University of Birmingham

School of Computer Science
Final year project

Reducing octree depth to more efficiently traverse during ray-trace computations by creating non-uniform voxel octrees

Author: **Felix Hiller** (1446415)

MSci Computer Science

supervised by
Dr Martín Escardó
Reader in Theoretical Computer Science
August 13, 2018

Contents

Abstract

Generating octrees splits a three-dimensional scene in the spatial centre. Our hypothesis is that by splitting the scene in an optimally chosen centre, we can decrease the time taken to subsequently ray-trace over the octree structure. We conjecture that the optimal splitting point for the cuboid voxels can be calculated in a small number of steps using modern techniques. In this paper we compare the performance of the traditional cube octree generation method and our new cuboid octree generation method to create the octree, and subsequently ray-trace over a scene. To properly test our hypothesis we have created a ray-tracer from the ground up so that we may implement our optimisation with as much control over the design as possible.

All the code and test files for the project can be viewed at `https://git-teaching.cs.bham.ac.uk/mod-ug-proj-2017/fxh416.git`. Information about building and running the project are located in Appendix sections B and C.

Acknowledgements

I would like to thank my parents for being so supportive and helpful throughout the project. I would also like to thank my housemates who were there to help whenever I found myself stuck, specifically Misha Wagner and Dan Jones for helping push me over some of the larger hurdles of the project. Finally I want to thank my supervisor Martin Escardo for his ongoing advice and motivation.

1 Introduction

The objective of this project is to improve the speed of ray-trace computations by changing how the objects inside any particular scene are stored and subsequently ray-traced over. In this section we describe how regular ray-tracing works and how our optimisation should improve ray-trace speeds.

1.1 Current methods of image synthesis

1.1.1 Rasterisation and ray-tracing

Image synthesis is done in two ways: rasterisation and ray-tracing. Rasterisation is the process of projecting a scene's three-dimensional vertices onto the screen using perspective projection, then checking whether the pixels reside within the contained shape. In comparison, ray-tracing generates an image by computing a ray for each pixel projected from the camera position and then checking the ray's intersection with the objects in the scene. If there's an intersection, we compute a shadow ray directed at each light source to compute the pixel's colour (as shown in Figure 1).

Without any optimisations, rasterisation is much faster than ray-tracing; however, rasterisation does not produce shaded images and further processing must be performed to achieve shading, unlike ray-tracing, which will generate realistic shading naturally. Additionally, ray-tracing can generate real-time images for massive models with many polygons[2]. Even with this extra post-processing that rasterisation has to do to achieve images of a similar quality, it's still faster than ray-tracing and is therefore used for the majority of real-time applications[3].

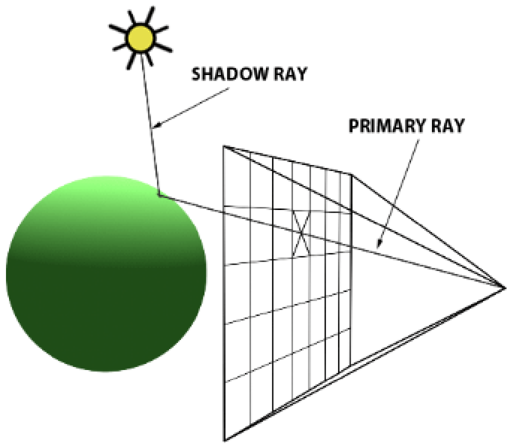

Figure 1: We fire a primary ray through the centre of each pixel to check object intersections, then cast a shadow ray for each light source, computing the pixel's colour[1].

With the large growth in processing power in recent years, interactive and non-interactive ray-tracing is being used in various industries that require a real-life accurate image or large models with millions of polygons[3]. By reducing the rendering times of ray-tracing, we might see wider-scale use across more varied industries or just improved use in its current industries.

1.1.2 Ray-trace optimisations

In order to reduce the largest culprit of ray-trace computation time (ray–object intersections), there are currently two main ideas: bounding volume hierarchies (BVH) and space subdivision. Both attempt to store the objects in different data structures and compare the rays with simpler primitive shapes to reduce the total number of ray–object intersection checks. While both methods introduce some overhead in interacting with the data structures, this is negligible compared with the speed-up introduced.

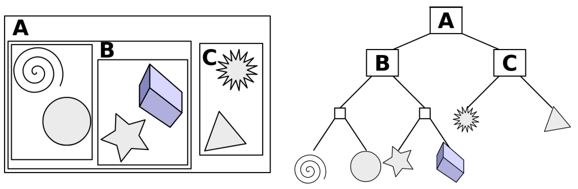

Figure 2: An example of a bounding volume hierarchy [4]

BVHs encapsulate the geometric objects in the scene with larger, simpler shapes (two-dimensional example shown in Figure 2); these are then placed into a tree structure [5]. Each leaf node of the tree is a geometric object, with each non-leaf node being a bounding volume to its children objects. When we compute a ray's intersection with the scene we first compute its intersection with the root node of the tree. If they intersect, we repeat recursively with the bounding volume's child nodes until a shape intersection is reached. Ideally the bounding volumes tightly surround the encapsulated objects, reducing the number of false-positive intersection tests we get (where the ray intersects with an object's bounding volume, but not the object itself, causing wasted clock cycles on ray–object intersection checks).

Instead of splitting our shapes in the scene into a tree structure we can subdivide the scene itself. Octrees encapsulate the geometric objects by dividing the scene into axis-aligned, equally sized voxels. Objects are then stored in the leaf nodes of the octree [6]. When we ray-trace across the scene, we can calculate the next voxel in the ray's path, and then subsequently check its intersection with any contained objects. If no intersection is found we can continue on to the next voxel.

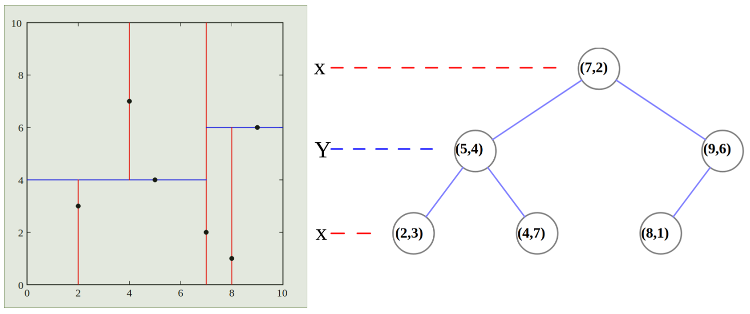

Figure 3:
Left: kD-tree for the point set: (2,3), (5,4), (9,6), (4,7), (8,1), (7,2)
Right: kD-tree for point set

An alternative to octrees are kD-trees. These are used to store the scene as a binary search tree structure. Each non-leaf node recursively partitions the scene in two hyperplanes that are perpendicular to an axis of the k-dimensional coordinate system (two-dimensional example shown in Figure 3). The axis to which the hyperplane is perpendicular changes every time we go down a layer of the tree. Unlike in Figure 3, kD-trees used in ray-tracing have leaf nodes that contain the objects within the specified sub-space[7]. kD-trees have the most optimal performance for static scenes; however, for dynamic/interactive scenes we must rebalance the tree every time the scene changes, and this rebalancing is extremely costly[8].

While for large, complex, static scenes, kD-trees have been shown to have the best performance of all modern data structures[8], for dynamic scenes it's still unclear which is the most optimal data structure. Some hybrid structures that attempt to improve performance of these base data structures have been suggested (irregular grids[9], bounding interval hierarchies[10]) and their performance in dynamic *and* static scenes is considerably greater than their standard counterparts.

1.2 Non-uniform voxel octree

A cause of many false-positive intersection checks is relaxed bounds for the contained objects, leaving a lot of room for rays to intersect the bounds, but not the enclosed shape, wasting clock cycles on ray–object intersection checks. This is shown in Figure 4, this occurs when our surface area heuristic (SAH; described in Section 2.2.3) has poorly constructed our octree.

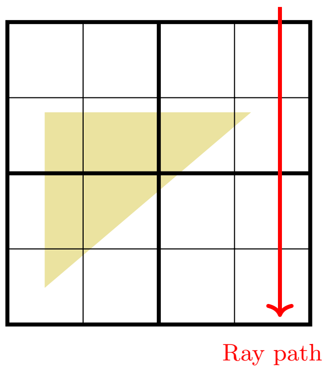

Figure 4: An octree with few divisions: when a ray passes through a cell that contains part of the object, a ray–object intersection calculation must occur

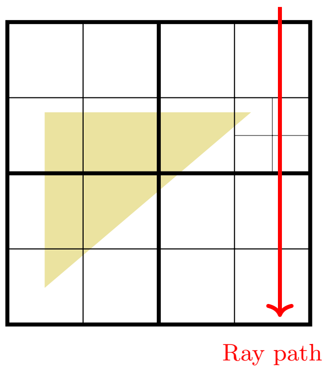

Figure 5: Simply subdividing further reduces the ray–object intersection checks, but increases the number of voxels in the tree, which increases the number of ray bound checks

With a properly constructed octree we reduce the number of ray–object intersection checks by subdividing further (shown in Figure 5). However, by increasing the number of voxels of the tree we increase the number of times our trace must find the next voxel in the octree. While these voxel checks are relatively quick in comparison with ray–object checks, they're still not completely free.

In order to reduce ray–object intersections and keep the size of the octree at a minimum, we have altered the shape of octree subdivisions to be non-uniform in size, better encapsulating the enclosed objects (shown in Figure 6). The only overhead encountered in relation to a regular octree structure is the storage of the size of each voxel. This is a trade-off on memory storage to computational time.

