Registration number 100062949

2016

Real-Time Ray Tracing

Supervised by Dr Stephen Laycock

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

Ray tracing is a technique used in computer graphics to produce highly realistic images by tracing the path of light. Due to its highly computational demand, ray tracing is typically used in off-line rendering for applications such as animation or CGI in film. For that reason, rasterisation has been preferred for real-time applications such as video games on consumer level hardware.

Due to the recent innovations in many core technologies, and the rise in programmable GPUs, real-time ray tracing is edging closer to replacing the classic rasterised graphics technique on consumer level hardware. This project aims to utilise these recent hardware developments to produce a ray tracer capable of rendering images in real-time while achieving effects such as shadows, reflections and refractions. The end result saw a performance of 15 frames per second while rendering a scene of 21k triangles at 720x480 resolution.

Acknowledgements

This project would not be possible without Turner Whitted's work in defining the original ray tracing specification; Nvidia, for the innovations in programmable GPUs through the use of CUDA; and Timothy J. Purcell et al., for demonstrating the first implementation of a ray tracer on a programmable GPU. Thanks also go to Dr. Stephen Laycock for his support and aid in this project.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

The ray tracing algorithm, first pioneered by Whitted (1980), is a computer image generation technique that renders images by tracing the path of light. In reality, an almost infinite number of light rays are emitted from a light source in all possible directions. These light rays then hit and bounce off of objects until they finally enter the eye. Simulating this on a computer is extremely time consuming and inefficient as not all rays will reach the eye. Instead, the light rays are traced backwards from the viewer into the scene. Using this approximated simulation of light, the ray tracer is able to model shadows when an area is blocked by a light source, as well as reflections and refractions for materials with reflective and refractive properties.

Ray tracing is known to be a very slow process due to the sheer number of rays that need to be traced. For every pixel in an image, a ray is traced to calculate it's colour, and for every effect such as shadows, reflections and refractions, further rays must be traced. This ultimately leads to long rendering times and is typically limited to off-line rendering for applications such as animation or CGI in film. However, due to the recent innovations in many-core technologies, and the rise in programmable GPUs, real-time ray tracing performance can be achieved, and will eventually replace the widely used rasterized graphics on consumer level hardware.

1.2 Aims

The aim of this project is to produce a ray tracer that is able to model the basic optical effects such as shadows, reflections and refractions. Real-time performance will be achieved by exploiting the GPUs parallel processing architecture though the use of Nvidia's CUDA on consumer level hardware. Primitive types such as spheres will be implemented due to their simplicity, and triangles will be featured to represent complex

geometry. Other features will also be implemented such as a moveable camera to inspect the scene, anti-aliasing techniques and texture maps to further enhance the image quality.

1.3 Issues

The main issue with ray tracing is the large amount of time it takes to render an image. According to Whitted (1980), up to 95% of the ray tracers rendering time is spent on intersection testing alone. Making it apparent that in order to achieve real-time performance, intersection testing must be accelerated through the use of data structures and efficient ray-primitive intersection algorithms.

2 Literature Review

The ray tracing algorithm, first pioneered by Whitted (1980), applies advanced illumination to an already well established computer image generation technique, the ray caster, Appel (1968). The ray tracer generates its images by simulating a rough approximation of light. Before reviewing these algorithms, some background information should be considered.

2.1 Background Information

2.1.1 Light

Since the ray tracer is modelled on the characteristics of light, some basic information about its behaviour is reviewed.

Light, to this day is not fully understood, the best we can describe it is by using two models, particles and waves. The particle model describes light as being made up of many tiny particles called photons, which travel in a straight line. The wave model represents the frequency of light, which describes it's colour.

In reality, a light source creates and emits photons in every possible direction. When these photons come in to contact with a surface, their direction will either be absorbed,

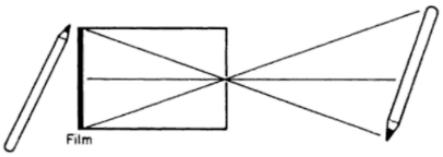


Figure 1: Pinhole Camera, Glassner (1989a)

The image is projected on to the film through the pinhole.

reflected or refracted. When the photon bounces off an object, it's frequency will change, meaning that it will change colour, Glassner (1989b).

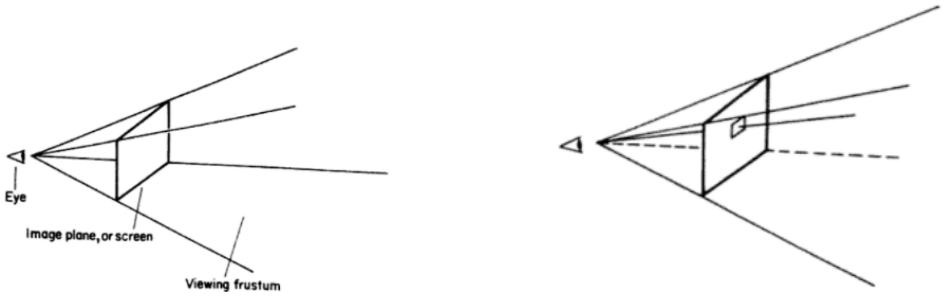
The human eye is able to see the surrounding environment when photons enter the eye. Due to the enormous number of photons travelling in all possible directions, only a small fraction of these will hit the eye which influences the image seen, Glassner (1989a).

In computer graphics, photons are modelled by using what is called a ray. A ray will represent a stream of photons travelling in the same direction. It is made up of an origin and a direction vector which represent a line segment in an Cartesian coordinate system.

2.1.2 Camera

Many camera models are used in computer graphics to visualise a scene. One popular camera model which can be used with a ray tracer is a called the pinhole camera, which is computerised version of the real pinhole camera model. The pinhole camera is a device that allows photographs to be taken. It consists of a light proof box with a small hole on one side, and an image film opposite, Glassner (1989a). The photographs are produced by light travelling through the hole, which projects an inverted image onto the film seen in Figure 1.

When used in computer graphics, the modified pin hole camera model "moves the plane of film out in front of the of the pinhole, and renames the pinhole as the eye" Glassner (1989a) Figure 2a. The lines projecting from the eye to the corners of the image plane represent the clipping wall, and the space inside of this wall is known as the viewing frustum. This can be thought of as an eye looking out of a window. To translate this image plane on to the screen, the plane is split up into many sections, and



(a) The modified pinhole camera used in computer graphics

(b) Each section of the image plane has a corresponding pixel

Figure 2: Modified Pinhole Camera Model, Glassner (1989a)

each section has a corresponding pixel, Figure 2b. This camera model is very popular in computer graphics as it is simple and easy to implement, Glassner (1989a).

2.2 Ray Casting

To simulate the characteristics of light, a naïve approach would be to trace rays from the light source to the viewer. Since only a fraction of these rays will reach the viewer, a more efficient method suggested by Appel (1968), traces the rays backwards from the viewer into the scene. The ray caster is based upon this approach, which traces rays through each pixel in the image plane. The colour of each pixel will be set to the shaded colour of the closest object the ray intersects, see section 2.3. If the ray does not intersect an object, the pixel will be set to the colour of the background, Figure 3.

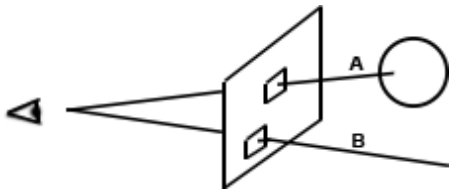


Figure 3: Ray Caster

Ray A intersects the sphere so the pixel will be set to the colour of the sphere.

Ray B does not intersect anything, in which the corresponding pixel is set to the background colour.

2.3 Shading

Shading functions are used once a ray intersects an object to calculate the illuminance of it's surface. There are three types of lighting that are generally used: ambient, diffuse and specular lighting. These three types of lighting are combined to produce the final result.

2.3.1 Ambient Lighting

In reality, photons are almost always present in any given scenario. Simulating constant light can be a very costly process, even in ray tracing. A cheap alternative is to apply the ambient colour manually, which is a simplistic model of global illumination.

2.3.2 Diffuse Reflection

Diffuse reflection is similar to ambient lighting, except the brightness of the colour is dependent on the direction of the light source, relative to the surface of the object. To calculate the brightness of the colour, a few details about the ray-object intersection must be known.

- The point of intersection, which is the xyz coordinate of the point where the ray hit the object
- The surface normal, which is a vector of unit length that is perpendicular to the surface of the intersection point
- A light ray which points to the position of the light source from the intersection point

The brightness of the diffuse colour will be determined by the angle between the light ray and the surface normal. The more this angle is inclined towards 0 degrees, the brighter the colour. However, if the angle surpasses 90 degrees, it can be determined that the light is behind the surface, in which the diffuse reflection will have no effect.

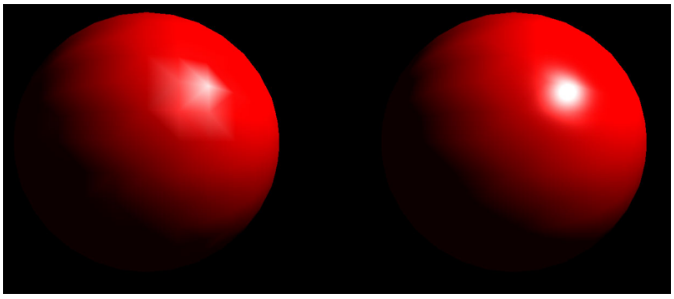


Figure 4: Comparison between Gouraud and Phong shading

Gouraud Shading (left), calculates the illuminance at each vertex, then interpolates the colour across the surface. Phong illumination (right), interpolates the normals across the surface before calcualting the illuminance.

2.3.3 Specular Reflection

The specular highlight is the reflection of the light source in the surface. To determine its brightness, the light's reflection ray about the surface normal needs to be computed. Similarly to diffuse lighting, the angle between the light reflection ray and the incidence ray is calculated. The more inclined this angle is to 0 degrees, the brighter the reflection. If the angle surpasses 90 degrees, no specular highlight will be visible. This is known as the Phong Reflection Model, Phong (1975).

2.3.4 Phong Shading Model

The Phong Shading model is a combination of two unique methods developed by Phong (1975) to illuminate a surface.

