Real-Time Ray Tracing

registration: 4759265

(Supervisor: Dr Stephen Laycock)

Abstract

As graphics accelerated cards improve and processors become increasingly multi-core, parallel implementations become popular. Both Intel and NVidia have recently investigated the potential of ‘Real-Time Ray Tracing’ on their latest architectures. This project focuses on the design and implementation of a ‘Ray Tracer’ capable of producing computer graphics images, incorporating lighting effects such as shadows and specular highlights. This project strives towards real-time performance on consumer level hardware.

Acknowledgements

This project would not be possible without the pioneering efforts of Whitted (1980) who developed the ray tracing technique. Help and guidance has been provided by the project advisor, Dr Stephen Laycock. Work by the Khronos Group who developed the OpenCL specification, has also helped to make a GPU implementation possible.

Contents

1 Introduction

1.1 Background and Motivation

Ray tracing is a technique that simulates the physics of light to render computer generated images capable of a high degree of visual realism. The Ray tracing technique has been around for many years, it was first proposed by Whitted (1980). Despite this ray tracing has taken a back seat, with massive research gone into classic rasterised 3D graphics, which has become the norm to computer rendering. Ray traced images are achieved by tracing the path of light through pixels in an image plane, but this form of rendering requires a much greater computational cost than traditional raster graphics, therefore limiting its applications of use. It has however become the main rendering technique used in most high-end 3D modelling and animation systems when creating still images, but until now real-time applications of ray tracing has yet to become possible, with the exception for expensive large industrial hardware. This means now with increasing hardware capacities, more research has started to be put into various forms of ray tracing. The improvements of graphics accelerated cards and processors becoming increasingly multi-core, means that real-time ray tracing is starting to become possible through the use of parallel implementations. This is especially useful on GPUs (Graphics Processing Unit Hardware) which contain hundreds or thousands of hardware cores that promises great opportunities to achieve what would be considered impossible in the past. Major companies such as Intel and NVidia have recently investigated the potential of real-time ray tracing on their latest hardware architectures. Ray Tracing is also very useful as there are many other algorithms, which expand upon the basic ray tracing technique to further generate photo-realistic results. This field of graphics is a hugely interesting field due to the potential of this emerging graphical technique that has been around for decades but only until now has it been possible to fully be realised.

1.2 Aims

This project focuses on the design and implementation of a ray tracer capable of producing computer graphics images. The ray tracer should incorporate a range of optical effects such as shadows, reflections and specular highlights. The project will strive towards real-time performance on consumer level hardware through optimisations and parallelisation. The final ray tracer will demonstrate its capabilities modelling a scene that exhibits the functions of the ray tracer, for example scenes exhibiting reflections and refractions in glass objects.

1.3 Areas of knowledge required

The ‘Ray Tracer’ is a process that creates images by simulating how light works in the real world. Unlike the forward ray tracing of the real world, it uses backward ray tracing which involves ray casting from the viewing point into each pixel of the image plane. Each ray is then tested to see if it intersects with objects in the scene. Knowledge and research of rays and ray-object intersections is needed. If a intersection does occur then that pixels colour is obtained by mathematically calculating the amount of light on the surface from the light sources in the scene until eventually a single colour is determined. This requires some knowledge and maths of the physics of light, also with research on the lighting calculating methods of calculating shading, reflections and transparency will be incorporated. To improve on real time performance parallelisation of the system will need to be applied, to greatly reduce the computational time through use of languages such as CUDA or OpenCL which utilise the many cores of today’s GPUs (Graphics Processing Unit Hardware). One of these languages needs to be learnt to be able to effectively code upon the GPU. To produce the system programming knowledge of a compatible programming language such as c++ is required. Additionally research into accelerated data structures such as Kd-trees or regular grids help real-time performance, but these generally these structures require pre-processing time which might be inappropriate for real-time applications.

2 Literature Review (Related Work)

Ray tracing is a process in computer graphics that originated many years ago through the work of Whitted (1980), who implemented basic ray tracing with advanced lighting to generate images of graphical quality. This was based upon the ray casting technique originating from Appel (1968). According to Whitted (1980) these high quality image results took hours to generate due to the high complexity of the intersection and lighting calculations. The basics of ray tracing can be represented through the pinhole camera model, described so that "each pixel is a small independent window onto the scene", (Glassner, 1993). Ray tracing involves ray casting from the viewing point into each of these pixels of the image. These rays are formed by a camera position and direction which are then traced through the two-dimensional array of pixels into a scene of lights and shapes. Each ray is then tested to see if it intersects with objects in the scene. The simplistic overview of the ray trace process is shown in Algorithm 1 and in Figure 2.1 demonstrated in Glassner (1993). Only recently with the increase of computing power has ray tracing started to become more popular again. Due to each pixels ray calculations being independent, ray tracing can benefit from a high degree of parallelism. One method of implementation ray tracing in parallel on a GPU is when "Object space is divided into parts (subspaces), each of which is allocated to a processor." (Kobayashi et al., 1987)

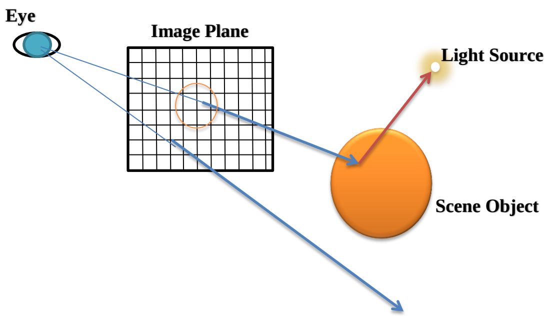

Figure 2.1: Diagram of the basic ray casting approach

Algorithm 1 Ray tracing pseudo-code.

```
for each pixel in $(x)$ do % Loop Screen Pixels Width
    for each pixel in $(y)$ do % Loop Screen Pixels Height
        for each object in $(scene)$ do %
            Find intersections with objects and determine which one is on top.
            if no intersections found then
                Colour background colour and EXIT.
            end if
            if Intersections found then
                Calculate pixel colour with lighting calculations.
            end if
        end for
    end for
end for
Display all pixels in a image.
```

2.1 Object Ray Intersections

Ray tracing rendering is achieved by computed collision detection between rays and objects in the scene. These objects include spheres, planes and polygons described by Amanatides (1984). Rays are cast from the camera, through each pixel of the image, and tested for these ray-object intersection. There is slight differences for checking various objects, but the principles are the same. The more objects in the scene the more calculations will be needed without optimization. Intersections are calculated through these various functions to find values of the ray parameter which objects are at the front and where the lighting needs to be calculated. Ray-triangle intersections is one way to represent polygons shown by Purcell et al. (2002). More intersection calculations can be seen in work by Shirley (2000).

