University of Derby
School of Computing & Mathematics

A project completed as part of the requirements for

*BSc (Hons) Computer Games Programming*

entitled

# Real-time GPU Ray Tracing with Volumetric Effects


Caitlin Wilks
cat@wilks.so

in the years 2013-2014

# Abstract

The rendering of volumetric effects such as refraction presents great challenge for rendering on the GPU. As the modern GPU is highly specialised towards rasterisation, a method of rendering that only considers the surfaces of objects being rendered, it is challenging to simulate effects that need to consider the interior of volumes in real-time.

This paper investigates methods of real-time volume rendering in the GPU that are capable of considering these volumetric effects, as well as the flexible encoding of heterogeneous volumetric structures. We present a real-time GPU-based 3D renderer capable of considering volumetric effects using an octree structure and ray tracing.

The full project is available at http://github.com/Catchouli/VolumeRender.

# Contents

# List of Figures

# 1. Introduction

## 1.1 Volumetric effects

Many effects in the real world are volumetric in nature, such as fluids, clouds, fire, fog, and dust. A volumetric effect is one for which the inner volume has an influence on the lighting of an object, rather than just its surface. These effects are highly sought after in video games, but existing solutions often do not model them volumetrically, instead utilising crude approximations. Despite being fast to render using existing hardware, it is difficult to produce realistic results with these approaches.

To produce truly realistic volumetric effects, it would be ideal to model the volumes that cause them, and then be able to render them directly. Unfortunately, this is not a trivial problem to solve, especially in modern 3D hardware which is focused on rendering geometry rather than volumes.

Take for example clouds. In the real world, the lighting of clouds is not caused by the way that light hits a surface. Perhaps the most important factor that results in the appearance of clouds is the scattering of light as sunlight hits the atmospheric particles that make up the cloud. As light is scattered in many directions, including inside the cloud, it becomes important to consider the inner volume of the cloud rather than the geometry that describes its shape.

In order to understand the problems involved in rendering volumetric effects on 3D rendering hardware, it is important to understand the mechanism by which 3D rendering is accomplished on this hardware: rasterisation.

(a) Clouds (b) A water droplet

Figure 1.1: The lighting of clouds and water droplets is volumetric in nature

(a) A naked flame

(b) Dust

Figure 1.2: Other effects which must be considered volumetrically to be rendered physically are fire and dust

## 1.2 Rasterisation

Rasterisation is currently the most popular rendering algorithm for generating three-dimensional images in video games. It is also the purpose for which modern graphics processing hardware is heavily specialised.

The basic mechanism by which rasterisation functions is converting polygons to a raster image, a rectangular grid of pixels with each pixel having a defined colour. Typically, triangles are utilised, as any geometric shape, whether convex or concave, can be easily triangulated. Graphics hardware then has the simple job of processing these triangles to produce a raster image.

### 1.2.1 Polygon scan conversion

A common method of accomplishing rasterisation is polygon scan conversion. By considering a single scan-line of a polygon at a time, identifying the edges, and filling in the scan-line in between them, a raster image of the polygon can be created. As in figure 1.3, this is trivial for a convex polygon such as a triangle, as there will only ever be two edges intersecting a scan-line.

Repeating this technique for every scan-line covered by the triangle produces a raster image of the polygon, as shown in figure 1.4.

### 1.2.2 Extension to 3D graphics

This technique is then extended to 3D graphics using a projection calculation which maps 3D geometry to a position on the screen. During rasterisation, the pixels can also be shaded using attributes such as the position of the point relative to any lights, and the normal of the plane the point lies on to produce a convincing result as shown in figure 1.7.

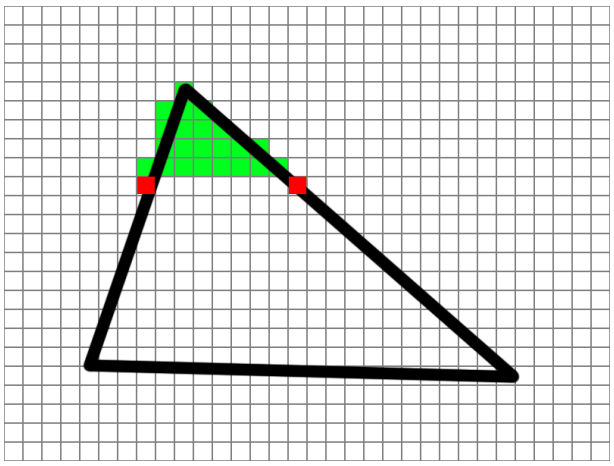


Figure 1.3: A triangle being rasterised. Green pixels have already been filled, while red pixels are the edges that have been identified for the current scanline

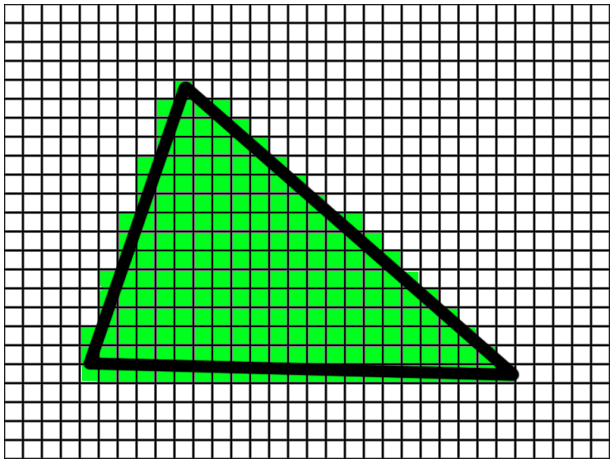


Figure 1.4: A polygon that has been rasterised by polygon scan conversion

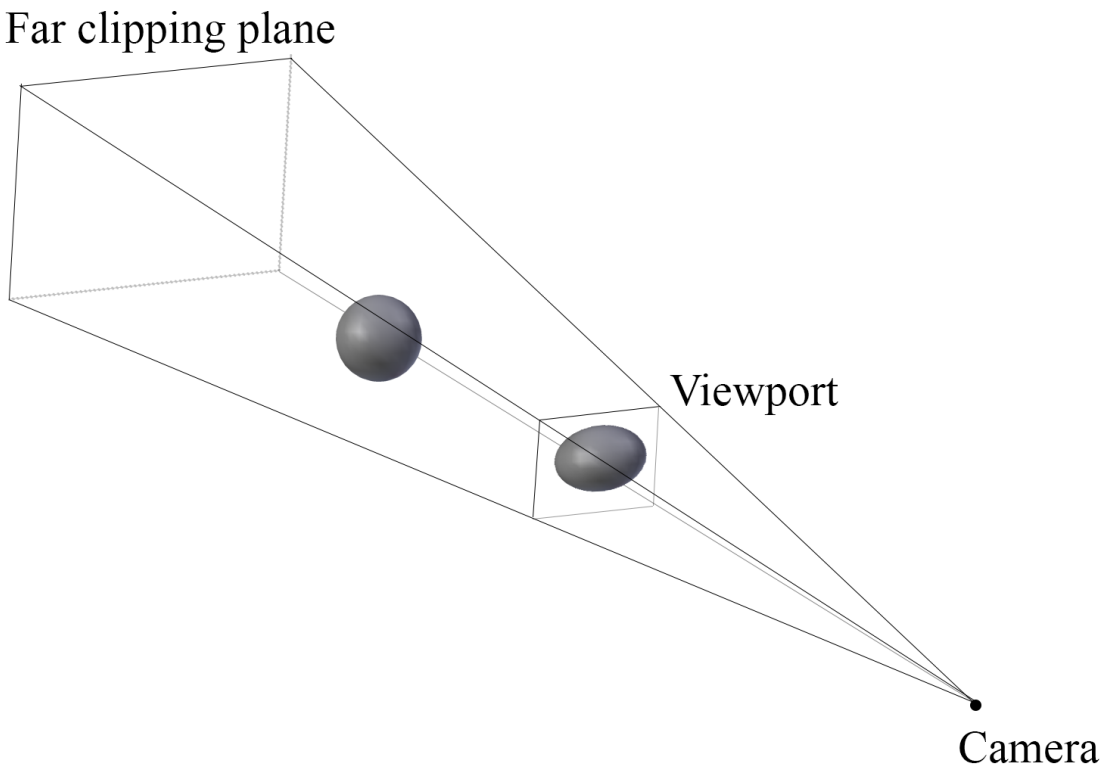



Figure 1.5: An illustration of how 3D objects are mapped to the screen using a perspective projection calculation. A 3D sphere is being rendered to the 2D viewport

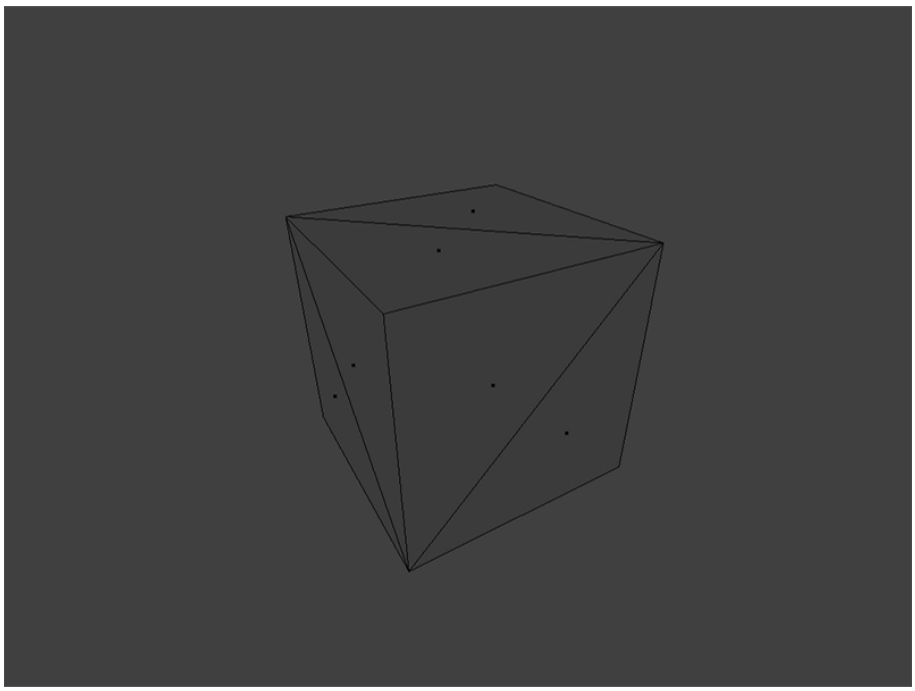


Figure 1.6: A demonstration of how a cube is triangulated

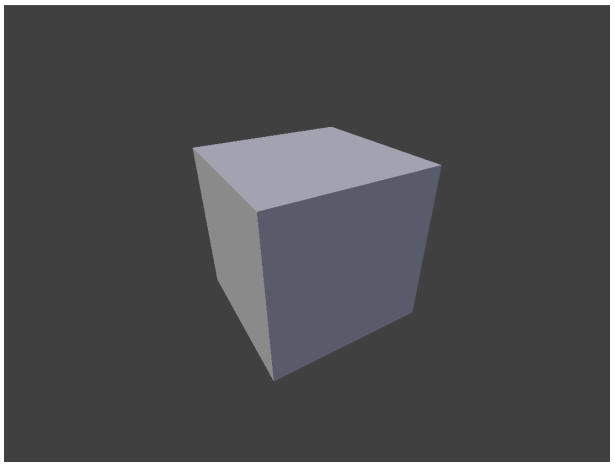


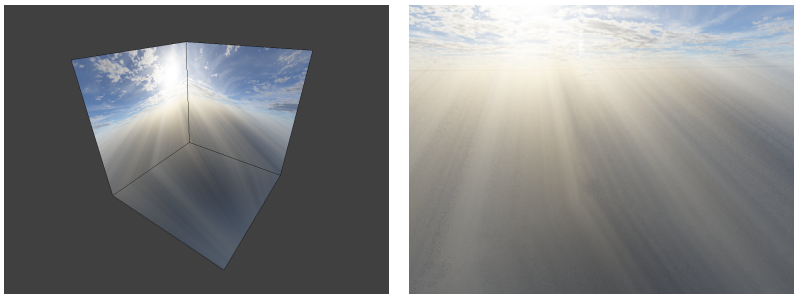
Figure 1.7: The result image once a cube has been rasterised

### 1.2.3 Triangulation

Triangulation is the process of dividing a surface into triangles so it can be easily rasterised. Triangles are convex in nature, and so are trivial to rasterise, (see section 1.2.1 on polygon scan conversion.) In addition, any flat geometric surface can be tessellated into triangles, and any curved surface can be reasonably approximated for the purposes of 3D rendering.

Take, for example, a cube mesh. A cuboid is made up of 6 rectangular faces. Each face can then be divided into two triangles.

The vertices of the cube are then transformed using a perspective projection, and the triangles are rasterised using polygon scan conversion.



(a) (b)

Figure 1.8: Mapping of clouds onto a skybox, and the result in-game

### 1.2.4 Limitations

Despite rasterisation producing convincing and, in some applications, even realistic results, it has a number of limitations when it comes to the rendering of volumetric effects.

For the purposes of rendering 3D volumes, the major limitation of rasterisation-based rendering is that it can only consider geometry. Therefore, it only considers the surfaces of objects that are being rendered, and not the volumes bounded within those surfaces. Without considering the inner volume, it becomes very difficult to render these effects realistically, and impossible to render them physically.

Another key limitation of rasterisation is that it only considers reflection, and even then, only an approximation of the light reflected from the current polygon. Only the current polygon is ever considered during rasterisation, making global effects such as reflection and refraction within the scene impossible.

Despite these key limitations, there has been some success in the mapping of volumetric effects to rasterisation hardware.

### 1.2.5 Existing rasterisation-based approaches

#### Skyboxes

Many games render clouds by mapping a cloud texture onto a sky box or sphere, and then rendering this in the background as shown in figure 1.8. This structure then surrounds the player, and is typically locked in place relative to the player, such that the player can never get closer to or farther away from it. The sky texture is then projected onto this structure, providing the background imagery for a scene.

While this may be sufficient for some applications, for example when clouds are in the distant background, clouds rendered in this manner can never be anything more than 2D images. Take for example a flight simulator. Using this type of software, it would be expected that the user would be able to fly close to, or even inside clouds, but as sky boxes can only ever be background imagery, such techniques are inappropriate for this type of application. Even for games in which the player is not expected to be able to

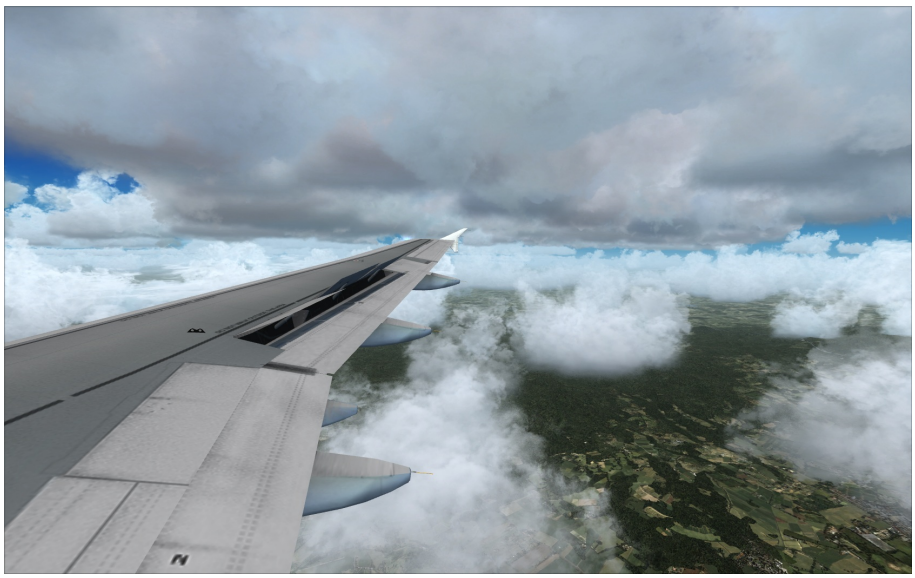
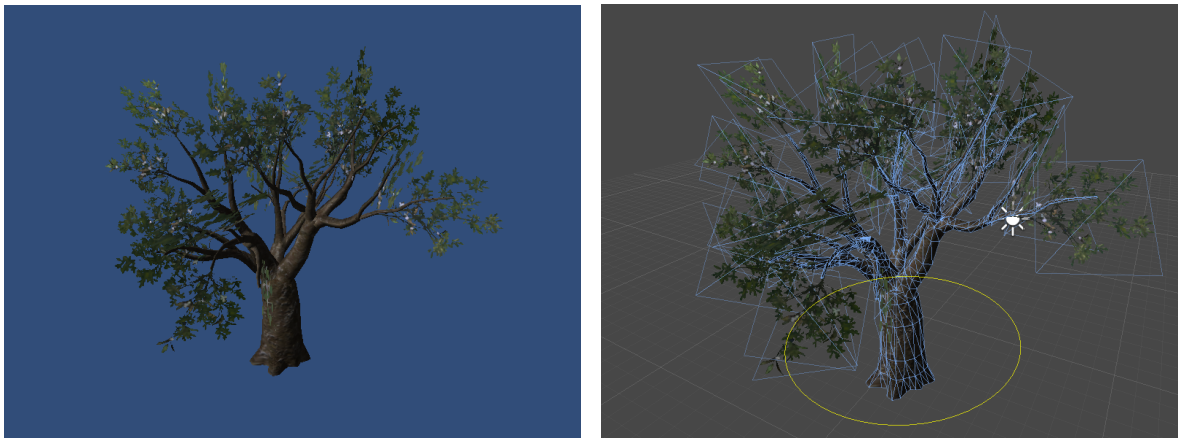


Figure 1.9: Clouds in Microsoft Flight Simulator X



(a) (b)

Figure 1.10: Use of texture-mapped quads to simplify a tree mesh. Each quad represents many leaves, greatly reducing the complexity when compared with other representations

reach the clouds, if the player is sufficiently close enough, the clouds should appear to move relative to each other, due to perspective or due to the effects of weather. Using a texture to represent the sky can not simulate this effect unless multiple layers are used, and even then this approach is extremely limited.

### Billboards

In 3D rendering, billboards are flat, texture-mapped planes that always face the camera. This allows them to appear at different scales and move relatively to each other based on the perspective of the camera. Billboards can be used to efficiently simulate a variety of volumetric structures, by overlapping a number of different images and blending them as appropriate to obtain desired appearance.

Billboards have been used effectively to simulate 3D volumetric clouds. Harris, 2002

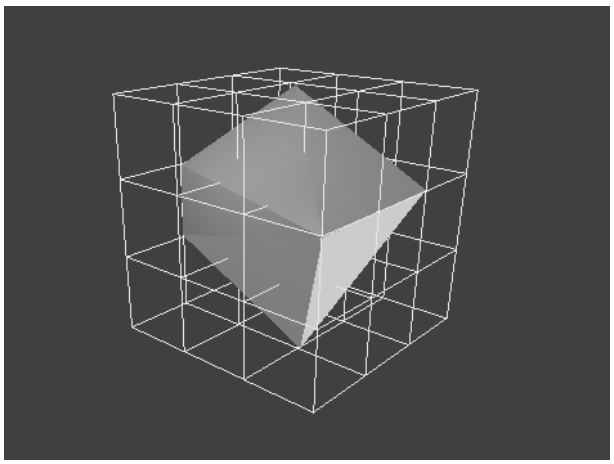


Figure 1.11: A volume sampled on a 3D scalar grid. The volume is sampled at the vertices of the grid, and the grid cells are then marched to produce a polygonal approximation of the volume

simulates the scattering of light within clouds, rendering the result to billboards which are then subtly used to represent the cloud volumes. As the billboards can be reused between multiple frames, this approach allows the realistic, physically based rendering of clouds.

This is a promising approach, and demonstrates the rendering of realistic looking clouds up close and far away as far away clouds can be reused. Harris's approach, however does not simulate global effects, so still falls short of the objective of simulating true volumetric clouds.

### Marching cubes

Marching cubes, a technique published in Lorensen and Cline, 1987, is a method of generating polygonal meshes from volumes in order to render them with standard polygon rasterisation hardware. Marching cubes extracts a polygonal approximation of an isosurface, a closed surface that separates the outside of a volume from the inside, defined by an isovalue for which all voxels with greater isovalues are inside the volume and all voxels with lower isovalues are outside.

The volume is sampled at each vertex of a regular grid, with each cell defined by its vertices and the isovalue of each vertex, as shown in figure 1.11. By comparing the isovalue of each vertex against the isovalue of the surface, and therefore determining which vertices lie inside, and which outside the volume, it is possible to create a polygonal approximation of the intersection of the isosurface with the current grid cell. Each grid cell is then considered individually, by "marching" from one cell to the next, in order to create a polygonal approximation of the volume. The detail level of the final mesh depends on the resolution selected for the scalar grid which is used to sample the volume.

The major advantage of this approach is that the resulting polygon mesh maps directly to existing rasterisation hardware, and is therefore as efficient to render as any

polygon mesh. However, the major disadvantage for the purposes of volumetric effects is that marching cubes only considers an isosurface, ignoring the inside of the volume and instead focusing on the rendering of the surface, making it inappropriate for considering volumetric effects.