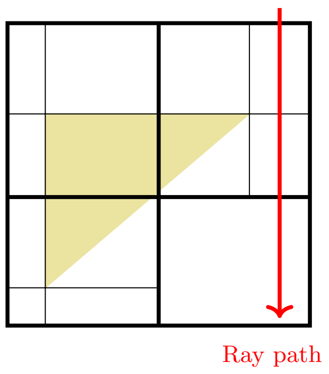

Figure 6: By altering subdivision shape, we reduce depth of tree but retain tight bindings around objects

Our traversal algorithm will be very similar to regular octree traversal [11]. Since advancing to the next bounding box is non-trivial, we must calculate the next voxel by using the current voxel's size in relation to our current direction.

Since the octree traversal for similar ideas has already been shown to decrease traversal time [9], if the non-uniform octree generation algorithm has similar generation times to uniform octrees we're likely to gain temporal improvements. The generation algorithm will be similar in concept to a regular octree generation algorithm in the way it chooses when to subdivide further using the SAH. However, before the octree subdivides it must decide an optimal midpoint to subdivide at. Following MacDonald and Booth's ideas [12], the most optimal splitting plane for any division lies along one of the bounding boxes of the contained objects. Additionally they found that the optimal splitting point also lies between the spatial median; and the object median, this helps to reduce the search space when trying to find an optimal midpoint. However, we aimed to reduce this even further through other techniques (discussed in Section 3).

2 Background

In this section we review and compare literature containing similar goals and ideas to our own. We also put forth the background mathematical theory needed to understand the rest of the paper.

2.1 Other literature

The conventional model for ray-tracing comes from the 1980 paper from Whitted [13], which describes the intersection process as a reflection of additional rays that point towards objects in the scene, acting as light sources. This was a new idea at the time; previously, light sources were assumed to be at a point infinitely far from the scene [14]. This is obviously flawed, as it limits the scenes we can draw effectively. Whitted's paper goes further, and describes the ray-tracing algorithm that is the basis for all ray-tracing to this day. While the algorithm described does attempt to optimise itself using bounding volume spheres, the algorithm is not optimised enough for any real-time or dynamic applications. Whitted shows us that over

95% of picture-generation time is spent calculating ray–object intersections[13], so this is an obvious vector for improvement.

In 1990 an SAH was proposed by MacDonald and Booth to optimise octree construction[12]: the heuristic is used to assign a cost to any particular octree construction. This is then used to compare any octree against its subdivided version; if the SAH stops decreasing, it informs us to stop subdividing the scene. The heuristic is still used in modern tree-generation techniques[15].

A type of non-uniform octree was suggested in 1995 by Kyu-Young Whang et al. dubbed "Octree-R"[16]. In their attempt to create a more optimal structure they applied similar methods to us to improve ray-trace times. However, they computed the optimal splitting plane identically to kD-trees, using a separate algorithm for each dimension that they split in, effectively performing kD-tree generation, but condensing three layers of the kD-tree into one. The algorithm they use computes the surface area of the subdivided nodes, as well as the surface area of the current node, which we feel is an impairment to their generation method, as the surface area computations are not needed and therefore a waste of compute time. Furthermore, computing all planes' optimal points separately reduces future opportunities for optimisation (further detail in Section 3).

In 1998 Klosowski et al. suggested using a hierarchy of bounding volumes[17], computing the ray's intersection with a node, then recursively computing intersections with that node's children, should they intersect. The bounding volumes' shapes now often change based upon several factors: the computational cost of the bounding object, the cost of updating the object for dynamic scenes, intersection costs and how closely the bounding volume encapsulates the object. As the bounding volumes increase in sophistication and complexity to better encapsulate their contained object, they lose the ease of transformation in dynamic scenes and increase intersection test times.

In 2000, improvements were made on the traversal times of octree structures. Glassner shows us that the process of discovering the next voxel our ray passes through can be sped up. We first find a point within the next voxel, then search a master linked list to find that voxel easily. Finding the point within the next voxel is inexpensive as we just need to find the maximum point of intersection that our ray has with the bounding planes of the current voxel, and then add some tiny value in the same direction (it is suggested that this tiny value is proportional to the size of the smallest voxel in our octree). This costs just one subtraction and one divide operation per bounding face of the current voxel, which is cheap compared to the cost of intersection[6].

We use the surface area heuristic (described in Section 2.2.3) to decide when to subdivide each node[12]. The SAH used is inherently greedy as it only considers the next subdivision, even if multiple subdivisions are needed to improve performance. While attempting to use the SAH in a non-greedy way would improve ray-tracing performance by generating a more optimal octree, the octree generation time would also increase. Whether this trade-off has overall benefits is unclear and may be an avenue to explore further. Glassner gives some methods for improving performance of the octree, such as hashing the node's identifier and storing its contained objects in a linked list which, while reducing memory consumption and speeding up voxel look-up, fails to reduce false-positive intersection checks.

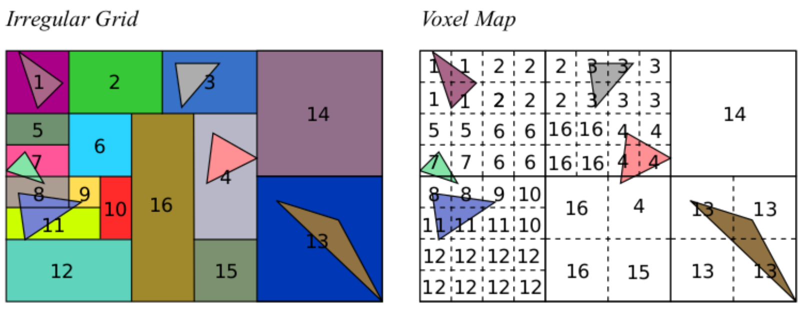

Figure 7: The irregular structure (left) and associated underlying octree structure (right) [9]

Recently (May 2017) Perard-Gayot, Kalojanov and Slusallek proposed an irregular grid data structure [9]. In the suggested data structure, the underlying octree remains as regularly subdivided axis-aligned voxels (shown in Figure 7). The novel idea is to index these voxels to effectively merge cells across octree bounds. This attempts to achieve the close binding of BVHs on complex structures while keeping the effective dynamic scene advantages of octree data structures. The algorithm for construction of their irregular structure makes effective use of GPU parallelisation to achieve comparatively speedy ray-tracing during dynamic scenes. However, their use of bounding boxes that are aligned with the underlying regular octree structure restricts how tightly these boxes can envelop their contained objects, potentially increasing the number of false-positive intersection checks using these bounding boxes.

The non-uniform octree construction algorithm needs to additionally decide where to make the split. This problem has been solved for kD-tree generation, as described by MacDonald and Booth [12]. However, kD-trees only split one axis at a time, so for the non-uniform octree construction we've altered the algorithm to take all dimensions into account (see equation 19).

2.2 Theory

This section outlines some basic mathematical theory and equations that are essential to understanding the design and implementation sections of the project. We recommend this section is used as a reference while reading the other sections of the project.

2.2.1 Parametric and implicit shapes

In order to effectively describe an object in a scalable way we must know its parametric notation; this simplifies ray–object intersections and helps us design our program.

Vectors We can describe all points along a vector as a point and a vector:

- O = any point on the vector (for our purposes, usually the origin of a ray)
- $\vec{v}$ = the direction of the vector

In order to find a point on the vector we multiply the vector $\vec{v}$ by some explanatory variable t and add the point p.

$$V(O, \vec{v}) = O + t\vec{v} \tag{1}$$

Spheres We can describe all points on the surface of a sphere as a centre point and a radius:

- $C =$ midpoint of the sphere
- $R =$ radius of the sphere

We can see whether any point P lies on the sphere by checking if the following equation holds:

$$(P - C)^2 - R^2 = 0 \tag{2}$$

The quadratic formula is used when computing sphere intersections. It provides solutions to a quadratic equation of the form $ax^2 + bx + c = 0$ where x is some unknown and a, b and c are constants. Each solution to the quadratic formula is a root of the quadratic equation given.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{3}$$

Planes We can describe the points along an infinite plane using a point and a normal vector.

- $p =$ a point on the plane
- $\vec{N} =$ the normal vector of the plane

We can see whether any point r lies on the plane by simply checking if the vector $r - p$ is perpendicular to the normal $\vec{N}$. This can be done by checking if the dot product between the two is equal to 0.

$$\vec{N} \cdot (r - p) = 0 \tag{4}$$

Triangles We can describe the points within a triangular plane using the barycentric technique and three points.

- $a =$ point one
- $b =$ point two
- $c =$ point three

We describe the points along a triangular plane by first calculating two vectors of our triangle $\overrightarrow{\mathrm{BA}} = b - a$ and $\overrightarrow{\mathrm{CA}} = c - a$, then we multiply each of them by an explanatory variable each $0 \leq u \leq 1$ and $0 \leq v \leq 1$, then add them together with our common point a.

$$T(a, b, c) = p + u \cdot \overrightarrow{\mathrm{BA}} + v \cdot \overrightarrow{\mathrm{CA}} \tag{5}$$

2.2.2 Vector operations

The majority of the algorithms used rely on these few vector operations. Understanding these operations eases understanding of the other algorithms used further on.