2.2 Illumination & Lighting

To produce highly realistic images ray tracing involves the combination of many lighting techniques together with the material properties of objects, for example surface colour. The most commonly used methods for lighting were pioneered by Blinn and Newell (1976) and Phong (1975) and are still widely used in lighting algorithms of today. To create more natural lighting a combination of Ambient lighting, Diffuse lighting and Specular lighting creates a good starting point. Ambient lighting is the base low lighting when no direct light source is applied which ensures the outlines of shapes are shown. Diffuse reflections produce soft reflections as seen on matte surfaces, while specular reflections are the shiny highlights found on smooth surfaces like mirrors. This lighting model called Phong Illumination lighting was pioneered and named by Phong (1975). It is the basic way to render objects that reflect light in a privileged direction without a full blown reflection and is also used in traditional raster graphics. This was improved upon by Blinn and Newell (1976) who added some considerations to the initial specular result. This involved including a Blinn vector with the normal to the surface producing a result called Phong-Blinn lighting. Other conditions are sometimes used such as transparency, bump-mapping and ambient-occlusion to help to create a even more realistic image. See also Cook and Torrance (1982) who also further developed "a reflectance model" combined with the other methods for lighting. These ray tracing lighting effects can be built up upon to form the combined effect from these modular components, for example the total intensity for a surface point being can be equal to the summation of ambient, diffuse and specular components, (Phong, 1975). This is then combined with recursive calculations for reflections and refractions, used by Atalay and Mount (2002), to form the final colour. The term ‘Ray Tracing’ comes from this recursive ray casting nature of these lighting factors, which is then ‘traced’ back to form the final colour.

2.3 Cameras

Ray tracing depends on a camera position often called the eye position and casting rays to an image plane to get a final image. There are a few different methods for camera types which are outlined by Carlbom and Paciorek (1978). Parallel Projections is one method, which shoots all rays in parallel into each pixel. The other is Perspective Projections which shoots all rays from a single point towards each pixel in the scene which is more physically accurate as it "represents an object as it would be seen by an observer positioned at a certain vantage point". This also incorporates the physical camera attribute known as focal length. Carlbom and Paciorek (1978) also discuss viewing transformations useful for moving and positioning the camera.

2.4 Anti-aliasing

There are many methods aimed at reducing aliasing (a unwanted staircase pixel effect in images). The universally used solution in ray tracing was initially developed by Whitted (1980) called super-sampling. Super-sampling is the idea that you sample a higher resolution image therefore using more individual colour samples per pixel in order to reduce aliasing problems. "The only way to anti-alias within standard ray tracing is to go to higher resolution", (Amanatides, 1984). For a final image of X by Y resolution, render to a higher resolution of double X and double Y, then taking the average of four samples to get the colour of one pixel. This is effectively a 4x super-sampling due to computing four times more rays per pixel, although therefore requiring a lot more computation.

2.5 Texture Mapping

Ray tracing can be combined with texture mapping to produce even more realistic results. Texture mapping is the process of applying texture colours and patterns usually from a bitmap or raster image to geometry in the scene, originally pioneered by Catmull" (1974). 2D or 3D textures are defined texels in a 2D or 3D array which translate the texture coordinates to the geometry described by Shirley (2000). Blinn and Newell (1976) pioneered this idea of UV texture coordinates to map textures onto curved surfaces. Additionally the texture can then also be combined with other optional techniques such as bump mapping and transparency mapping. Bump mapping is a technique in computer graphics for simulating bumps and wrinkles on the surface of an object later introduced by Blinn (1978).

2.6 Existing Implementations

Ray tracing is most commonly used in modelling packages such as 3DS Max and Maya when rendering an image of the scene. Other software such as POV-Ray by Buck (2004), are striving towards getting real-time ray tracing performance. The hardware company's NVidia has researched and produced there own software OptiX (Parker et al., 2010) running in parallel with CUDA and optimised through accelerated data structures, similar to the work of this project. A different approach is the RPU, ray processing unit. The RPU was proposed by Woop et al. (2005). Introducing an "architecture and a prototype implementation of a single chip, fully programmable Ray Processing Unit." which is a hardware implementation approach to real-time performance of ray tracing. Woop et al. (2005) stated that it "already renders images at up to 20 frames per second".

2.7 Real-Time Speed Improvements

2.7.1 Parallel GPU with CUDA^{TM}

CUDA is a parallel computing platform and programming model created and developed by NVIDIA. Most relatively new NVIDIA GPUs implement the CUDA architecture, "All NVIDIA's currently available cards (GeForce, Quadro, and Tesla brands) have CUDA work distribution units that are optimized for homogeneous units of work." describes Aila and Laine (2009). CUDA offers easily implementable extensions available for C and C++. Unlike OpenCL, CUDA-enabled GPUs are only available from Nvidia. Budge et al. (2008) demonstrates an implementation with "CUDA framework based on fast stack-based kd-tree traversal. Each ray is mapped to a thread, and a single kernel is used for the entire ray tracing pipeline including shading".

2.7.2 Parallel GPU with OpenCL

OpenCL (Open Computing Language) is another framework for writing programs that execute across platforms consisting of central processing unit (CPUs) and graphics processing unit (GPUs). OpenCL gives any application access to the graphics processing unit for non-graphical computing. OpenCL also allows the use of the graphics processing unit for computation beyond graphics for example the parallel processing of ray tracing. A useful feature is that "OpenCL is designed to efficiently share with OpenGL" described by Munshi (2008). The major key feature of OpenCL is portability on most GPUs (via its abstracted memory and execution model). Although the portability is "not guarantee that the same code will run on all different types of devices with near-optimal performance without rewriting the most performance- or data-intensive parts of the code targeting a specific device.", (Cho et al., 2010). The OpenCL specification added by Munshi (2008) provides much information on implementing an OpenCL application.

2.7.3 Spatial Partitioning

Rendering the scene can be optimised by partitioning the scene so that each ray does not have to be checked against every piece of geometry. Spatial partitioning works by grouping multiple objects together into Bounding Volume Hierarchies (BVH), so that only select regions of the scenes geometry need to be intersection checked describes (Glassner, 1993). There are many BVH methods with a variety used for ray tracing including commonly Kd-Trees, Octrees, Binary Space Partitioning and Regular Grid implementations. Kd-tree partitioning constructs a k-dimensional tree which is a special case of a binary space partitioning tree. Hou and Zhou (2011) describes how to utilize the GPU efficiently by parallelising both within and across kd-trees structure. Octree implementation is described by Jing and Song (2008) saying, "the root node expresses a cube that contains the whole objective. If the cube is filled with the objective fully, stop dividing; else the cube is divided into eight small cubes of same size." Regular grid implementations involve partitioning the scene into same size voxel boxes, making construction and traversal easier. Fujimoto et al. (1986) demonstrates a method to traverse these data structures with a ray to find the objects it intersects. A major disadvantage of these spatial partitioning methods is that for dynamic scenes they require the tree structure to be either partially or fully rebuilt each time the scene changes.