#### Texture-based methods

Texture mapping has also been successfully used to map volume rendering onto rasterisation hardware. Engel and Ertl, 2002 utilises 3D textures representing slices, each of which represents a view-aligned slice through the volume. Overlapping samples are then composited using hardware alpha blending, in order to create a 3D image of the volume, considering the volume rather than just its surface.

The advantages of this method are that it maps onto traditional graphics hardware very well, resulting in extremely high performance on very common consumer hardware. Unlike approaches which only consider isosurfaces, such as marching cubes, this approach allows us to consider the internals of the volume by means of blending, meaning that it should be possible to model volumetric effects using this approach.

Unfortunately, this approach does not allow us to consider the effects of light that involve reflection, refraction, or scattering, as that would require changes in the direction of light, and this approach can only consider overlapping parts of the volume.

## 1.3 Ray tracing

Another approach to volume rendering, that does not attempt to map volume rendering problems to rasterisation, is ray tracing. Ray tracing functions by following rays of light backwards from the eye and out into the scene, as shown in figure 1.12. By casting a ray outwards into the scene for each pixel on the screen, and shading that pixel in accordance with the properties of the hit surface, it is possible to produce a 3D image of the scene similar to that produced by means of rasterisation.

Once the primary ray has intersected the scene, secondary rays can be spawned to determine the contribution of light from reflection, refraction, and also to check whether the path to each light in the scene is occluded in order to determine the contribution of that light. By spawning these rays, ray tracing can easily produce realistic global reflection and refraction within the scene.

Another advantage of ray tracing over rasterisation is that it can be adapted to any data for which an intersection test with a ray can be devised. For example, the intersection of a ray with a volume stored within an array can be determined by stepping along the ray from the eye and stopping only when an intersection is detected or the volume is exited.

Once the ray-tracing algorithm has been adapted to consider the volumes, it is also possible to let it peer into volumes in order to consider the interior of the volume when

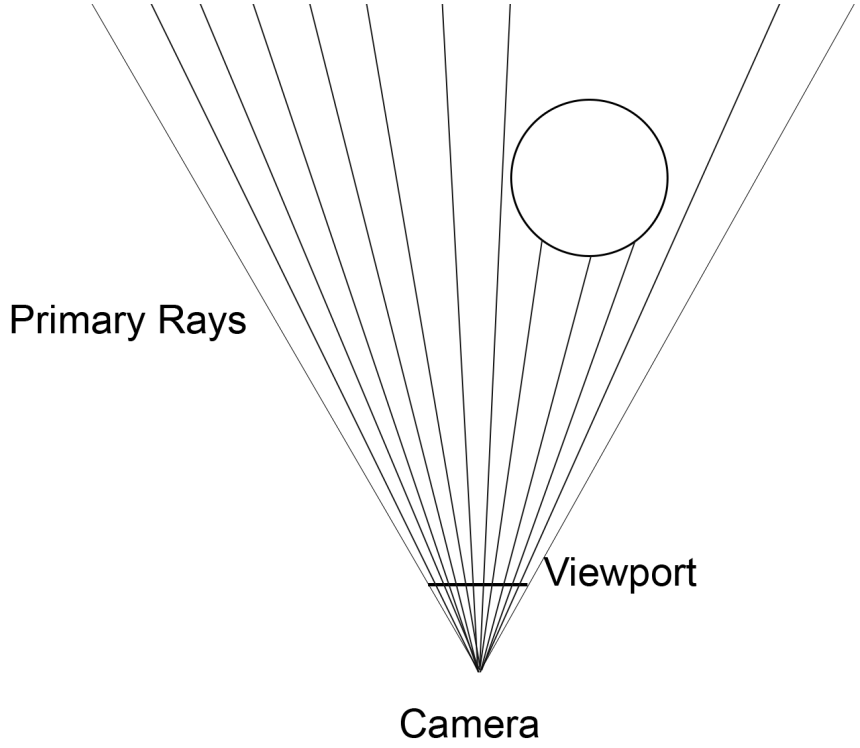



Figure 1.12: Primary rays being cast into the scene to determine intersection with the sphere. A primary ray is spawned for every pixel in the viewport

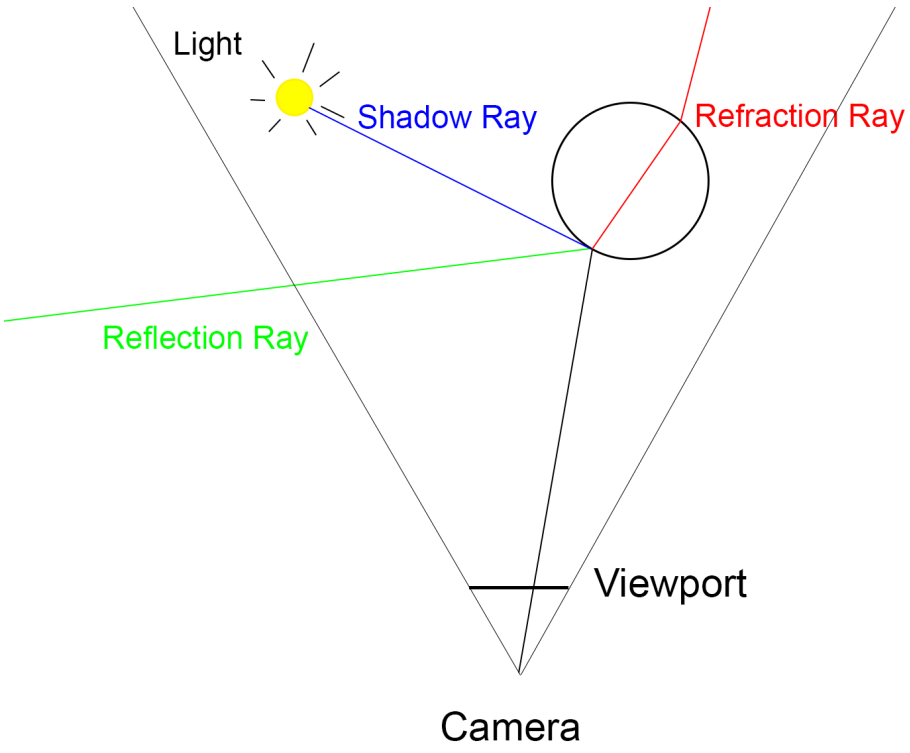



Figure 1.13: Once primary intersection is determined, secondary rays can be spawned in order to determine reflection, refraction, and shadow coverage

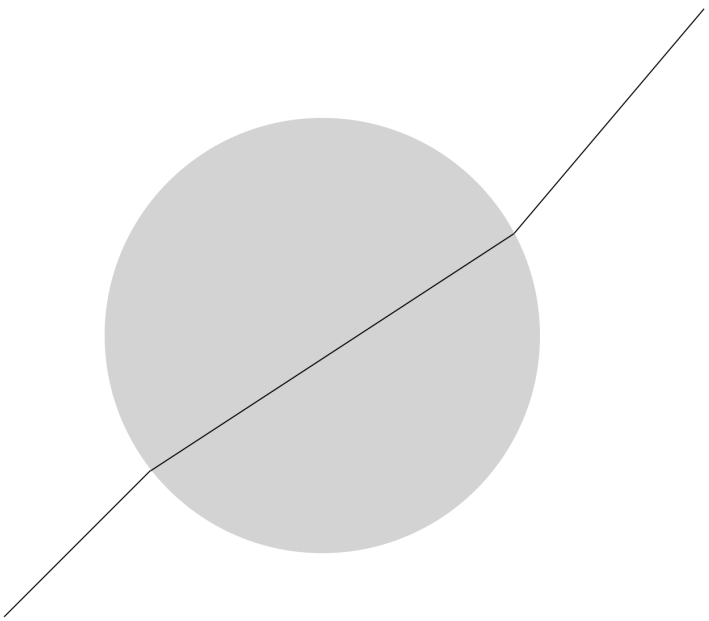


Figure 1.14: Ray traced refraction through a homogeneous volume

calculating the colour of the final pixel on the screen. In this way, it is similar to the approach used in Engel and Ertl, 2002, in that overlapping parts of the volume can be considered.

For example, as in figure 1.14, a volume with a homogeneous index of refraction would be possible to consider by just considering the difference in index of refraction at the surfaces, and therefore would be possible to model using geometry alone.

On the other hand, modelling refraction through a volume with a varying index of refraction, as in figure 4.8, this approach would quickly become infeasible for volumes of high heterogeneity. By allowing the ray tracing algorithm to consider the volume involved, rather than just the geometry, volumes high in heterogeneity can also be modelled, allowing complex visual effects to be simulated.

In order to do this in real-time, however, it is necessary to parallelise this process in order to render at a reasonable resolution to allow image quality as high as that of rasterisation.

## 1.4 General-purpose processing on the GPU

Around the turn of the century, the traditional graphics pipeline became more flexible with the introduction of the programmable graphics pipeline. This has turned the GPU into a massively parallel SIMD (single instruction, multiple data) processor, capable of performing highly parallelised computation by running the same instructions on many pieces of data at once.

With the emergence of general purpose programming languages such as OpenCL and

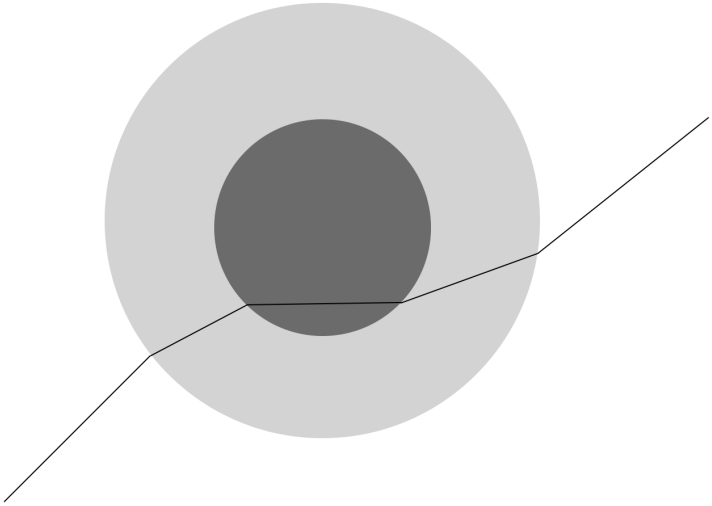


Figure 1.15: Ray traced refraction through a heterogeneous volume

CUDA, the GPU has become a general purpose processing tool capable of processing any parallelisable algorithm extremely quickly, redefining the problem of volume rendering on the GPU from mapping volume rendering problems to fixed rasterisation operations, to the utilisation of the GPU's massively parallelisable SIMD computation model.

It turns out that ray tracing maps very favourably to this model, as each ray must do roughly the same computation, but with different input data. By processing these rays on the GPU, for example, every pixel of the screen could be considered in parallel, allowing high screen resolutions to be considered without greatly increasing the time of computation.

## 1.5 Storage

Another challenging problem in volume rendering is representing true heterogeneous volumes. Naive volume storage approaches using arrays typically store data at a fixed resolution. The problem with this approach is that, in order to store any of the volume at a high enough resolution to represent it accurately, the whole volume must be stored at that resolution. For complex scenes, this can very quickly become prohibitively expensive.

In addition, it is also unnecessary. Large regions of the scene may be empty, homogeneous, or identical to other regions. With such an approach, these regions have exactly the same memory footprint as any other identically-sized region of the volume.

Ideally, we should be able to store any volume at a high enough resolution such that it can be represented accurately, without any unwanted visual artifacts. Numerous researchers have suggested approaches that avoid this problem by using hierarchical data

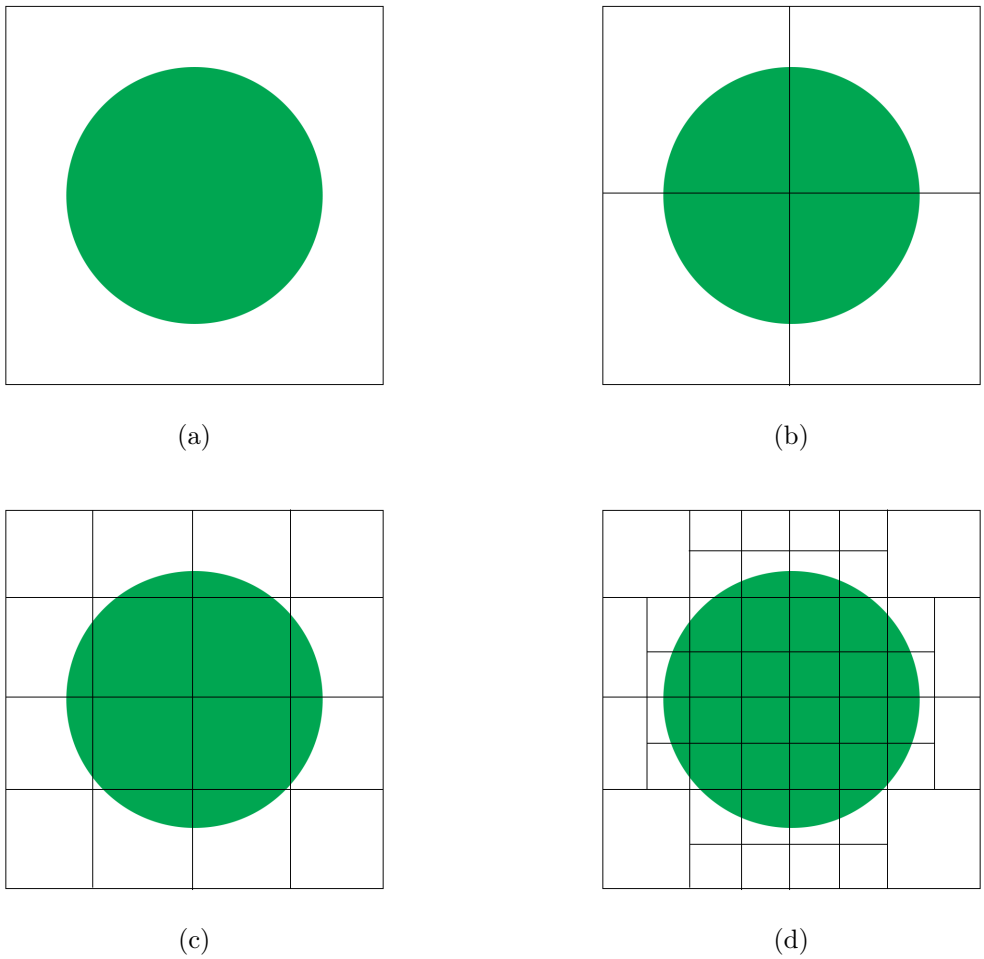


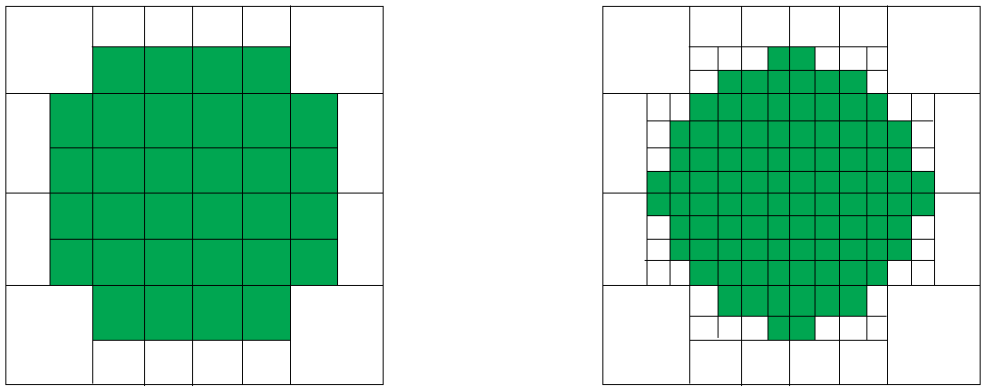

Figure 1.16: The construction of a hierarchical volume representation. As demonstrated by (d), empty nodes are not divided further

structures.

### 1.5.1 Hierarchical data structures

Hierarchical, or tree, data structures, are structures for representing 3D volumes that start by encoding data at a low resolution, covering large areas of the volume, and becoming higher resolution the deeper one traverses into the hierarchy. Once the encoded region is a sufficiently good approximation of the original volume, the hierarchy does not need to go any lower.

An additional advantage of this type of structure is that one does not need to encode areas of empty space, and during traversal, large areas of empty space can be skipped at the more coarse levels of the hierarchy. The construction of such a structure has been demonstrated in figure 1.16.



(a) (b)

Figure 1.17: Rendering the leaves of the tree results in an approximation of the volume. As the number of subdivisions is increased, the approximation improves

## 1.6 Objectives

Our primary objective is to develop a renderer that can simulate true volumetric effects by means of ray tracing. This renderer should be able to handle heterogeneous volumes as well as homogeneous volumes, taking into account global effects, where the rest of the scene can affect the rendering of any one part.

In addition, this renderer should be able to function with real-time performance on ubiquitous consumer-level graphics hardware. To accomplish this, we intend to utilise general purpose GPU programming and develop a highly parallelisable method of rendering that utilises the SIMD nature of graphics hardware.

Our renderer also needs to support a compact volume format which is capable of representing highly heterogeneous regions as well as homogeneous regions, with a small enough memory footprint to fit in GPU memory. In order to accomplish this we intend to take advantage of hierarchical data structures which allow us to store volumes of varying resolution and avoid storing empty space.

# 2. Literature Review

## 2.1 Rasterisation rendering

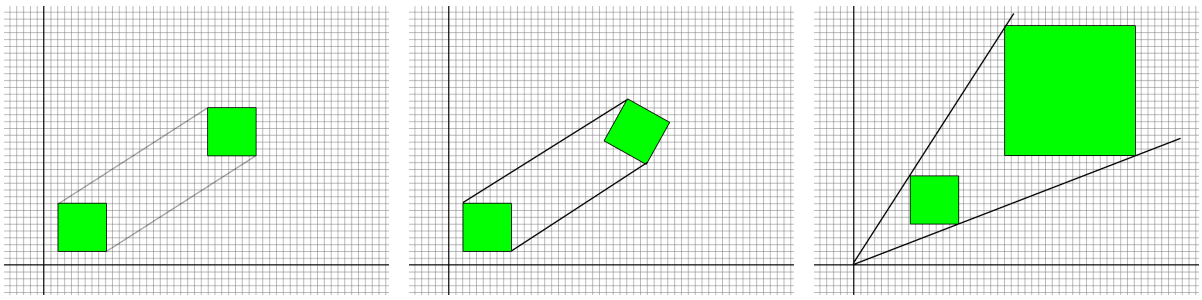
Modern graphical hardware is heavily specialised for the tasks involved in rasterisation. Rasterisation refers to the conversion of mathematically defined vector geometry to a raster image on screen.

There are two main operations involved in rendering using rasterisation: transformation, the act of converting geometry defined in 3D space to coordinates on the screen, taking into account perspective; and rasterisation itself, the act of converting this transformed vector geometry into a raster image for display. These operations, and the operations required to complete them, are the purpose for which modern graphics hardware is specialised.

### 2.1.1 Transformation

In graphics, the term transformation refers to the mathematical mapping of one set of coordinates to another. The main transformations involved in 3D rendering include translation, rotation, scaling, and projection. Translation is defined as a transformation which offsets a point by a certain amount in each axis, whereas rotation is a transformation which rotates a point around the origin by some angle in some axis. Scaling is a transformation which increases the distance from 0 in each axis by a multiplier.

Projection is a transformation which describes the mapping of three-dimensional points onto a two-dimensional plane. In order to give the rendered image perspective, we use a



(a) Translation (b) Rotation (c) Scaling

Figure 2.1: Geometric transformations

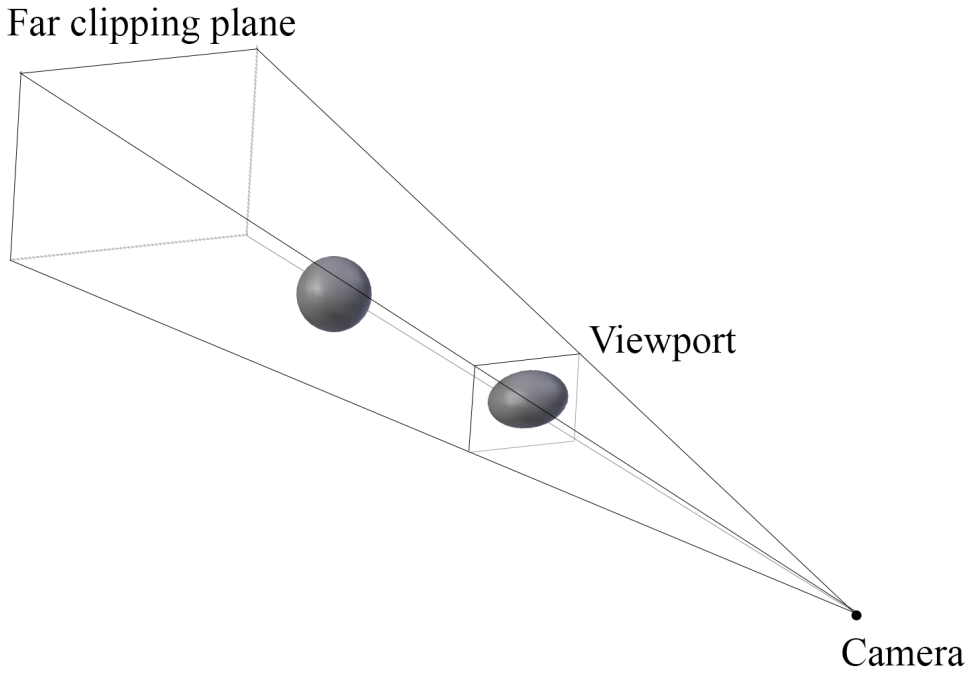



Figure 2.2: A perspective projection transformation

perspective projection. A perspective transformation takes into account the field of view (FOV) of the viewer, as shown in figure 2.2, in order to provide a perspective-correct representation of the scene.