The first method, Phong reflection, as already mentioned in Section 2.3.3, calculates the specular highlight of a surface. An algorithm named Blinn-Phong Shading, Blinn (1977) extends Phong's work by applying some optimisations. It calculates the specular highlight by computing a vector known as the halfway vector. This halfway vector is a vector of unit length which points in the direction between the incidence ray and the light ray's directions. The strength of the specular highlight will now be determined by the angle between the halfway vector and the surface normal. This eliminates the costly process of calculating a reflection direction for the light source.

The second part of the Phong Shading Model is named Phong Interpolation, which takes three normals from the vertices of the intersected triangle, and interpolates them to simulate a smooth surface. This provides more realistic results when compared to other shading methods like Gouraud Shading, Gouraud (1971), Figure 4.

2.4 Ray Tracing

The ray tracer developed by Whitted (1980), improves upon Appel's ray caster by introducing an advanced illumination technique which simulates shadows, reflections and refractions. To achieve these effects, many additional types of rays are generated and traced throughout the scene. There are four different types of rays: primary, shadow, reflectance and transmission.

The primary ray is the first initial ray that is traced from the camera into the scene, much like the rays in Appel's ray caster. Instead of terminating once a colour has been found, more rays may be spawned if the ray intersects an object to further calculate the illuminance of that point. Shadow rays are used to simulate shadows, and the reflectance and transmitted rays are spawned depending on the characteristics of the surface material to simulate reflections and refractions.

2.4.1 Shadows

Shadows on a surface point are present if the point is not in view of a light source. To determine whether the point is in view of a light source, a shadow ray is generated, with it's origin at the position of the surface point, and it's direction vector pointing towards the light position. This ray is then tested against objects in the scene for an intersection. If an intersection has been found that lies between the surface point and the light source, it can be concluded that the surface point is not in view of the light source. On the other hand, if an intersection is not found, then the surface point is in view, Figure 5.

Once it can be concluded that a point is in full view of a light source, a shading model will be applied to illuminate the surface that incorporates ambient, diffuse and specular lighting. If the point is not in view of a light source, a shading model which only uses ambient lighting will be used to simulate the ambience throughout the scene. In a scene

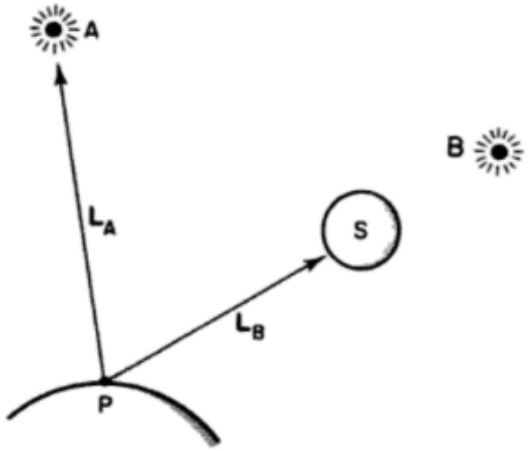


Figure 5: Shadow Rays, Glassner (1989a)

Shadow rays, L_A and L_B are cast from point P, to light sources, A and B. The point P is in view of light A as the ray L_A does not intersect an object. However, light source Bs view of P is blocked because ray L_B intersects object S.

that has multiple light sources, this process will be repeated for each light, making the computational efforts linear in the number of lights.

2.4.2 Reflection

Reflection rays are used to determine the reflected colour of a point on a surface. The calculation of the reflective ray direction simply follows the law of reflection, which states the angle of reflection is equal to the angle of incidence, Whitted (1980), Figure 6.

The reflected ray is traced in the same manner as the primary ray. It is tested against the objects in the scene for an intersection. If one is found, the illuminance at that point will be calculated, which is added to the resulting pixel colour.

2.4.3 Refraction

Transmitted rays are used to calculate the refracted colours of the surface point. The direction of this ray can be calculated by using Snell's Law, Whitted (1980), Figure 6. This is a function that incorporates the incidence ray, the surface normal, and the refractive indices of the current and intersected material. Like the reflected ray, the transmitted

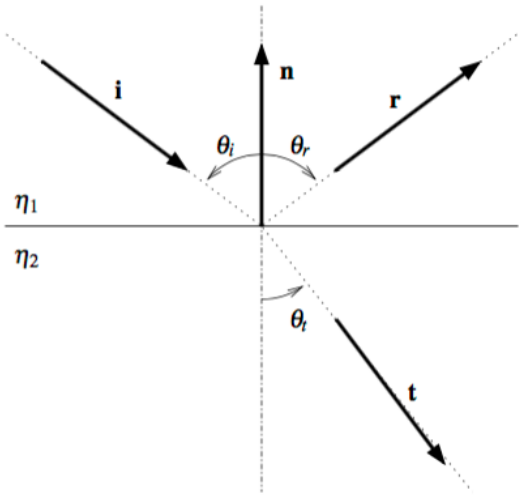


Figure 6: Reflection and Reflection Ray

Reflection: The angle between the reflected ray, $\hat{R}$ and the normal, $\hat{N}$ is equal to the angle of the incident ray, $\hat{I}$ and $\hat{N}$.
Refraction: The transmitted ray, $\hat{T}$ is calculated from a function of $\hat{I}$, $\hat{N}$, and the refractive indices, $n1$ and $n2$, which obeys Snell's Law.

ray also behaves in the same way as the primary ray by being traced further through the scene to find a colour.

2.4.4 Problems

According to Whitted (1980), the ray tracer takes a very long time to render an image. Specifically, it took 74 minutes to render Whitted's famous ray traced image, Figure 7. Whitted broke down the time spent for each task and discovered that the intersection tests among the rays and the objects account for 75 percent of the total rendering time. For more complex scenes, this percentage increased to over 95 percent. Although consumer hardware to date is able to perform tasks considerably faster than hardware in 1980, optimisations in areas such as ray intersection tests are still needed to achieve real-time performance. Sections 2.5 and 2.6 address these issues.

Another problem highlighted by Whitted (1980), is the lack of support for diffuse reflections. As only specular reflections are supported, glossy surfaces which heavily

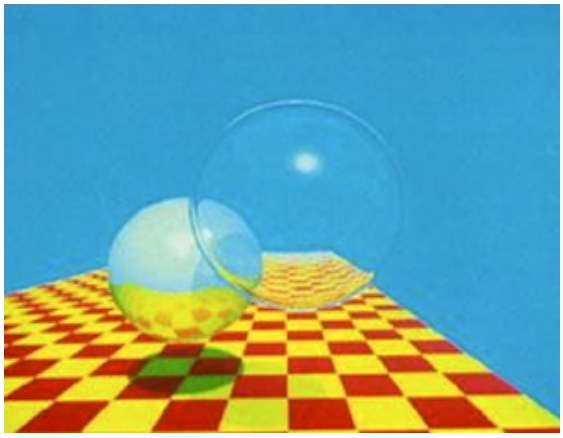


Figure 7: Ray traced image, Whitted (1980)

One of the first ray traced images rendered in 1980, demonstrates shadows, reflections and refractions.

reflect diffuse lighting, can not be accurately modelled. However, a method called distributed ray tracing, Cook et al. (1984) solves this problem, as well as many others by tracing multiple rays, instead of just one ray for each desired effect. However, distributed ray tracing is beyond the scope of this project as real-time performances are much harder to achieve, due to the number of rays traced.

2.5 Ray Intersections

In this section, ray-primitive intersections will be reviewed to develop knowledge of how intersections can be computed and to find the most efficient algorithms to achieve better real-time performance.

2.5.1 Sphere

The sphere is one of the most commonly used primitives in ray tracing due to the simplicity and speed of computing an intersection test against a ray, Haines (1989). Haines presented two ray-sphere algorithms; algebraic, and geometric based. The geometric solution is considerably more efficient than the algebraic solution as the operations are not as computationally expensive.

2.5.2 Plane

Planes are also a very popular primitive to feature in a ray tracer, as like the spheres, they are simple and efficient to test for an intersection. Haines (1989) presented an algorithm which is able to conclude whether a ray has or has not intersected a plane in an efficient manner. Since a plane is infinite, it only needs to be known if the ray's direction points towards the plane.

2.5.3 Triangle

Complex geometry in computer graphics are typically built from hundreds, or even thousands of triangles, making them a very important component of a ray tracer. Since a large number of triangles may be tested against a ray, it is very important to use an efficient intersection algorithm.

Since a triangle is a 2D shape, it is able to lie within the face of a plane. The first step of the ray-triangle intersection test is to compute this triangle's plane. A test between the ray and this plane, seen in Section 2.5.2, will be performed to determine if there is a possibility of an intersection. If a collision between the ray and plane is detected, the intersection point will be computed, and further investigation will be carried out to determine if the intersection point lies inside the triangle. This is done by using the inside-outside test, Sutherland et al. (1974). The inside-outside test is performed three times between the point of intersection, and each of the triangle's edges. If any one of these tests fails, it can be concluded that the ray does not intersect the triangle. Only if all the tests pass, then the point of intersection lies inside of the triangle.

The second algorithm, uses what is called barycentric coordinates to test if the ray-plane point of intersection lies inside of the triangle. Many variations of this algorithm exist, Möller and Trumbore (1997), Badouel (1990), Shirley and Morley (2003), however, they all follow the same general pattern.

Barycentric coordinates are used to express the position of the intersection point by using three scalars: u, v and w. To calculate the barycentric coordinates of the point which is currently expressed in an xyz Cartesian coordinate system, a technique known as Cramer's Rule is used. A collision can be concluded if the sum of these coordinates equal 1. If the sum is does not equal 1, no collision occurred. These algorithms are

considered to be very fast, and efficient since the barycentric coordinates can also be used for triangle interpolation with data such as colours, normals or texture coordinates.

Wald (2004), presents an optimisation of the barycentric coordinates based algorithm, which projects the triangle's vertices and intersection point on to a 2D plane in a way that does not disrupt the barycentric coordinate values. The computations for calculating the barycentric coordinates can then be optimised since only two dimensions are considered instead of three.

2.6 Data Structures

Performing ray intersection tests against objects in a scene is the most computationally expensive part of a ray tracer, as already discussed in Section 2.4.4. One way to minimise the time spent performing these intersections is to store all the primitives within the scene, in to some form of data structure, Arvo and Kirk (1989). Two types of data structures will be reviewed: bounding volume hierarchies (BVH), and spatial subdivision techniques. The fundamental difference between these structures is that the BVH "selects volumes based on a given set of objects", and spatial subdivision structures "selects a set of objects based on given volumes", Arvo and Kirk (1989).