Dot product Algebraically the dot product of two vectors ($\vec{a}$ and $\vec{b}$) can be defined as the sum of the multiplication between each corresponding dimension:

$$\vec{a} \cdot \vec{b} = \sum_{i=1}^{n} a_i b_i \tag{6}$$

It can also be useful to calculate the dot product geometrically; this method is used to calculate the angle between two vectors. This is done by multiplying the magnitude of both vectors by each other, then multiplying by the cosine of the angle between the vectors.

$$\vec{a} \cdot \vec{b} = |a||b| \cos(\theta) \tag{7}$$

Cross product The cross product of two vectors ($\vec{a}$ and $\vec{b}$) gives us the vector that is perpendicular to both vectors. In three-dimensional space it can be calculated by the following equation:

$$\vec{a} \times \vec{b} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_2b_3 - a_3b_2 \\ a_3b_1 - a_1b_3 \\ a_1b_2 - a_2b_1 \end{pmatrix} \tag{8}$$

Normalisation Normalisation of a vector $\vec{n}$ converts it into a unit vector (a vector of length 1). A unit vector is displayed as $\hat{n}$ and is computed by dividing each dimension of the vector by the vector's total length.

$$\hat{n} = \frac{n}{|n|} = \begin{pmatrix} n_1 \div |n| \\ n_2 \div |n| \\ ... \\ n_i \div |n| \end{pmatrix} \tag{9}$$

2.2.3 Heuristics

Surface area heuristic (SAH) This heuristic is used to decide whether to subdivide a particular node of a tree. We calculate the SAH of the octree with and without the subdivision in question, if the subdivision decreases the octrees SAH score it is kept. As described by MacDonald and Booth[12] in 1990:

$$\text{SAH} = \frac{C_i \cdot \sum_{i=1}^{N_i} \text{SA}(i) + C_l \cdot \sum_{l=1}^{N_l} \text{SA}(l) + C_0 \cdot \sum_{l=1}^{N_l} \text{SA}(l) \cdot N(l)}{\text{SA}(\text{Root})} \tag{10}$$

where

- N_i = number of interior nodes
- N_l = number of leaves
- $N(l)$ = number of objects stored in leaf l
- $\mathrm{SA}(i)$ = surface area of interior node i
- $\mathrm{SA}(l)$ = surface area of leaf node l
- C_i = cost of traversing an interior node
- C_l = cost of traversing a leaf
- C_0 = cost of testing an object for intersection

Heuristic for optimal splitting point in kD-trees This heuristic is used alongside the SAH to choose an optimal splitting point for kD-trees. The heuristic computes the new splitting point by multiplying the surface area of each side of the given splitting plane by the number of objects in that side. Ideally we minimise this function by choosing a splitting plane that finds the maximum number of objects in the smallest surface area:

$$f(b) = LSA(b) \cdot L(b) + RSA(b) \cdot (n - L(b)) \tag{11}$$

where

- b = position of the splitting plane as a percentage of the current node
- n = total number of objects in the current node
- $LSA(b)$ = surface area to the left of the splitting plane at b
- $RSA(b)$ = surface area to the right of the splitting plane at b
- $L(b)$ = number of objects to the left of the splitting plane at b

This function fails to take into account objects that span the splitting plane.

3 Computing optimal centre points

Non-uniform octrees have been attempted once before[16], but we think the optimal midpoint searching method of looping through all the potential splitting points in the search space to evaluate their midpoint score algorithm was slow and unoptimised. Our contribution is how we find this centre point faster, whilst retaining the speed benefits of the non-uniform octree.

To find the optimal midpoint for our octrees we altered the kD-tree splitting method (described in Section 2.2.3) to work in three dimensions. Initially it was important to incorporate all dimensions into one function so that this function could be minimised through derivation, but it additionally helped with incorporating the non-linear optimisation solution.

3.1 Adapting kD-tree splitting methods

We first try to reduce the complexity of the kD-tree splitting function. This is done firstly to ease the use of optimisations later on, and secondly to decrease the time our function takes to evaluate.

This can be done by noting that the left surface area and right surface area (LSA and RSA) are both only proportional to the given b variable which is a percentage of the current node's total width between 0 and 1.

Instead of calculating the surface area as follows (for the x plane):

$$\mathrm{LSA}_x(b) = 2 \cdot (b \cdot xy + yz + b \cdot zx) \tag{12}$$

we can instead just use the input value b, so the surface area Equation 12 becomes (for any plane p)

$$\mathrm{LSA}_p(b) = b \tag{13}$$

Note that all that has changed is the removal of all the constants from Equation 12 (it is later observed that using $b \cdot x$ is more efficient in practice as it has been pre-computed before the function call, but the argument still holds).

RSA is therefore (for any plane p):

$$\mathrm{RSA}_p(b) = (1 - b) \tag{14}$$

The one-dimensional function can now be rewritten:

$$f_p(b) = b \cdot L(b) + (1 - b) \cdot R(b) \tag{15}$$

where p is any plane.

To work with three dimensions we want to combine Equation 15 with itself for all dimensions (x, y and z) to give some value. We can do this via addition or multiplication:

$$f_{\mathrm{a}}(x, y, z) = f_x(x) + f_y(y) + f_z(z) \tag{16}$$

or

$$f_{\mathrm{m}}(x, y, z) = f_x(x) \cdot f_y(y) \cdot f_z(z) \tag{17}$$

These functions will both behave identically in that, given two potential midpoints (X_1, Y_1, Z_1) and (X_2, Y_2, Z_2),

$$f_{\mathrm{a}}(X_1, Y_1, Z_1) < f_{\mathrm{a}}(X_2, Y_2, Z_2) \iff f_{\mathrm{m}}(X_1, Y_1, Z_1) < f_{\mathrm{m}}(X_2, Y_2, Z_2) \tag{18}$$

This also extends to the other comparison operators $>$ and $=$.

Therefore the only difference between the two functions is how the compiler can optimise. At a theoretical level addition should be faster; however, depending on how the CPU can pipeline this set of instructions, there are cases where multiplication can be faster. This is highly dependent on the instruction set of the CPU running the code, and therefore cannot be ideally optimised for all architectures. For the purposes of derivation below, we used the multiplication function f_{m}. Inputting our $f_x(), f_y()$ and $f_z()$ into the multiplication function gives:

$$\begin{aligned} f(x, y, z) =& (x \cdot L_x(x) + (1 - x) \cdot R_x(x)) \\ & \cdot (y \cdot L_y(y) + (1 - y) \cdot R_y(y)) \\ & \cdot (x \cdot L_z(z) + (1 - z) \cdot R_z(z)) \end{aligned} \tag{19}$$

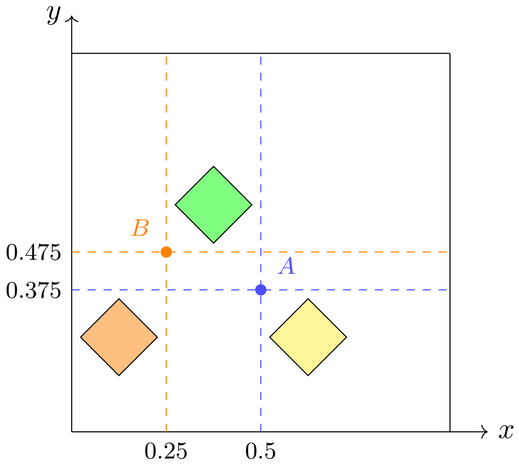

Figure 8: Simple two-dimensional scene, with two potential midpoints

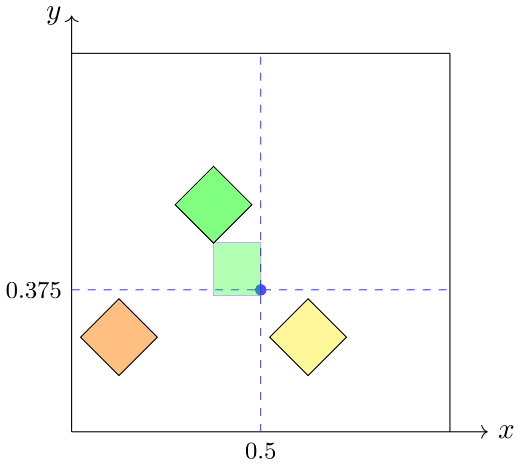

Figure 9: Our simple example with the reduced search space visualised

The validity of this function can be shown through a simple two-dimensional example. We can compute our function $f(x, y, z)$ for both midpoints in Figure 8. Note that both midpoints separate all shapes in the scene into their own sub-voxel – this is the ideal situation for any midpoint (splitting all objects evenly). It can be seen clearly that the A midpoint is a better subdivision point, as it splits more diamonds into smaller voxels than the B midpoint.

Evaluating the A midpoint:

$$f(0.5, 0.375) = (0.5 \times 2 + (1 - 0.5) \times 1) \times (0.375 \times 2 + (1 - 0.375) \times 1) = 2.0625 \qquad (20)$$

Evaluating the B midpoint:

$$f(0.25, 0.475) = (0.25 \times 1 + (1 - 0.25) \times 2) \times (0.475 \times 2 + (1 - 0.475) \times 1) = 2.5812 \qquad (21)$$

In order to find the "best" midpoint according to our function we need to test it with a variety of inputs for each subdivision. If we follow MacDonald and Booth's idea, the most optimal splitting planes lie between the spatial median and the object median points, and within this reduced range, the optimal split occurs at the edge of one of the objects[12]. We can see from Figure 9 that our example optimal A midpoint does indeed lie in the reduced search space.

For each shape within the reduced search space, we must check all eight corners of its bounding box for a potential optimal midpoint. A scene containing 50,000 objects with $\sim 5\%$ of the objects lying within the search space may have to compute this function $\sim 20,000$ times to find the best midpoint for the first split alone. If it splits perfectly evenly then it would need to check a further 2,500 midpoints for each new voxel it creates.

3.2 Finding optimal centre point with non-linear optimisation

We first attempted derivation to avoid evaluating $f(x, y, z)$ for all potential midpoints. Instead solving the derivation for all local minima, which would be considerably quicker. The function $f(x, y, z)$, however, is non-continuous, and therefore cannot be derived. We attempted to alter the function to make it continuous for derivation, but with poor results.