In rasterisation rendering, we aim to take vector values defining the "world space" (position within the scene) coordinates of the geometry of the surfaces we are attempting to render and map them to the screen such that it produces a 2D image of the 3D scene.

In order to accomplish this, we consider three main transformations: perspective projection, which is defined by a projection transformation; the model transformation, which uses translation, rotation, and scale transformations in order to represent the object's position in the scene; and the view transformation, which uses a translation and a rotation transformation in order to represent the camera's position and rotation in the scene.

We use 4x4 matrices to represent these transformations, as this allows us to represent perspective projection and translation transformations (as well as the other required transformations.) As the matrices are square, an added advantage that they can be multiplied together to result in a combined transformation known as a model-view-projection (MVP) matrix.

This overall transformation defines the transformation of points from world space to a position on the viewport, in "viewport space". This normalised coordinate space, often from -1 to 1 on both axes, where -1 and 1 are the extremities of the field of view, must then be scaled to their final coordinates on the screen, in "screen space", based on the dimensions of the viewport. Vertices outside of this range are outside of the field of view

of the projection.

This scaling can be described very simply. For a viewport space defined from -1 to 1 on both axes, where -1 represents the left and bottom of the viewport, and 1 represents the top and right of the viewport, this transformation can be done by adding 1 to both coordinates to get them in the range 0..2, dividing both coordinates by 2 to get them in the range 0..1, and multiplying by the dimensions of the viewport, to get the coordinates in screen space. This can be written algebraically as:

$$x_{screen} = (1 + x_{viewport}) \cdot \frac{width_{viewport}}{2} \tag{2.1}$$

$$y_{screen} = (1 + y_{viewport}) \cdot \frac{height_{viewport}}{2} \tag{2.2}$$

In different rendering frameworks the definition of the viewport can vary, but the principles are the same. The basic transformations required to make this conversion are simply a 2D translation and scale.

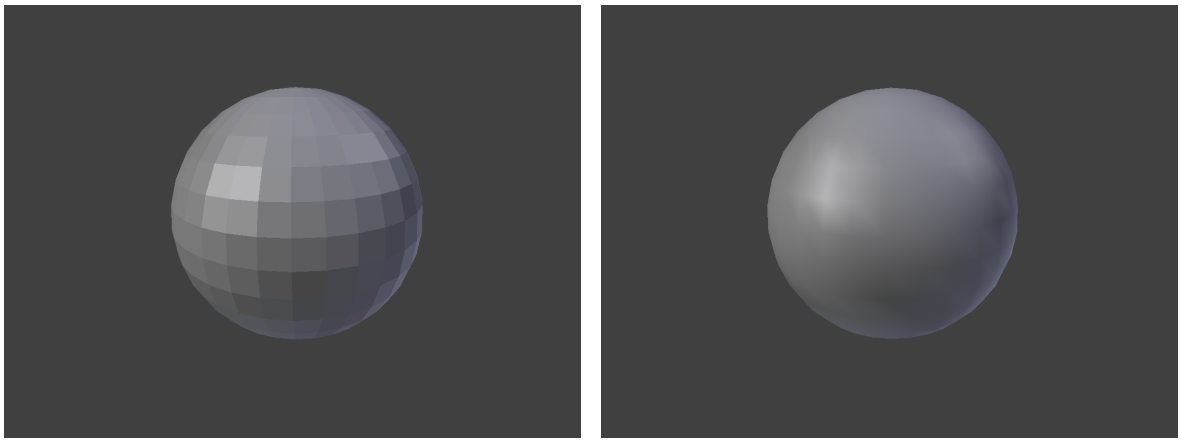
### 2.1.2 Rasterisation

Once geometry has been transformed into screen space, the process of rasterising triangles in order to display them on the screen is incredibly simple (see section 1.2.1 on polygon scan conversion for one possible method.) However, this is not the only step which is required to produce a 2D representation of a 3D scene. The next problem is how we should colour each pixel in the rasterised triangle in order to make it look 3D. Once a pixel is determined to be inside the triangle, the pixel is then shaded in order to determine its final colour on the screen.

#### Shading

Shading refers the process of determining a colour for a pixel on the screen. With rasterisation-based rendering techniques, this colour is usually determined using attributes from the triangle. Estimations of a number of lighting effects are used in order to determine this colour. There are three main contributions normally taken into account with rasterisation-based rendering techniques: ambient reflection, diffuse reflection, and specular reflection.

In order to calculate the way a surface is lit, a normal vector describing the normal of the plane on which a point on the surface sits. One approach is to use the normal of the plane on which the triangle sits, which produces results that are not smooth as shown in figure 2.3. On the other hand, an approach called Gouraud shading (Gouraud, 1971), or smooth shading, aims to simulate the lighting of a smooth surface by defining a normal for each vertex, typically an average normal for all faces sharing that vertex, and then shade each vertex of the triangle, finally interpolating the colour in between to produce



(a) Flat shading (b) Smooth shading

Figure 2.3: The difference between flat and smooth shading

a smooth fade. This effect is demonstrated in 2.4 by defining each vertex to have one of the primary colours and then fading in between them, creating a smooth fade.

### Ambient reflection

Ambient reflection refers to a small amount of light which is scattered around the scene. As rasterisation-based approaches often do not consider global effects, the ambient contribution to a surface's reflection intensity, $I_{ambient}$, is typically modelled as a constant for a given scene, depending on the desired effect.

### Diffuse reflection

Diffuse reflection refers to the scattering of light by a matte surface. Diffuse reflection is usually modelled using Lambert's cosine law, which describes the way light is scattered by a perfectly matte, or lambertian, surface.

Using Lambert's cosine law, the intensity of reflected light due to lambertian reflectance, $I_{diffuse}$, can be calculated by taking the dot product of the normalised light-direction vector $\vec{L}$ with the surface normal $\vec{N}$, as below:

$$I_{diffuse} = \vec{L} \cdot \vec{N}$$

This value can then be multiplied by the colour of the surface and the intensity of the incoming light in order to determine the overall diffuse component of the surface's reflection.

### Specular reflection

Specular reflection refers to the bright spots of light that are reflected by shiny objects. Specular highlights are visible where the surface normal lies half way between the light

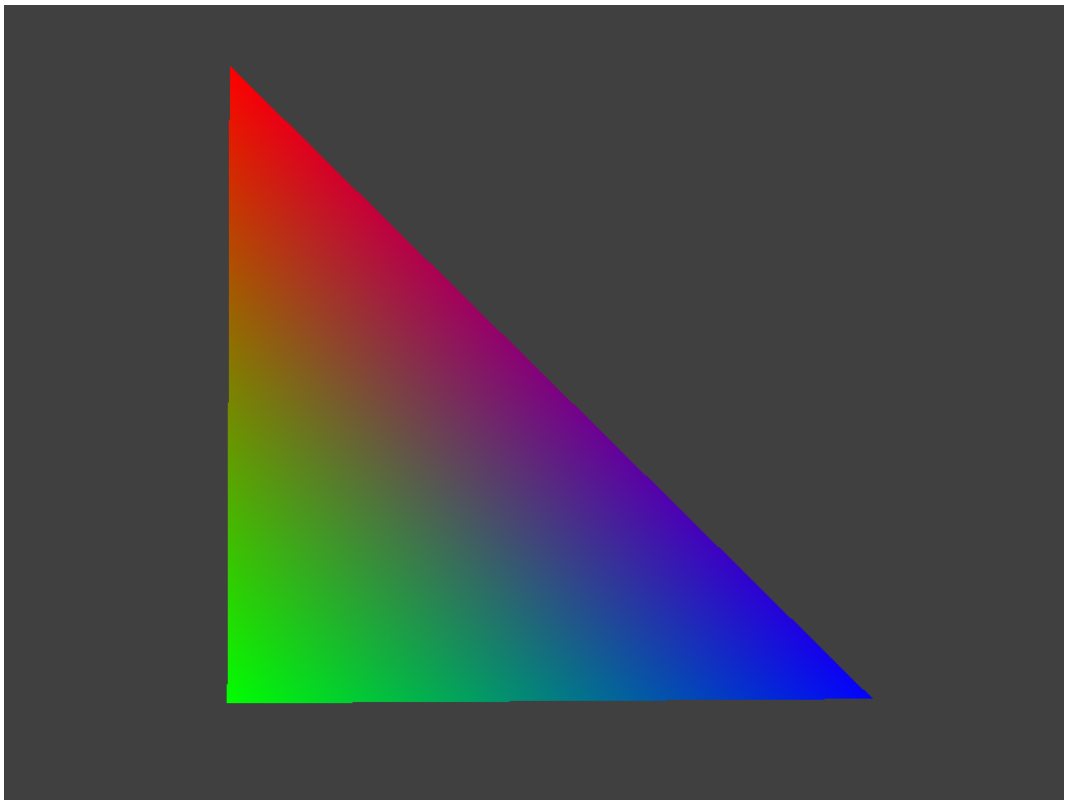


Figure 2.4: Colour interpolated across a triangle

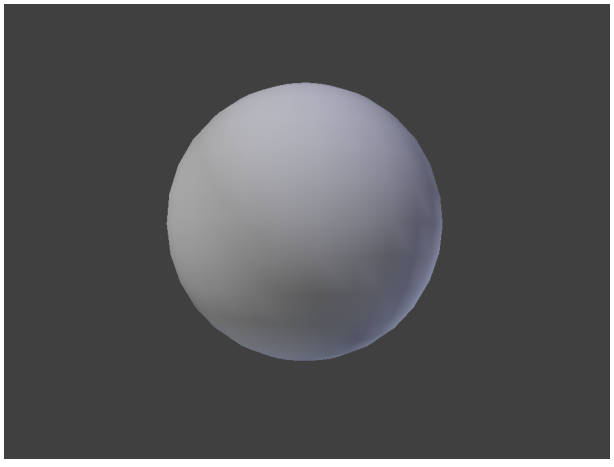


Figure 2.5: An example of diffuse reflection

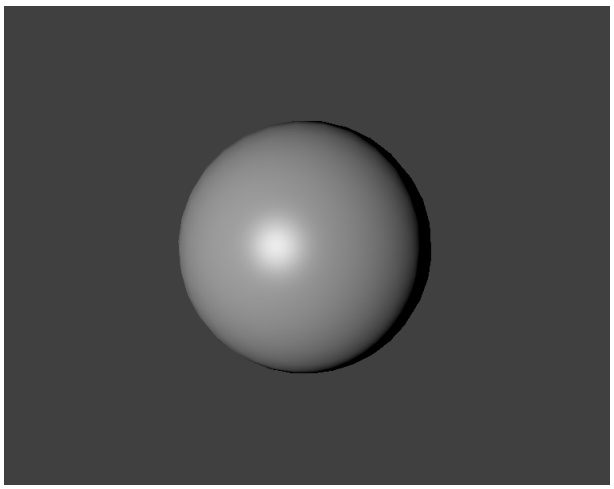


Figure 2.6: The shiny look of this sphere is caused by specular reflection

direction and the view direction.

Two main models are used to estimate the contribution of specular reflection to the overall reflection of a surface: the Phong shading model (Phong, 1975), and a computationally lighter modification known as the Blinn-Phong shading model (Blinn, 1977).

These approaches improve upon Gouraud shading by interpolating the normal across the triangle, rather than just the shaded colour, increasing the computational intensity of shading, but allowing smooth specular highlights to be calculated.

The specular component in Blinn-Phong shading is calculated by computing a half-way vector $\vec{H}$ between the view and light directions (Blinn, 1977):

$$\vec{H} = \frac{\vec{L} + \vec{V}}{|\vec{L} + \vec{V}|}$$

Or, more simply:

$$\vec{H} = normalise(\vec{L} + \vec{V})$$

This vector is shown in figure 2.7.

The specular contribution to the overall reflection can then be calculated as:

$$I_{specular} = (\vec{H} \cdot \vec{N})^s$$

Where s is the specular power, which determines the intensity of specular highlights produced by the surface. How varying the specular power affects the intensity of the specular reflections produced is shown in figure 2.8.

### Overall reflection

The overall reflection of these surfaces can then be calculated additively:

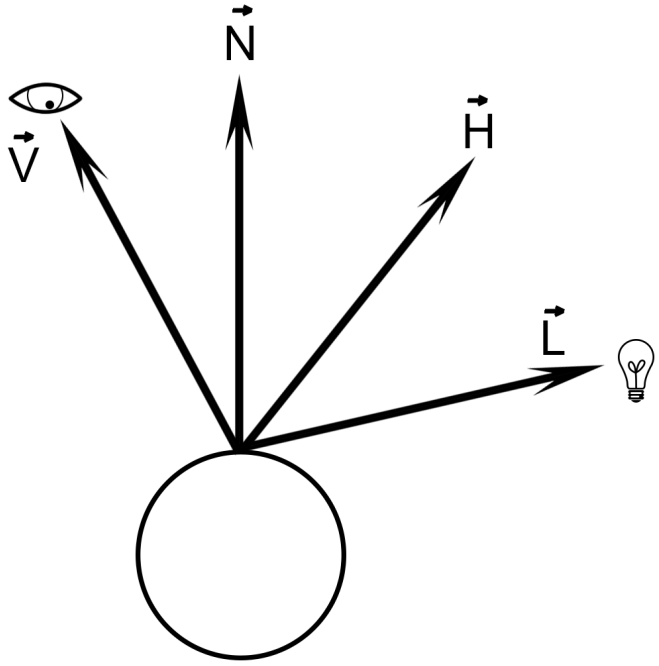
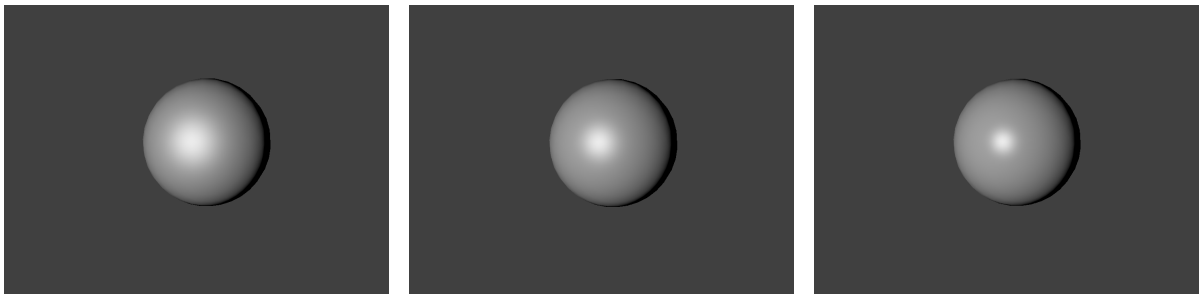



Figure 2.7: The relationship between the direction of viewing, light direction, the half-angle in between the direction of viewing and the light direction, and the normal of the surface being viewed. $\vec{L}$ is the normalised light vector which points from the surface being shaded to the light source. $\vec{N}$ is the normal of the surface being shaded. $\vec{H}$ is the direction vector half way between the angle of viewing, $\vec{V}$, and the angle of the light direction, $\vec{L}$



(a) s = 16 (b) s = 32 (c) s = 64

Figure 2.8: The effect of varying specular power

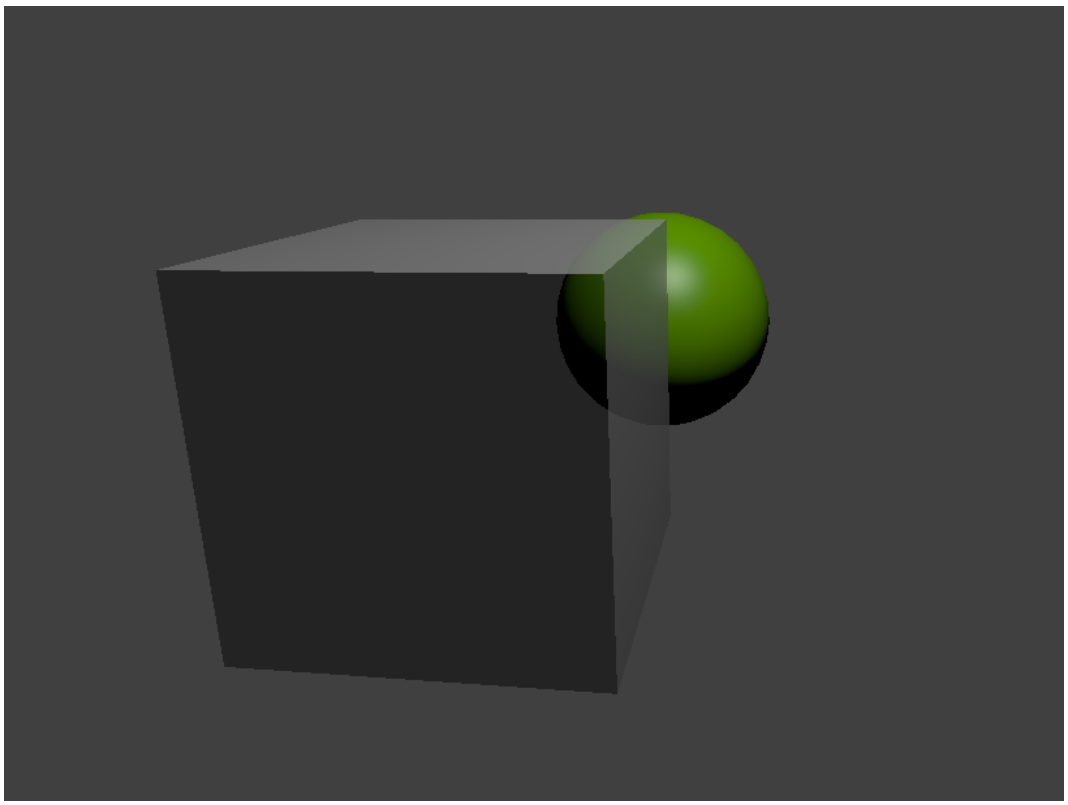


Figure 2.9: Demonstration of alpha blending - a cube is alpha blended with the sphere in the background to give the appearance of transparency

$$I_{overall} = I_{ambient} + I_{diffuse} + I_{specular}$$

**Blending**

Blending is another aspect of shading, which allows rasterised surfaces to give the appearance of partial or full transparency. By adding a fourth component to colours, an alpha component, we can define how opaque an object is, where a surface with an alpha value of 1 is fully opaque, and a surface with an alpha value of 0 is fully transparent.

By considering the alpha component of surfaces, and applying blending to additively combine the surface's colour with the background colour, the background colour is allowed to show through, as shown in figure 2.9.

## 2.2 Rasterisation-based volume rendering

Researchers have used many approaches to map volume rendering operations directly to rasterisation hardware, as modern graphics hardware is extremely specialised for the purposes of rasterisation. As this sort of hardware is ubiquitous in modern computers, methods of rendering volumes in real-time using this hardware are highly sought-after. In order to understand why these approaches are not suitable for achieving our objectives,

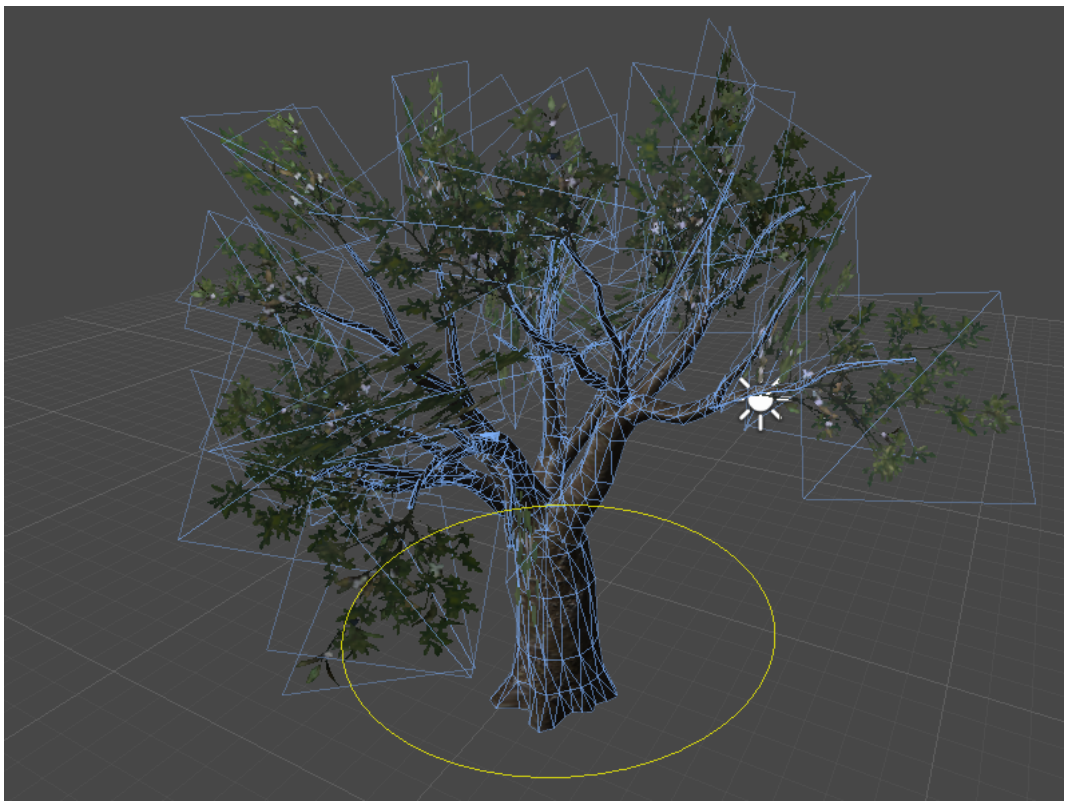


Figure 2.10: Billboard cloud tree rendering

it is important to understand these approaches in order and appreciate their limitations.