2.6.1 Bounding Volume Hierarchy

The bounding volume is an enclosure, typically a bounding box, or a sphere, that contains a given object. This volume is tested for an intersection against a given ray to determine if the ray has potential of intersecting the object, Arvo and Kirk (1989). Only if the ray intersects the volume, is the ray then tested further against all of the primitives that belong to the object.

The bounding volume hierarchy developed by Rubin and Whitted (1980), is a data structure that wraps groups of nearby clustered bounding volumes, inside a new bounding volume. This process is done recursively, building a tree of bounding volumes, in which the root node will engulf all objects in the scene, and the leaf nodes will contain each individual object.

When testing a ray against the BVH for an intersection, the first step will be to check if

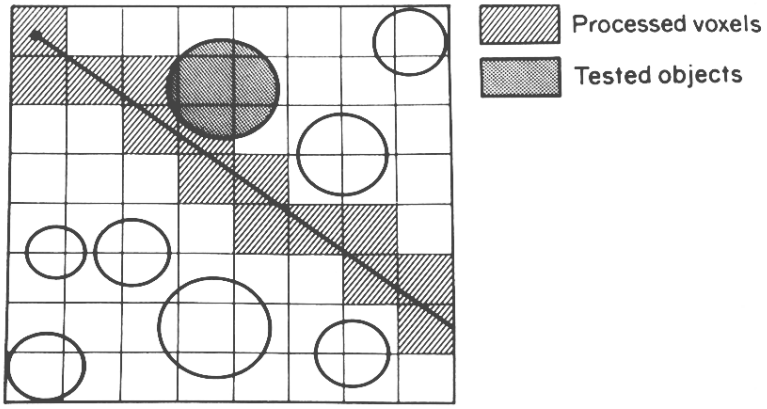


Figure 8: Regular grid traversal, Arvo and Kirk (1989)

The nodes that are grey have been intersected by the ray during the traversal process. The dark grey sphere is tested for an intersection with the ray as it lies inside an intersected node.

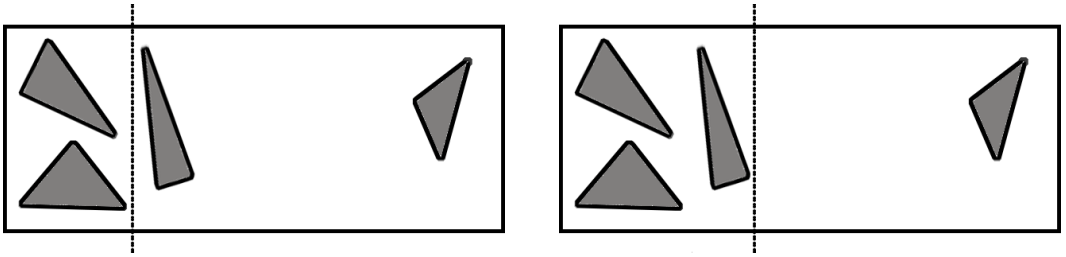
the ray hits the root node. If an intersection is found, the tree will be traversed, checking for an intersection at every level. A ray will only be tested against all the primitives of an object once a leaf node has been found and intersected by the ray. This changes the time complexity for intersection tests from being linear in the number of primitives, to a logarithmic procedure.

2.6.2 Regular Grid

The regular grid, discovered by, Fujimoto et al. (1988), is a type of spatial subdivision structure, where uniform sized voxels are organised in to a 3D grid.

To build the grid, it is 'overlaid' on the scene, in which the voxels will hold references to the objects they intersect.

To traverse this data structure for an intersection, the contents of the closest voxel that is pierced by the ray are tested for an intersection. If the voxel does not contain any primitives, or no intersection is found, an efficient incremental method is used to retrieve the next closest voxel hit by the ray. This process will be repeated until either a ray intersects a primitive, or no intersection is found within all the voxels hit by the ray, Figure 8. This traversal behaviour is only possible because of the uniformly places



(a) Median spatial splitting

Here, the splitting plane at the spatial median of the four triangles.

(b) Surface area heuristic splitting

The splitting plane is chosen at a position that maximises empty space.

Figure 9: Kd-tree splitting plane techniques

voxels in the grid, Arvo and Kirk (1989).

2.6.3 Kd-tree

Bentley and Ottmann (1979), presented an algorithm called the kd-tree, an abbreviation for k-dimensional tree. It is a non-uniform spatial partitioning structure, in which space is sectioned in to regions of varying size, Arvo and Kirk (1989).

To build the tree, the root node first constructs a voxel which encompasses every primitive within the scene. An axis aligned plane along the x, y or z dimension will be chosen which splits the voxel to divide the primitives. This forms a tree by recursively splitting each voxel until there are only a certain number of primitives left, or some threshold is met to prevent the tree from becoming too large. The most trivial and naïve method to split the voxel is called spatial median splitting, in which a plane at a random dimension is chosen at the spatial median, Wald and Havran (2006), Figure 9a.

The position of the splitting plane in a kd-tree directly affects the number of traversal steps and ray-primitive intersections, Wald and Havran (2006). Since the median spatial splitting algorithm is very naïve, a more sophisticated approach which uses Surface Area Heuristics (SAH), maximises empty space close to the root when choosing a splitting plane, Figure 9b. This essentially decreases the number of traversal steps which has potential to increase the performance up to several times faster, Wald and Havran (2006).

When traversing this data structure to find an intersection, the voxel of the root node is

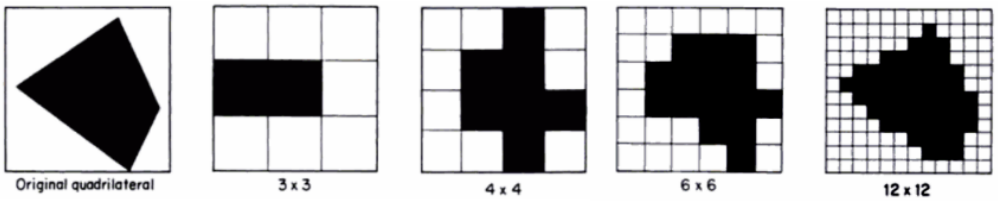


Figure 10: Spatial Aliasing, Glassner (1989a)

An object is represented in four different resolutions. No matter how high the resolution, the staircase effect, jaggies will always be present. All that can be done is to minimise the effect.

tested against the ray. If the ray pierces the voxel, the tree is traversed starting at the root. The node that is the furthest from the ray origin is pushed on to a stack. The nearest node is traversed further resulting in a recursive process. Once a leaf node is found, it's contents are tested against the ray. If an intersection is found, the tree traversal terminates and returns the intersection data. If no intersection is found, the next node will be retrieved from the stack and this process will repeat again, Foley and Sugerman (2005).

If the ray tracer is to be parallel computed on the GPU, see Section 2.8, this traversal algorithm will be very inefficient, as a stack will need to be stored in memory for every ray at the same time, Horn et al. (2007). Foley and Sugerman (2005) presented a traversal algorithm named, 'kd-restart', which is able to traverse a kd-tree without keeping a stack, making it much more efficient on the GPU.

2.7 Anti-Aliasing

Aliasing is the effect of digital computers not being able to represent a continuous signal, Glassner (1989a). One type of aliasing that should be considered in a ray tracer is called spatial aliasing.

Spatial aliasing, best described in Figure 10, is the effect of uniformly placed pixels, sampling a scene, Glassner (1989a). In ray tracing, the primary rays are directed through the middle of each pixel. No matter how high the resolution, the primary rays will never be able to sample every aspect of the scene, which will result in the staircase effect,

known as jaggies. Another problem with aliasing is that some detail within a scene may be missed out entirely. If a small object falls between two adjacent pixels, the rays might miss them, and not include them in the image, Crow (1977). Some anti-aliasing techniques exist which aim to reduce the visibility of jaggies.

2.7.1 Super-Sampling

Instead of directing a primary ray through the centre of a pixel, a technique called super-sampling, Whitted (1980), directs multiple rays through the pixel in a uniform manner. The pixel colour is set to the average of the colours found by the rays. This is effectively increasing the resolution, then downsizing it to reduce the effect of jagged edges, Glassner (1989a).

According to Glassner (1989a), super-sampling is a very expensive procedure. If nine primary rays are traced for each pixel, the rendering time for the image will be approximately nine times longer. This decrease in performance makes this approach unsuitable when striving for real-time performance.

2.7.2 Stochastic Sampling

The reason jaggies are so obvious to the viewer is because the image is sampled in a uniform manner. Stochastic sampling, Cook (1986), is a method which samples rays non-uniformly by applying some small random offset to the direction. The irregularity of the samples makes jaggies less obvious, ultimately improving the image quality. Although different from super-sampling, this technique can be combined with super-sampling for further improvements, Cook et al. (1984).

2.8 Parallel Computing

Parallel computing is the process of many calculations running simultaneously on a computer. Due to the computational independence of each ray, the ray tracer is easy to implement using a parallel architecture, Parker et al. (1999).

Consumer CPU hardware to date consists of a few cores that are highly powerful. The ray tracer can take advantage of this by dividing the image up into sections, and assign

each core of the CPU to render each section simultaneously. Splitting the render up into many sections and executing them in parallel greatly increases performance, which is ideal for real-time performance, Parker et al. (1999).

The GPU is a processing unit that consists of thousands of small, more efficient cores. Purcell et al. (2002), developed the first GPU ray tracer, which takes advantage of the enormous number of cores available by tracing many more rays simultaneously. Since Purcell's implementation, graphics programming languages such as CUDA and OpenCL have become available, which make GPU programming much more accessible.

3 Design and Implementation

This section details the design and implementation phase of the ray tracer. All of key areas in the literature review have been considered and implemented. Each sub section will follow the same general patten. That is a brief description of the method used and it's purpose, the design, followed by (if appropriate) mathematical equations, pseudo code, and figures to further illustrate the algorithm or the outcome.

3.1 Camera and Primary Rays

The camera used in this implementation is designed to provide the parameters of which the primary rays are built upon. It does so by following the pin hole camera model by defining a 2D image plane, and an origin vector in a Cartesian coordinate system. The primary rays are then shot from the camera's origin though each pixel in the image plane.