3 Design, Methods and Implementation

To build the final real-time ray tracer, the following methods, algorithms and techniques are combined in program code to produce a working real-time ray tracing application. Methods are detailed below on the techniques and algorithms used, as well as design and implementation decisions made.

3.0.4 Scope

To create a real-time ray tracer a system has been written using C++. This is combined together with WIN32 and OpenGL to render the produced ray traced image output to the screen. The GPU implementation is achieved with OpenCL, but it is beyond the scope of the project to also implement the program with CUDA. Ray intersections with spheres, planes and triangles are implemented, but other additional geometry is not included as it wouldn't provide much benefit to the system. The scope is also limited to just implementing a ray tracer, other additions such as Radiosity and other optional lighting effects are too complicated to be added this project, but could be considered in future work. Testing is performed to determine performance factors, but only upon available hardware of AMD and Intel, therefore the system is not required to work on NVidia hardware and is untested.

3.1 Basic Approach

The fundamental structure of ray tracing shown earlier in Figure 2.1 together with Algorithm 1 producing the basic approach towards implementing a ray tracer. This method becomes the starting point to implementing a ray traced solution with a modular design allowing many other lighting options to be gradually attached. This method loops for all the pixels in the screen for both the width and height. This is the major problem for ray tracing being slow, but can be easily parallelised due to each rays independent nature (see OpenCL section 3.8). For each of these pixels a ray is constructed from an eye viewing position and is traced through the current pixel. A ray is constructed from a origin point R_o and a direction R_d. In this case the specified camera (eye) position is the starting point for every ray and the direction is the vector between the eye point heading towards the current rays pixel. These values are denoted by x,y,z values. This ray is then checked for intersections with objects in the scene, the naive approach is to check all objects for the closed intersection, but later an accelerated data structure can be used to reduce the number of intersections(see Regular Grid section 3.10). A point along a ray can be determined using the ray equation: $P = R_o + R_d * t$ where t is a parameter of distance along the ray. The intersection methods for objects described below calculate this t parameter, allowing intersection points to be calculated and determine which object is in front. From this point the pixel colour is calculated using various lighting factors and the objects properties.

3.2 Intersections

Ray Sphere Intersections The simplest objects for computing intersections are spheres due to there easy mathematical description. Spheres can be rendered very easily using ray tracing compared to traditional rasterisation approaches which require the use of polygons. The mathematical representation of a sphere consists of a centre point $Sc = (Xc,Yc,Zc)$ and a radius $S_r r$. The intersection can either be computed using the quadratic formula to compute t0 and t1 values or a geometric approach which is slightly faster. The basic geometric algorithm implemented (Glassner, 1993) is detailed in Figure 3.1.

origin to centre vector $\equiv OC = S_c - Ray_{Origin}$

closest approach $\equiv CA = OC \cdot Ray_{Direction}$

half chord distance $\equiv D = CA^2 - OC^2 + S_r{}^2$

Check half chord distance: If D < 0, then no intersection as ray misses the sphere.

$t = CA - \sqrt{D}$ for rays originating outside the sphere,

$t = CA + \sqrt{D}$ for rays originating inside or on the sphere.

t < 0 then sphere is behind the ray.

Figure 3.1: Ray-Sphere Intersection Geometric Approach

Ray Plane Intersections An infinite plane is another mathematical object that can be easily rendered in ray tracing. The mathematical representation of a plane consists of a point on the surface of the plane $P_c = X_c, Y_c, Z_c$ and the normalised surface normal of the plane P_n. From these parameters intersection points with a ray can be calculated. The algorithm used is shown in Figure 3.2.

```
compute dot product $\equiv vd = P_n \cdot Ray_{Direction}$
vd == 0 then ray and plane are parallel.
$t = (P_n \cdot (P_c - R_o))/vd$
t < 0 then plane is behind the ray.
```

Figure 3.2: Ray-Plane Intersection Approach

Ray Triangle Intersections To produce more complex objects polygons are needed such as triangles, which is the standard approach to create a object/scene in Computer Graphics. Triangles are similar to planes but are finite instead of infinite. The representation of a triangle is define as three points $T_{v1xyz}, T_{v2xyz}, T_{v3xyz}$ and the normal to the triangles face T_n. Since a triangle is so similar to a plane, the same approach is used to calculate the intersection point shown above using T_n and one vertex. Once the point on the plane is found this needs to be checked to see if the point is inside the sides of the triangle. Scott (2005) describes two techniques, "the same side technique" and the other using barycentric coordinates. The barycentric approach is described as slightly more efficient and therefore chosen for implementation. Firstly picking one of the points, all other locations on the plane are considered as relative to that point. e.g T_{v1} which can also be used as the plane origin point. With the point on the plane known, the point equation can be used and rearranged to find the unknowns to check whether the point is inside the sides of the triangle. This is shown in in Figure 3.3 with triangle vertexes represented by $A = T_{v1}, B = T_{v2}, C = T_{v3}$

Point equation: $P = T_{v1} + u * (T_{v3} - T_{v1}) + v * (T_{v2} - T_{v1})$
Rearrange: (P - A) = u * (C - A) + v * (B - A)
Substitute: $V_0 = (C - A), V_1 = (B - A), V_2 = (P - A) -> V_2 = u * V_0 + v * V_1$
Rearrange and substitute to produce two equations, used to solve the two unknowns:
$u = ((V_1.V_1)(V_2.V_0) - (V_1.V_0)(V_2.V_1))/((V_0.V_0)(V_1.V_1) - (V_0.V_1)(V_1.V_0))$
$v = ((V_0.V_0)(V_2.V_1) - (V_0.V_1)(V_2.V_0))/((V_0.V_0)(V_1.V_1) - (V_0.V_1)(V_1.V_0))$
if u or v < 0 then wrong direction and outside the triangle.
if u or v > 1 then walked too far and outside the triangle.
if u + v > 1 then crossed the opposite edge and outside the triangle. else point is inside the triangle.