### 2.2.1 Billboards

Billboards are texture-mapped planes which typically have no rotation component. In other words, they are always facing directly at the camera. Billboards have long since been used to render complex 3D objects in real time. By reducing complex geometry to relatively few billboards, it is possible to create a convincing enough looking representation of a 3D object. By utilising transparency and alpha blending capabilities of rasterisation hardware, it is possible to allow multiple layers of billboards to show through.

A common example as used in games is for tree rendering. A technique demonstrated by Garcia, Sbert, and Szirmay-Kalos utilises a set of billboards, referred to in the paper as a cloud of billboards, in order to vastly simplify complicated tree models defined by hundreds of thousands of polygons into a few fixed billboards, each of which can approximate many leaves. This technique is demonstrated in figure 2.10.

Similar techniques have also been successfully developed for volume rendering. Harris, 2002 combined the simulation of multiple forward scattering within a cloud volume with billboards used to accelerate rendering. By utilising impostors, billboards subtly intended to replace 3D objects, this approach accelerates rendering by rendering to the billboards and reusing the results between frames. This method allows for realistic real-time rendering of clouds.

The limitation of this approach is that it does not simulate global effects, and as such, the rest of the scene does not have an effect on the rendering of an individual cloud volume. In fact, the cloud rendering approach utilised in this paper is very specific in the effects which it simulates, simulating very homogeneous looking clouds that are not affected very much by the scene or surrounding clouds. Although this may be a suitable compromise for some games, it falls short of our objectives to simulate true volumetric effects in real-time. Nonetheless, the impostor technique may be suitable in some cases in order to exploit frame-to-frame coherence.

### 2.2.2 Texture-based approaches

Engel and Ertl, 2002 proposes an approach that utilises a 3D texture as proxy geometry, exploiting hardware alpha blending as a means to composite overlapping samples in order to create a 3D image of a volume.

The advantage of this approach is that it maps onto existing rasterisation hardware very well, allowing real-time volume rendering on relatively cheap, ubiquitous graphics hardware. The major limitation of this method, however, is that it only considers overlapping parts of the volume. If one wishes to calculate lighting effects which involve light being reflected or refracted in different directions as we do, this approach is not suitable. In addition, Engel and Ertl found the precision of rasterisation hardware limited when it came to texture representations, which resulted in artifacts in the rendered image.

## 2.3 Ray tracing

A more generic approach called ray tracing, similar to the approach in Engel and Ertl, 2002, aims to simulate the path of light in reverse. Ray tracing starts by firing rays outwards into the scene from the camera (Appel, 1968), traditionally firing one ray per pixel and finding the first intersection of the scene with the ray.

Once an intersection is found, additional rays can be spawned in order to take account of the effects of reflection and refraction. Unlike Engel and Ertl's approach, this allows the ray of light to follow a path that isn't just straight, allowing rendering to account for the contribution to lighting of other parts of the scene that aren't directly overlapping a particular pixel.

In addition to allowing light reflected from the rest of the scene to be taken into account, rays can also peer directly into volumes in order to determine the effects of lighting within them. This allows us to consider volumes that are not just homogeneous in nature, as the ray can be affected within the volume by the internal structure of the volume.

**Algorithm 1** Per-pixel rendering

```
1: procedure RENDER_PIXEL(x, y, tree, projection)
2:     out_colour ← (0, 0, 0, 0)
3:     ray ← calculate_ray(x, y, projection)
4:     intersection ← RAYCAST(tree, ray)
5:     if intersection.hit then
6:         out_colour ← SHADE(tree, ray, intersection)
7:     end if
8:     returnout_colour
9: end procedure
10:
11: procedure SHADE(tree, ray, intersection)
12:     out_colour ← (0, 0, 0, 0)
13:     for each light do                                  ▷ Shade for each light
14:         light_dir ← light_pos − intersection.position      ▷ Check shadow
15:         shadow_ray ← (intersection.position, light_dir)
16:         shadow_intersection ← RAYCAST(tree, shadow_ray)
17:         if shadow_intersection.hit then
18:             continue
19:         end if
20:         out_colour += DIFFUSE_REFLECTION(intersection, light)
21:         out_colour += SPECULAR_REFLECTION(intersection, light)
22:     end for
23:
24:     reflection_ray ← REFLECT(ray, intersection)        ▷ Reflection
25:     reflection_intersection ← RAYCAST(tree, reflection_ray)
26:     if reflection_intersection.hit then
27:         out_colour += SHADE(tree, reflection_ray, reflection_intersection)
28:     end if
29:
30:     refraction_ray ← REFRACT(ray, intersection)        ▷ Refraction
31:     refraction_intersection ← RAYCAST_EMPTY(tree, refraction_ray)
32:     if refraction_intersection.hit then
33:         out_colour += SHADE(tree, refraction_ray, refraction_intersection)
34:     end if
35: end procedure
```

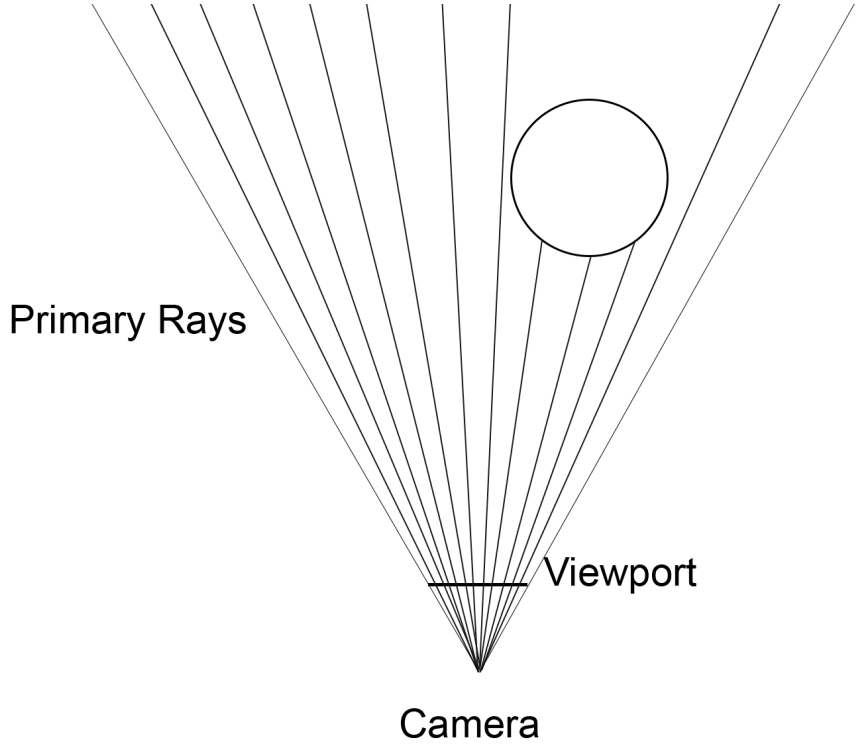



Figure 2.11: In ray tracing, primary rays are cast through each pixel on the viewport

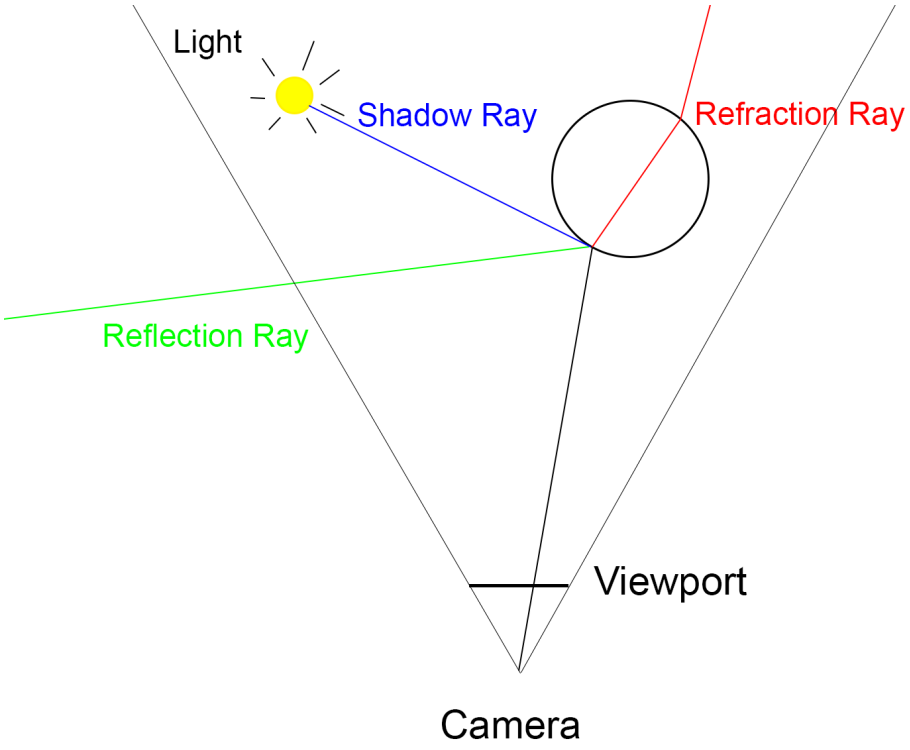



Figure 2.12: Additional rays can then be spawned for each intersection to calculate shadow coverage, reflection, and refraction

### 2.3.1 Direct volume rendering

Ray tracing allows the direct rendering of a volume, without having to first sample the underlying volume into another format. Any volume (or structure) for which an intersection test can be defined can be rendered by means of ray tracing.

Roth, 1982 presented an algorithm for directly rendering constructive solid geometry (CSG), a method for representing 3D shapes which involves combining various geometric shapes using boolean operations. This work described the calculation of eye rays for orthographic as well as perspective projections, as well as a hierarchical data structure for storing and rendering CSG directly. This work also described a number of primitives as well as the relevant necessary intersection tests. Additionally, Roth demonstrated the application of this algorithm for producing realistically shaded solids for use in computer aided design, and also used shadow rays in order to produce ray traced shadows, and even described a method for anti-aliasing of the resulting image.

The direct rendering of volumes simplifies computation, removing the need for us to rely on intermediate representations of volumes, and allowing us to consider the volume itself as all information may be relevant to the rendering of the current pixel.

Although this allows broad coverage of geometric shapes, it is difficult to represent highly heterogeneous volumes with CSG, as it would require a very large number of separate geometric shapes to model changes in heterogeneous attributes. In addition, it would be impossible to render smoothly changing heterogeneous attributes without a very large number of geometric shapes, at which point it would be more efficient to use a grid-based sampling approach. For this reasons, it is unsuitable for our objectives.

### 2.3.2 Space subdivision

The next important advancement in ray casting technology was presented by A. S. Glassner, 1984. By using an octree, a hierarchical volume representation, to partition space, and storing a list of objects in each node of the tree, he greatly reduced the number of comparisons required for each ray. This allowed the rendering of hundreds, or even thousands of objects in a scene without greatly increasing the computation time for each object. Despite this, an object format which is capable of storing heterogeneous volumes is still required.

On the other hand, Wilhelms and Gelder, 2000 presented a space-efficient octree representation that is designed for representing voxel data directly. By not storing empty space within the tree, a technique referred to as sparse voxel octrees (SVOs), a considerable amount of space is saved, reducing the memory footprint of the stored voxel data. Additionally, as the voxel data is stored directly using this kind of approach, it is possible to represent heterogeneous volumes with this structure.

Amanatides and Woo, 1987 presented a significant optimisation to the traditional

ray tracing algorithm, by presenting a method of mathematically determining the next voxel in a traversal and automatically stepping directly to its boundary. Using this new algorithm, traversing from one voxel to its neighbour requires only two floating point comparisons and one floating point addition. This greatly reduces computation time for ray casts, and is the basis for a large amount of derivative work on efficient ray casting.

Arvo, 1988 extends this technique to the octree, presenting an efficient linear-time voxel walking algorithm for octrees with a best case complexity of O(log N). Arvo's approach is a top-down approach which ensures that nodes are only considered once per ray and visits the voxels in the correct order, resulting in the O(log N) best case complexity. Arvo's algorithm involves checking the intersection of the ray with the root node of the octree, and recursively shortening the spans of the ray that are considered for intersection.

### 2.3.3 GPU implementation

Due to the fact that ray tracing utilises one primary ray per pixel, a fact that both limits it and works to its advantage in its basic implementation, the process is highly parallelisable. Additionally over the last decade, a breakthrough occurred in graphics hardware, which saw GPUs becoming massively parallel general-purpose stream processors, as opposed to the original fixed-function format. Purcell, Buck, Mark, and Hanrahan, 2002 investigates this trend and explains how ray tracing can be mapped to graphics hardware. Purcell et al. demonstrate how to reformulate ray tracing as a streaming computation, making it appropriate for GPU implementation. The process is as follows:

For each pixel of the screen:

1. Generate eye rays
2. Traverse acceleration structure
   (a) Do intersection tests
3. Shade hit
   (a) Generate shading rays (repeating 2-3 as necessary)

Purcell et al. were the first to demonstrate that efficient real-time ray tracing in the GPU is possible without any changes in architecture.

### 2.3.4 Efficient GPU traversal

Aila and Laine, 2009 discusses the mapping of elementary ray tracing operations such as traversal and intersection onto GPUs. This work focuses on understanding the efficiency of GPU ray traversal rather than directly presenting an efficient ray traversal implementation, highlighting the fact that very little is understood about the performance of fast

ray traversal algorithms. By comparing the performance of these algorithms against a theoretical upper bound, Aila and Laine demonstrate that previous methods are off by a factor of 1.5x - 2.5x theoretical optimal performance, and highlight previously unidentified inefficiencies in hardware work distribution.

Their work demonstrates that reliance on persistent threads rather than hardware work distribution mechanisms can improve the performance of the fastest GPU trace() kernels significantly.

### 2.3.5 Efficient octree storage

Wilhelms and Gelder, 2000 present a space-efficient octree representation that does not allocate memory for empty space - what has become known as a sparse voxel octree (SVO). Wilhelms and Gelder's structure utilises a pointer-less "linear octree" structure which avoids storing pointers between every node.

Laine and Karras, 2010 expand on these ideas by storing pointers between blocks of octrees and otherwise determining a particular child's address using an index and a parent's 8-bit child mask. As the child's pointer can be obtained from a lookup table using this index and child mask, it is unnecessary to store a pointer for every voxel, making the structure far more efficient.

As Laine and Karras demonstrate, the SVO structure can be efficiently ray traced with an implicit level of depth (LOD) mechanism, as traversal can be terminated as soon as a found voxel is smaller than the current pixel on the screen. At high resolutions, however, even with the efficient SVO storage structure presented by Laine and Karras, mid-sized scenes still take up an excessive amount of storage, with Laine and Karras's test scenes utilising up to 4GBs of memory, the upper bounds of memory on today's GPUs.

However, while extremely efficient to ray cast, and fairly compact when compared to other structures, Laine and Karras's structure is still fairly heavy on memory usage. In their paper, Laine and Karras address this by only encoding surfaces, an approach that is obviously not possible if volumetric effects are desired.

To combat memory usage, they additionally implement a concept called contours, in which surface voxels are restrained to a non-cubic shape using a pair of parallel planes, and once this shape provides a good enough approximation of the surface, the tree does not need to be generated any further. Laine and Karras's implementation for this is very efficient, and automatically takes into account multiple levels of contours in order to produce a shape extremely close to the source data. A similar process could also be used to prevent unnecessary depth in homogeneous regions, where the current approximation of the volume is sufficiently good.

Additionally, more recent work by Kampe, Sintorn, and Assarsson has greatly reduced the memory usage of SVOs by allowing identical regions of the tree to share pointers, allowing the reuse of data between regions. This new structure is referred to as a direct

analytic graph, or sparse voxel DAG. Kampe et al. have found that, using this technique, the memory usage of even highly irregular scenes can be reduced by 1 to 3 orders of magnitude. Despite this being most effective for more homogeneous scenes, the gains shown for highly irregular scenes are also very promising.

### 2.3.6 Animation

Due to the nature of deformation, animation has previously not been possible with voxel structures because it would leave gaps between voxels. Additionally, deforming a hierarchical structure means that there is no longer a guarantee that all child nodes lie inside their parent nodes. Bautembach, 2011 presented an approach to the animation of voxel octrees by expanding the parent voxels so that they still encompass all child voxels. Bautembach's approach demonstrates an efficient and effective approach capable of character animation with sparse voxel octrees, something which had previously been considered infeasible.

### 2.3.7 Caching

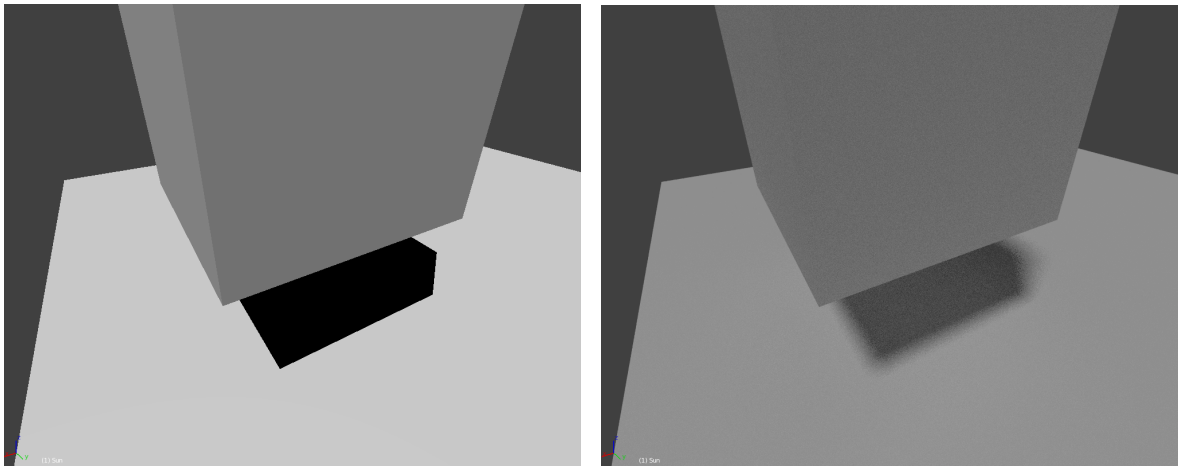
Ruff, Clua, and Fernandes, 2013 presented a caching-like strategy which is capable of storing data generated in previous frames which can then be used to accelerate rendering in later frames. Ruff et al. claim that their approach can be integrated to any existing ray tracing solution, and will allow data to be reused between frames. A significant limitation of this approach, however, is that it is tailored to static scenes. Ruff et al. point out that any movement of the scene objects would generate inconsistencies in cached data, although they do provide some potential solutions to this problem.

### 2.3.8 Limitations of ray tracing

Despite the numerous advantages of ray tracing, it has one major disadvantage: ray tracing can not simulate fuzzy phenomena, such as soft shadows. Figure 2.13 demonstrates shadows as simulated by ray tracing. Shadows produced by ray tracing are defined by explicit boundaries, where in real life, the edges of shadows are not well defined. In order to understand why ray tracing has this limitation, it is important to understand exactly what it is trying to approximate.