Once derivation failed we moved on to other methods to minimise our function $f(x, y, z)$ while reducing the search space. Non-linear optimisation is the process of optimising an objective function (maximising or minimising), possibly bound by a number of constraints. Many of the choices we've made in our non-linear optimisation solution aim to reduce the time spent generating an optimal midpoint as much as possible, while retaining the accuracy of the midpoint generated.

The problem we have is an *unconstrained bounded minimisation problem.* Our objective function is the function $f(x, y, z)$ and the lower and upper bounds are those of the spatial median and the object median. There are many optimisation algorithms that exist, but all are specific to different types of mathematical problem.

3.2.1 Global vs local optimisation

Global optimisation is the problem of finding a feasible point (x, y, z) that minimises the objective function over the *entire* feasible region. This is very difficult, especially as the number of dimensions of the objective function increase, and one can never be certain that the true global optimum has been found without searching the entire feasible space. In comparison, local optimisation only tries to find a local minimum within the feasible region from some starting feasible point. This is extremely quick and easy to do even for large dimensional problems; however for our particular problem we don't know where to start without using global optimisation first, and additionally our objective function $f(x, y, z)$ has large areas of identical scores intermittent with many local minima.

Ideally, without a time constraint on generating the octree, we would run a global optimisation algorithm and then refine the solution by running a local optimisation algorithm afterwards. However, this is a time-constrained problem. If we wish to achieve speeds similar to regular octree generation, we need to reduce our required standard of accuracy for our generated midpoints. For this reason we chose to rely solely on global optimisation and tweak the parameters of the algorithm to produce the best results.

3.2.2 Gradient-based vs derivative-free algorithms

Gradient-based algorithms improve efficiency of the optimisation process. However, this requires that the objective function be derivable. Since this function is non-derivable, this requires us to use a derivative-free algorithm. Derivative-free algorithms work best for small dimensional problems (where the number of dimensions is less than five hundred), as they must evaluate $f(x, y, z)$ at *least* twice for each dimension.

3.2.3 Choice of algorithm

All algorithms tested performed better than iterating and testing all available points in the reduced search space. This was especially obvious when working with large scene files.

The global optimisation algorithms we considered and tested were:

- ISRES (Improved Stochastic Ranking Evolution Strategy)[18]
- MLSL (Multi-Level Single Linkage)[19]
- Augmented Lagrangian algorithm[20]

Both MLSL and Augmented Lagrangian use an additional local optimisation algorithm to perform the global optimisation, so we further had to choose a complimentary local derivative-free optimisation algorithm. We considered and tested:

- COBYLA (Constrained Optimisation BY Linear Approximation)[21]
- Sbplx[22]

We set an upper-bound time constraint on all algorithms tested, because if any algorithm takes too long to converge, this will affect generation time significantly and kill any potential time gains. Since our problem is highly dependent on the input scene, we used several different small tests for the above algorithms to choose the best algorithm for our general problem.

Augmented Lagrangian algorithm ended up being the poorest performing algorithm. No matter what accompanying local optimisation algorithm we used, it performed $\sim 4\%$ slower than all the other global optimisation algorithms. This is possibly because the Lagrangian algorithm was designed to work specifically on problems with constraints, and without any constraints we are performing extra work computationally, with no gain.

ISRES performed surprisingly well, considering that it is a gradient-based algorithm. Unlike the other global optimisation algorithms, ISRES needs no accompanying local optimisation algorithm, and this could be the reason for its comparative speed. However, it also had the highest inconsistency of all tested algorithms. This stemmed not from the generation time, but from the octree it was creating (and, therefore, occasionally it would generate a slow, non-optimal tree). This is due to ISRES's stochastic nature, and therefore is not suitable for our uses.

MLSL performed excellently with both accompanying algorithms. There was little difference between COBYLA and Sbplx as the timing differences were negligible. Sbplx has more theoretical basis for improvement, as it is based on Subplex, which facilitates optimisation of non-continuous functions (of which $f(x, y, z)$ is one), whereas COBYLA was designed with no such theoretical basis. For this reason we chose MLSL as our global optimisation algorithm using Sbplx as its local optimisation algorithm to generate our optimal midpoints.

4 Ray-tracer design

In this chapter we give an overview of the structure of the ray-tracer, and then go into depth on the specific algorithms, their use and some of the data structure's design in each section of the project.

4.1 Main overview

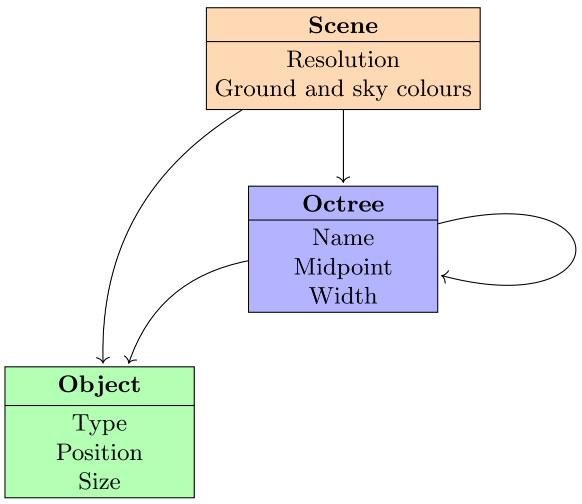

Figure 10: Main class relation structure

The class structure is simple (shown in Figure 10): we generate a scene from a file, which then generates an octree structure containing the shapes of the scene.

When we want to generate the image, we input a pixel value (x, y) to the scene to generate a ray and intersect the ray with the octree, outputting a colour. This colour is then saved in an array. Once a colour is generated for every pixel we save the array to a file. Pseudocode for the main program can be seen below:

```
scene = load(/*filename*/)

// For all the pixels
for i = 1 to scene.xResolution {
    for j = 1 to scene.yResolution {
        // Trace over the octree and save the colour generated
        saver[i][j] = scene.trace(i,j)
    }
}

saver.toFile(<filename>)
```

4.2 Objects

We have several types of object contained within our scene and octree:

- Lights
- Cameras
- Shapes
  - Spheres
  - Triangle meshes

Only two types of shape are included, because that's all we require to properly compare the two types of octree.

4.2.1 Types of light

Following the *.x3d* ISO standard[23], we have three types of light source:

- Directional light: emits light in a direction from infinitely far away
- Point light: emits light in all directions from a specified point for a certain distance
- Spot light: emits light from a point in a certain direction with a maximum angle

Their similarities and differences in properties can be seen in Figure 11. Further information on how these properties are used in calculation of colour can be found in Section 4.3.3.

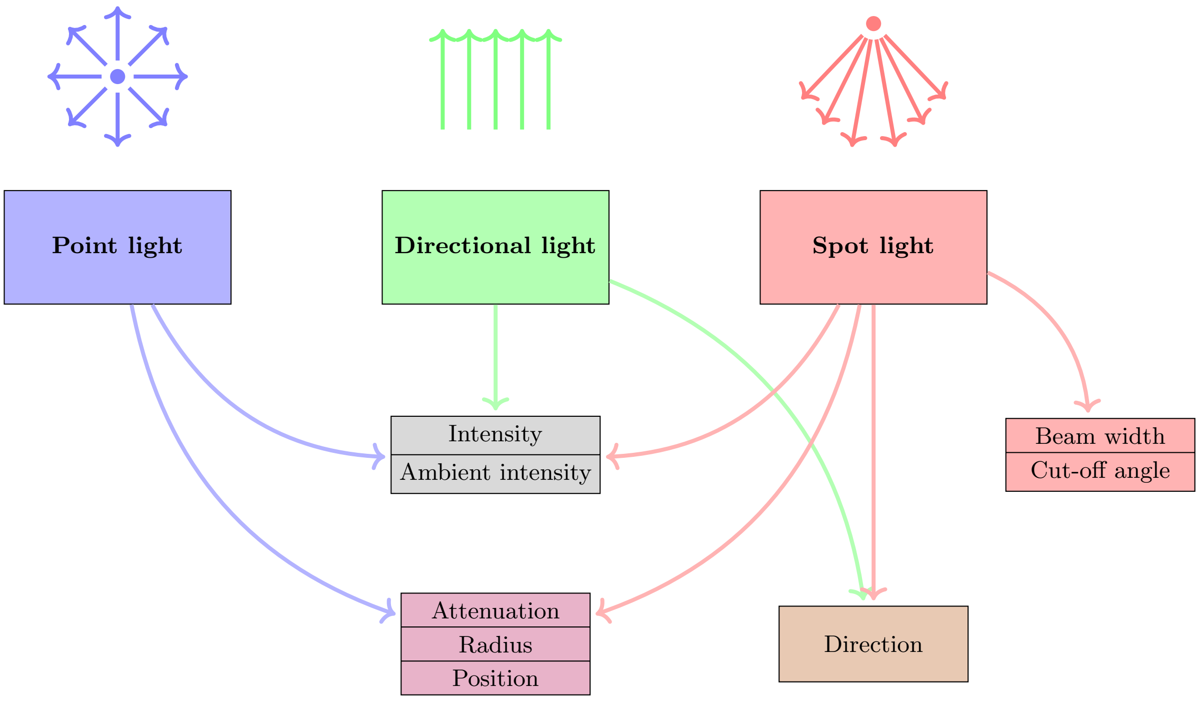

Figure 11: Each light type's values visualised and compared

4.2.2 Camera information

The camera contains several components of key information for generating the primary rays of the image: position, resolution and field of view (note that rotation is not included – this is explained in Section 4.7). How these values are used to calculate the primary rays of the scene is detailed in Section 4.3.1.

