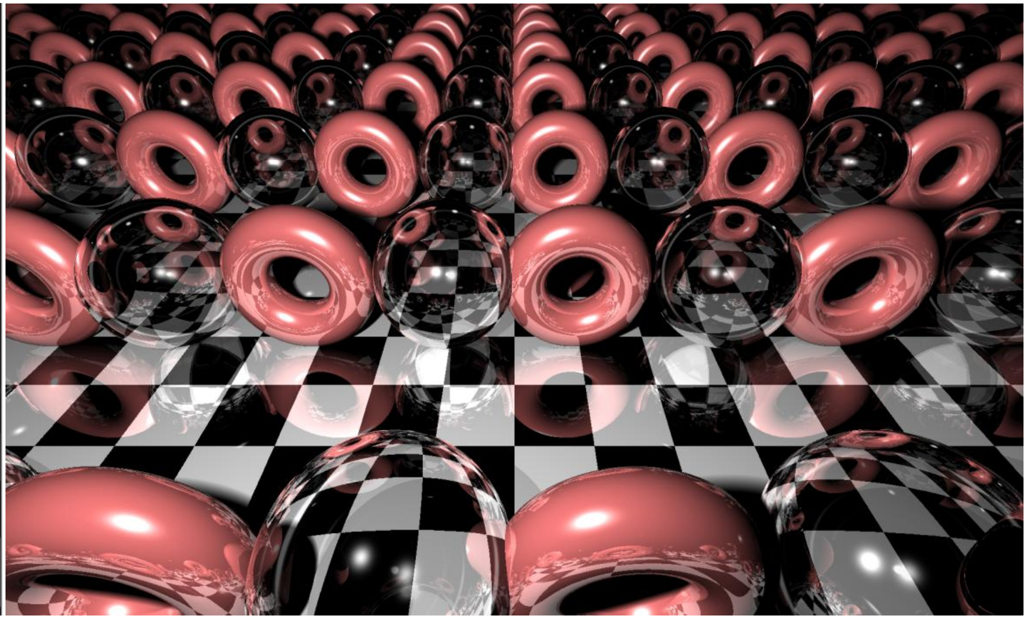
A ray traced scene  with 3 lights, 3 spheres, 8xAA



A ray marched scene with domain repetition, torus and spheres.



A ray traced scene with 1 low light, 4 spheres, 8xAA



A ray marched scene with domain repetition, refracted spheres and torus.