Figure 3.3: Ray-Triangle Intersection Barycentric Approach

3.3 Phong Illumination

To compute the colour of the screen pixel for a intersected object, the objects defined colour is combined with various lighting factors. Ray Tracing uses the same basic local lighting as model, implementing the Phong illumination model (Figure 3.4). The full Phong model of illumination contains terms for ambient, diffuse, and specular reflections. These components are simply added together (3.1).

$$Pixel_{RGB} = Ambient_{RGB} + Diffuse_{RGB} + Specular_{RGB} \tag{3.1}$$

The ambient model of illumination is very similar to the self-luminous model of object colour. It provides default colour to objects when they are not directly lit. But, this gives slightly unrealistic lighting effects and is a greatly over-simplified model of the world. This can be implemented as

$$Ambient_{RGB} = (O_R * 0.1, O_G * 0.1, O_B * 0.1)$$
where O_{RGB} is the intersected objects defined colour.

The next part is diffuse reflection, also known as Lambertian reflection. The brightness of a Lambertian surface depends on the angle between the light direction and the surface normal. Viewer's eye position is irrelevant to the diffuse lighting computation.

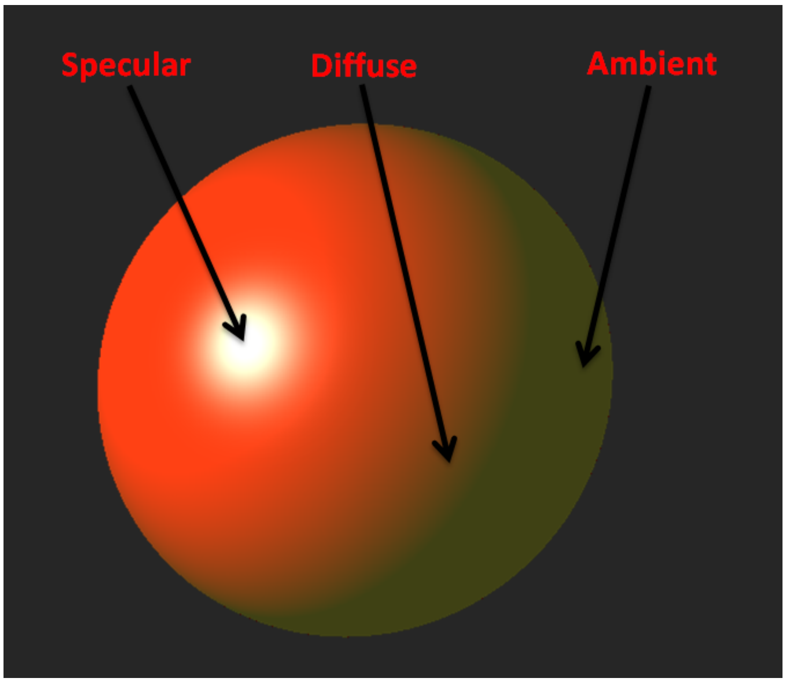

Figure 3.4: Phong illumination components

$$Diffuse_{RGB} = Light_{RGB} * O_{RGB} * (Light_{Direction} \cdot O_N)$$

The specular reflection aspect allows the simulation of render shiny surfaces, causing surface highlights. The specular term varies based on the position of the observer. The specular term chosen is the derived version based on the work of Jim Blinn, called Blinn-Phong (Blinn and Newell, 1976). This is expressed in terms of the surface normal, the lighting direction and the viewing direction with a specular-reflection exponent component. The degree to which specular reflection falls off can be set by the blinn power exponent value.

$$Specular_{RGB} = (O_N \cdot H)^{BlinnPower}$$
$$\text{where } H = normalise(Light_{Direction} - Ray_{Direction})$$

3.4 Camera

A basic static camera system is easy to implement simply using the pixel values stated before, but to get a moving camera to view the scene from different angles requires another approach. A perspective view using the eye position and a direction can be used. Directly trying to modify values of the camera can easily cause screen distortion with the change in the field of view, making spheres no longer appear round. Therefore a rotation and translation matrix is multiplied with both the eye and direction to move the camera without the field of view changing. This is easily achieved by getting the ModelView matrix off of the stack and multiplying such as below. In this case the ModelView matrix is consisting of a translation, rotation in x and rotation in y:

$$ModelViewMatrix \equiv MV = \begin{bmatrix} \cos\theta_y 1 & 0 & \sin\theta_y & T_x \\ 0 & \cos\theta_x 1 & -\sin\theta_x & T_y \\ -\sin\theta_y & \sin\theta_x & \cos\theta_x \cos\theta_y 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$Ray_{Start} = \begin{bmatrix} Eye_x \\ Eye_y \\ Eye_z \\ 1 \end{bmatrix} \cdot \begin{bmatrix} MV \end{bmatrix} \; Ray_{Direction} = \begin{bmatrix} Pixel_x \\ Pixel_y \\ 0 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} MV \end{bmatrix}$$

3.5 Shadows

To produce shadows where objects can block light from other objects, a concept of a shadow ray is introduced. This shadow ray is used to test if a surface is visible to a light. If the viewing ray intersects an object then a shadow ray is constructed between this point on the surface and each light source in the scene. This ray is traced between this intersection point and the light. If any opaque object is found in between the surface and the light, the surface is in shadow and so the light does not contribute to its shade. This ray between the object and light needs to test for intersections the same way as the view ray earlier. When in shadow ‘Ambient’ light is applied, therefore some lighting may be resident in shadows. However this approach produces hard shadows with a instant switch between lit and in shadow, where as in reality the lit area gradually blends to become shadow. Soft shadows techniques require many more rays, severally affecting the real-time performance and therefore not used.

3.6 Reflection

A key feature of ray tracing is reflections, which are difficult to simulate using other algorithms, but are a natural result of the ray tracing algorithm. This is a recursive process which involves Ray Tracing secondary rays from the point of intersection. A reflected ray in constructed in the mirror-reflection direction for shiny surfaces. It is then intersected with objects in the scene, the closest object it intersects is what will be seen in the reflection. The final reflectance pixel colour is the returned colour multiplied by the reflectance factor of the object. (See Figure 3.6). The incidence ray (eye ray) is reflected at the intersection point, resulting in the reflection ray. The angle between the normal is the same for both sides. The formula in Figure 3.5 is applied.

$$ReflectedRay_{Direction} = O_N * (O_N \cdot IncidentRay_{Direction}) + a \text{ and}$$
$$IncidentRay_{Direction} + a = O_N * (O_N \cdot IncidentRay_{Direction})$$
Therefore by rearranging and subsituting removing a to get:
$$ReflectedRay_{Direction} = IncidentRay_{Direction} - 2 * (O_N \cdot IncidentRay_{Direction}) * O_N$$
$$ReflectedRay_{Origin} = IntersectionPoint_{xyz} + ReflectedRay_{Direction} * 0.01$$
$$Pixel_{RGB} += RayTrace(ReflectedRay) * O_{Reflectance}$$