4.2.3 Ray–sphere intersection tests

We check whether a ray intersects a sphere (and find its points of intersection) by setting up a quadratic equation and solving using the quadratic formula (see Section 3). We form the quadratic equation by combining the equation for our line (Equation 1) and the equation for our sphere (Equation 2) to form:

$$(O + t\overrightarrow{v} - C)^2 - R^2 = 0 \tag{22}$$

We now expand our brackets to form a quadratic equation where our unknown is t:

$$O^2 + t^2\overrightarrow{v^2} + 2Ot\overrightarrow{v} + C^2 - 2CO - 2Ct\overrightarrow{v} - R^2 = 0 \tag{23}$$

$$\text{rearranging} \implies \overrightarrow{v^2}t^2 + 2\overrightarrow{v}(O - C)t + (O^2 + C^2 - 2CO - R^2) = 0 \tag{24}$$

Now it's clear our quadratic terms are:

$$\begin{aligned} a &= \overrightarrow{v}^2 \\ b &= 2\overrightarrow{v}(O - C) \\ c &= (O^2 + C^2 - 2CO - R^2) \end{aligned}$$

and we plug these values into the quadratic formula to find the solutions of t.

4.2.4 Ray–triangle intersection tests

Our design follows the Möller–Trumbore intersection algorithm [24]. This algorithm has been optimised for C compilation and is the standard for ray–triangle intersection tests.

More recently a faster ray–triangle intersection test has been proposed, which claims to speed up image generation by 1–6% [25]. As this doesn't affect our experiment (because a speed-up of ray–triangle intersection tests would speed up both octree types proportionally), we've chosen to go with the standard Möller–Trumbore algorithm.

4.3 Scene design

The scene holds all objects (described in Section 4.2) and the octree (described in Section 4.4). Its generation comes from a scene descriptor file, and is entirely dependent on the file type.

The scene also contains the tracing algorithm that calculates the return colour for the given (x, y) pixel. The algorithm can be split into two parts: creating and tracing the primary and subsequent shadow rays over the octree to gather information; and calculating the colour based on the retrieved information.

4.3.1 Creating primary rays

The algorithm for generating a scene's primary rays were reproduced from the tutorial website **scratchapixel**[1]. To generate an origin point and a direction vector for any pixel, we must use:

- the given (x, y) pixel
- the camera's resolution
- the camera's position
- the camera's field of view

The ray's origin point is simple as it will be equal to the camera's position. The direction vector, however, is a bit more complex. We can generate the vector by imagining an image plane that is a single unit of distance away from the origin point of the camera along an axis, with each pixel represented on this plane (see Figure 12). The direction vector is then the centre point of the pixel on the two-dimensional plane in three-dimensional space minus the camera's origin.

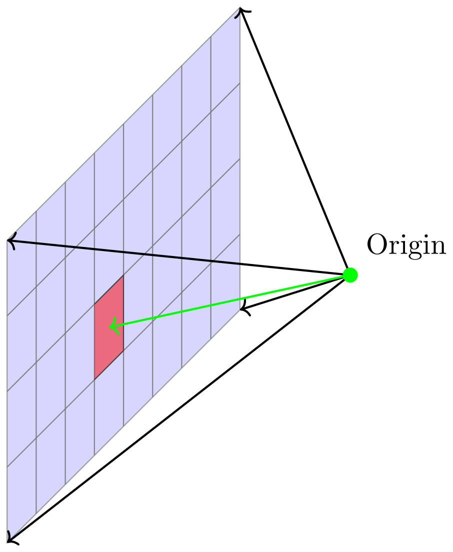

Figure 12: Camera image plane for a resolution of 8 by 4

Since the image plane is a single unit of distance away from the origin, we can set the direction vector's z value to 1.

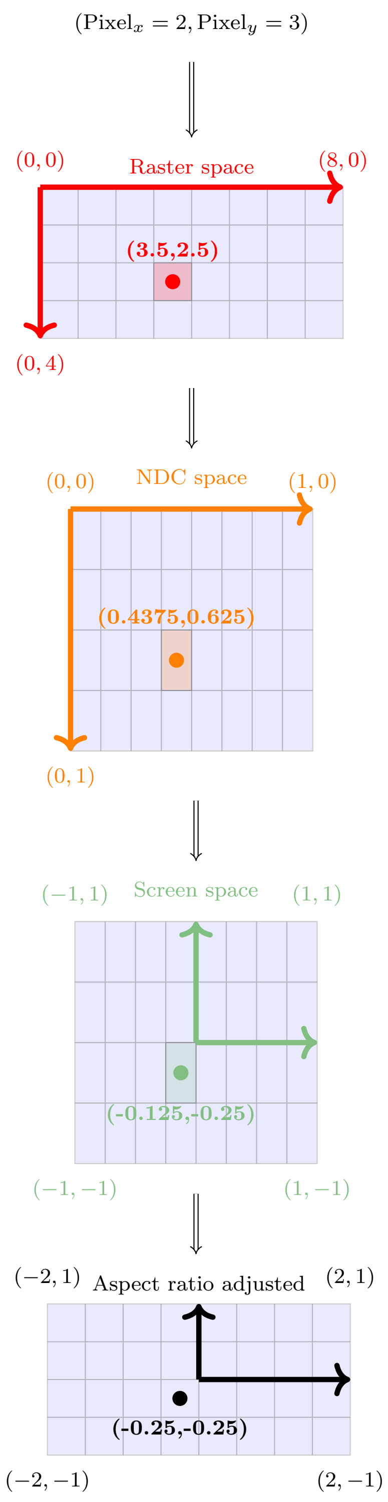

Figure 13: Visualisation of each representation of pixel input $(2,3)$

To find the vector's x and y values, we must take the coordinates given in **raster space**, convert them to **NDC space** (Normalised Device Coordinates), then convert that to our image plane **screen space**. Finally we have to adjust the image plane screen space to take into account the aspect ratio.

Raster space is simply the centre of the pixel specified, in a world where all pixels are 1 unit by 1 unit, starting from $(0,0)$ (see Figure 13). We can calculate raster space simply by adding 0.5 to both the x and y input pixels.

NDC space coordinates are represented in the range from 0 to 1 on both axes. To calculate **NDC space** coordinates, each coordinate is divided by the maximum length of its axis:

$$\text{PixelNDC}_x = \frac{(\text{Pixel}_x + 0.5)}{\text{width}} \tag{25}$$

$$\text{PixelNDC}_y = \frac{(\text{Pixel}_y + 0.5)}{\text{height}} \tag{26}$$

Screen space is similar to raster space, but ranges from -1 to 1 on both axes (exactly as our image plane is located). We can achieve this through the following equations:

$$\text{PixelScreen}_x = 2 \cdot \text{PixelNDC}_x - 1 \tag{27}$$

$$\text{PixelScreen}_y = 1 - 2 \cdot \text{PixelNDC}_y \tag{28}$$

Equation 28 is inverted as the given y pixel starts at the top of the image.

Note in Figure 13 that our conversion to **NDC space** squashes the pixels into rectangles. To achieve correct, square pixels we need to multiply the x pixel by the **aspect ratio** which is the width divided by the height:

$$\text{AR} = \frac{\text{width}}{\text{height}} \tag{29}$$

$$\text{PixelScreen}_x = (2 \cdot \text{PixelNDC}_x - 1) \cdot \text{AR} \tag{30}$$

Finally we need to account for the given field of view θ. For this we need to increase or decrease the the size of every pixel evenly across both the x and y axes. We do this by multiplying the current values for x and y by the length of the vertical distance from the midpoint of the screen space, to the top of the screen space, as given by the θ value. This value is the length of O from Figure 14. Using some simple trigonometry we can formulate the equation for O:

$$\tan\left(\frac{\theta}{2}\right) = \frac{\text{opposite}}{\text{adjacent}} \tag{31}$$

$$\implies \tan\left(\frac{\theta}{2}\right) = \frac{O}{1} \tag{32}$$

$$\implies O = \tan\left(\frac{\theta}{2}\right) \tag{33}$$

Notice that a θ value of 90°gives $O = 1$.

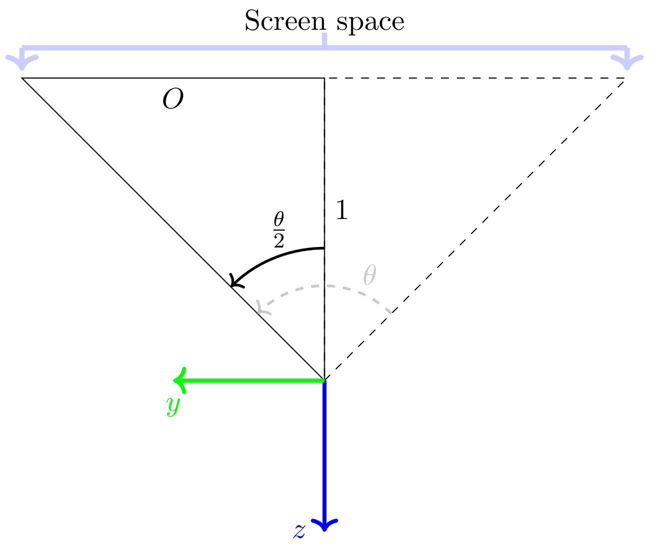

Figure 14: Calculation of multiplying value from given field of view angle visualised

Once we've calculated the value for O we multiply it by both aspect ratio adjusted pixels, which gives the final equations:

$$\text{PixelScreen}_x = \left(\frac{2 \cdot (\text{Pixel}_x + 0.5)}{\text{width}} - 1\right) \cdot \left(\frac{\text{width}}{\text{height}}\right) \cdot \tan\left(\frac{\theta}{2}\right) \tag{34}$$

$$\text{PixelScreen}_y = \left(1 - \frac{2 \cdot (\text{Pixel}_y + 0.5)}{\text{height}}\right) \cdot \tan\left(\frac{\theta}{2}\right) \tag{35}$$

4.3.2 Tracing rays through the scene

Once we have a primary ray we need to check it against our octree. For this we use the octree trace algorithm described in Section 4.4.4. If the trace algorithm returns no collision we return a colour $\text{Col}_{\text{background}}$ based on our own simple algorithm that takes a percentage of the sky and the ground colour based on the y component of the primary ray:

$$\text{Col}_{\text{background}} = \left(\frac{\text{primary}.y + 1}{2}\right) \cdot \text{Col}_{\text{sky}} + \left(\frac{1 - \text{primary}.y}{2}\right) \cdot \text{Col}_{\text{ground}} \tag{36}$$

If we have a collision with the primary ray, before calculating the return colour we must first calculate a shadow ray for each light source in the scene. The shadow ray's origin is the hit point of our object, and the shadow ray's direction vector is the normalised vector between the hit point and the light source.