#### The Rendering Equation

Kajiya, 1986 presented a single integral equation which generalises a variety of rendering techniques. This equation has become known as "The Rendering Equation", the title of his 1986 paper. Solving it produces an accurate simulation of many effects, including soft shadows, depth of field, ambient occlusion, global illumination and motion blur. The



(a) hard shadows (b) soft shadows

Figure 2.13: The hard edges of shadows generated by ray tracing, next to the soft shadows created by path tracing

Figure 2.14: The soft edges of a real shadow

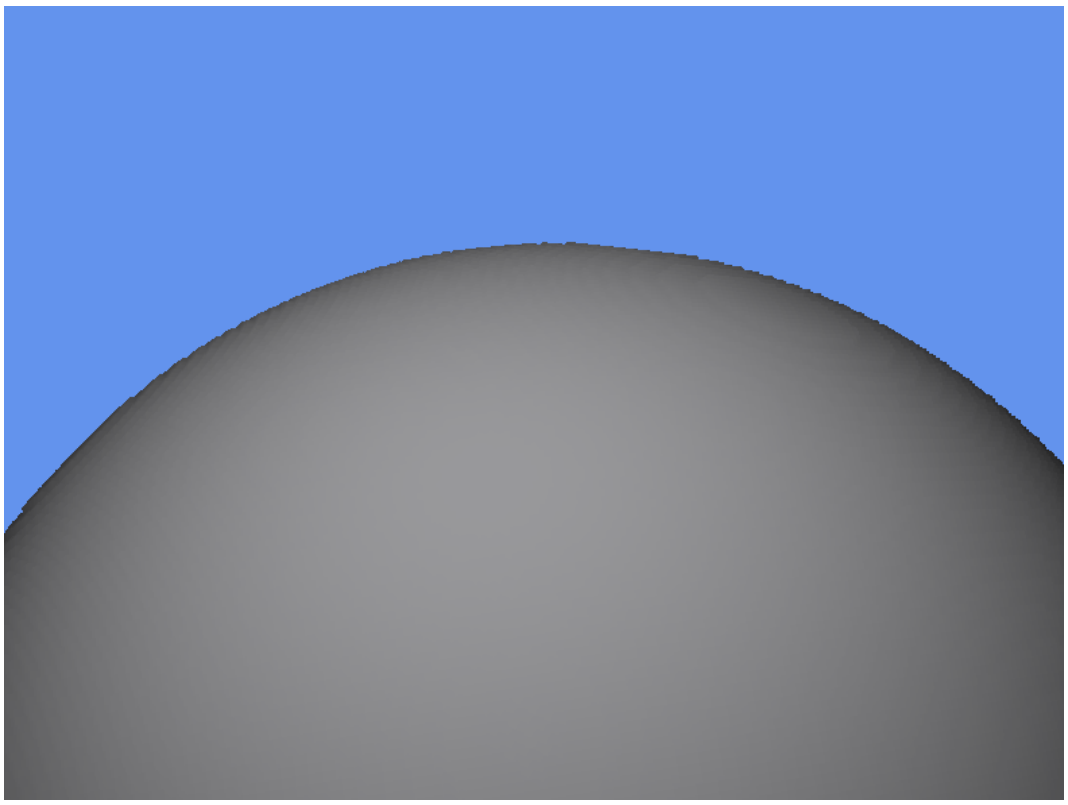


Figure 2.15: The aliased edges caused by point sampling approaches

physical basis to this equation is as an approximation to Maxwell's equation for electromagnetism. Solving this equation is the main challenge in realistic rendering (Dimov, Penzov, & Stoilova, 2007).

Ray tracing evaluates this integral at a single point, the point of the intersection of a ray with the scene, and as such, it is not a good approximation. For this reason, ray tracing is known as a point-sampling approach. A number of approaches have been developed to address this limitation, such as cone tracing, distributed ray tracing, and path tracing.

### Cone tracing

Amanatides presented a new method of ray tracing using cones rather than half-line rays in order to sample each pixel. The spread angle of the cone is chosen such that the radius of the cone's base is the size of the pixel according to the projection. The difficulty in using this method is the complexity of the intersection tests of a cone with the scene. Amanatides describes the necessary intersection calculations between a cone and various objects.

As cone tracing is no longer a point-sampling method, anti-aliasing is no longer required, and yet still only one sample per pixel is required. Amanatides' approach also allows for built in level of detail, soft shadows and blurred reflections. Given efficient intersection tests, this allows cone tracing to achieve similar performance to ray tracing

with numerous advantages.

### Distributed ray tracing

Cook, Porter, and Carpenter, 1984 also provides a solution to these problems with traditional ray tracing. By distributing the directions of the rays according to the analytic functions they sample, Cook et al.'s approach allows ray tracing to render fuzzy phenomena. By utilising this technique, Cook et al. enable ray tracing to produce blurred reflections, translucency and soft shadows, motion blur and depth of field.

The advantage of this approach over Amanatides' cone tracing approach is that the ray intersection calculations do not change, merely the ray directions are distributed (Cook et al., 1984). Similarly to Amanatides, 1984, the approach presented in this paper also does not require any more rays than vanilla ray tracing. The disadvantage to this approach is that the rays are still point-sampled, and therefore, anti-aliasing techniques are still required in order to reduce the effects of aliasing on the edges of objects.

### Path tracing

Kajiya's own approach involves a Monte Carlo solution of the rendering equation. This Monte Carlo solution involves randomly distributing rays over the integral's domain, and then averaging them. This produces an extremely accurate and unbiased image, which naturally simulates many visual effects, including soft shadows, global illumination, and depth of field. Unfortunately, in order to get high quality images from this approach, a large number of rays must be traced in order to avoid very visible noisy artifacts.

Despite this, Bikker and van Schijndel have managed to get a path tracer running in real-time by utilising the persistent threads technique demonstrated by Aila and Laine, 2009. This approach still displays noisy artifacts at first however, with the approximation improving over time as the camera is stationary and more samples are averaged. Despite this, the noise is still being improved, and with more computing power, the ability to average more samples per pixel faster will naturally reduce the amount of noise generated over time.

# 3. Methodology

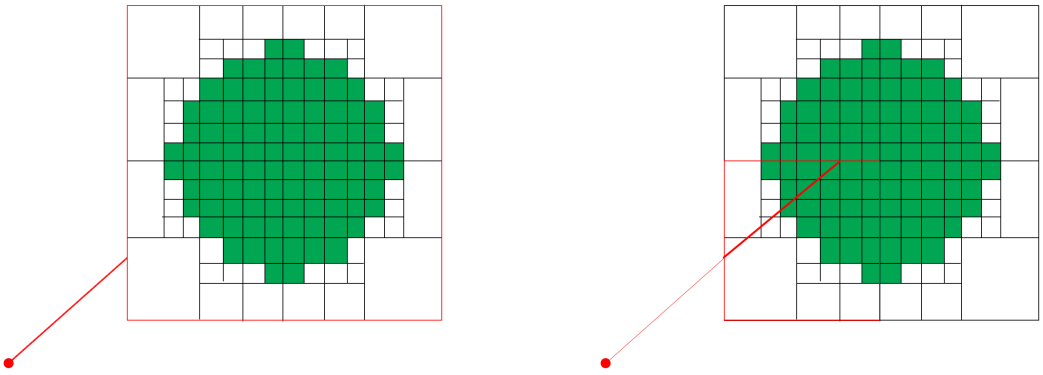
## 3.1 Ray tracing

Our ray tracer uses the GPU-optimised parametric octree traversal algorithm presented by Laine and Karras, 2010. This algorithm is extremely efficient for traversing along rays through a sparse octree as it hierarchically avoids empty space, finding the first intersection of the ray extremely fast.

Pseudo-code for the ray casting algorithm is given in algorithm 2. Our data is stored in a regular octree structure, meaning that each node of the tree is divided into 8 identically sized octants. The algorithm then begins by determining the intersection of the root node with the ray, as well as determining the first entered child octant. This is shown in figure 3.1. The main loop then begins.

The algorithm then checks that the current voxel exists. If the current voxel exists, the exit conditions are checked: the projection of the current voxel is smaller than the pixel on-screen, and whether the voxel is a leaf node, in which case there are no more subdivisions below it. If either of these conditions are met, the algorithm terminates, returning the current voxel. Otherwise, the algorithm executes push, advancing the traversal to the first entered child voxel.

If the current voxel does not exist, the algorithm advances the traversal to the next voxel intersected by the ray on the current level. The direction of advancement is then



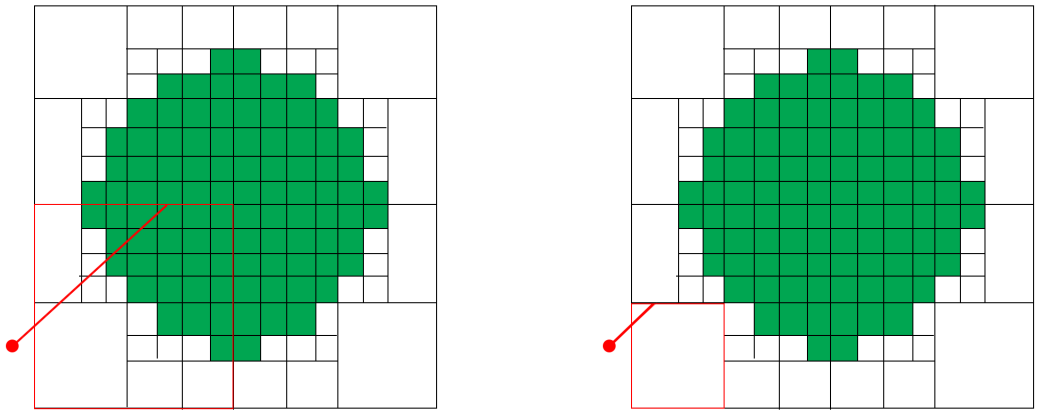
(a) Intersection of the root with the ray

(b) Initial child selection

Figure 3.1: Initial intersection test

**Algorithm 2** Ray cast algorithm pseudo-code (Laine & Karras, 2010)

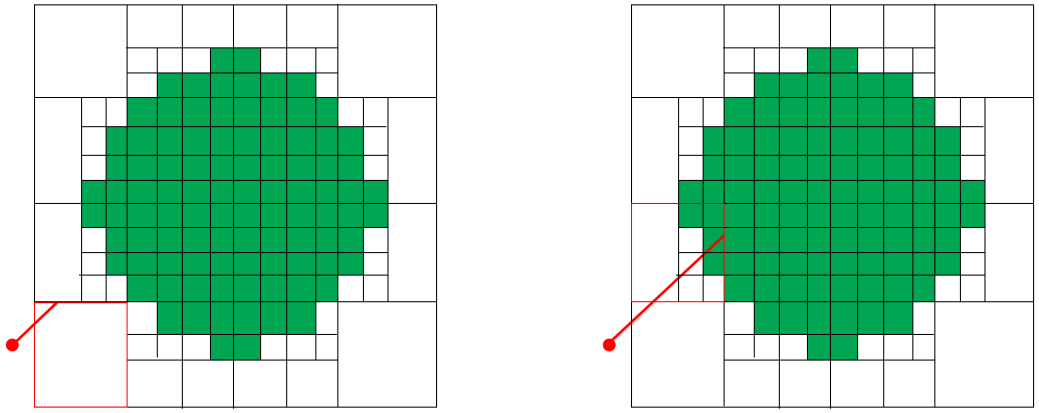
```
1: procedure RAYCAST(root, ray)
2:     current_voxel ← intersect(root, ray)                ▷ Intersection with root
3:     while not terminated do                                              ▷ Traverse
4:         if current_voxel exists then
5:             if voxel smaller than pixel then                          ▷ Exit conditions
6:                 return current_voxel
7:             else if voxel is a leaf then
8:                 return current_voxel
9:             else
10:                current_voxel ← push(currrent_voxel, ray)        ▷ Descend into child
11:            end if
12:        end if
13:        current_voxel ← advance(current_voxel, ray)   ▷ Advance into next sibling
14:        if advance direction disagrees with ray direction then
15:            current_voxel ← pop(current_voxel, old_voxel)        ▷ Pop last common
   parent
16:        end if
17:    end while
18: end procedure
```



(a) The current voxel prior to push

(b) The current voxel after push

Figure 3.2: The push operation, 2D case



(a) The current voxel prior to advance

(b) The current voxel after advance

Figure 3.3: The advance operation, 2D case

checked against the ray direction to ensure the advance was valid. If the direction of advance does not coincide with the direction of the ray, in which case one of the parent nodes of the current voxel differs from one of the parent nodes of the previous voxel, pop is used to return to the last common parent of the two voxels and determine the next child intersection along the ray, after which time traversal continues. The push, advance, and pop operations are described fully by Laine and Karras, 2010.

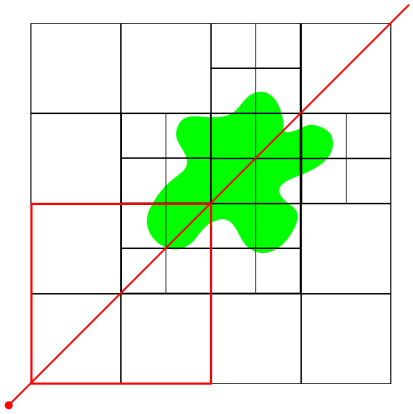
As empty space is quickly advanced over, preventing unnecessary traversal deeper into the tree, and voxels are never revisited, this algorithm quickly determines whether a ray intersects with a volume stored within an octree. This efficient traversal allows us to cast rays against a volume to produce a real-time ray traced image.

### 3.1.1 Storage

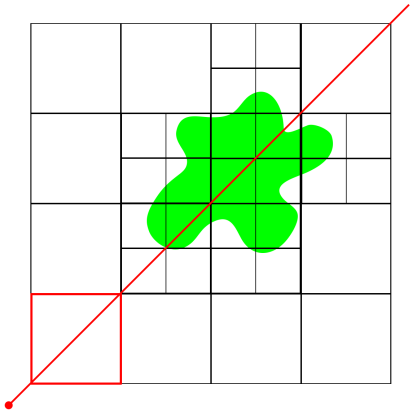
Our data structure uses the same basic storage mechanisms as Laine and Karras, 2010. The tree structure is stored separately from the related shading data, as the majority of processing time is spent in traversal. The extra time it takes to look up shading attributes for shading is therefore traded for a more compact traversal structure. A pointer to a lookup table pointing to attributes for the whole volume is stored at 8 kilobyte boundaries within the tree data.

The tree structure encodes each non-leaf node as a child descriptor, containing a pointer to the children of the current node, an 8-bit mask detailing which of the current node's children exists, and another 8-bit mask which determines which of the node's children are non-leaves (in other words, which children have their own child descriptors stored at the child pointer.)

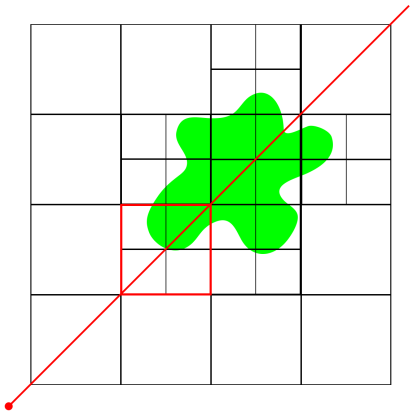
The child descriptors for the children of a node are then stored together in a block, referenced by this child pointer. The non-leaf mask can then be used to access a particular child using an index from 0 to 7 using a 256 bit lookup table which relates the non-leaf



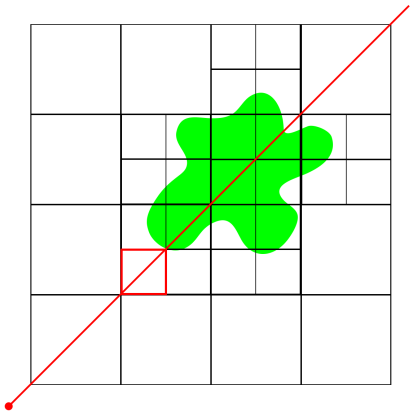
(a) The initially intersected voxel



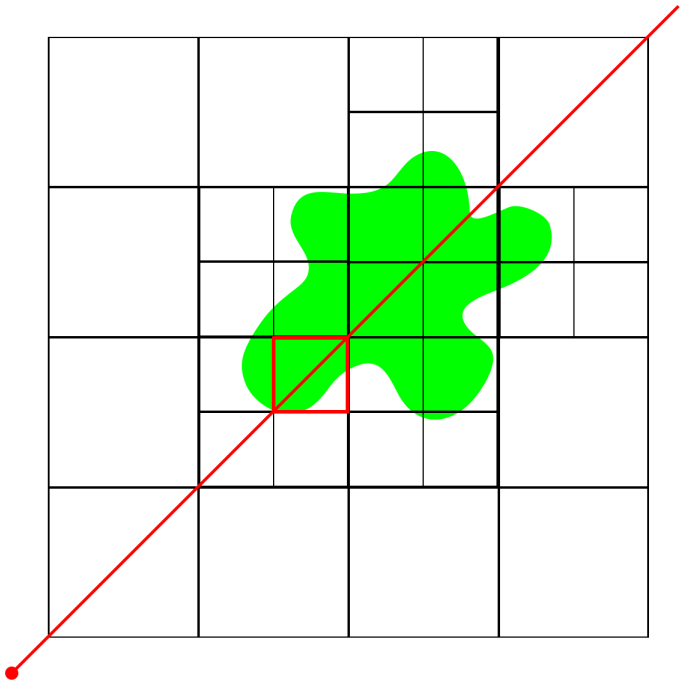
(b) Push



(c) Advance



(d) Push



(e) Advance, the first intersected voxel is found

Figure 3.4: How empty space is avoided, 2D case

mask of a parent voxel to the index of the desired child:

$$child\_offset = lookup(non\_leaf\_mask << child\_index)$$

The lookup table contains values from 0 to 7, containing the offset of the desired child given a particular non-leaf mask. This lookup table and the relevant implementation is shown in appendix Appendix A.

### Voxel attributes

The attributes stored for each voxel are the main way in which our storage scheme varies from that used in Laine and Karras, 2010. In Laine and Karras, 2010, only a colour and a normal, representing the volume after ambient occlusion is pre-calculated, are required for shading. For our purposes, additional attributes are required per-voxel, such as index of refraction and reflectance.

Attributes are stored after the traversal information, along with a lookup table that points to blocks of attributes, grouped by parent voxel. The index in the lookup table can be obtained by considering the offset of the parent's child descriptor from the start of the data, as so:

$$lookup\_index = \frac{parent\_desc\_addr - start\_addr}{child\_desc\_size}$$

The child descriptor size is a constant, referring to the size in bytes of each child descriptor.

This index is then used to retrieve the block of attributes containing the current voxel's attributes, and then the $child\_offset$ obtained above is used to obtain the attributes for the current voxel:

$$voxel\_attributes = lookup\_table[lookup\_index] + child\_offset$$

The resulting pointer points to a structure containing the packed attributes required to shade the current voxel. For our renderer, that includes colour, normal, reflectance, and index of refraction, where the colour also contains an alpha channel which is used to determine transparency. The number of bits used to store each attribute is detailed below:

| Attribute | Size in bits | Details |
|---|---|---|
| Colour | 32 bits | 8 bits for red, green, blue, and alpha components between 0.0 and 1.0 |
| Normal | 32 bits | Encoded as detailed in Laine and Karras, 2010, providing up to 14 bits of precision for smooth normals |
| Reflectance | 16 bits | A floating point value between 0.0 and 1.0 |
| Index of refraction | 16 bits | A floating point value between 1.0 and 4.0 |

The 16-bit floating point values for reflectance are stored as integers between $-2^{15}$ and $2^{15}$, as these are the minimum and maximum values that can be stored in a signed 16-bit integer. These values are scaled such that the minimum value (0.0 for reflectance and 1.0 for refraction) is encoded as $-2^{15}$, and the maximum value as $2^{15}$. Minimum and maximum values were chosen to maximise the precision of the packed values, although the values are packed as 16-bit integers purely for the reason that two 16-bit integers fit exactly into a 32-bit bit-field. If more attributes were added as needed, the sizes of these packed values could potentially be reduced as needed.

The minimum and maximum values for reflectance were chosen as values outside of this range are not physically sensible, as reflectance is defined as the fraction of received electromagnetic radiation (in this case, light) that is reflected by a surface. Values between 0 and 1 for refraction are also not physically sensible, although 4 was chosen as an arbitrary cut-off.

Floating point values are packed into integral bitfields using the following formula, where min and max are the minimum and maximum values to be considered, and bits is the number of bits allocated:

$$range = max - min$$

$$int\_max = 1 << (bits - 1)$$

$$packed\_value = floor((value - min) \cdot \frac{int\_max}{range})$$

These values are then unpacked as needed. In some cases, this might not be necessary. For example, when checking for changes in index of refraction, the integral value can be used instead.