In order to calculate the primary ray's direction for each pixel in the 2D image plane, the xy coordinate of each pixel must be computed. To do this, the pixel index must be retrieved in a normalized form, [-1 1], using Equations 1 and 2. The addition of 0.5 on to the pixel position ensures the coordinate indicates the centre of the pixel.

$$PixelCoord_x = 2 * \frac{Pix_x + 0.5}{Width} - 1 \tag{1}$$

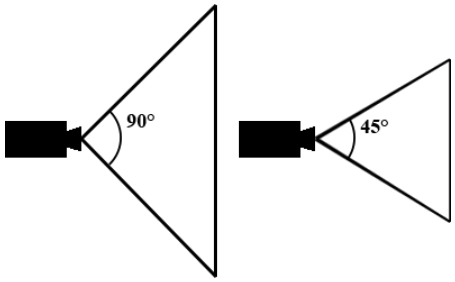


Figure 11: Pin-hole camera FOV

The camera with 90 degrees FOV has a larger image plane than the camera with 45 degrees FOV. The larger the image plane, the more of the scene is visible to the camera.

$$PixelCoord_y = 1 - 2 * \frac{Pix_y + 0.5}{Height} \tag{2}$$

However, this is only suitable for square screens, if the dimensions are not of equal size, the pixels will become stretched. To fix this, the coordinate of the largest image plane dimension will need to be scaled by the aspect ratio, Equation 3.

$$PixelCoord_k * AspectRatio \tag{3}$$

Furthermore, the field of view can also be manipulated. The field of view is an angle, that defines the size of the image plane, Figure11. The larger the image plane, the more of the scene can be seen at one time. Equation 4, finds the length between the centre of the image plane to the edge, given a FOV angle.

$$tan(\frac{fov}{2}) \tag{4}$$

Like the aspect ratio, Equation 3, this acts as a scaling factor, and is applied to the x and y pixel coordinates.

Now that the image plane x and y pixel coordinates have been found, all that is left is to compute the z coordinate. Since the image plane is positioned in front of the origin, the z coordinate can be set to 1 unit in front of the camera's origin. A direction can now be given by subtracting the pixel coordinate by the camera's origin.

Algorithm 1 PrimaryDirection(Direction P, Camera C, Texcoord xPos, yPos) **return** void

1: $xPix = (xPos + 0.5) * C_{inverseWidth}$
2: $yPix = (yPos + 0.5) * C_{inverseHeight}$
3: **if** $C_{width} > C_{height}$ **then**
4: $\quad xPix = (2 * xPix - 1) * C_{aspectRatio} * C_{FOVScale}$
5: $\quad yPix = (1 - 2 * yPix) * C_{FOVScale}$
6: **else**
7: $\quad xPix = (2 * xPix - 1) * C_{FOVScale}$
8: $\quad yPix = (1 - 2 * yPix) * C_{aspectRatio} * C_{FOVScale}$
9: $P = normalise(xPix, yPix, 1 - C_{origin})$
10: $P = P * C_{viewMatrix}$

Functionality has also been added to allow the camera to be moved around the scene. This is achieved by featuring three direction vectors of unit length, front, right and up. The front vector points from the camera origin and through the centre of the image plane. The up vector points upwards, perpendicular to the front vector, and the right vector points to the right and is perpendicular to the front and up vector. A view matrix is constructed from these vectors to express the direction the camera is looking at. Each of the primary ray directions are then rotated by the view matrix. An implementation of the primary ray direction calculation can be seen in Algorithm 1.

3.2 Ray Intersections

When tracing the rays throughout the scene, ray-primitive intersection tests are needed to determine if and where a ray hits an object. Two types of primitives were implemented in this application, spheres and triangles. The most efficient algorithms reviewed in the literature were chosen, in order to maximise performance.

3.2.1 Sphere

The geometric ray-sphere intersection test was chosen from, Haines (1989), as this was the most efficient algorithm when compared to the algebraic solution. The first step to determine whether a ray hits the sphere, is to find if the ray originates inside of the sphere. To do so, the squared distance from the ray's origin, to the sphere's origin is found, Equation 5 and 6.

$$rayToSphere = S_{orign} - R_{Origin} \tag{5}$$

$$rayToSphere2 = rayToSphere \cdot rayToSphere \tag{6}$$

If $rayToSphere2 > S_{Radius^2}$, then the ray lies outside of the sphere, and it can not yet be concluded if an intersection is occurring. If $rayToSphere2 < S_{Radius^2}$ it can be determined that the ray lies inside of the sphere, meaning a collision must occur.

In either case, the shortest distance between the sphere's origin and the ray direction must be calculated, Equation 7.

$$rayShortestDistance = rayToSphere \cdot R_{Direction} \tag{7}$$

If $rayShortestDistance < 0$, and the ray originates outside of the sphere, then the sphere is behind the ray, meaning that no intersection has occurred. If $rayShortestDistance > 0$, then the sphere is in front of the ray which indicates that a collision is possible.

The next step is to find the half chord, which is the distance between the point of the shortest distance and the sphere's surface, Equation 8. This can be found by computing:

$$halfChord = S_{origin} - rayToSphere^2 + rayShortestDistance^2 \tag{8}$$

If $halfChord < 0$ and the ray originates outside of the sphere, then a collision did not occur and the algorithm can be terminated. In all other cases, it can be concluded that a collision has occurred.

Now, all that is needed is to calculate t, which is the shortest distance along the ray that intersected the sphere. For rays that originate inside of the sphere, t can be found

using Equation 9.

$$t = rayShortestDistance + \sqrt{halfChord} \tag{9}$$

and rays that originate outside of the sphere, Equation 10.

$$t = rayShortestDistance - \sqrt{halfChord} \tag{10}$$

Now that an intersection has been found, and the distance along the ray where the closest intersection is known, all that is needed is to calculate the point of intersection, which is given using, Equation 11.

$$P = R_{Origin} + R_{Dir} * t \tag{11}$$

3.2.2 Triangle

The Möller and Trumbore (1997), ray-triangle intersection was chosen to be implemented within this ray tracer because of it's efficiency and calculation of barycentric coordinates. The barycentric coordinates represent a point on a triangle which can be used later in the program when interpolating colours, normals or texture coordinates, Section 3.10.

The first step of Möller's algorithm is to determine whether the ray intersects the plane that the triangle lies in. To do this, the determinant of the ray direction and the triangle's plane is calculated, Equation 12. *pvec* is given using Equation 13, and the edges can be calculated using Equations 14 and 15, where $V0$, $V1$ and $V2$ are the triangles vertices.

$$det = edge1 \cdot pvec \tag{12}$$

$$pvec = R_{Dir} \times edge2 \tag{13}$$

$$edge1 = V0 \cdot V1 \tag{14}$$

$$edge2 = V0 \cdot V2 \tag{15}$$

If $det \equiv 0$, then the ray is parallel to the triangle's plane, meaning that no intersection occurs and the algorithm can be terminated. Since no back face culling is being performed in this ray tracer, it does not matter if the determinant is negative.

The next step is to compute the barycentric coordinate, u, Equations 16 and 17.

$$u = \frac{tvec \cdot pvec}{det} \tag{16}$$

$$tvec = R_{Orgin} - V0 \tag{17}$$

If $0 < u < 1$, does not hold true, then the algorithm can be terminated as the ray does not fall inside of the triangle. If this test is passed successfully, then the coordinate, v needs to be calculated, Equations 18 and 19

$$v = \frac{qvec \cdot pvec}{det} \tag{18}$$

$$qvec = tvec \times edge1 \tag{19}$$

If $0 < v < 1$ and $0 < u + v < 1$, the ray must pierce the triangle at the position uv. If this condition is not met, then the ray does not intersect the triangle.

Lastly, the distance t, along the ray at the point of intersection is calculated, Equation 20.

$$t = \frac{edge2 \cdot qvec}{det} \tag{20}$$

If $t < 0$, then the intersection is behind the ray's origin, therefore no intersection has occurred. if $t > 0$, then the ray successfully intersects the triangle.

Like the sphere, the point of intersection can be found using, Equation 11.

Some optimisations can be incorporated into this algorithm to increase the performance. The first optimisation performed was to pre-compute the edges of the triangle, as these variables remain constant throughout the life time of the program. Another optimisation was to replace the costly divisions by the determinant, with a multiplication by the inverse determinant. The algorithm and optimisations can be viewed in Algorithm 2.

Algorithm 2 TriangleIntersection(Ray R, Vertex V0, Edge V0V1, V0V2, Scalar t, u ,v) **return** Boolean

```
1:  pvec = cross(R_dir, V0V2)
2:  det = dot(V0V1, pvec)
3:  if fabs(det) < EPSILON and fabs(det) > -EPSILON then
4:      return false
5:
6:  invDet = 1 / det
7:  tvec = R_dir - V0
8:  u = dot(tvec, pvec) * invDet
9:  if u < 0 or u > 1 then
10:     return false
11:
12: qvec = cross(tvec, V0V1)
13: v = dot(R_dir, qvec) * invDet
14: if v < 0 or u + v > 1 then
15:     return false
16:
17: t = dot(V0V2, qvec) * invDet
18: if t < 0 then
19:     return false
20: return true
```

3.3 Data Structure

A data structure is needed to contain the scene's geometry to eliminate unnecessary intersection tests. The data structure chosen was the kd-tree, Bentley and Ottmann (1979), because it can be traversed quickly and efficiently, Wald and Havran (2006), which is ideal for a real-time ray tracer.

When building the tree, Surface Area Heuristics (SAH) have been used to find the optimal splitting plane, which maximises empty space close to the root. This allows for more early exits when a ray is traversing the tree, resulting in faster performance.

Algorithm 3 Build(Primitives P, Voxel V) **return** Node

```
1: if TerminationCritertia then
2:     return new Node(P, V)
3: plane = findSAHSplittingPlane(P, V)
4: (V_L, V_R) = splitVoxel(plane, V)
5: (P_L, P_R) = splitPrimitives(plane, P)
6: return new Node(plane, Build(P_L, V_L), Build(P_R, V_R))
```

To build the kd-tree, Algorithm 3 is used to recursively partition space until only one primitive is left, or some threshold is met to prevent the tree from becoming too large. The space is partitioned by finding a splitting plane using SAH, Wald and Havran (2006). Once the splitting plane has been found, the voxel of the current node is split on the axis and the position of the splitting plane. Next, the primitives are sorted into two lists. One list for the primitives to the left of the splitting plane, and the other for the primitives to the right. This function is then called recursively, once for the primitives and voxel to the left, and once for the primitives and voxel to the right. The result will be a tree of nodes which sub divide space into smaller subsections.