We then check these shadow rays against our octree using the same octree trace algorithm as the primary ray. For every shadow ray that doesn't intersect an object in the octree (i.e. the light illuminates that hit point), we compute that light source's colour contribution (see Section 4.3.3). All the colour contributions for all light sources are summed and returned as the pixel's colour.

4.3.3 Colour contribution algorithm

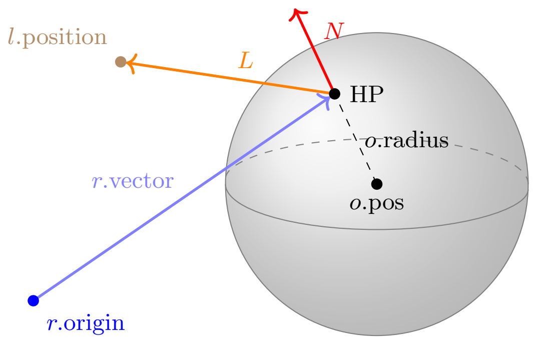

Figure 15: Visualisation of all elements that contribute to the colour algorithm

We follow the *.x3d* specification for colour generation[23]. For light source l that hits object o at the intersection point HP between object o and ray r with hit normal N (see Figure 15), the colour contribution of that light source is specified as follows:

$$\text{Col}_l = (\text{Att} \cdot \text{SLF} \cdot l.\text{colour} \cdot (\text{Col}_{\text{amb}} + \text{Col}_{\text{diff}} + \text{Col}_{\text{spec}})) \tag{37}$$

The Att variable is the amount to reduce the light's intensity as the distance from the light increases; it's calculated from the light l's attenuation variables

- l.ax = base amount of attenuation, normally set to 1
- l.ay = attenuation for distance from ray to object
- l.az = attenuation for distance from light to object

as follows:

$$\text{Att}' = l.\text{ax} + l.\text{ay} \cdot \text{distance}\,(r.\text{origin}, \text{HP}) + l.\text{az} \cdot \text{distance}\,(l.\text{position}, \text{HP})^2 \tag{38}$$

$$\text{Att} = \begin{cases} 1, & \text{if } l.\text{type} = \text{Directional} \\ \frac{1}{\max(1, \text{Att}')} & \text{otherwise} \end{cases} \tag{39}$$

Directional light has no attenuation, as it has no drop-off distance.

The SLF is a Boolean variable only relevant if l is a spotlight and adjusts the light contribution based on how directly the spotlight is facing the object. We first find the angle between the spotlight facing direction $\overrightarrow{D}$ and the vector from the light to the hit point $-\overrightarrow{L}$. This is done as follows:

$$\text{spotAngle} = \arccos\left(\max\left(0, \left(-\overrightarrow{L}\right) \cdot \overrightarrow{D}\right)\right) \tag{40}$$

If the angle is greater than the spotlight's cut-off angle value, then SLF $= 0$ (i.e. the spot light does not illuminate the hit point). If the angle is smaller than the spot light's beam width, then SLF $= 1$ (i.e. the spot light illuminates the hit point completely). Otherwise the angle is between the cut-off angle and the beam width, so the amount the spot light illuminates the hit point is equal to:

$$\text{SLF} = \frac{\text{spotAngle} - \text{cutoff}}{\text{beamWidth} - \text{cutoff}} \tag{41}$$

The last three variables of Equation 37 are the object's colour contribution to the pixel. Col_{amb} is the diffuse colour of the object, multiplied by the ambient intensity of the light source (i.e. the colour that the light gives the object without any relation to the hit normal N):

$$\text{Col}_{\text{amb}} = l.\text{ambient.intensity} \cdot o.\text{diffuse\_col} \tag{42}$$

Col_{diff} is the diffuse colour contributed according to the angle between hit normal N and the vector from the hit point to the light $\overrightarrow{L}$. This can be calculated as follows:

$$\text{Col}_{\text{diff}} = l.\text{intensity} \cdot o.\text{diffuse\_col} \cdot \max\left(0, \overrightarrow{L} \cdot N\right) \tag{43}$$

Col_{spec} is the colour contributed by the specular highlights of the object. This is dependent on the object's shininess value, and is computed as follows:

$$\text{Col}_{\text{spec}} = l.\text{intensity} \cdot o.\text{specular\_col} \cdot \max\left(0, \frac{\overrightarrow{L} - r.\text{vector}}{\left\|\overrightarrow{L} - r.\text{vector}\right\|}\right)^{128 * o.\text{shininess}} \tag{44}$$

4.4 Octree design

Each uniform octree needs to have its own: name, midpoint, **width in** x, y **and** z, surface area (for SAH computation) and either a list of contained shapes or a list of leaf nodes. All uniform octrees in the structure need access to the same values:

- for computing the SAH:
  - the root octree
  - the depth of the octree
  - the number of voxels in the octree
- for finding the next octree during ray-tracing:
  - the list of all octrees
  - the smallest width among all voxels

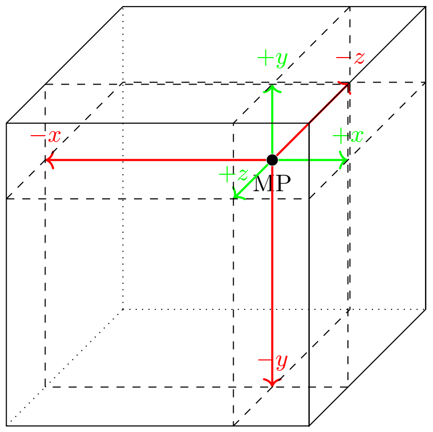

Figure 16: Non-uniform octree voxel described by a midpoint and six widths

The non-uniform octrees have all of the above values, except that, because they can have their midpoint in the non-spatial centre, they must have six width values, three positive $(+x, +y, +z)$ and three negative $(-x, -y, -z)$. This is shown in Figure 16.

Because of the almost identical properties of uniform and non-uniform octrees we thought it easier to combine our uniform and non-uniform octree classes. The only difference being the way the octrees are subdivided (in this way all the uniform octrees are just non-uniform octrees that will only ever subdivide in the spatial centre). This will induce some additional memory complexity to the uniform octree structures, but the amount induced will be minimal (three float values per voxel) and shouldn't affect temporal performance. The benefits of designing the octrees in this way allow us to re-use the traversal and ray-trace functions for both octree types, which is extremely helpful since these are complex functions with potential for many errors and bugs.

4.4.1 Octree labelling

It's important to label our octrees so that they can each be easily identified. For this we follow Glassner's proposed naming scheme[6], where each octree has an integer name from 1 to 8, where each integer corresponds to its relation to its parent's midpoint – this is illustrated in Figure 17. The only exception to this is the root octree, which is named 0.

Each parent then passes its own name onto its children as a prefix. For example, splitting a node named 43 would generate eight octree children, named 431 to 438.

This naming scheme guarantees that each node has a unique name that accurately describes its position in the octree structure, which we can make use of during ray-tracing (more detail in Section 4.4.4). Non-uniform octrees are named in an identical fashion.

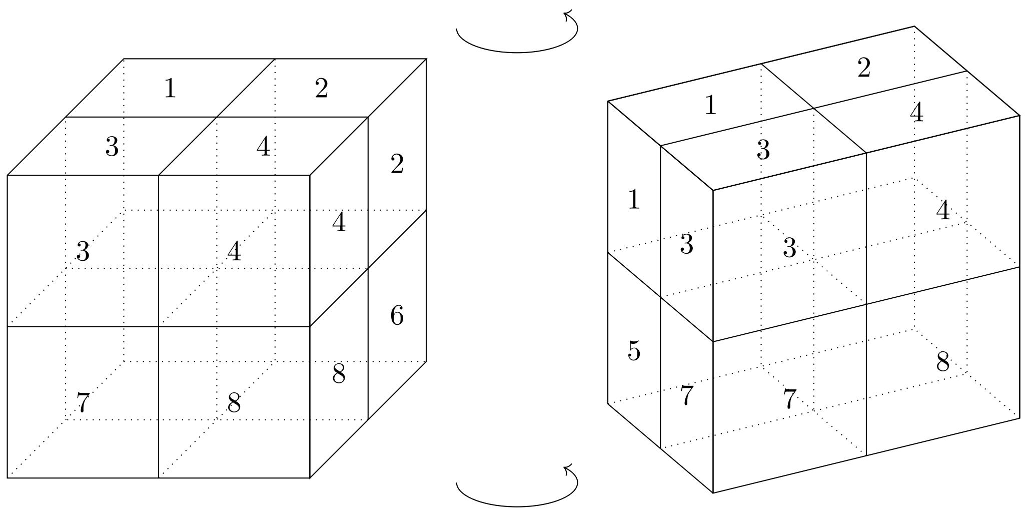

Figure 17: A subdivided octree with labelled children nodes

4.4.2 Octree generation

Creating either uniform or non-uniform octree structures is guided by the SAH in a theoretically simple algorithm. To subdivide any octree node we use the following algorithm:

```
subdivide() {
    // Calculate the SAH before subdivision
    sah_before = root.SAH()
    // Create the leaves and partition the contained objects into them
    this.split()
    // Calculate the SAH after subdivision
    sah_after = root.SAH()
    if (sah_after < sah_before) {
        // If the subdivision is good, continue subdividing on all leaves
        // Additionally we must update our global value of the smallest width
             in our octree
        for int i = 1 to 8 {
            this.leaves[i].update_smallest_voxel_width()
            this.leaves[i].subdivide()
        }
    } else
        // If the subdivision is bad, delete all leaves
        this.unsubdivide()
}
```

The only difference between the uniform and non-uniform octree structures is how the split() function works.