## 3.2 Reflection rays

In order to determine the contribution of light reflected by the scene, a reflection ray must be cast in the angle of reflection in order to find any surfaces that may be reflecting light

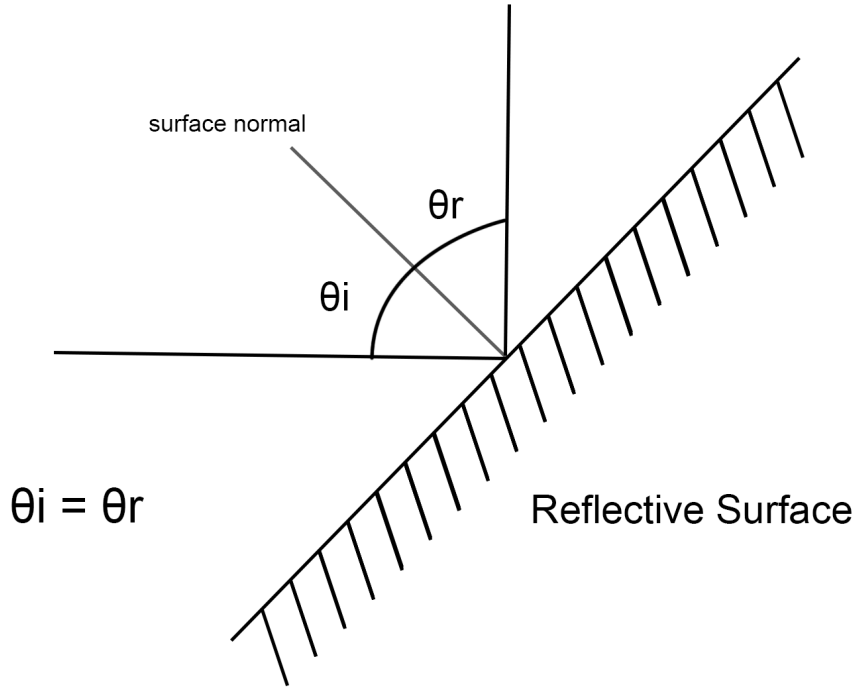



Figure 3.5: The relationship between the angle of incidence $\theta_i$ and the angle of reflection $\theta_r$

back. The law of reflection tells us that the angle of incidence, the angle between the normal of a surface and the ray hitting it, is equal to the angle of reflection (Heath, 1999). The GLSL (OpenGL Shader Language) specification gives a vector form of this equation using the dot product (Kessenich, Baldwin, & Rost, 2014), where $I$ is the incident vector and $N$ is the normal of the surface:

$$reflection\ direction = \vec{I} - 2 * \vec{N} * (\vec{N} \cdot \vec{I})$$

When shading a surface, a reflection ray is cast in this direction in order to check for intersections with the rest of the scene. If any such intersections are found, the intersected surface is shaded recursively down to a specified limit, and then added onto the overall reflected colour for the current surface.

## 3.3 Refraction rays

In order to refract a ray through a surface, the direction of the refracted ray must be calculated. The refracted angle can be calculated by utilising Snell's law, which states a relationship between the angle of incidence $\theta_1$, the angle of refraction $\theta_2$, the index of refraction of the current medium $n1$, and the index of refraction of the medium being entered $n2$ (A. Glassner, 1989):

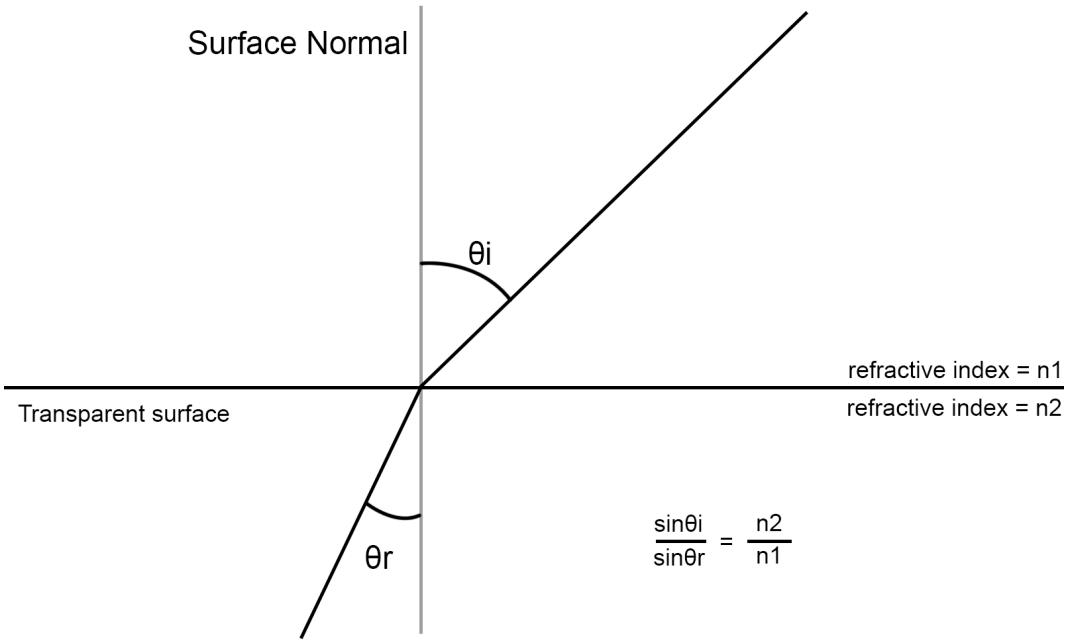



Figure 3.6: The relationship between the angle of incidence $\theta_i$ and the angle of refraction $\theta_r$

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{n2}{n1}$$

By using this relationship, it is possible to determine a direction vector for the refracted ray. The GLSL specification again gives a method of calculating a direction vector for a refracted ray, given an incidence vector $I$, a surface normal $N$, and the ratio of the current index of refraction to the new index of refraction *eta*:

Listing 3.1: Calculation of a refraction ray

```
vector refract(vector I, vector N, float eta)
{
        k = 1.0 - eta * eta * (1.0 - dot(N, I) * dot(N, I));

        if (k < 0.0)
                return vector(0.0, 0.0, 0.0);
        else
                return eta * I - (eta * dot(N, I) + sqrt(k)) * N;
}
```

A refracted ray can then be cast through a material in order to determine the path of light through a medium. When the ray reaches a boundary beyond which the index of refraction changes, the direction of the refraction ray must be recalculated.

One additional consideration must be made, however. The specified normal must be a normal on the plane through which the ray is refracting. In the case of a refraction ray leaving a material, rather than entering it, the normal usually used for rendering a surface will need to be reversed in order to obtain this normal. This can easily be handled by checking the dot product of the normal with the direction of the refracted ray:

$$dot = ray.direction \cdot normal$$

If this dot product is greater than zero, then the surface normal is facing outwards from the direction of the ray, and as such, must be reversed to obtain the normal of the surface through which the ray is being refracted:

$$normal = -normal$$

### 3.3.1 Tracing rays through materials

In order to handle translucent materials, we must be able to trace a ray through a translucent material. The problem is, given an intersection with a surface, finding the point at which a refraction ray cast from this intersection point exits the surface. One possible solution is to modify the ray casting algorithm, redefining the exit conditions such that it only exits when empty space is found, as in algorithm 3.

The algorithm will now descend until it finds a leaf node, advancing and popping only once a leaf is found. This algorithm will find the exit point of a ray, but it has the side effect of checking every voxel along the ray's path, at every level. This is very inefficient, and does not take advantage of the volume's tree structure.

On the other hand, if we were to consider the refraction ray discretely, we could follow the path of the ray through the solid, casting back towards the original point of

**Algorithm 3** Ray casting through solids

```
1: procedure RAYCAST_SOLID(root, ray)
2:     current_voxel ← intersect(root, ray)                    ▷ Intersection with root
3:     while not terminated do                                  ▷ Traverse
4:         if current_voxel exists then
5:             current_voxel ← old_voxel
6:             current_voxel ← push(currrent_voxel, ray)        ▷ Descend into child
7:         end if
8:         old_voxel ← current_voxel
9:         current_voxel ← advance(current_voxel, ray)          ▷ Advance into next sibling
10:        if advance direction disagrees with ray direction then
11:            current_voxel ← pop(current_voxel, old_voxel)    ▷ Pop last common
    parent
12:        end if
13:        if current_voxel nonexistent then                    ▷ Return when empty space is found
14:            return old_voxel
15:        end if
16:    end while
17: end procedure
```

intersection at intervals. Once the active span of the ray as returned by the ray casting algorithm is non-zero, in other words, when the ray has started in empty space and found a surface, we have found the probable exit point of the refraction ray. In order to do this, an interval must be chosen. We chose to use a fixed user-configurable interval, but it may be possible to use a dynamic interval based on the volume being ray traced. The pseudo-code for this operation is given below.

**Algorithm 4** Discrete ray cast

```
1: procedure DISCRETE_RAYCAST(root, ray)
2:     max_iterations = scene_width / fixed_interval
3:     for i = 0; i < max_iterations; ++i do
4:         RAYCAST(origin + i * direction, -direction)
5:         if resulting span > 0 then                           ▷ We found empty space
6:             break
7:         end if
8:     end for
9:     return found surface
10: end procedure
```

### 3.3.2 Refraction through heterogeneous materials

It has already been established that the refraction ray's direction must be recalculated when moving from a material with one index of refraction to a material with a different index of refraction. The problem, then, is how to accomplish this for heterogeneous materials.

The above discrete ray cast algorithm can be easily modified to take account of the change in refractive index as it traverses through a material.

**Algorithm 5** Discrete ray cast with varying refractive index

```
1: procedure DISCRETE_RAYCAST(root, ray)
2:     max_iterations = scene_width / fixed_interval
3:     refractive_index = hit_refractive_index
4:     for i = 0; i < max_iterations; ++i do
5:         RAYCAST(origin + i * direction, -direction)
6:         if resulting span > 0 then                    ▷ We found empty space
7:             break
8:         end if
9:         if hit_refractive_index ≠ refractive_index then
10:            if normal · direction > 0 then            ▷ Check if exiting material
11:                normal = −normal
12:            end if
13:            direction = REFRACT(direction, normal, refractive_index / hit_refractive_index)
14:        end if
15:    end for
16:    return found surface
17: end procedure
```

## 3.4 Shading

Our shader uses a Blinn-Phong shading model, as described in chapter 2.

When shading a surface, a ray is cast towards each light source to check if it is occluded, in which case the intersection is in shadow. Reflection and refraction rays are also cast and the results are then mixed additively with the final output colour.

We use recursion to handle multiple levels of reflection and refraction to a predefined maximum. Simplified pseudo-code for rendering and shading is given in algorithm 6.

## 3.5 Parallelisation

Ray tracing is parallelised with one thread per pixel on the screen. This thread is then responsible for the primary ray and all secondary rays, and all work involved in generating that pixel, as shown in algorithm 6. Pixels are considered independently to allow this process to be parallelised. These threads are spawned on the GPU in 32x32 grids to maximise occupancy. This is the maximum grid size on current generation CUDA GPUs, and allows us to achieve high levels of GPU utilisation as shown in figure 3.8.

This approach allows us to achieve real-time rendering for simple scenes, and given the results of Laine and Karras, 2010 should also allow more complex scenes to be rendered. However, the limitation of this approach is that, as more work is added to each thread,

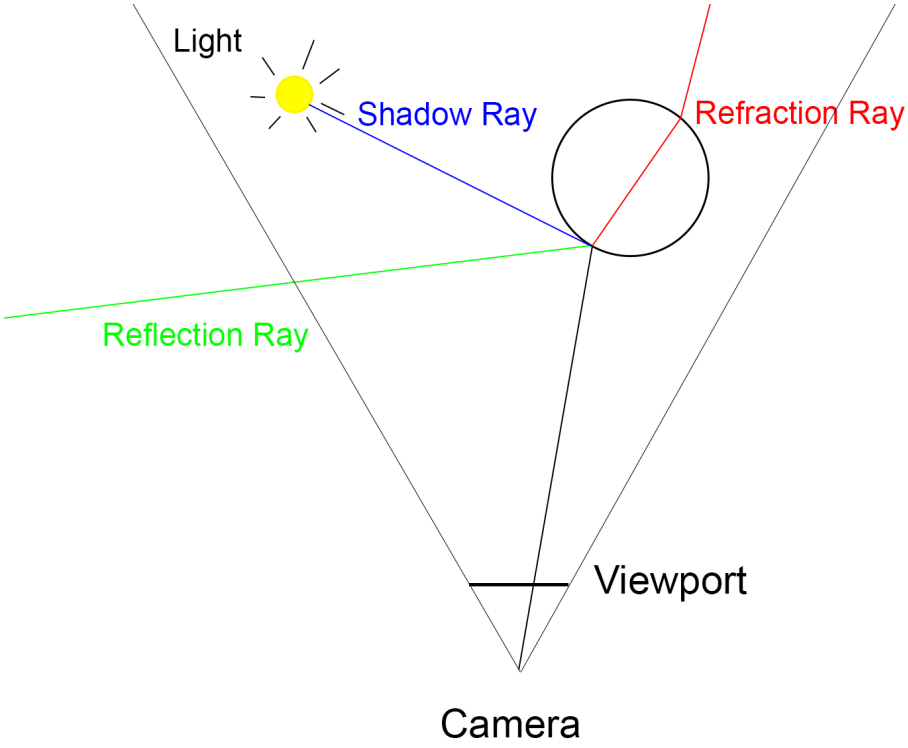



Figure 3.7: Secondary rays being spawned for calculating shadow coverage, reflection, and refraction

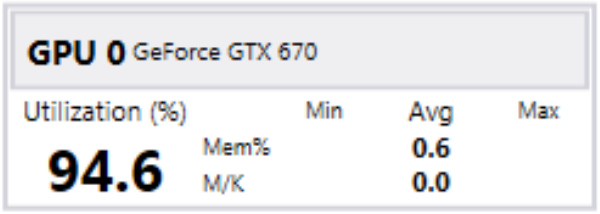



Figure 3.8: GPU utilisation for our rendering kernel, as determined using Nvidia's nsight profiler

**Algorithm 6** Per-pixel rendering

```
1: procedure RENDER_PIXEL(x, y, tree, projection)
2:     out_colour ← (0, 0, 0, 0)
3:     ray ← calculate_ray(x, y, projection)
4:     intersection ← RAYCAST(tree, ray)
5:     if intersection.hit then
6:         out_colour ← SHADE(tree, ray, intersection)
7:     end if
8:     returnout_colour
9: end procedure
10:
11: procedure SHADE(tree, ray, intersection)
12:     out_colour ← (0, 0, 0, 0)
13:     for each light do                                  ▷ Shade for each light
14:         light_dir ← light_pos − intersection.position      ▷ Check shadow
15:         shadow_ray ← (intersection.position, light_dir)
16:         shadow_intersection ← RAYCAST(tree, shadow_ray)
17:         if shadow_intersection.hit then
18:             continue
19:         end if
20:         out_colour += DIFFUSE_REFLECTION(intersection, light)
21:         out_colour += SPECULAR_REFLECTION(intersection, light)
22:     end for
23:
24:     reflection_ray ← REFLECT(ray, intersection)        ▷ Reflection
25:     reflection_intersection ← RAYCAST(tree, reflection_ray)
26:     if reflection_intersection.hit then
27:         out_colour += SHADE(tree, reflection_ray, reflection_intersection)
28:     end if
29:
30:     refraction_ray ← REFRACT(ray, intersection)        ▷ Refraction
31:     refraction_intersection ← RAYCAST_EMPTY(tree, refraction_ray)
32:     if refraction_intersection.hit then
33:         out_colour += SHADE(tree, refraction_ray, refraction_intersection)
34:     end if
35: end procedure
```

every thread in a grid must follow the most computationally heavy computation path due to the highly parallelised nature of GPUs. Possible solutions to this problem are presented in chapter 5.

## 3.6 Test data

Benchmarks for the software were taken by recording the average number of frames per second (FPS) over one minute for a given screen resolution, as well as data resolution. Screen resolution is measured in pixels, while data resolution is measured in nodes of the tree at the lowest level of the tree. This resolution is given by the formula:

$$resolution = 2^{max\_scale\ -\ 1}$$

Such that data generated down to the $11^{th}$ level has a resolution of $2^{10} = 1024$ in each dimension. This is also referred to as a resolution of $1024^3$.

Each scene has been tested across various screen resolutions as well as data resolutions in order to demonstrate how the algorithms scale with respect to screen ruesolution and data resolution.

In order to evaluate our renderer's performance we have considered the differences in performance for a sample scene with no shadows, reflection or translucency, as screen resolution and data resolution are varied. We consider this data our control sample, and following samples for rendering with shadows, reflection, and translucency is compared with this control.

We have then evaluated the relationships between screen resolution, data resolution, and performance for these samples, as well as compared the overall performance for these samples against our control to determine the effect on performance of considering these effects.

These tests were conducted on a single Nvidia GTX 670.

## 3.7 Implementation

Our implementation utilises CUDA for all rendering, bypassing the rasterisation process almost entirely. An OpenGL texture is created at program launch, and this texture is then written to by the rendering kernel using OpenGL-CUDA interoperation.

### 3.7.1 OpenGL-CUDA interop

OpenGL-CUDA interop is accomplished using a pixel buffer object (PBO) that is shared between CUDA and OpenGL. A PBO is stored in GPU memory, eliminating unnecessary copying to main CPU memory. When required by CUDA, the PBO is mapped to a device

pointer accessible by cuda, and must be unmapped before it can be accessed again by OpenGL. The PBO can then be unpacked into an OpenGL texture directly, allowing the CUDA-generated buffer to be written to the screen.

A code listing demonstrating how a PBO and a texture are created, and how this PBO is registered with CUDA is available below in code listing 3.2. A code listing for The gpuErrChk macro's code listing is available in appendix Appendix B. Once the PBO is created, it can be mapped to a device pointer as shown in code listing 3.3. Once unmapped, its contents can then be unpacked to a texture for display on the screen as demonstrated in 3.4. As all of these operations occur directly within GPU memory, unnecessary coping to main memory is avoided.

### 3.7.2 Matrix maths

Matrix maths is accomplished using GLM, the OpenGL Mathematics library. This is a C++ library which implements all of the features of GLSL and includes CUDA support, making it suitable for writing 3D rendering code and shaders in CUDA.

Listing 3.2: Creation of the OpenGL PBO and texturre

```
void* glFb;

GLuint pbo;
GLuint texid;

// Create pixel buffer object
glGenBuffers(1, &fbo);

// Create buffer texture
glGenTextures(1, &texid);

// Initilaise pixel buffer object with size
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo);
glBufferData(GL_PIXEL_UNPACK_BUFFER,
        viewport.w * viewport.h * 4 * sizeof(GLubyte),
        nullptr, GL_DYNAMIC_DRAW);
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, 0);

// Initialise texture with width, height, and format
glBindTexture(GL_TEXTURE_2D, texid);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0,
                                    GL_RGBA, GL_UNSIGNED_BYTE, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glBindTexture(GL_TEXTURE_2D, 0);

// Register pixel buffer object with cuda
gpuErrchk(cudaGraphicsGLRegisterBuffer(&glFb, pbo,
                            cudaGraphicsRegisterFlagsNone));
```

Listing 3.3: Binding the PBO so that CUDA kernels can access it

```
// Bind cuda graphics resources
gpuErrchk(cudaGraphicsMapResources(1, &glFb, 0));

// Get a device pointer to it
gpuErrchk(cudaGraphicsResourceGetMappedPointer(&ptr, &size,



// Pass device pointer to CUDA kernel

// Unmap pbo
gpuErrchk(cudaGraphicsUnmapResources(1, &glFb, 0));
```

Listing 3.4: Unpacking of a PBO to an OpenGL texture

```
glBindBuffer(GL_PIXEL_UNPACK_BUFFER, pbo);
glBindTexture(GL_TEXTURE_2D, texid);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0,
                                        GL_RGBA, GL_UNSIGNED_BYTE, 0);
```

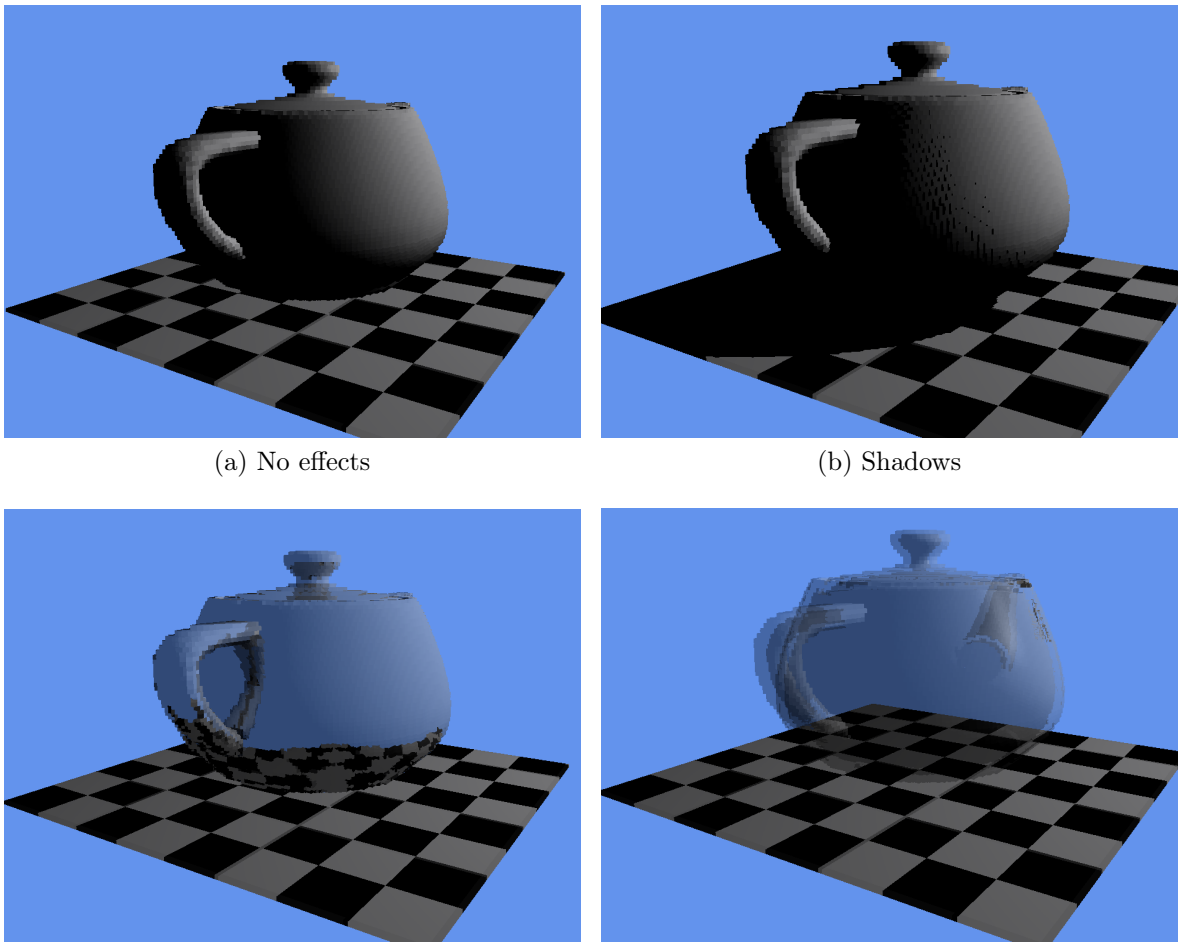
# 4. Results and Analysis

In line with our objective of creating a real-time GPU ray tracer that is capable of ray tracing heterogeneous translucent materials, there are three main factors to consider when evaluating our work. The first factor is performance. In order to run at real-time frame rates, a minimum performance threshold must be met. The next factor is the size of our data set. The data must be compact enough to fit in GPU memory and still be high resolution enough such that a high quality image can be generated. Additionally, the final image quality must be considered, to determine if the renderer can correctly handle the desired visual effects, and identify any problems with our approach.