Figure 3.5: Reflection algorithm

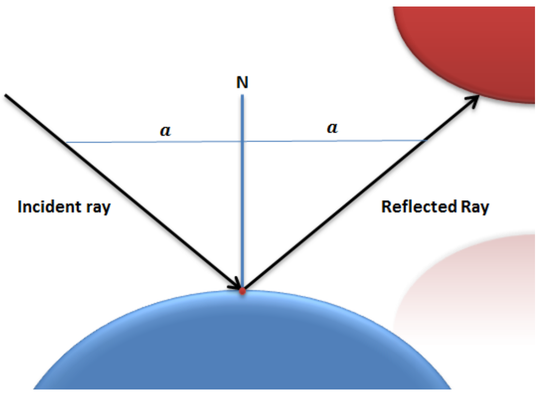

Figure 3.6: Reflected light ray at a point on the surface of a sphere

3.7 Refraction

The next lighting aspect added is to simulate the refraction of light (or sometimes called transmission), such as for the effect of light bending in glass and other translucent materials. This is another recursive technique that requires tracing secondary refracted rays. The amount of bending depends on the objects material properties defined by a refraction index value. The following algorithm described in Glassner (1993) is the vector form derived from Snell's Law of the physics of refraction of light. From the incident ray and a normalized object normal vector n, its possible to work out the normalized refraction ray. Some parts of this algorithm are similar to the specular reflection calculation done earlier. (See Figure 3.7)

$$\cos\theta_1 = -(O_N \cdot IncidentRay_{Direction})$$
$$\cos\theta_2 = 1 - refractionIndex^2 * (1 - \cos\theta_1{}^2)$$
$$RefractedRay_{Direction} =$$
$$(refractionIndex * IncidentRay_{Direction}) + (refractionIndex * \cos\theta_1 \text{ - } \sqrt{\cos\theta_2})n$$
$$ReflectedRay_{Origin} = IntersectionPoint_{xyz} + ReflectedRay_{Direction} * 0.01$$
$$Pixel_{RGB} += RayTrace(RefractedRay) * O_{Refractance}$$

Figure 3.7: Refraction Algorithm

3.8 OpenCL

To implement the ray tracer concurrently, the OpenCL platform was chosen. This is due to no NVidia hardware being readily available, as well as providing other benefits described earlier. OpenCL-coded routines, called kernels, can execute in parallel on GPUs and CPUs from many popular manufacturers. Also OpenCL supports code closely linked to OpenGL, although debugging tools weren't available, therefore making testing very difficult. OpenCL projects consist of the host application and the executable kernel which is run in parallel on the hardware devices. OpenCL contains data structures and functions that are unique, and not found in other platforms such as CUDA.

The main five data structures that make up the host application are:
$cl\_device\_id, cl\_context, cl\_kernel, cl\_program$ and $cl\_command\_queue$.
Together these form steps that initialise, compile and execute the Kernel. The host application is comprised of these Steps:

Step 1. First select the device to run upon, e.g CL_DEVICE_TYPE_CPU or CL_DEVICE_TYPE_GPU and create the device based on the system platform.

Step 2. Create an OpenCL context from the device platform obtained. This allows devices to receive kernels and transfer data.

Step 3. Next create a command queue from the context, so each device can receive kernels through a command queue.

Step 4. Next load the Kernel code file, these are a '.cl' file external to the program. This solution opted to allow for interchangeable kernel files for testing purposes. The file consists of the code that is to run concurrently, in this case the ray tracing for each pixel.

Step 5. The host application next needs to compile the code from the kernel file into a OpenCL Program data structure, which the host can select the kernel from.

Step 6. The kernel object is then created which the host application can then distribute to devices.

Step 7. Next if the program requires data to be passed to and from the kernel, then buffer objects need to be created. This program transfers pixel data and other parameters to the kernel. define and set kernel arguments. Buffer objects declare the memory for these parameter and are then assigned to the kernel object.

Step 8. Finally the kernel needs then to be enqueued onto the devices through the command queue.

Step 9. Optionally to read buffer contents after execution the buffer contents then need to be mapped back into a data structure in the host application, in this case pixel colour values.

Memory Spaces The kernel parameters can be stored using different memory methods to optimise the performance of the memory transfer. Normally parameters are use the Global memory space as it is the largest capacity memory subsystem on the compute device, but is considered the slowest memory subsystem. This is required for the large data sets of scene data and the pixel grid. Other memory spaces available are local memory which is shared by work-group, private memory which is per work-item memory and constant memory which is a region of read-only memory. Constant memory is the fastest memory space and has been used to greatly improve the data transfer performance of some parameters that are constant across all executed kernels and require little memory. Additionally the parameters that require global memory can be optimised through specifying that the memory uses the host memory pointer, avoiding extra memory copying.

OpenCL Shortcomings Through implementing a program in OpenCL there are a number of problems that need to be address such as memory overflow due to limiting size of memory spaces, additionally out of bounds problems can occur if indexes aren't correctly assigned. The lack of debug information makes development and testing additionally harder. The ray tracer requires recursion to be implemented correctly but GPU graphics devices generally have yet to include recursive functionality, but is likely to be included in future hardware. AMD platforms are the exception and support one layer of recursion allowing the ray tracer program to be implemented, but resulted in incompatibility with current NVidia hardware.

3.9 File Loader

Scene data to be rendered to the ray tracer needs to be easily customisable and interchangeable. Therefore a file loader is needed to load text based data. This is achieved by reading in a file name and opening the data in the C++ host application. A custom file format for spheres and planes was needed to be created. Additionally the file loader is configured to open object format (.obj) files to load triangle meshes. The scene data is then passed to the OpenCL kernel ray tracer. The different object formats are distinguished by an identifier at the start of each line. The format listed below:

v - Vertex	f - Triangle face
o - Standard primitive	t - Total objects to load
r - Scene background colour	

3.10 Regular Grid

The naive approach to ray tracing is to loop through all the objects in the scene when checking for intersection tests. Performance can be greatly improved with an accelerated data structure. This ray tracer chose to implement a regular grid structure to reduce the number of intersections. This type of spatial division subdivides the scene space in regular sized sub-boxes. As the ray passes through the grid of boxes, checks are made to see if they contain geometry that we should ray trace against. This means rather than checking rays against all objects in the scene, only need to do intersection calculations with the boxes the ray passes through. This idea is illustrated in 2D in Figure 3.8.