4.4.3 Splitting a leaf node

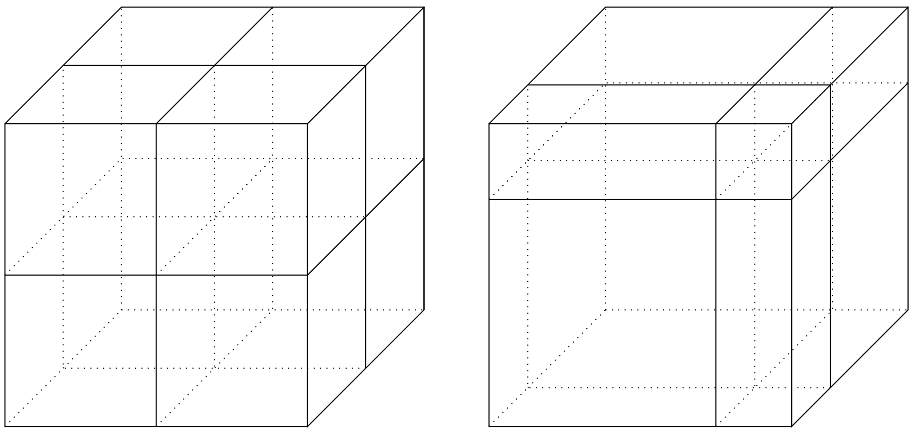
Figure 18: A uniformly subdivided voxel and a non-uniformly subdivided voxel

To split a leaf node we need to understand what each leaf node will look like in relation to our current node. No matter the type of voxel, the eight subnodes generated by our split will have spatially centred midpoints (and therefore identical positive and negative widths).

In the case of the non-uniform octree, only once we start to split the node do we choose an optimal midpoint and alter the positive and negative widths to keep the voxel bounds identical. Once we've done this we can continue with the split as identically to the uniform octree. The split happens as follows:

1. Create all subvoxels:
   (a) Using the name of the subvoxel to guide, compute each subvoxel's centre point, maximum point and minimum point.
   (b) Work out new voxel widths from centre point, maximum point and minimum point.
   (c) Create the subvoxel and add as a child to the current voxel.
2. Place the objects of the current voxel into the correct subvoxels based on their position to the current voxel's midpoint.

The creation of the subvoxels is not simple – the pseudocode for the beginning of the split algorithm can be seen in Appendix D to aid its understanding. The pseudocode omits the placing of the objects into the subvoxels as that part of the algorithm is relatively simple. The algorithm follows the naming scheme described in Section 4.4.1.

4.4.4 Ray-tracing through the octree

The octree trace algorithm is given a ray to compare to, and must return a Boolean value of whether the ray intersects with any shapes inside the octree. If the function returns true, it must also return:

- the shape the ray intersects, to get further shape details

- the distance from the ray to the shape intersection, to generate the shadow ray

A given ray may start within the octree already. For this reason we must only check for intersections from the origin of the ray onwards, and ignore any intersections behind the ray's origin position.

This algorithm can be boiled down into two separate parts: checking the ray's intersection with the shapes inside the current voxel and finding the next voxel to check.

Checking the ray's intersection with voxel objects can be done by simply looping through all the shapes in the leaf voxel and using each shape's intersection test (described in Section 4.2.3). If we hit no shape, we move onto the next voxel; if we hit a shape, we want to save the distance we hit the shape at and continue through the shape list, only updating the hit shape if the distance is closer to our ray origin.

Finding the next voxel we follow the method presented by Glassner[6], which requires us to find a point within the next voxel and then search our octree for the leaf voxel containing that point.

Finding a point within the next voxel is relatively easy: we first find the furthest point from our ray where the ray intersects the bounding planes of the current voxel, then we add the ray direction multiplied by a small number (i.e. we want to move only *slightly* inside the next voxel). This small number is proportional to the smallest voxel width in our octree.

Once we've found the point we search for it in the octree with a simple algorithm:

1. Set the **currentVoxel** to the root node.
2. Check the point's location relative to the **currentVoxel**'s midpoint.
3. Set the **currentVoxel** to the corresponding leaf found in 2.
4. If our **currentVoxel** is not a leaf node, return to 2.
5. Return the **currentVoxel**.

4.5 File type choices

Scene file type: *.x3d* There are many open source three-dimensional scene file types, but since this is a rather unimportant part of the project we chose a file type that is:

- easy to read in
- quick to create test files
- easy to automate creation of files

.x3d was chosen because of its fulfilment of the above, and additionally:

- there exists a detailed specification of *.x3d* available online[23]
- the free modelling software **blender** supports exporting to *.x3d*

Output image type: *.bmp* *.bmp* was chosen because at a base level it contains no compression, so we can easily export our two-dimensional array of colours to a file.

4.6 Serial vs parallel

It's clear that a graphics card, or graphics processing unit (GPU), is optimal for image synthesis, as the problem is relatively easily parallelised and can make excellent use of many cores. However, all speed-up gained by parallelising our ray-tracer would speed up generation and ray-tracing equally for both uniform and non-uniform octree structures, and therefore directly help us compare the effectiveness of our octree structure.

Ideally, given more time for the project, we would next implement GPU parallelisation, because then we could directly compare ray-trace times with other state-of-the-art ray-tracing techniques, but since it's not directly needed, it is omitted.

4.7 Rotation

For this project we planned no rotation for any elements (cameras and shapes). While including rotation helps to flesh out the ray-tracer and is relatively easy to implement, it doesn't aid us in testing our octree comparison, and was not a priority to be included. Here we describe its potential design using Rodrigues' rotation formula[26].

To rotate any vector or point in three dimensions around an axis, we have to generate a rotation matrix and multiply it by our vector or point. To rotate vector or point $\vec{v}$ around normal vector axis $\vec{k}$ by θ°we follow the equation:

$$V_{\text{rot}} = \vec{v} \cdot \theta + \left(\vec{k} \times \vec{v}\right) \cdot \sin(\theta) + \vec{k} \cdot \left(\vec{k} \cdot \vec{v}\right) \cdot (1 - \cos(\theta)) \tag{45}$$

The direction of rotation follows the right-hand rule of the $\vec{k}$ vector[27].

4.8 Experimental design

In several parts of the design it was important to try various values or expressions to achieve the best result. This is most obvious in the cost calculations for the SAH: we wrote a small Python script that used the library `skopt` to minimise our program's time complexity by altering the functions used to generate the costs[28] (this required a large rewrite of the program to accept command line inputs that would change how these costs were calculated). While this version of the program can no longer be found in the code repository without trawling through the commit history, doing these experiments clearly showed us that certain equations achieved much better results than others (the equations picked can be seen in Section 5.4.1).

We also used a kind of experimental design while choosing the parameters for our non-linear optimisation algorithms. While we didn't use the Python minimisation scripts, we did perform many hand-designed experiments tweaking the `maxeval` and `maxtime` parameters, observing the change in timing. These experiments gave us a large range of “good” values for the parameters, but no specific changes in the parameters were found to improve the generated octree structures. This is a potential vector for further study to improve the non-uniform structure.

During testing we found that the non-uniform structure was still creating incorrect images, because it was trying to create too small voxels (problem detailed in Section 5.4.2). When trying to alleviate this problem, we altered the way in which the point inside the next voxel was being found; however, this caused the octree to miss out leaf voxels, missing some parts of objects entirely. The fix found for this was through experimentally changing the modifier used on the smallest voxel width during next voxel calculations; we additionally experimented with changing the smallest size of voxel allowed to be created within the structure.

5 Implementation and experiments

The project was written in `C++` and split into five main classes: `Main`, `Scene`, `Octree`, `Shape` and `BitmapSave`. The project makes heavy use of `C++11`’s `shared_ptr` type, which is a regular `C++` pointer with a counter of the number of references to it. Once the counter reaches zero, it calls the object’s destructor (it may be likened to Java’s garbage collection). `shared_ptr`s are not completely memory safe, however, as cyclical references can cause memory leaks (for this reason, they’re not used in the octree structure). Our use of `shared_ptr` is omitted from the following and replaced with regular pointer syntax (`*`) as they are functionally identical.

The project’s non-octree-related ray-tracing code implementation was aided by the online resource scratchapixel[1], which is a tutorial site dedicated to explaining the fundamentals of ray-tracing. The project was written to be easily extensible – to this end we’ve made use of `C++`’s templating to make as many functions and classes generic as possible.

5.1 Working in three dimensions

A vector/point type is needed for working heavily in three dimensions, so we created two generic classes: `Vec3<T>` and `Vec2<T>`. Both are tuple types containing three and two values respectively of type `T`.