As this implementation of the ray tracer is utilising the power of parallel computing on the GPU, Section 3.11, the traditional stack based tree traversal approach is not feasible as a stack will have to be stored for every thread simultaneously. Therefore, the stackless traversal algorithm, kd-restart, Foley and Sugerman (2005) was implemented as it is designed specifically with the GPU's memory limitations in mind. Instead of storing a stack of nodes, the kd-restart algorithm tracks the minimum and maximum point along the ray during the traversal, and increments them until either an intersection is found, or there are no more nodes left to check. This traversal algorithm is better illustrated in Figure 12. Algorithm 4 demonstrates the process of traversing the structure.

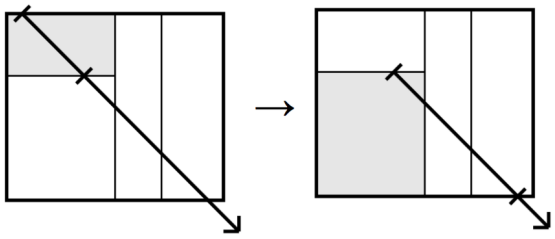


Figure 12: Kd-restart traversal, Foley and Sugerman (2005)

No intersection was found in the first node, therefore tmin and tmax are advanced further across the tree.

Algorithm 4 KDSearch(Ray R, Node root) **return** Boolean

```
1:  if (globaltMin, globaltMax) = IntersectRootVoxel(R, root) then
2:      tMin ← globaltMin, tMax ← globaltMin
3:      t ← ∞
4:      while tMax < tMin do
5:          node = root
6:          tMin = tMax, tMax = globaltMax
7:          while !node_isLeaf do
8:              k = node_planeAxis
9:              tSplit = (node_planePos - R_origin) / R_dir_k
10:             (first, second) = sortNodes(node_Left, node_Right, R)
11:             if tSplit ≥ tMax or tSplit < 0 then
12:                 node = first
13:             else if tSplit ≤ tMin then
14:                 node = second
15:             else
16:                 node = first
17:                 tMax = tSplit
18:         t = intersectNodePrimitives(R, node)
19:         if t ≤ tMax then return True
20: return False
```

3.4 Phong Illumination Model

When an intersection between the ray and an object is found, the Phong illumination model, Phong (1975), is used to calculate the colour of the light sources effect on the surface. The Phong model incorporates ambient lighting, diffuse reflection, specular reflection and attenuation. Each object in the scene has a material property associated with it which features ambient, diffuse, and specular colours. These colours are mixed with the light sources; ambient, diffuse and specular colour to produce a final result.

3.4.1 Ambient lighting

The calculation of the ambient lighting is the simplest and cheapest of the three types of lighting. It's computed by multiplying the light source's ambient colour with the material's ambient colour, Equation 21.

$$A_{RGB} = L_{A_{RGB}} * M_{A_{RGB}} \tag{21}$$

3.4.2 Diffuse Reflection

The brightness of the diffuse reflection, also known as Lambertian reflection, is directly impacted by the position of the light source relative to the object's surface. The diffuse colour will be at it's brightest when the light is directly in front of the surface, and at it's dimmest when the light is behind the surface. One way of computing the relative position is by finding the angle between the surface's normal, and the light direction, Equation 22. Where $\hat{N}$ is the surface normal and $\hat{L}$ is the light direction. The light direction can be computed, using Equation 23, where P is the point where the ray intersected the surface, and O is the position of the light.

$$cos\theta_L = \hat{N} \cdot \hat{L} \tag{22}$$

$$L = P - O \tag{23}$$

Similarly to the calculation of the ambient colour, the diffuse colour can be found by multiplying the light source's diffuse by the material's diffuse property, then the angle

between the surface and the light, Equation 24.

$$D_{RGB} = L_{D_{RGB}} * M_{D_{RGB}} * cos\theta_L \tag{24}$$

3.4.3 Specular Reflection

The brightness of the specular highlight depends on the angle between the incident direction and the reflected light direction. However, Blinn (1977) eliminates the need of calculating the costly reflection vector by computing the angle between the incident direction and a halfway vector. The half way vector is a direction vector that points in a direction that is between the light direction and the incident direction, Equation 25.

$$\hat{H} = \frac{\hat{L} + \hat{V}}{|\hat{L} + \hat{V}|} \tag{25}$$

The cosine angle between the halfway vector and the incident direction is raised by an exponent which determines how shiny the surface is. The greater the exponent, the smaller and tighter the specular highlight will become, resulting in a shinier looking surface. The final specular highlight colour can be calculated using Equation 26, where e is the exponent.

$$S_{RGB} = L_{S_{RGB}} * M_{S_{RGB}} * cos\theta_L^e \tag{26}$$

3.4.4 Attenuation

Light source attenuation has been incorporated into the project to simulate the strength of light over distance. Equation 27 calculates the attenuation factor where d is the distance from the light source to the surface, and K_c, K_l and K_q are constant, linear and quadratic terms that determine the strength of the light. Min is used to clamp the attenuated result from becoming greater than 1 to prevent the colours from becoming too bright.

$$F_{att} = min\left(\frac{1}{K_c + K_l * d + K_q * d^2}, 1\right) \tag{27}$$

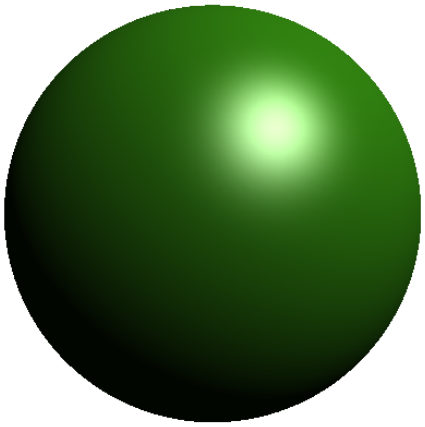


Figure 13: Phong illumination used when ray tracing a sphere

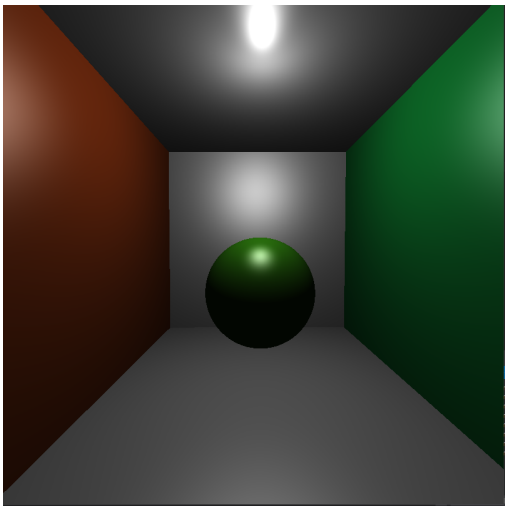
3.4.5 Final Illumination Result

To complete the Phong illumination model, the attenuation and each of the colours are combined to produce the final illuminated result, Equation 28. The result of the Phong illumination model in this implementation can be seen in Figure 13.

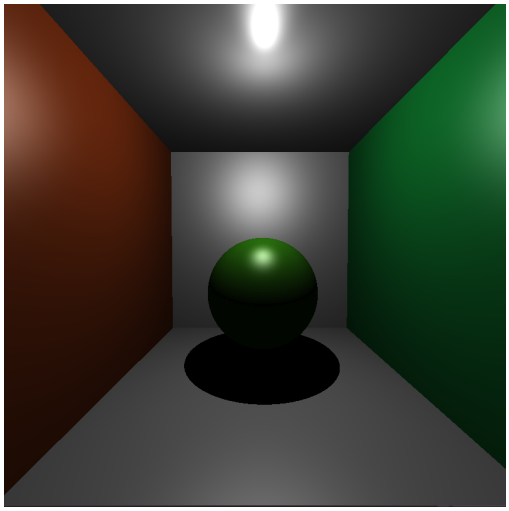
$$I_{RGB} = A_{RGB} + F_{att} * (D_{RGB} + S_{RGB}) \tag{28}$$

3.5 Shadows

Shadows are a result of an object blocking light from reaching a certain area. Therefore, before the illumination model is applied, some calculations need to be performed to determine if an object lies between the surface and the light source. This is done by casting a shadow ray, from the intersection point to the light source using Equation 23, and testing it for an intersection. If no intersection is found, then the surface is in full view of the light source, in which the illumination model can be applied. If an intersection is found between the shadow ray and an object, then it can be concluded that the light is blocked, in which only ambient lighting is applied to the surface. An implementation of this can be found in Figure 14, with a comparison between shadows on and off.



(a) Shadows off



(b) Shadows on

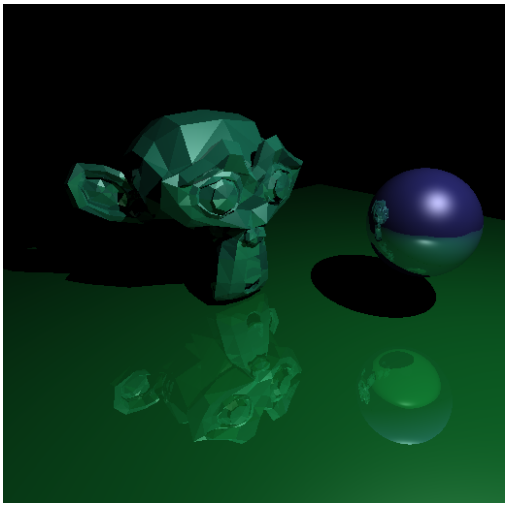
Figure 14: Ray Traced Shadows

3.6 Reflections

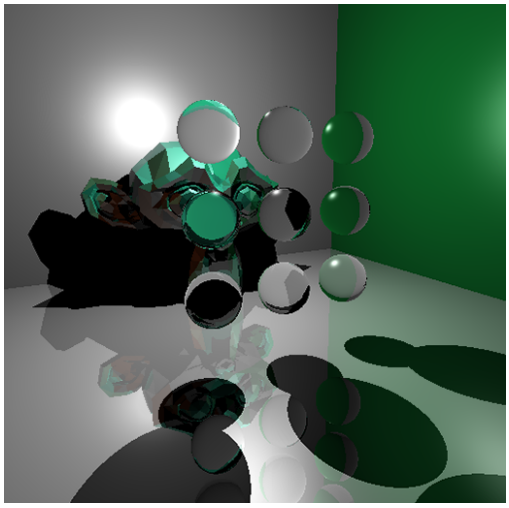
Reflections can be rendered in a ray tracer by tracing what is called a reflection ray. The reflection ray is created when a ray intersects a reflective object to calculate the reflected colour on the surface. The direction of the reflection ray can be given by abiding the law that the angle of reflection must equal the angle of incidence, Whitted (1980). To find the direction, Equation 29 can be used where R is the reflected direction, $\hat{I}$ is the direction of incidence, $\hat{N}$ is the surface's normal and $cos\theta_I$ is the angle between the incidence and normal vectors.

$$R = \hat{I} + 2 * cos\theta_I * \hat{N} \tag{29}$$

Since the reflected ray is cast from the position where the incident ray hit the object, the point of intersection is set as the reflected ray's origin. Now, the trace function is called recursively using this newly created ray to calculate the reflected colour. Before the colour can be applied to the final pixel colour, it must multiplied by some reflectance value specified by the material which determines how reflective the object is. An implemented result of reflections can be seen in Figure 15a.