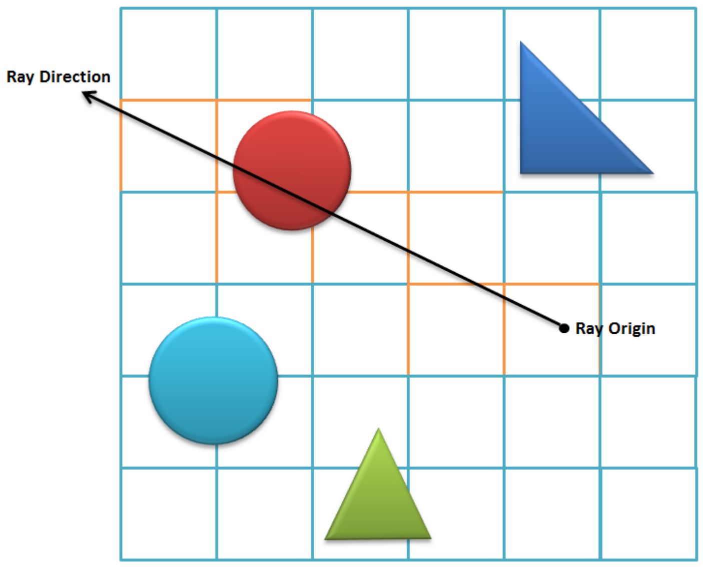

Figure 3.8: Traversing a ray through a 2D Grid

The first step is to generate the grid for the scene data. The grid dimensions are defined from the maximum and minimum position of the objects in the scene creating a bounding box around the whole scene, this is then cut into 3D box cells using defined number of boxes e.g 10 by 10 by 10 cells. This structure is implemented with a concept of a 4D array with the forth dimension being the list of objects that this cell intersects. To insert objects into grid cells the bounding boxes of the object primitives are found (See Figure 3.9) and placed into the cells that intersect the box (See Figure 3.10).

Sphere Bounding Box is:

ObjectBB.min = spherecenter - radius in xyz;

ObjectBB.max = spherecenter + radius in xyz;

Triangle Bounding Box is:

ObjectBB.min = smallest vertex coordinates in xyz

ObjectBB.max = largest vertex coordinates in xyz

(Infinite planes cannot be inserted into a finite regular grid and therefore need to be handled separately as before.)

Figure 3.9: Finding bounding boxes of object primitives

Objects are then placed into cells by checking intersection of the ObjectBB and each cell. This is achieved by checking:

```
if($ObjectBB.min >= GridCell.min$ AND $ObjectBB.max <= GridCell.max$)
then object inside the grid cell...
else if($GridCell.max < ObjectBB.min$ OR $GridCell.min > ObjectBB.max$)
then no intersection occurs as object is completely outside the cell...
else
then they intersect...
```

Figure 3.10: Approach to insert object primitives into cells

When performing ray tracing the grid needs to be traversed with the rays passing through it, testing for intersections with the scenes geometry. We traverse the grid cell by stepping cell by cell following the ray's direction (See Figure 3.8). The 3D-Digital Differential Analyser (or DDA) algorithm is used to achieve this (Fujimoto et al., 1986). The first step before traversing the grid is to check if the ray hits the grid at all, which can be done with a simple ray-box intersection test. If the ray intersects the grid then the coordinates of the cell where the ray enters the grid are computed. Once we know the start position of the ray in the grid (which requires to convert the hit point or the ray's origin if the ray is inside the grid to cell coordinates), we simply use the DDA algorithm to efficiently walk through the grid in the direction of the ray and test for an intersection with the geometry contained by every cell the ray passes through. The traversal algorithm is easily explained in 2D, starting with the ray starting point. To be able to step through the cells the distance to move 'tMaxX' to cross the vertical cell boundary is found, and 'tMaxY' to cross the horizontal cell boundary. Once these factors are found, traversal can be stepped by using the minimum of these values which determines how far we can travel before we hit the first cell boundary. This method is shown in Algorithm 2 below (Amanatides et al., 1987). This was easily extended into 3D for our implementation simply adding a Z axis component.

Algorithm 2 2D Grid Stepping Traversal

```
loop
  if tMaxX < tMaxY then
    tMaxX = tMaxX + tDeltaX.
    XCell = XCell + stepSizeX
  else
    tMaxY = tMaxY + tDeltaY
    YCell = YCell + stepSizeX
  end if
end loop
```

4 Results, Testing and Performance Analysis

To test the performance of the implemented ray tracer, several scenes were made. These tests see the various performance factors of different aspects of the ray tracer, identifying potential bottlenecks to the real time performance as well as seeing how well it runs on different hardware systems including both parallelised GPU and CPU. The test scenes listed in Table 4.2 are used (Screen-shots in Appendix A) and tested upon systems listed in Table 4.1. All scenes are using a standard setup of 1 light and a 10 by 10 by 10 grid structure unless specified otherwise. Results are collected in two ways by measuring direct time to complete running the OpenCL kernel in seconds and the overall performance of the system measured in frames-per-second (fps). The systems tested upon are mid-range consumer hardware attached to a normal PC. Interchangeable kernel files and scene files are used to test various aspects of the system, to more accurately identify key areas of interest.

Table 4.1: Hardware Devices

Device	**Type**	**Cores**	**Clock Speeds**
AMD FX 8350 Piledriver CPU	CPU	8	4.00GHz
AMD Radeon HD 7870	GPU	1280 Stream Processors	1000MHz

Table 4.2: Test Scenes

Scene File	**Description**	**Primitives**	**Max Per Cell**
scene.txt	Basic Test Scene with planes, triangles and spheres.	621	22
grid.txt	Grid of spheres with maximum of 2 per grid cell	1001	2
rand50.txt	50 Randomly placed spheres	53	6
rand500.txt	500 Randomly placed spheres	503	18
rand5000.txt	5000 Randomly placed spheres	5003	113
teapot.obj	Blinn and Newell (1976)'s Teapot made from triangles.	1057	102
bunny.obj	Turk and Levoy (1994)'s Rabbit made from triangles.	4968	74
objects.obj	Various objects made from triangles.	5436	401

Each scene has been tested on each of the hardware devices shown in Table 4.3. Real-time performance has been achieved using the GPU, but CPU implementations run slower. Increasing the number of grid data per cell, introduces a lot more computation time and memory costs. This can be seen in the results to be proportional to the performance. The grid data structure can help to improve performance, but the benefit

Table 4.3: Scene Performance Data

Scene name	**CPU(FPS)**	**CPU(s)**	**GPU(FPS)**	**GPU(s)**
Scene	2	0.48	17	0.04
Grid	9	0.08	30	0.01
50RAND	7	0.10	28	0.02
500RAND	4	0.20	19	0.03
5000RAND	1	0.70	7	0.11
Teapot	2	0.67	16	0.05
Bunny	1	1.00	7	0.13
Objects	3	0.36	17	0.04

appears to vary being highly dependant on grid densities. There are also differences in the data due to the number of rays that don't intersect objects. This is demonstrated with the 'objects.obj' scene that has the largest number of triangles performing better than other scenes with considerably less primitives such as the 'bunny.obj' scene. There are many factors effecting the scene performance with differences in shapes and camera positions. This makes detailed comparison of these scenes harder to quantify. The best comparison can be achieved by looking at the random sphere data scenes that share the same camera positions and overall space layout.