## 4.1 Performance

In evaluating the performance of our renderer there are three main criteria to consider: that it can produce reasonable quality images in real-time at high screen resolutions; that it scales well with the resolution of the data, such that the overall quality of the image produced is comparable to that of rasterisation-based renderers; and that it scales well with the resolution of the screen, such that it can be used at the higher screen resolutions desired for games.

Figure 4.1 shows the resulting image from our sample scene at $512^3$ resolution with each effect enabled.



(a) No effects

(b) Shadows

(c) Reflection

(d) Translucency

Figure 4.1: Our test scene at a $512^3$ resolution with various options turned on

### 4.1.1 Table of results

| Scene | Screen Res | Data Res | Avg FPS |
|---|---|---|---|
| 1 Teapot on checker board plane | 768x768 | $512^2$ | 99.34 |
| | | $1024^2$ | 85.17 |
| | | $2048^2$ | 71.55 |
| | 1024x1024 | $512^2$ | 63.05 |
| | | $1024^2$ | 54.45 |
| | | $2048^2$ | 46.09 |
| | 1920x1080 | $512^2$ | 31.18 |
| | | $1024^2$ | 26.38 |
| | | $2048^2$ | 22.68 |
| 2 Teapot with shadow on checker board plane | 768x768 | $512^2$ | 77.46 |
| | | $1024^2$ | 65.32 |
| | | $2048^2$ | 55.22 |
| | 1024x1024 | $512^2$ | 49.54 |
| | | $1024^2$ | 42.04 |
| | | $2048^2$ | 36.10 |
| | 1920x1080 | $512^2$ | 23.24 |
| | | $1024^2$ | 19.40 |
| | | $2048^2$ | 16.89 |
| 3 Reflective teapot on checker board plane | 768x768 | $512^2$ | 69.83 |
| | | $1024^2$ | 58.64 |
| | | $2048^2$ | 49.13 |
| | 1024x1024 | $512^2$ | 46.68 |
| | | $1024^2$ | 37.99 |
| | | $2048^2$ | 31.93 |
| | 1920x1080 | $512^2$ | 20.44 |
| | | $1024^2$ | 16.88 |
| | | $2048^2$ | 14.18 |
| 4 Translucent teapot on checker board plane | 768x768 | $512^2$ | 39.48 |
| | | $1024^2$ | 32.32 |
| | | $2048^2$ | 25.25 |
| | 1024x1024 | $512^2$ | 24.38 |
| | | $1024^2$ | 20.11 |
| | | $2048^2$ | 15.96 |
| | 1920x1080 | $512^2$ | 10.07 |
| | | $1024^2$ | 8.30 |
| | | $2048^2$ | 6.14 |

### 4.1.2 Real-time rendering

For the purposes of our evaluation, we are defining real-time as above 30 frames per second. This is not only the minimum frame rate at which a game is expected to run, but is also above the minimum of approximately 20 required for smooth motion. Ideally, 60 frames per second is desired, as this is the refresh rate of most desktop monitors, as well as a common frame rate at which games are expected to run.

To determine if these thresholds are being met, we measure frame rate over one minute and average it.

### 4.1.3 Scaling with screen resolution

Our renderer is expected to perform at the high resolutions demanded by games. As our render is parallelised per pixel, in order to test how well this parallelisation functions, we have taken measurements for three screen resolutions: 768x768 (approximately 550k pixels,) 1024x1024 (approximately 1000k pixels,) and 1920x1080 (approximately 2000k pixels.) 1920x1080 was chosen as our benchmark resolution as it is at the upper bounds of standard display resolutions for video games, and as such, our renderer should be able to handle it. The lower resolutions were chosen due to an approximate doubling of pixels between each one, and as such it should allow us to determine whether our performance scales well with screen resolution.

### 4.1.4 Scaling with data resolution

A 3D renderer must be able to handle data of a resolution high enough to display the data smoothly. For this reason, we have chosen to measure the performance at varying data resolutions in order to determine how our methods scale with the resolution of the data.

A $1024^3$ resolution (1024 subdivisions in each axis) was chosen as this is the resolution needed to show our data up close with similar quality to a rasterisation-based renderer. Samples were also taken at $2048^3$ and $512^3$ resolutions in order to determine how the performance is affected by varying data resolutions.

**Rendering with no effects (control)**

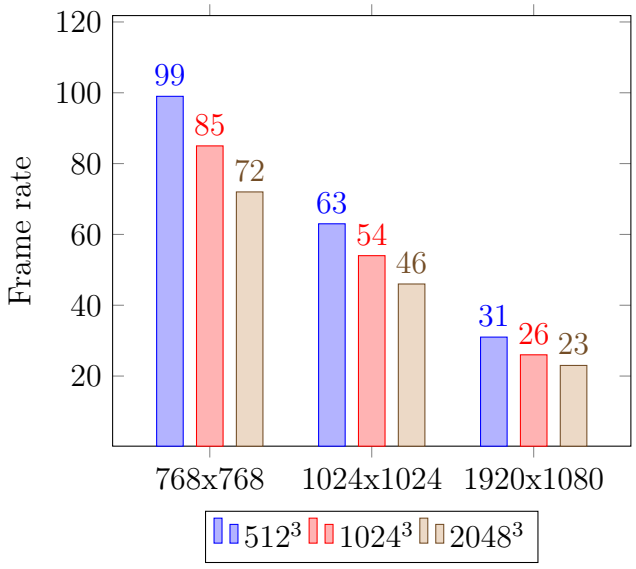



Figure 4.2: A plot of frame rate against screen resolution for our control

This data functions as our control sample, and allows us to analyse the effect on performance that different visual effects have. By examining the differences in performance with this mode of rendering, and considering the averages and ranges of these differences in performance, we can determine exactly what is affecting performance.

As shown by the graph above, our renderer exhibits a decrease in performance as the number of pixels on the screen is increased. The number of pixels increases by 78% from 768x768 to 1024x1024, with performance decreasing by 36.0% on average with a range of 0.94%. For a linear decrease, one would expect the performance to increase by $100(1 - \frac{1}{1.78}) = 44\%$, meaning that our parallelisation results in a slightly better than linear decrease in performance when the number of pixels on the screen are increased.

On the other hand, the number of pixels from 1024x1024 to 1920x1080 increases by 98%, with performance decreasing by an average of 51.0%, with a range of just 0.25%. In other words, with the number of pixels on the screen doubled, the performance halves, meaning that, at high screen resolutions such as these, the performance increases linearly.

This is disappointing, and highlights a weakness in our parallelisation scheme. At higher resolutions, our renderer drops down to and below the lower end of our desired threshold for real-time rendering. With better parallelisation, one would expect a less than linear decrease in performance, with a perfectly parallelised process operating in constant time when the number of pixels is varied.

On the other hand, rendering appears to scale well with data resolution. As the resolution of the data is increased by 8 times at each step, the performance decreases by an average of 14.7% between steps, with an average range of only 1.6%. As such, the data structure performs very well, and should allow the rendering of a range of scenes of varying resolutions.

### Rendering with shadows

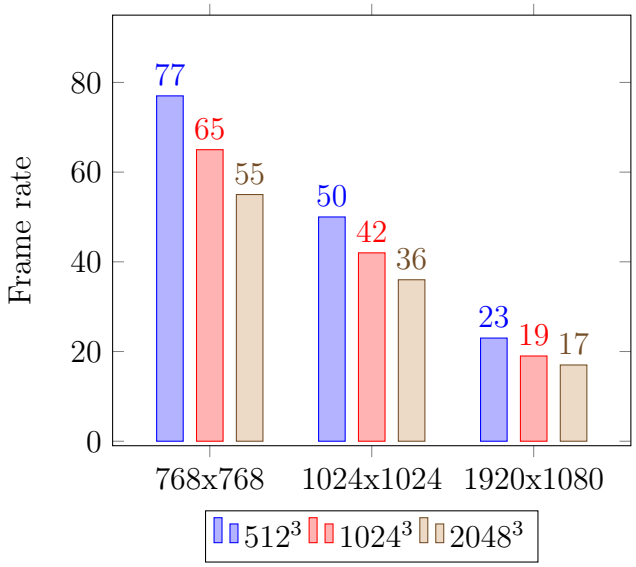



Figure 4.3: A plot of frame rate against screen resolution for rendering with shadows

As shown by the graph above, the same relationships are seen when shadows are enabled, with a minor decrease in performance. Shadow rays decrease performance by an average of 23.5%, with a range of just 5.0% between varying screen resolutions and data resolutions.

The range in performance decrease as screen resolution is varied ranges from 1.0% to 1.3%, while the same range for varying data resolutions is 3.6% to 4.0%, implying that data resolution is the major contributor to shadow performance. This makes sense, as the only difference between our control and shadow rendering (with reflection turned off, at least) is that a single extra ray per pixel is cast through the data. Despite this, the difference is very low, allowing the same relationships as in the control samples to show through.

Despite this decrease in performance, the performance still scales as well as the performance in our control samples, meaning that the major factors in scaling should still be the same factors as those for our control.

### Rendering with reflection

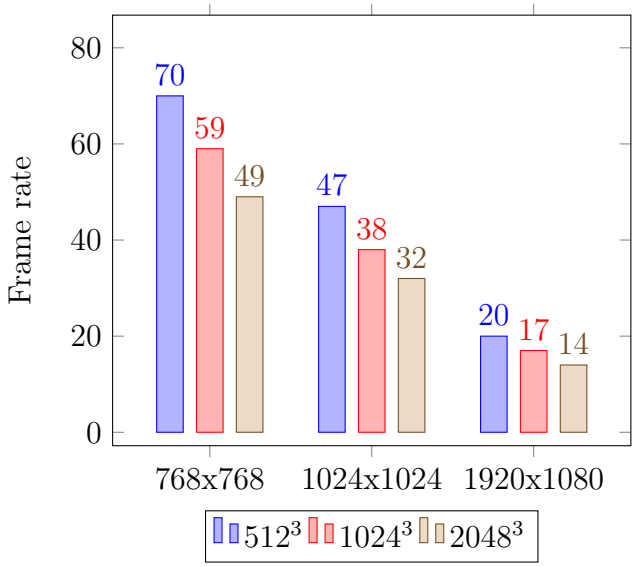



Figure 4.4: A plot of frame rate against screen resolution for rendering with reflection

Similarly to rendering with shadows, this data exhibits the same relationships as our control. This time, the average decrease in performance is 32.0%, with a much greater overall range of 11.5%. The performance decrease ranges from 1.6% to 4.8% as the screen resolution is increased, while ranging from 5.8% to 8.5% as data resolution is varied, indicating that, again, data resolution is the major factor affecting differences in performance of reflection.

Despite this, these ranges are still minor compared to the average difference, allowing the data to still show similar relationships to those shown by the control data.

Additionally, the performance does not seem to decrease significantly as data resolution is increased.

### Rendering with translucency

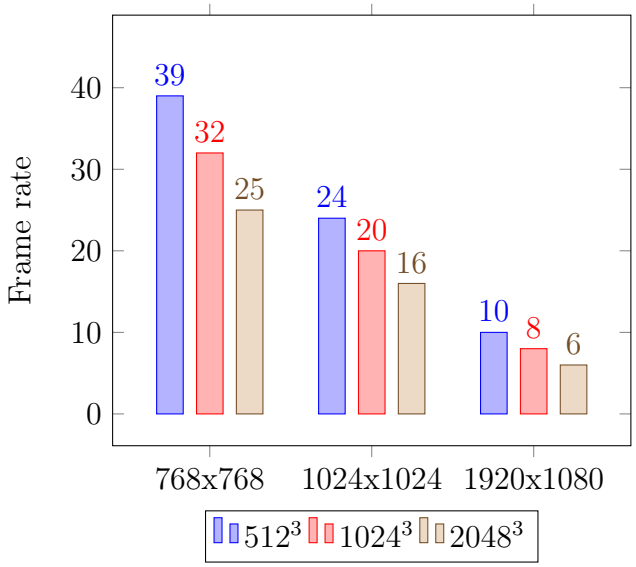



Figure 4.5: A plot of frame rate against screen resolution for rendering with translucency

This data also exhibits the same relationships as our control, but the reduction in performance is far greater when compared to shadows and reflection. The average reduction in performance for rendering our translucent object was 65.1%, with an overall range close to that of reflection at 12.7%. The range for varying screen resolution was 4.0% - 5.2%, while the range for varying data resolution was 5.5% - 8.2%. These are, again, not a major problem, as the data still exhibits similar relationships to the control data.

Despite this, the data for translucency shows a good relationship between performance and data resolution, with rendering scaling well as data resolution is increased. This makes sense, as our method of determining translucency is discrete, meaning it considers the volume at discrete, fixed intervals rather than the resolution of the data.

The overall decrease in performance, however, is an issue, with performance at 1920x1080 dropping into single digits. Even at lower resolutions, the performance dips below real-time frame rates.

We believe the decrease in performance comes from adding more work to the thread that renders each pixel. As more work is added to the thread, every thread must follow the most computationally expensive execution path due to the SIMD nature of the GPU. In order to address this, we believe that investigation into alternate parallelisation schemes is required.

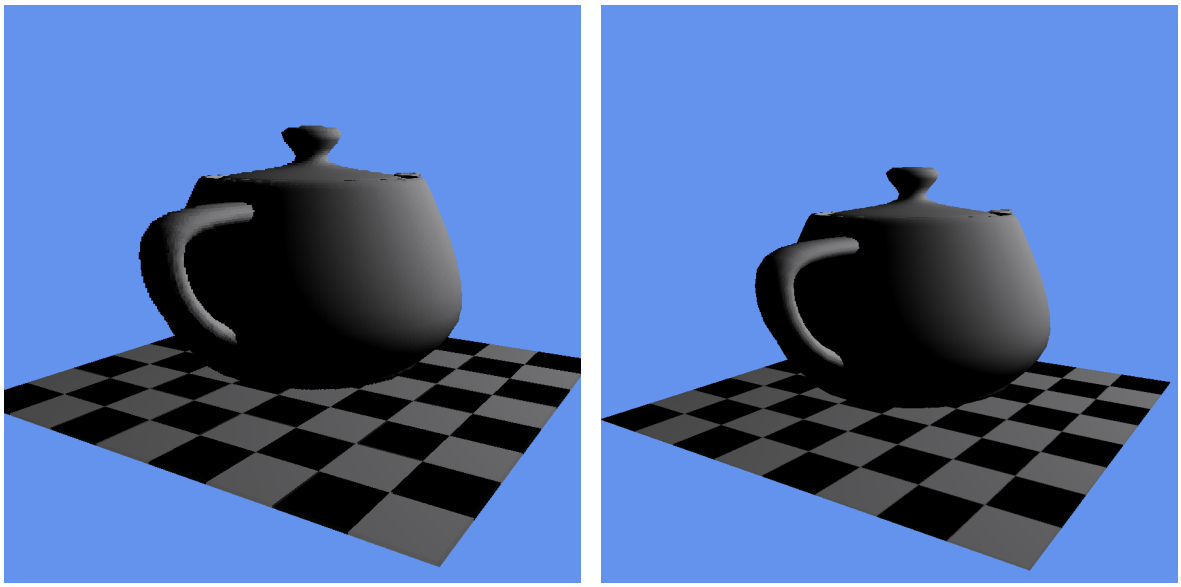
The persistent threads technique demonstrated in Aila and Laine, 2009 seems like it could improve the parallelisation of our renderer. Firstly, the number of threads spawned on a GPU is related to achieving maximum occupancy on that GPU rather than the number of pixels on the screen, which we have shown to drop off in performance gains at higher resolutions.

Additionally, as the processes for the initial intersection with the volume, shadows, reflection and refraction all require casting rays within the volume, we believe that it may be possible to create persistent threads that can handle any of these operations, rather than having one thread that handles all of them.

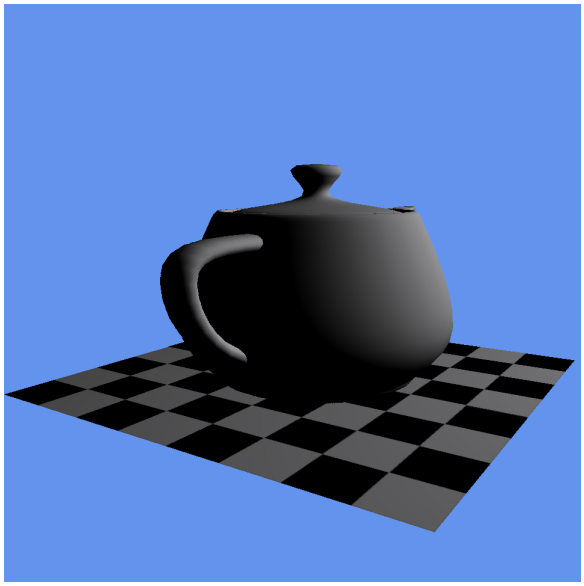
## 4.2 Memory usage

Memory usage is an important factor to consider in real-time rendering. Ideally, all required data should be compact enough to be stored in memory. Even if this is not the case, it is desired for the memory usage to be as low as possible to reduce the amount of memory that must be considered for streaming. In order to evaluate memory usage, we have looked at the memory usage of our test scenes from section 4.1. Our data structure encodes volume data as well as shading data, which must be taken into account when analysing this data.

Figure 4.6 shows the memory usage of our test scene at each resolution.



(a) $512^3$ resolution, memory usage 34,354kB (b) $1024^3$ resolution, memory usage 137,141kB

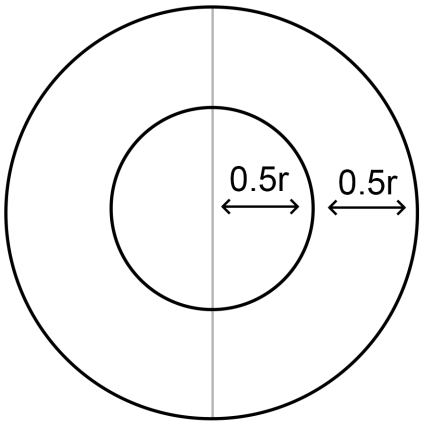


(c) $2048^3$ resolution, memory usage 543,832kB

Figure 4.6: Our test scene at various resolutions and the resulting memory usage

As can be seen, the memory usage increases significantly for every power-of-two increase in resolution. In practice, a $1024^3$ resolution is often sufficient for smooth rendering, however up close, greater resolutions may be required. Despite the structure being compact enough to fit into GPU memory for our example scene, the memory usage is significantly greater than polygon-based meshes. Despite this structure including both volume and shading data, it still has unsatisfactorily high memory usage.

Two major approaches have been been suggested by other works to reduce the memory usage of the sparse voxel octree: contours (Laine & Karras, 2010), which utilise non-cubical voxels to increase the approximation of the original data, removing the need for deeper encoding once the approximation becomes sufficient; and sparse voxel DAGs




(a)

Figure 4.7: A 2D cross section of the sphere used in this experiment. In one data set, the centre is hollow. In the other, it has a refractive index of 1.0

(Kampe et al., 2013), which allow identical regions to share pointers, and can reduce the memory usage of such structures by one to three orders of magnitude. Despite the high memory usage of our structure, these techniques should also be applicable to our work.