(a) The reflection of the environment can be seen in the sphere and on the surface of the plane.



(b) The monkey and planes appear distorted in the transparent spheres. The refractive indices of the spheres are set to 1.4.

Figure 15: Implemented reflections and refractions

3.7 Refractions

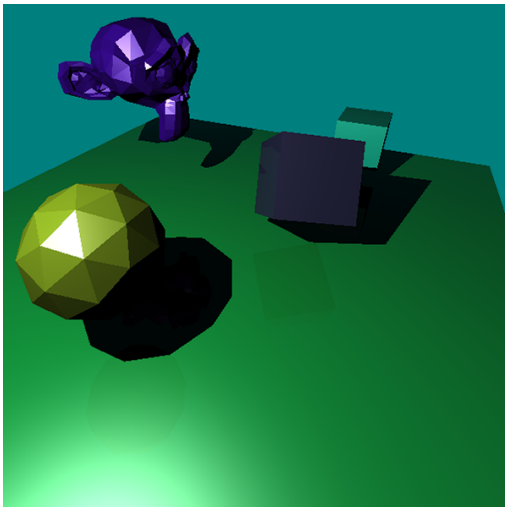
The refraction component of the ray tracer models transparent materials such as glass and water. Similarly to the calculation of reflections, the transmitted colour is computed by recursively tracing a transmission ray, however, the direction of the ray is calculated using Snell's law. Equation 30, gives the direction of the transmitted ray.

$$T = \frac{n1}{n2} * \hat{I} + \frac{n1}{n2} * cos\theta_I - \sqrt{sin^2\theta_T} * N \tag{30}$$

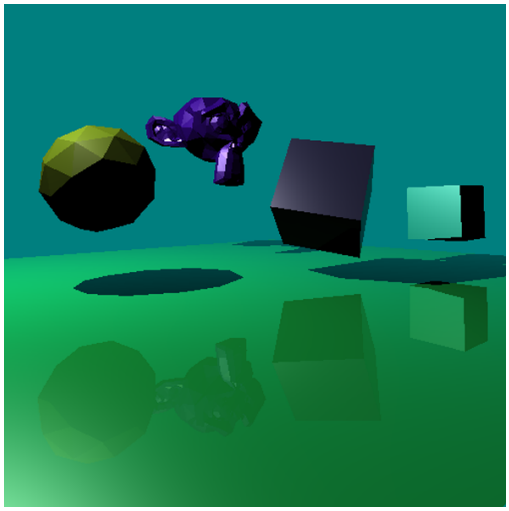
Where T is the transmitted direction, $n1$ is the refractive index of the current material, and $n2$ is the refractive index of the material that the transmitted ray is entering. $sin^2\theta_T$ is the result given using Snell's law, Equation 31.

$$sin^2\theta_T = 1 - \left(\frac{n1}{n2}\right)^2 * (1 - cos^2\theta_I) \tag{31}$$

Although, a material that has transparent properties is not always guaranteed to refract light. This is the result of total internal reflection, where the transmitted ray direction



(a) The reflections in the plane are weak.



(b) The colour of the reflections are stronger.

Figure 16: Reflected colours with Schlick's approximation

reflects off of the surface, instead of travelling though it. Total internal reflection is more likely to occur when $cos\theta_I$ is inclined towards 0 and can be detected when the result from Snell's law, $sin^2\theta_T$, is negative.

In this implementation, if total internal reflection occurs, the transmitted colour is not computed, and is skipped. An implemented result of refraction can be viewed in Figure 15b.

3.8 Fresnel Reflection

When light hits an object, the amount of light that is reflected and refracted alters based on the angle the light intersected the surface, this is known as Fresnel reflection. Schlick's approximation of the Fresnel factor, Schlick (1994), calculates the intensities of the reflected and refracted colours given the angle of incidence and the refractive indices. The intensity of the reflected colour is given using, Equation 32, and the intensity of the transmitted colour is given using Equation 34.

$$R\theta_I = R_0 + (1 - R_0) * (1 - cos\theta)^5 \tag{32}$$

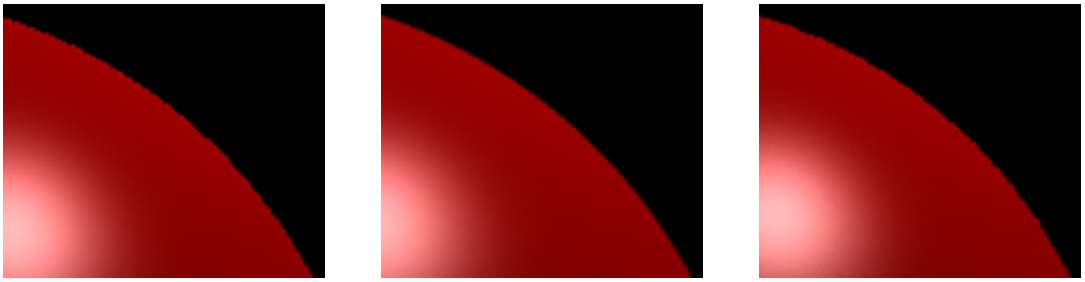
$$R_0 = \left(\frac{n1 - n2}{n1 + n2}\right)^2 \tag{33}$$

$$T\theta_I = 1 - R\theta_I \tag{34}$$

Figures 16a and 16b, show the strength of the reflected light from two different angles.

3.9 Anti-Aliasing

The brute force anti-aliasing technique super-sampling, and stochastic sampling have been implemented in to this project to reduce the jaggy effect, Figure 17a.



(a) No Anti-Aliasing (b) Super-samping (c) Stochastic Sampling

Figure 17: Anti-Aliasing Techniques

3.9.1 Super-sampling

Super-sampling effectively renders the image at a higher resolution, then downsizes it to fit the screen. This implementation saw a super-sampling anti-aliasing method which shoots four primary rays through each pixel. In Section 3.1, the primary rays are shot through the centre of the pixel by specifying a 0.5 offset to the x and y pixel coordinate. In order to shoot four rays through different regions of the pixel, different offsets are introduced for each primary ray traced. Figure 17b, demonstrates the effect of a super-sampling implementation.

3.9.2 Stochastic Sampling

This implementation of stochastic sampling traces two randomly offset primary rays per pixel. Instead of tracing four rays like super-sampling to reduce the aliasing effect, this method makes jaggies much less noticeable by tracing rays in a non-uniform manner. This is done by applying a random offset between 0 and 1 when calculating the coordinate of the pixel.

Although the results from this method are not as visually pleasing as the super-sampling method, it is much more efficient because of the number of rays traced. Figure 17c, shows the effect of stochastic sampling.

3.10 Texture Maps

This ray tracer has benefited from the use of texture maps. Textures are a good way of adding more realistic results due to the significant increase in detail. In order to map a 2D texture on to the face of a 3D object, texture coordinates are assigned to the vertices of the object that represent a segment of the 2D texture. Using the barycentric coordinates already calculated from ray-triangle intersection testing, the texture coordinates can be interpolated using Equation 35.

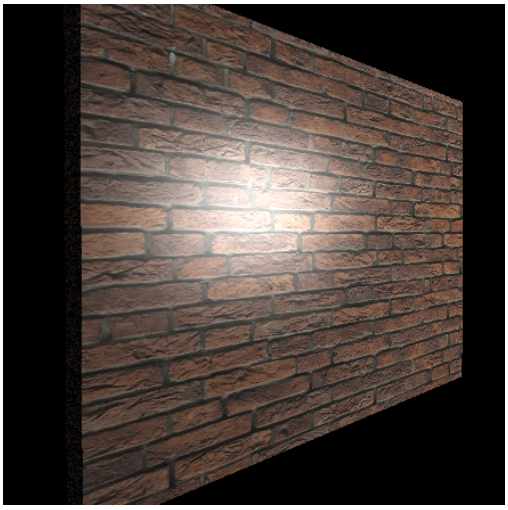
$$(u,v) = TO_{uv} + u*(T1_{uv} - T0_{uv}) + v*(T2_{uv} - T0_{uv}) \tag{35}$$

The sphere u v coordinates can be calculated using Equations 36 and 37.

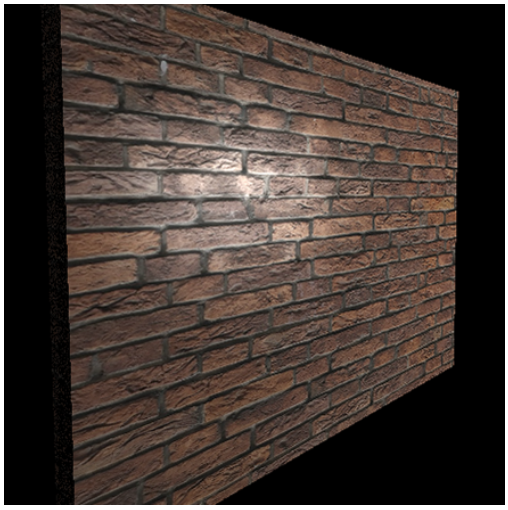
$$u = 0.5 + \frac{\arctan^2_{P_{XY}}}{2\pi} \tag{36}$$

$$v = 0.5 - \frac{\arcsin_{P_Y}}{\pi} \tag{37}$$

The texture coordinates given from these equations can be used to index a pixel colour from the diffuse, specular and bump map.



(a) A diffuse texture of a brick wall



(b) A combination of the diffuse and specular map.