Testing the benefit of the regular grid data structure To fully quantify the effectiveness of the grid data structure the basic 'scene.txt' file is tested with a separate kernel implementation that contains none of the grid acceleration code and uses only the naive approach of performing intersections with all the objects in the scene. This can compare the performance benefit in Table 4.4

Table 4.4: Performance data of scenes with no regular grid optimisation

Scene file	**CPU(FPS)**	**CPU(s)**	**GPU(FPS)**	**GPU(s)**
50RAND	4	0.20	28	0.02
500RAND	1	1.80	7	0.12
5000RAND	0	12.80	1	0.90

The performance difference between the regular grid implementation and intersecting all the objects is quite significant. With the increasing amounts of spheres, the grid saves a lot of computation time helping greatly to make real time performance more possible. The downsides of the grid come in the form of increased memory usage and restricting moving objects in the scene as the grid needs to be modified or rebuilt if a object moves. Other accelerated data structures such as Octrees could be tested in future work.

Another factor that could help to improve performance is to alter the grid density. If a higher grid density is used, each cell could potentially contain fewer objects and therefore less intersection tests would need to be performed. However this comes at the cost of much greater memory use. The effects of increasing the grid size can be seen in Tables 4.5 and 4.6. This shows that for the standard scene the effect of increasing the grid doesn't change the performance much as the memory cost outweighs the benefit of the reduction of objects per cell. However a performance increase can be seen for the 'bunny.obj' with the increase in grid density gaining performance speed. This is down to the maximum number of objects per cell decreasing from '74' to as low as '24'. This therefore results in fewer intersections, outweighing the cost of the increased memory. Tweaking this grid density for each scene could help to optimise their real-time performance.

Grid Density	FPS	Seconds
10by10by10	16	0.041
20by20by20	16	0.048
30by30by30	16	0.046

Table 4.5: Performance after changing the grid density for scene.txt

Grid Density	FPS	Seconds
10by10by10	7	0.132
20by20by20	10	0.084
30by30by30	11	0.077

Table 4.6: Performance after changing the grid density for bunny.obj

Testing the performance of different standard screen resolutions The difference in screen resolutions greatly affects the speed of the ray tracer as with increased numbers of pixels means more rays have to be traced. Therefore 'scene.txt' has been tested with different commonly used resolutions to see the impact on performance (Table 4.7). It was expected for this effect to be linear, but this is not shown in Figure 4.1. It has been concluded that this is due to a changing viewing angle as well as the many factors that affect the performance of the ray tracer. These results also show that the ray tracer can achieve good real-time performance at lower resolutions than the default resolution (XGA 1024x768), which is greater than many other implementations of ray tracers shown in Glassner (1993).

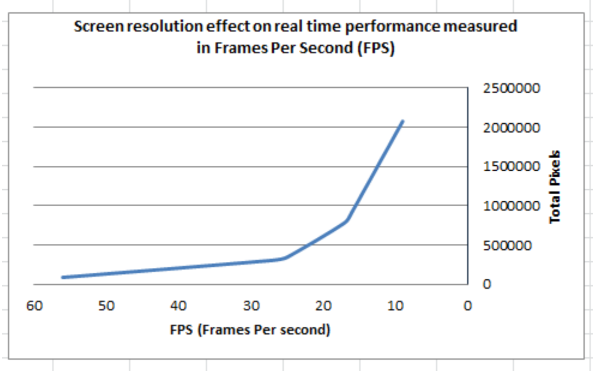

Figure 4.1: Graph of performance against total number of pixels

	Resolution	FPS
XGA(Default)	1024x768	17
Window	360x250	56
VGA	640x480	27
480p	720x480	25
720p	1280x720	16
1080p	1920x1080	9

Table 4.7: Performance data of scene.txt with different screen resolutions

Performance analysis of the different lighting components To try to find which areas of the ray tracer mostly affect the performance, a number of kernel files have been made with various aspects of the ray tracing removed. These results were obtained using the 'teapot.obj' scene as this fairly includes the same amount of refraction, reflections and shadows on each triangle and can be accurately used to test how much these factors effect the system. The results are shown in Table 4.8. These imply that even with removing reflections and shadows from a scene the performance difference if only slight. However the refraction calculations appear to have the most performance impact, providing a large performance boost when this calculation was removed. Reflections and refractions are quite similar in code, so such a large performance different is likely down to underlying hardware acceleration. Therefore further work into optimising this section can be done, to further increase real-time performance. Additionally a test with just the Phong illumination was also performed as it produces a similar image shading result to traditional OpenGL rasterisation rendering approaches and seems to perform reasonable well in real time.

Table 4.8: Removal of lighting features to identify computationally expensive sections of code

Kernel file	Kernel description	FPS
MainKernel.cl	Normal default kernel file	16
NoReflection.cl	Kernel with no reflections	17
NoRefraction.cl	Kernel with no refractions	24
NoShadow.cl	Kernel with no shadows	18
JustPhong.cl	No advanced lighting, just Phong Illumination	32

Testing the performance implications of memory transfer To see what other factors affect the performance of the final ray tracer, another kernel file was made with all ray tracing removed. This allows the performance of memory transfer being passed through the kernel to be recorded. Therefore the performance cost of the running application is shown in Table 4.9. This memory only kernel is tested with increasing numbers of spheres stacked on top of each other to gradually increase the grid memory size. These results show that even without performing any ray tracing, other factors such as the screen resolution and memory transmission seem to greatly slow down the performance. The effect of increasing memory size slows down the FPS. Therefore the memory transfer is a large factor in slowing down the performance. Though the basic scene without any ray tracing only runs at '31 FPS'. This is before performing any ray tracing functions such as grid traversals, intersections and lighting. This demonstrates that the host application is a performance issue and additional performance can be achieved by reducing the cost of these memory transfers.

Table 4.9: Performance of memory transfer without performing ray tracing

Scene file	Description	FPS
scene.txt	Normal default scene	31
sphere1.txt	1 centred sphere (maxcell 1)	33
sphere500.txt	500 stacked centred spheres (maxcell 500)	31
sphere5000.txt	5000 stacked centred spheres (maxcell 5000)	24

Identifying the host application as being a major performance factor has prompted additional tests on aspects of the application shown in Table 4.10. This demonstrates that there is some overhead in the texture drawing. The biggest cost is the execution of the OpenCL kernels as expected. Although even without any ray tracing or OpenCL kernels the application peaks at '84 FPS'. Therefore there is a factor of the host application that is greatly reducing performance and would therefore greatly help to provide additional performance in the future if this could be improved.