## 4.3 Image quality

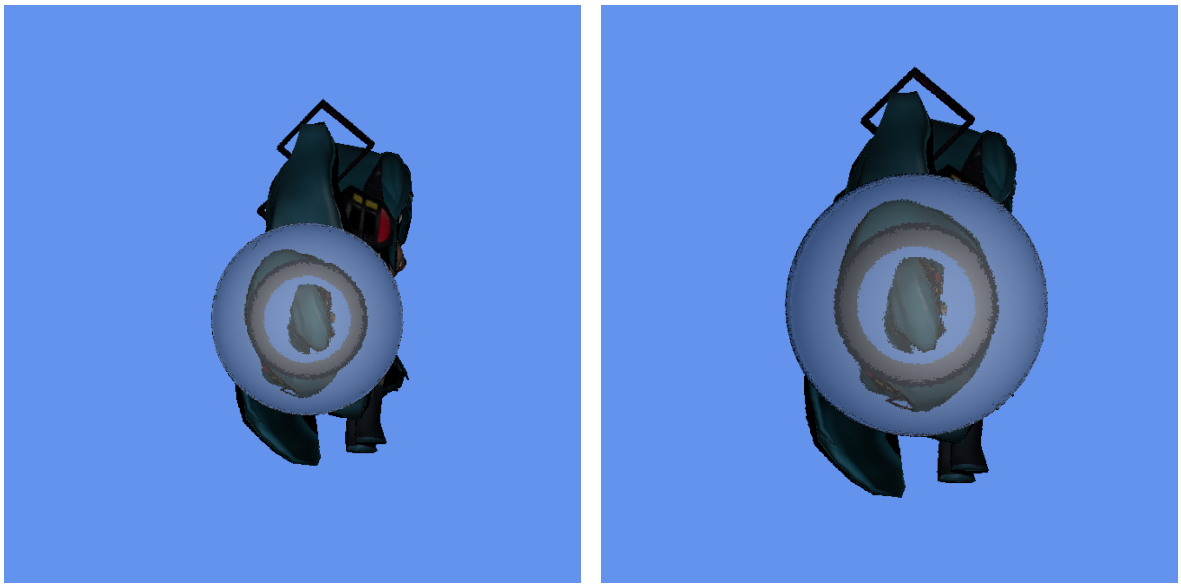
### 4.3.1 Accuracy of heterogeneous refraction

In order to test the accuracy of our method of calculating heterogeneous refraction, we utilise two sets of data: a sphere with refractive index 1.5, with a hollow half-radius sphere embedded in it (see figure 4.7); and a sphere with refractive index 1.5, with a half-radius sphere embedded inside it of refractive index 1.0. In theory, as the refractive index of air is considered to be 1.0, the resulting images should be identical.

As can be seen in figure 4.8, our heterogeneous refraction works as intended. It is important to note, however, that this does not prove that the result is physically accurate. Further work is required, comparing the output of our renderer to real objects that exhibit varying indices of refraction, in order to determine its physical correctness.

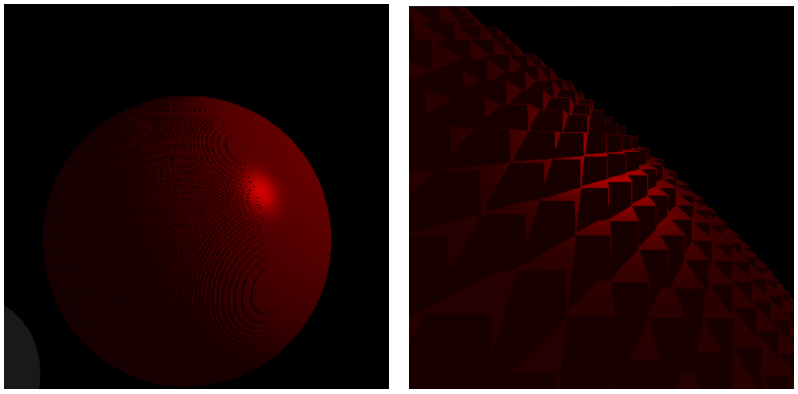
### 4.3.2 Issues due to the cubical nature of voxel data

As can be seen in figure 4.9, the cubical nature of voxels poses a great issue when considering secondary rays. As a sphere is a convex shape, it should, in theory, be unable to cast any shadows on its own surface. Despite this, the voxel approximation of a sphere casts shadows on itself, causing unsightly black artifacts. This problem generalises to any



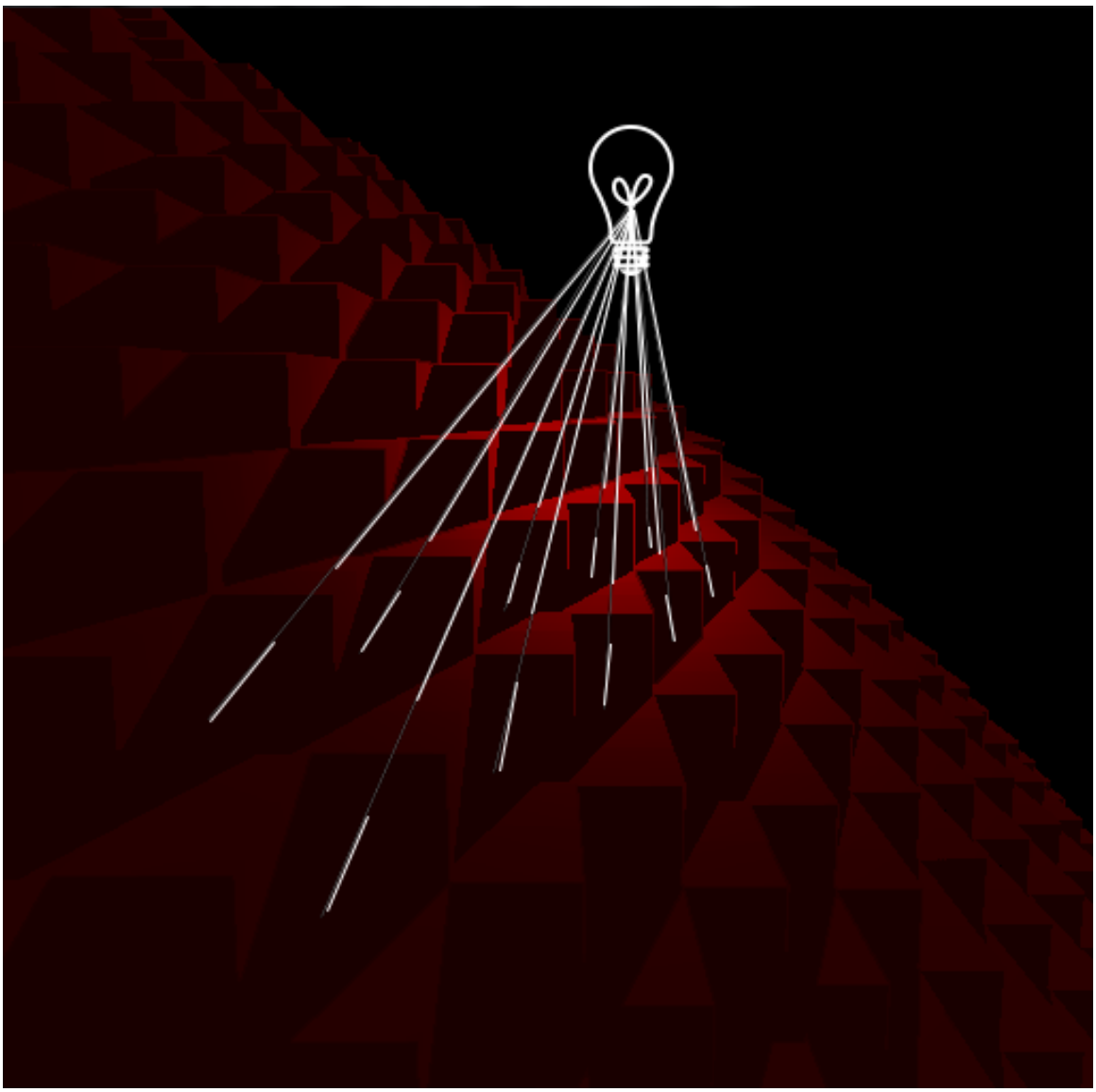
(a) Hollow centre (b) Refractive index 1 centre

Figure 4.8: The resulting images are identical



(a) A voxel approximation of a sphere (b) The phenomenon up close

Figure 4.9: A voxel approximation of a sphere casting shadows on itself



(a)

Figure 4.10: The path of a shadow ray from the surface. The ray does not make it to the light, as it intersects with other voxels first

curved surface, as well as other types of secondary rays, such as reflection and refraction rays. Figure 4.10 demonstrates why this occurs, by considering the path of a shadow ray from part of the surface that is in shadow.

One possible solution to this problem is to increase the data resolution. As data resolution increases, the problem lessens. The problem with this solution is that it's hard to predict, in a general way, what resolution the data will need to be stored at in order to prevent these types of artifacts in any given situation. Even lessening the effect in this way is extremely memory inefficient.

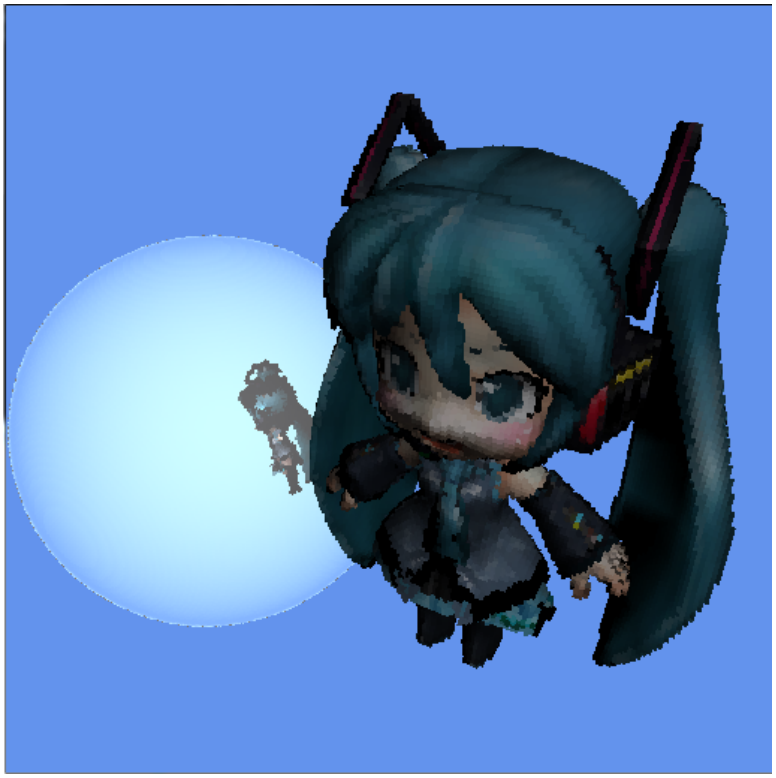
There are two main alternatives to this solution. Since the problem only occurs for surface voxels, one possible solution is to make the ray casting algorithm aware of surface normals in order to perform a sort of back-face culling. The results of this are shown in figure 4.12. The major disadvantage of doing this is that it adds a number of expensive

(a)

Figure 4.11: The problem also occurs with reflection

calculations to the inner loop of the ray cast, greatly reducing the overall performance of the renderer. On top of this, it would be more desirable to extend such a solution to one that does not involve considering surfaces, in order to stay in line with our volumetric rendering goals.

A more generic version of this solution would use non-cubical voxels in order to improve the approximation a voxel can make of a surface. Laine and Karras's contours technique, which can efficiently consider non-cubic voxels using a pair of parallel planes to bound a volume inside the voxel, does this. We believe that this technique would also solve our problems in this area, as well as greatly lessening the memory usage of our structure as discussed in section 4.2.



(a)

Figure 4.12: Making the ray casting algorithm aware of the normals of surfaces can solve the problem, but has disadvantages

# 5. Conclusions and Future Work

In this work we have produced a volume renderer which, in line with our objectives, is capable of rendering heterogeneous structures in real-time by utilising the highly parallelised architecture of the GPU.

Our renderer is capable of obtaining comparable quality to rasterisation-based rendering while also producing volumetric effects such as refraction through a heterogeneous volume. The data structure we have chosen is capable of encoding highly heterogeneous as well as homogeneous volumes, allowing for a dynamic sampling of the original volume which enables the renderer to perform at varying data resolutions.

We believe that this is a promising approach which is capable of producing real-time volumetric effects for consumer applications such as video games at real-time frame rates, without having to rely on non real-time techniques such as the impostors technique used in Harris, 2002 which allow data to be reused between frames. Because of this, our renderer is capable of producing highly dynamic effects that don't rely on coherent results.

Despite our success in achieving our objectives, there are some limitations to our work.

## 5.1 Memory usage

The memory usage of our data structure is sufficient for simple scenes, but as we store sub-surface volumes, unlike approaches such as that used in Laine and Karras, 2010 which only encode the surfaces as volumes, our memory usage very quickly becomes unsatisfactorily high as the resolution of the data is increased. Despite this, the encoding of sub-surfaces is necessary in order to consider volumetric effects, otherwise the advantages over polygon-based approaches are diminished.

This excessive memory usage may become an issue for complex scenes, where a higher resolution is required in order to encode larger structures down to similar precisions. Promising approaches for reducing the memory usage of the sparse voxel octree have been researched. One such approach is the contours utilised in Laine and Karras, 2010, while another is the sparse voxel DAG presented in Kampe et al., 2013.

## 5.2 The disadvantages of cubical voxels

Another key limitation of our work is that, due to the cubical nature of voxels, secondary rays cast from smooth surfaces may be able to intersect with themselves where they otherwise would not be able to. This produces many undesired visual effects, including the voxels of a sphere casting shadows on the sphere's surface.

We believe that this is a limitation that can be overcome by considering surface voxels to not be cubical. The contours approach by Laine and Karras, 2010 would also solve this problem, as it is an efficient method of accomplishing non-cubical voxels while adding little work to the inner loop of each ray cast.

## 5.3 The scaling of performance as screen resolution is increased

At resolutions higher than 1024x1024, our performance begins to scale linearly with screen resolution. Although this may be acceptable with higher performance, it highlights a weakness of our parallelisation scheme. It would seem that alternate paralellisation schemes, such as the persistent threads approach from Aila and Laine, 2009 may benefit our work immensely.

## 5.4 Future work

Our work demonstrates that real-time volumetric effects within the GPU are possible, but falls short of producing real-time frame rates at full-screen resolutions. For this reason, future work would be best targeted at improving the parallelisation of our approach such that it scales better at these resolutions.

Additionally, as we only consider volumetric refraction, adapting our work to consider more complex volumetric effects such as the sub-surface scattering effects encountered when considering the lighting of clouds would allow a far wider range of real-life objects to be rendered volumetrically.

Once these limitations have been overcome, our work points to the possibility of such effects appearing in consumer software, such as games, in the near future.

# References


Aila, T. & Laine, S. (2009). Understanding the efficiency of ray traversal on gpus. In *Proc. high-performance graphics 2009*.

Amanatides, J. (1984). Ray tracing with cones. In *Proceedings of the 11th annual conference on computer graphics and interactive techniques* (pp. 129–135). SIGGRAPH '84. New York, NY, USA: ACM. doi:10.1145/800031.808589

Amanatides, J. & Woo, A. (1987). A fast voxel traversal algorithm for ray tracing. In *In eurographics '87* (pp. 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* (pp. 37–45). AFIPS '68 (Spring). Atlantic City, New Jersey: ACM. doi:10.1145/1468075.1468082

Arvo, J. (1988). Linear-time voxel walking for octrees. ftp://ftp.sgi.com/other/bspfaq/faq/ltvw.html. Accessed: 2014-05-01.

Bautembach, D. (2011). Animated sparse voxel octrees. *Thd Thesis. University of Hamburg*.

Bikker, J. & van Schijndel, J. (2013). The brigade renderer: a path tracer for real-time games. *Int. J. Computer Games Technology, 2013*. Retrieved from http://dblp.uni-trier.de/db/journals/ijcgt/ijcgt2013.html#BikkerS13

Blinn, J. F. (1977, July). Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph. 11*(2), 192–198. doi:10.1145/965141.563893

Cook, R. L., Porter, T., & Carpenter, L. (1984). Distributed ray tracing. In *Acm siggraph computer graphics* (Vol. 18, pp. 137–145). ACM.

Dimov, I., Penzov, A., & Stoilova, S. (2007). Parallel monte carlo approach for integration of the rendering equation. In T. Boyanov, S. Dimova, K. Georgiev, & G. Nikolov (Eds.), *Numerical methods and applications* (Vol. 4310, pp. 140–147). Lecture Notes in Computer Science. Springer Berlin Heidelberg. doi:10.1007/978-3-540-70942-8_16

Engel, K. & Ertl, T. (2002). Interactive high-quality volume rendering with flexible consumer graphics hardware.

Garcia, I., Sbert, M., & Szirmay-Kalos, L. (n.d.). Tree rendering with billboard clouds. Citeseer.

Glassner, A. S. (1984). Space subdivision for fast ray tracing. In *In eurographics '87* (pp. 15–22).

Glassner, A. (1989). *An introduction to ray tracing*. Academic Press. Academic Press. Retrieved from http://books.google.co.uk/books?id=YPblYyLqBM4C

Gouraud, H. (1971, June). Continuous shading of curved surfaces. *Computers, IEEE Transactions on, C-20*(6), 623–629. doi:10.1109/T-C.1971.223313

Harris, M. J. (2002). Real-time cloud rendering for games.

Heath, T. (1999). *A history of greek mathematics: from aristarchus to diophantus*. Elibron Classics Series. Adamant Media Corporation. Retrieved from http://books.google.co.uk/books?id=zGIYbEtzD-QC

Kajiya, J. T. (1986). The rendering equation. In *Computer graphics* (pp. 143–150).

Kampe, V., Sintorn, E., & Assarsson, U. (2013, July). High resolution sparse voxel dags. *ACM Trans. Graph. 32*(4), 101:1–101:13. doi:10.1145/2461912.2462024

Kessenich, J., Baldwin, D., & Rost, R. (2014, January 22). The opengl shading language. Retrieved from http://www.opengl.org/registry/doc/GLSLangSpec.4.40.pdf

Laine, S. & Karras, T. (2010). Efficient sparse voxel octrees. In *Proceedings of the 2010 acm siggraph symposium on interactive 3d graphics and games* (pp. 55–63). I3D '10. Washington, D.C.: ACM. doi:10.1145/1730804.1730814

Lorensen, W. E. & Cline, H. E. (1987). Marching cubes: a high resolution 3d surface construction algorithm. In *Proceedings of the 14th annual conference on computer graphics and interactive techniques* (pp. 163–169). SIGGRAPH '87. New York, NY, USA: ACM. doi:10.1145/37401.37422

Phong, B. T. (1975, June). Illumination for computer generated pictures. *Commun. ACM*, *18*(6), 311–317. doi:10.1145/360825.360839

Purcell, T. J., Buck, I., Mark, W. R., & Hanrahan, P. (2002, July). Ray tracing on programmable graphics hardware. *ACM Trans. Graph. 21*(3), 703–712. doi:10.1145/566654.566640

Roth, S. D. (1982). Ray casting for modeling solids. *Computer Graphics and Image Processing*, *18*(2), 109–144. doi:http://dx.doi.org/10.1016/0146-664X(82)90169-1

Ruff, C., Clua, E., & Fernandes, L. (2013, August). Dynamic per object ray caching textures for real-time ray tracing. In *Graphics, patterns and images (sibgrapi), 2013 26th sibgrapi - conference on* (pp. 258–265). doi:10.1109/SIBGRAPI.2013.43

Wilhelms, J. & Gelder, A. V. (2000). Octrees for faster isosurface generation. *IEEE TRANSACTIONS ON MEDICAL IMAGING*, *19*, 739–758.

# Appendix A Lookup table used for accessing child voxels using a parent's child descriptor

Listing 1: The lookup table and function used for accessing child voxels using a parent's child descriptor

```
// A table for getting the child index for the child child_index
// the index in this array = (parent's_childmask << child index)
__constant__ int32_t child_index_table[] =
{
        0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4,
        1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
        1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
        1, 2, 2, 3, 2, 3, 3, 4, 2, 3, 3, 4, 3, 4, 4, 5,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
        2, 3, 3, 4, 3, 4, 4, 5, 3, 4, 4, 5, 4, 5, 5, 6,
        3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
        3, 4, 4, 5, 4, 5, 5, 6, 4, 5, 5, 6, 5, 6, 6, 7,
        4, 5, 5, 6, 5, 6, 6, 7, 5, 6, 6, 7, 6, 7, 7, 8,
};

__device__ int32_t get_child_index(uint32_t mask,
                                   uint32_t child_idx)
{
        return child_index_table[(mask << child_idx) & 0xFFu];
}
```

# Appendix B CUDA Utils

Listing 2: Macro that checks CUDA calls for errors (gpuErrChk)

```
#define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }
void gpuAssert(cudaError_t code, char *file,
                                        int32_t line, bool abort)
{
        if (code != cudaSuccess)
        {
                fprintf(stderr,"GPUassert: %s %s %d\n",
                cudaGetErrorString(code), file, line);

                system("pause");

                if (abort)
                        exit(code);
        }
}
```