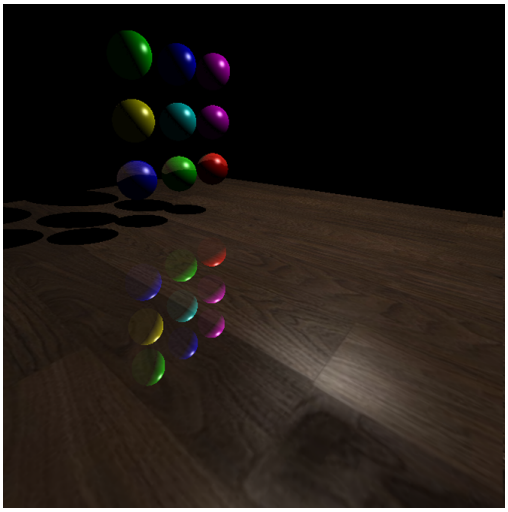
Figure 18: Diffuse and specular map

3.10.1 Diffuse Map

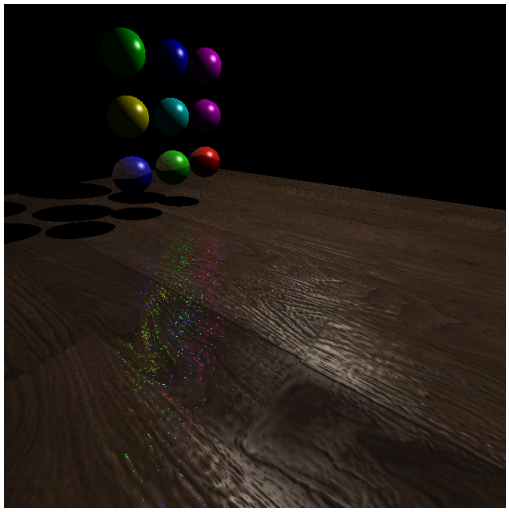
Diffuse maps are used in this ray tracer to give an object different colours in different regions for enhanced detail. The ray tracer picks out the RGB colour using the given texture coordinates, and assigns it as the diffuse colour. As seen in Figure 18a, the diffuse map has been used to map a brick texture on to a surface, which makes the object appear more realistic.

3.10.2 Specular Map

Specular maps are used to model areas with various reflective properties in the Phong illumination model. Each pixel in the specular map signifies a level of intensity, ranging from 0 (black), to 1 (white) within an RGB format. The appropriate specular intensity is retrieved using the texture coordinates, and is plugged into the illumination model. An implementation of the specular map, Figure 18b clearly shows an enhancement to the visual fidelity.



(a) The bump map turned off. Here the surface of the wooden floor appears smooth due to the perfectly reflected spheres.



(b) The bump map turned on. Here the surface appears uneven as the specular highlight and reflections are scattered.

Figure 19: Bump map comparision

3.10.3 Bump Map

Instead of specifying normals at the vertices of a triangle and interpolating them across it's surface, bump maps, Blinn (1978), can be used to model normals that point in various non-uniform directions. The normals in the texture are represented by a pixel that contains RGB values. The red colour signifies the X direction, green, Y, and blue, Z. Since the colours in a texture range between [0 1], the normals must be converted into the range [-1 1], using Equation 38.

$$N = 2 * RGB - 1 \tag{38}$$

This normal then needs to be converted from texture space, in to tangent space, which is a space local to the surface of the primitive. The tangent space is defined at each vertex by specifying a tangent, bi-normal and normal vector, which represent the XYZ axis of this space. These three vectors are interpolated across the polygon and used to create what is called a TBN matrix, which is a matrix that rotates the normal retrieved from

the bump map into tangent space, Equation 39.

$$N = N * TBN \tag{39}$$

where

$$TBN = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \\ N_x & N_y & N_z \end{bmatrix} \tag{40}$$

T_{xyz}, B_{xyz} and N_{xyz} represents the interpolated tangent, bi-normal and normal vectors.

This method benefits the ray tracer in two ways. The first being an illusion of increased geometry, because the normal displacement affects the objects illumination giving the impression of more triangles. The next, is simplifying a mesh by decreasing its triangle count and relying on the bump map to maintain the detail, resulting in better real-time performance. Figure 19a and 19b demonstrate the impact a bump map has on the appearance of a surface.

3.11 Cuda

The parallel programming language, CUDA was used to accelerate the ray tracer by taking advantage of the GPUs many core architecture. Since the colour of each pixel is calculated independently from all of the other pixels, the hundreds of threads on the GPU can be allocated to a pixel in the image to trace simultaneously.

3.12 Final Ray Trace Result

The final result saw an implementation of a real-time ray tracer that achieves all of the effects demonstrated in this section. The following four figures: 20, 21, 22 and 23 showcase the final renderings produced by this implementation. The performance achieved with these scenes will be explored in the following section.

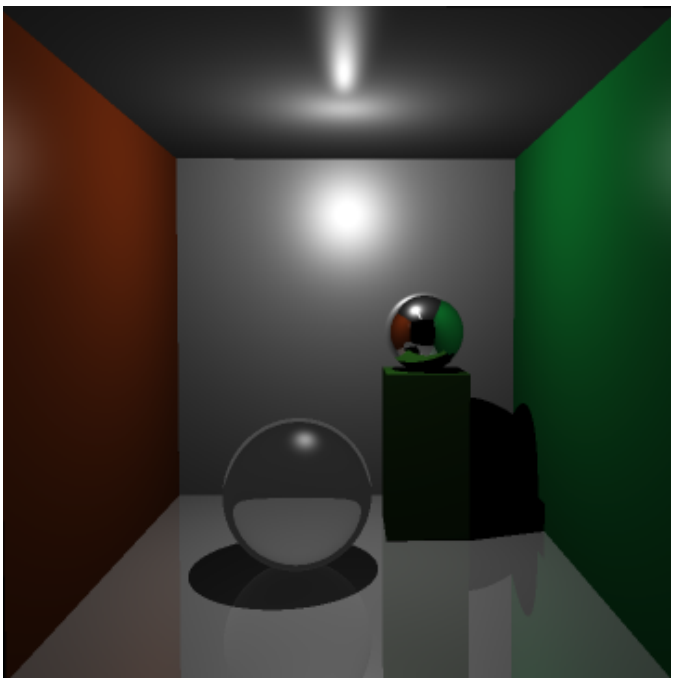


Figure 20: Cornell Box Scene

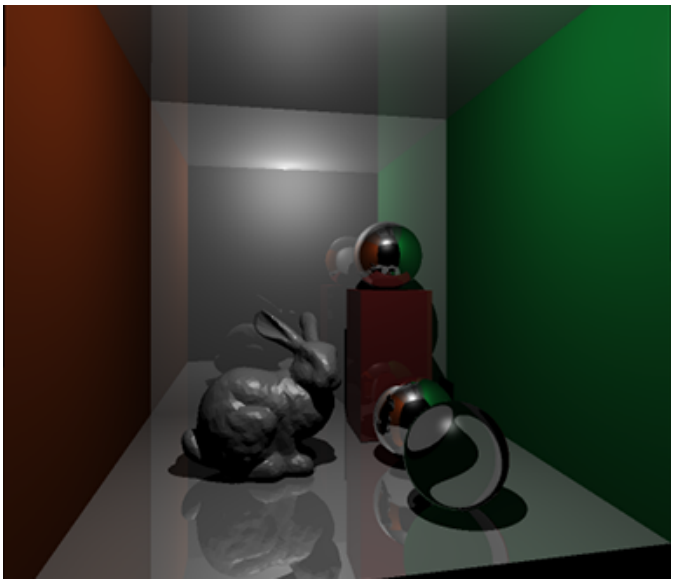


Figure 21: Stanford Bunny Scene

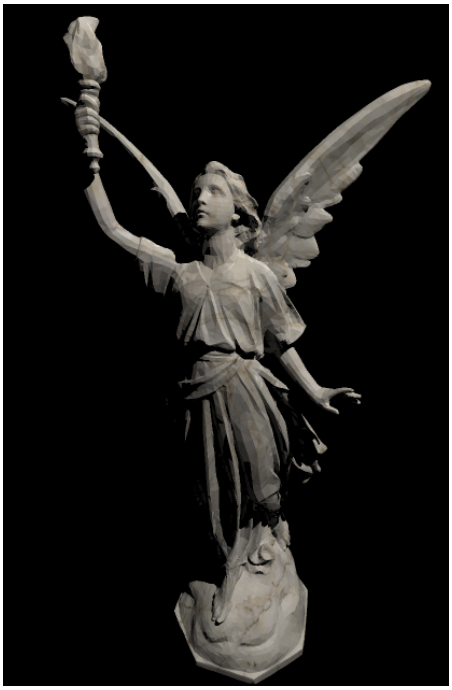


Figure 22: Lucy Scene

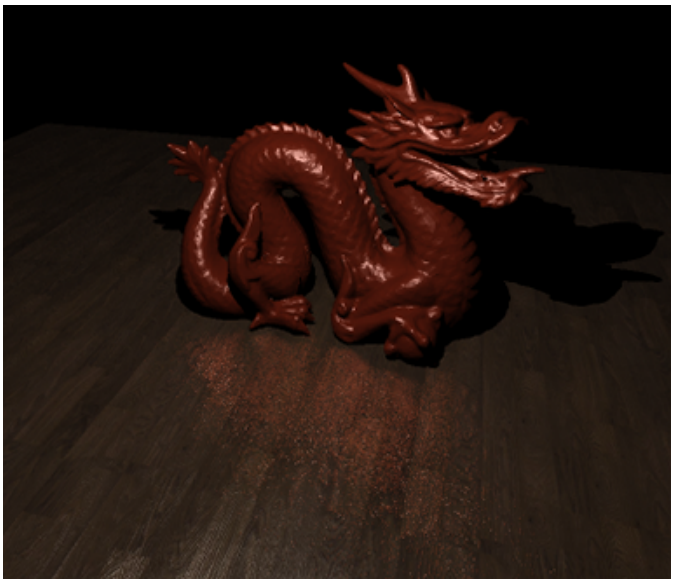


Figure 23: Dragon Scene

Scene	Description	No. Primitives
Cornell Box	The popular Cornell Box features five walls, two spheres and a box, Fig 20.	26
Stanford Bunny	The Stanford bunny model, taken from the Stanford model repository, Fig 21.	4,968
Lucy	A model of a Christian angel from the Stanford model repository, Fig 22.	33,446
Dragon	A model of a Chinese dragon from the Stanford model repository, Fig 23.	100,002

Table 1: List of test scenes

A list of the test scenes used in this section. The Stanford models were downloaded from, Stanford (2014).