Table 4.10: Performance impacts of sections of the host application

Program modification	**FPS**
Normal Program	16
Program with no data output copying from the kernel	21
Program with no OpenCL kernel execution	84
Program with no texture assignment or drawing	24

Overall there are many factors that can affect the performance of this ray tracing. Number of objects, positioning in the scene, lighting factors, and which hardware it is running on all change the real-time performance. The biggest contributors to the ray tracing performance is the screen resolution and grid memory sizes. The host application has been identified to have low performance for future improvement.

5 Future Work

The field of ray tracing is a highly expandable field of work. There is certainly a lot of future work and research to be explored in this field of computing. Future work into additional areas of lighting and into improving the real-time performance are the key areas to making ray tracing more mainstream in computer graphics rending.

There are many other features that can be added to this ray tracer to produced greater lighting and scene effects. One addition could be to add improved anti-aliasing. Texture mapping onto the scenes objects is also a technique discussed. This could improve the image scene results, but was opted out in this implementation due to the cost of sending more memory to the kernel parameters. Other lighting factors such as 'Ambient

Occlusion' and 'Soft Shadows' are additions for improving image quality, but greatly increases the number of rays to be cast.

Future work into further improving the ray tracing real-time performance is a key area. This could be achieved by exploring additional data structures and looking at other methods of implementing the ray tracer in parallel. This should aim at increasing rendering speed with large data sets and addressing the memory transfer bottlenecks.

Specific to this ray tracer the host application of this could be investigated and improved. Also work into getting the solution to work cross platform, such as looking into CUDA implementations to compare speeds.

Future applications of this ray tracer include rendering of lighting for scenes imported using the built file loader. The ray tracer could also be used for the basis of a simple game offering good rendering effects. Commercial applications could involve work with haptic devices and rending scenes with large numbers of spheres such as chemical biomolecules.

6 Conclusions

This project focused on the design and implementation of a ray tracer capable of producing computer graphics images. A final ray tracer program has been created which exhibits a range of optical effects including shadows, reflections and refractions. The resulting ray tracer works upon consumer hardware and has strived towards real-time performance. Methods to improve performance have been implemented such as a spatial decomposition grid and a parallel implementation using OpenCL. Through these optimisations real-time ray tracing has been achieved. All initial aims have been produced and to a reasonably good standard.

Much more progression and research can also be achieved in this field in the future to further render more complex scenes with greater resolutions in real-time. With improving hardware all the time, real-time ray tracing is definitely starting to be possible. It is unlikely at this point in time to be able to be used in games with large scene data, but ray tracing is definitely possible for other real-time applications or composite approaches. Perhaps in the coming years, more applications will start to use the real-time ray tracing technique together with GPU implementations making the technique more widespread. This is likely to happen as GPUs continue to get more cores and become increasingly powerful.

References

Aila, T. and Laine, S. (2009). Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG ’09, pages 145–149, New York, NY, USA. ACM.

Amanatides, J. (1984). Ray tracing with cones. *SIGGRAPH Comput. Graph.*, 18(3):129–135.

Amanatides, J., Woo, A., et al. (1987). A fast voxel traversal algorithm for ray tracing. In *Proceedings of EUROGRAPHICS*, volume 87, pages 3–10.

Appel, A. (1968). Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS ’68 (Spring), pages 37–45, New York, NY, USA. ACM.

Atalay, F. B. and Mount, D. M. (2002). *Ray Interpolants for Fast Ray-Tracing Reflections and Refractions*.

Blinn, J. F. (1978). Simulation of wrinkled surfaces. *SIGGRAPH Comput. Graph.*, 12(3):286–292.

Blinn, J. F. and Newell, M. E. (1976). Texture and reflection in computer generated images. *Commun. ACM*, 19(10):542–547.

Buck, D. (2004). Persistence of vision (tm) raytracer.

Budge, B., Anderson, J., Garth, C., and Joy, K. (2008). A straightforward cuda implementation for interactive ray-tracing. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pages 178–178.

Carlbom, I. and Paciorek, J. (1978). Planar geometric projection and viewing transformations. *ACM Comput. Surv.*, 10(4):465–502.

Catmull", E. E. ("1974"). *"A Subdivision Algorithm for Computer Display of Curved Surfaces"*. PhD thesis, "Dept. of CS, U. of Utah".

Cho, S., Im, D., Jang, O., Song, H. J., Paulovicks, B., Sheinin, V., and Yeo, H. (2010). Opencl and parallel primitives for digital tv applications. *IBM Journal of Research and Development*, 54(5):7:1–7:14.

Cook, R. L. and Torrance, K. E. (1982). A reflectance model for computer graphics. *ACM Trans. Graph.*, 1(1):7–24.

Fujimoto, A., Tanaka, T., and Iwata, K. (1986). Arts: Accelerated ray-tracing system. *Computer Graphics and Applications, IEEE*, 6(4):16–26.

Glassner, A. (1993). *An Introduction To Ray Tracing*. Academic Press Limited.

Hou, Q. and Zhou, K. (2011). A shading reuse method for efficient micropolygon ray tracing. *ACM Trans. Graph.*, 30(6):151:1–151:8.

Jing, G. and Song, W. (2008). An octree ray casting algorithm based on multi-core cpus. In *Proceedings of the 2008 International Symposium on Computer Science and Computational Technology - Volume 02*, ISCSCT '08, pages 783–787, Washington, DC, USA. IEEE Computer Society.

Kobayashi, H., Nakamura, T., and Shigei, Y. (1987). Parallel processing of an object space for image synthesis using ray tracing. *The Visual Computer*, 3(1):13–22.

Munshi, A. (2008). Opencl. *Parallel Computing on the GPU and CPU, SIGGRAPH*.

Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M. (2010). Optix: a general purpose ray tracing engine. *ACM Trans. Graph.*, 29:66:1–66:13.

Phong, B. T. (1975). Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317.

Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. (2002). Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21(3):703–712.

Scott, J. (2005). Point in triangle test @ONLINE.

Shirley, P. (2000). *Realistic ray tracing*. A K Peters.

Turk, G. and Levoy, M. (1994). Zippered polygon meshes from range images. pages 311–318.

Whitted, T. (1980). An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349.

Woop, S., Schmittler, J., and Slusallek, P. (2005). Rpu: a programmable ray processing unit for realtime ray tracing. *ACM Trans. Graph.*, 24(3):434–444.

List of Figures

List of Tables

Appendix

Print-screens of the scenes produced by the ray tracer.