4 Results and Performance Analysis

To test the performance of the real-time ray tracer, a series of benchmarks have been performed to identify the strengths and weaknesses in the key areas. The test scenes used, found in Table 1, were performed on consumer level hardware with an Intel i5 6600k CPU and an Nvidia GTX 970 GPU.

4.0.1 Optical Effects

The first test performed measures the frame per second (FPS), of each individual effect in the ray tracer, Table 2. The average and maximum FPS was captured with various anti-aliasing techniques at a resolution of 720x480 with one light source using the Cornell box scene. This particular test has been chosen to identify the performance implications of each optical effect.

As expected, the highest amount of FPS are achieved while rendering a scene using only Phong shading with no anti-aliasing. However, shadows appear to cause the most significant drop in the frame rate when rendered. This is possibly due to shadow rays being traced for every object hit, where as reflection and refraction rays are only traced for a subset of the objects. When all of the effects are rendered simultaneously, the FPS

	No Anti-Aliasing		Stochastic Sampling		Super-Sampling	
Effect	Avg	Max	Avg	Max	Avg	Max
Shading	38	41	18	19	10	11
Shadows	28	31	14	17	7	8
Reflections	31	32	15	18	8	10
Refractions	36	41	18	19	9	10
All	20	23	10	12	5	5

Table 2: Optical effects testing

The first column 'Effect' details the effect under test. The 'Avg', and 'Max' columns show the average and maximum frames achieved.

is approximately halved.

The stochastic anti-aliasing technique appears to roughly half the frame rate in all areas when compared to no anti-aliasing. This is to be expected as two primary rays, instead of one, are being traced per pixel. The super-sampling anti-aliasing technique continues to cut the FPS in half again, making it roughly 75% slower than no anti-aliasing methods, and 50% slower than stochastic sampling, this is because of the four primary rays being traced per pixel.

4.0.2 Kd-tree Splitting Plane

When building the kd-tree, spatial median and surface area heuristics were used to choose the position of the splitting plane. This test aims to identify the difference in performance between these two methods. The results, Table 3, show that the FPS while using SAH is consistently higher than the spatial median splitting. This is due to the SAH maximising empty space which means early terminations are more likely when the ray is traversing the tree. It can be seen that the number of nodes in the kd-tree while using spatial median splitting is much greater than the SAH tree. This is due to the termination criteria in the spatial median splitting being based on a depth and minimum primitive limit, where as SAH splitting is based on the cost probability of intersection tests. Given these results, the SAH approach is a better choice when striving

	Median			SAH		
Scene	Nodes	Avg	Max	Nodes	Avg	Max
Cornell Box	59	16	19	3	20	23
Stanford Bunny	30,113	6	6	21,577	12	15
Lucy	210,191	4	5	131,023	8	9
Dragon	512,144	1	1	420,689	3	4

Table 3: Kd-tree splitting plane testing

The first column 'Scene' details the scene under test. The 'Nodes' column highlights the number of nodes in the tree. The 'Avg' and 'Max' columns show the average and maximum frames achieved.

for real-time performance.

4.0.3 Resolution

This test is aimed to identify the performance differences of the ray tracer with varying resolutions. It was performed using the Cornell box test scene with one light source. From the results displayed in Table 4, it was expected to see a reduced number of frames as the resolution increased, due to the larger number of primary rays traced. For resolutions such as 720p and 1080p, the FPS is not high enough for it to be classed as real-time, whereas VGA and 480p resolutions are borderline real-time.

Table 4: Resolution testing

	No Anti-Aliasing			Stochastic Sampling			Super-Sampling		
Resolution	Rays	Avg	Max	Rays	Avg	Max	Rays	Avg	Max
640x480 (VGA)	0.31M	27	29	0.61M	15	16	1.23M	6	7
720x480 (480p)	0.35M	20	23	0.69M	10	12	1.38M	5	5
1280x720 (720p)	0.92M	13	17	1.84M	8	9	3.69M	3	4
1920x1080 (1080p)	2.07M	8	8	4.15M	3	4	8.29M	2	3

The first column 'Resolution' details the resolution under test. The 'Rays' column highlights the number of primary rays traced. The 'Avg' and 'Max' columns show the average and maximum frames achieved.

5 Future Work

Real-time ray tracing is a field of study that is becoming very important as it edges closer to replacing the classic rasterisation on consumer level hardware. The evidence gathered from this project indicates that achieving real-time performance in a ray tracer is a difficult task to accomplish, especially when rendering complex scenes. Future work should strive to accelerate the real-time performance by improving upon key bottlenecks, such as intersection testing. Further research in to data structures that decrease the number of traversals; and more efficient ray-primitive intersections algorithms will also address this problem. Other bottlenecks such as memory latency should also be considered. The use of global GPU memory used to store data like geometry should be replaced by faster storages such as shared, constant, or texture memory to improve memory read and write times. Load balancing on the GPU could be used in future work to increase performance by ensuring that all threads on the GPU are always busy.

Other future work that aims to improve the visual quality could also be considered, such as soft shadows. Soft shadows could be used to eliminate the ugly, hard edged shadows to make them appear more realistic. Depth of field could also be used to blur objects in the background for better realism and sense of depth.

6 Conclusion

The target of this project was to produce a ray tracer that is able to render images in real-time on consumer level hardware. The final result saw an implementation of a real-time ray tracer on the GPU using CUDA. The ray tracer supports spheres and triangles which can be used to represent models from an .obj file, and were contained in an SAH kd-tree data structure to accelerate intersection testing. The basic ray tracing optical effects were achieved such as shadows, reflection and refractions, as well as more advanced effects like light attenuation and Fresnel reflection. Further enhancements were made to increase the visual quality such as texture mapping for diffuse lighting, specular intensity and normals to simulate an uneven surface. Anti-aliasing techniques were also incorporated such as super and stochastic sampling to eliminate jaggies.

The results achieved in this project saw a performance of 23 FPS while rendering the Cornell box scene, and 15 FPS with the Stanford bunny (21k triangles) at a resolution of 720x480. Higher resolutions saw a drop in frame rate such as 17 FPS at 1280x720 and 8 FPS at 1920x1080 when rendering the Cornell box scene.

Real-time ray tracing is a growing field of research which will eventually replace the popular rasterised graphics technique. This project demonstrates that a ray tracer with real-time capabilities is possible on consumer level hardware thanks to the recent innovations in GPU many-core technology. As new efficient algorithms arise and consumer hardware becomes more powerful, real-time ray tracing is likely to replace rasterisation, and become the new standard in the near future.

References

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.

Arvo, J. and Kirk, D. (1989). A survey of ray tracing acceleration techniques. In Glassner, A. S., editor, *An Introduction to Ray Tracing*, chapter 6, pages 201–262. Academic Press Ltd., London, UK, UK.

Badouel, D. (1990). Graphics gems. chapter An Efficient Ray-polygon Intersection, pages 390–393. Academic Press Professional, Inc., San Diego, CA, USA.

Bentley, J. and Ottmann, T. (1979). Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, 28(9):643–647.

Blinn, J. F. (1977). Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198.

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

Commons, W. (2006). Illustration of the components of the phong reflection model (ambient, diffuse and specular reflection). File: Phong components version 4.png.

Cook, R. L. (1986). Stochastic sampling in computer graphics. *ACM Trans. Graph.*, 5(1):51–72.

Cook, R. L., Porter, T., and Carpenter, L. (1984). Distributed ray tracing. *SIGGRAPH Comput. Graph.*, 18(3):137–145.

Crow, F. C. (1977). The aliasing problem in computer-generated shaded images. *Commun. ACM*, 20(11):799–805.

Foley, T. and Sugerman, J. (2005). Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '05, pages 15–22, New York, NY, USA. ACM.

Fujimoto, A., Tanaka, T., and Iwata, K. (1988). Tutorial: Computer graphics; image synthesis. chapter ARTS: Accelerated Ray-tracing System, pages 148–159. Computer Science Press, Inc., New York, NY, USA.

Glassner, A. S. (1989a). An overview of ray tracing. In Glassner, A. S., editor, *An Introduction to Ray Tracing*, chapter 1, pages 1–32. Academic Press Ltd., London, UK, UK.

Glassner, A. S. (1989b). Surface physics for ray tracing. In Glassner, A. S., editor, *An Introduction to Ray Tracing*, chapter 4, pages 121–160. Academic Press Ltd., London, UK, UK.

Gouraud, H. (1971). Continuous shading of curved surfaces. *IEEE Trans. Comput.*, 20(6):623–629.

Haines, E. (1989). Essential ray tracing algorithms. In Glassner, A. S., editor, *An Introduction to Ray Tracing*, chapter 2, pages 33–78. Academic Press Ltd., London, UK, UK.

Horn, D. R., Sugerman, J., Houston, M., and Hanrahan, P. (2007). Interactive k-d tree gpu raytracing. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 167–174, New York, NY, USA. ACM.

Möller, T. and Trumbore, B. (1997). Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, 2(1):21–28.

Parker, S., Martin, W., Sloan, P.-P. J., Shirley, P., Smits, B., and Hansen, C. (1999). Interactive ray tracing. In *Proceedings of the 1999 Symposium on Interactive 3D Graphics*, I3D '99, pages 119–126, New York, NY, USA. ACM.

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.

Rubin, S. M. and Whitted, T. (1980). A 3-dimensional representation for fast rendering of complex scenes. In *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '80, pages 110–116, New York, NY, USA. ACM.

Schlick, C. (1994). An inexpensive BRDF model for physically-based rendering. 13(3):C/233–C/246.

Shirley, P. and Morley, R. K. (2003). *Realistic Ray Tracing*. A. K. Peters, Ltd., Natick, MA, USA, 2 edition.

Stanford (2014). The stanford 3d scanning repository.

Sutherland, I. E., Sproull, R. F., and Schumacker, R. A. (1974). A characterization of ten hidden-surface algorithms. *ACM Comput. Surv.*, 6(1):1–55.

Wald, I. (2004). Realtime Ray Tracing and Interactive Global Illumination.

Wald, I. and Havran, V. (2006). On building fast kd-trees for ray tracing, and on doing that in o (n log n).

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