More importantly they also have all vector manipulation functions (from Section 2.2.2) and several additional functions that we found useful or eased readability as member functions. We also overloaded most basic operators to ease their use and readability in code.

The `Vec3<T>` type is extremely versatile and is used heavily throughout the code. Its main use is that of a point or vector in three-dimensional space; however, it has additionally been used as a colour variable in the BitmapSaver and Scene classes. We also created a ray structure to pass between the scene and the octree more easily, this ray structure is just the combination of two `Vec3<T>`s, representing the origin point and vector of the ray.

5.2 Shape class

All scene objects (lights, cameras, shapes) are represented with this one class – inside the class is a type enumeration that identifies the current type of the object. The enumeration acts a tag on a union of specific shape data. The enumeration tag and union are as follows:

```
enum class Shapes {
    UNIDENTFIED,
    SPHERE,
    CAMERA,
    TRIANGLEMESH,
    LIGHT
};

union Shapesdata {
    SPHERE_STRUCT *sphere;
    TRIANGLEMESH_STRUCT *triangles;
    CAMERA_STRUCT *camera;
    Light *light;
};
```

We use a union here to reduce memory consumption because a single scene can contain a huge number of shape instances. This also eases extensibility, as adding a new type of object only involves:

- adding the tag to the enumeration
- adding the type `struct` to the union
- adding a constructor method
- (if the type is a shape to be intersected) adding an intersection method

Because almost all of the entities represented by the shape class have some common values, these are included in the shape class itself, instead of having to be re-included by all the different shape structures. These common values are:

- `float`: shininess
- `Vec3<float>`: position
- `Vec3<float>`: diffuseColour
- `Vec3<float>`: specularColour

Because of the way the *.x3d* file is parsed, we need to allow for the building up of a shape as the parser reaches the relevant line. This means we cannot just have a single constructor that populates all fields in the shape class: instead we use additional setters and constructors for each type of shape's data.

All shapes that can be ray-traced over contain a function that returns the surface data at any particular point on the object's surface. This is used only to retrieve the hit object's normal at the hit point location, but can be modified to also return additional information relating to the shape's colour (e.g. retrieving texture colour at the hit point).

5.2.1 Sphere structure

The sphere is the simplest of all the structures. Because spheres can be represented so simply, the sphere structure only needs to contain its radius. Whenever we compute the intersection test for a sphere (Section 4.2.3) we use its radius squared. Since this needs to be computed so often (for every ray–sphere intersection), it makes sense to additionally store the radius squared in this structure as well.

5.2.2 Triangle mesh structure

A triangle mesh structure is a container for a group of triangles and their normals. The data structure is designed in a way to reduce the memory footprint of the triangle mesh. To describe all triangles, we store two lists:

- `std::vector<Vec3<float>> position`: a list of all three-dimensional coordinates within the triangle mesh
- `std::vector<int> vertexIndexArray`: a list of integers that index the `position` list

For every triangle there are three consecutive entries in the `vertexIndexArray`. This allows our triangles to share the coordinates of their neighbouring triangles. This storage of triangle meshes is the industry standard in image synthesis.

With triangle meshes we can either return face normals (sometimes called surface normals) or face weighted vertex normals to the colour contribution algorithm (described in Section 4.3.3). Returning the face normal of whichever triangle face was hit causes our faces to be flatly shaded (see Figure 19) whereas returning face weighted vertex normals produces a smoother gradient of lighting (see Figure 20). The face weighted vertex normals $\overrightarrow{N}$ are calculated from:

- the vertex normals $\overrightarrow{A}, \overrightarrow{B}$ and $\overrightarrow{C}$ of the hit triangle with points A, B and C
- the hit position on the hit triangle u, v (this hit position is returned by the Möller–Trumbore algorithm, see Sections 2.2.1 and 4.2.4)

as follows:

$$\overrightarrow{N} = (1 - u - v) \cdot \overrightarrow{A} + u \cdot \overrightarrow{B} + v \cdot \overrightarrow{C} \tag{46}$$

Because the choice of normal to return only creates a stylistic difference in images, we allow the file to dictate which kind of normal to use for shadow calculations per object.

When the normals of each vertex are given in the file, the program will load them in and will use face weighted vertex normals for ray-tracing. However, when neither normal type is provided in the input file, we must generate them ourselves.

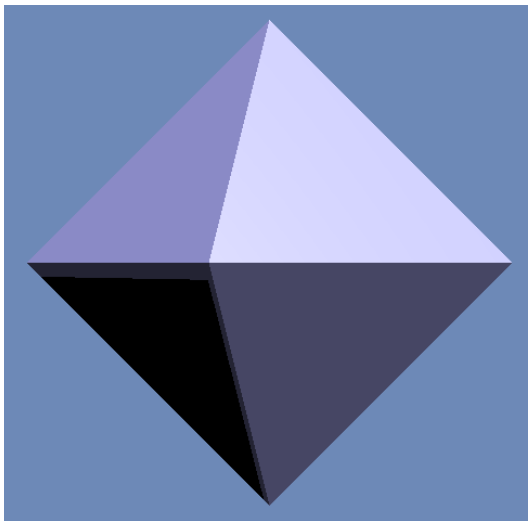
Figure 19: Traced using face normals

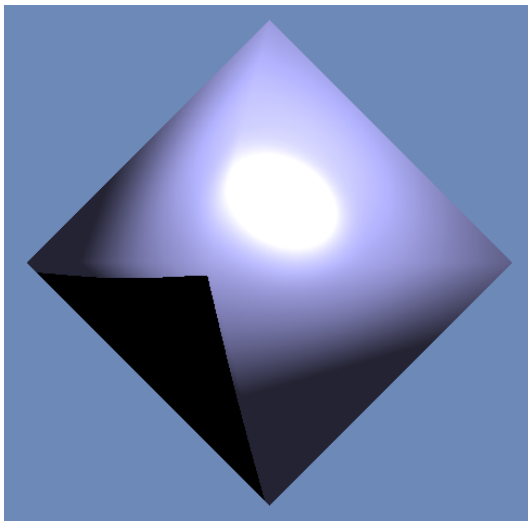
Figure 20: Traced using weighted vertex normals

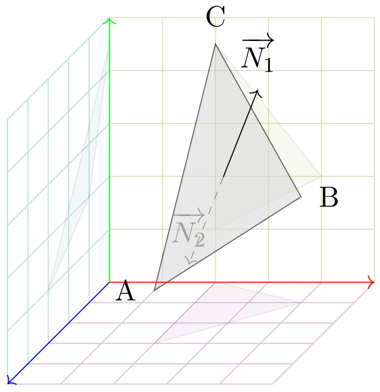

Figure 21: The two face normals we can generate from points A, B and C

Generating triangle face normals from a triangle's points A, B, C involves taking the cross-product between two of the vectors of the triangle's edges. The order of the cross-product taken changes the result (see Figure 21) and is reliant on the order the triangle's coordinates are specified by the input file (clockwise or counter-clockwise). Industry has no particular standard on the order of triangle coordinate specification, but we follow the *.x3d* standard of counter-clockwise:

$$\overrightarrow{N_1} = (B - A) \times (C - A) \tag{47}$$

To generate a vertex normal we must take the average of all normals of faces adjacent to the vertex (i.e. normals of all faces that contain our vertex as a point in their triangle). This means we still must generate the face normals as above first.

During octree generation we use a shape's width to place it inside the leaf nodes and guide our non-uniform midpoint heuristic. In order to save computational time during this process, we additionally save the width of each triangle mesh instead of recomputing every time it's needed. This is important for triangle meshes containing many triangles, because to calculate the width of a triangle mesh we check every vertex position in the triangle mesh structure.

5.2.3 Light structure

As there are three types of light, we need to be able to distinguish between them. This is achieved by giving our `Light` structure its own enumeration tag:

```
enum LightTypes {
    SPOTLIGHT,
    POINTLIGHT,
    DIRECTIONALLIGHT
}
```

Because the differences between the light types are minimal (a couple of float values) we don't use a union structure for each light type's individual values as the memory overhead is so low (especially since the number of light objects in a scene is vastly lower than the number of shapes).

The `light` class is implemented just as designed in Section 4.2.1, with the only addition being that of default values for undefined parameters which we've taken from the *.x3d* documentation.

5.3 Scene class

Our `scene` class variables are merely descriptors for the scene in the given *.x3d* file. The `scene` class contains member functions that parse the specified *.x3d* file into its variables and a single function for generating a pixel's colour from the camera's perspective.

We use the following variables to describe the scene:

- `Vec3<float>: skycolour`
- `Vec3<float>: groundcolour`
- `Shape: sceneCamera`
- `Octree: rootNode`
- `std::vector<Light*>: lights`

The `scene` class only supports a single camera currently because multiple camera support wasn't a priority for the project. It would be a relatively easy change to implement, however, since we would replace our single camera member object with a list of camera objects.

5.3.1 Parsing *.x3d*

Because *.x3d* is a type of *XML* file, we are using the **boost** library's `property_tree` type to read the file in as *XML*. This property tree structure is then traversed by a function named `parseX3D` to create and fill the `scene`'s variables.

The function `parseX3D(type optimiser)` performs a number of steps:

1. Parses the global information (camera, ground colour, sky colour).
2. Calls a subsidiary function `parseNode()` on the remaining nodes to:
    - fill the octree with shapes

    - populate the list of lights
3. Resizes the root octree node to accommodate newly added shapes.
4. Subdivides the octree according to given type (uniform or non-uniform).

5.3.2 Tracing

The trace algorithm of `scene` creates the rays that are passed to the octree structure (which then performs the ray-tracing). If there is no collision, we use the algorithm described in Section 4.3.2 to generate a background colour.

If there's a primary ray collision, the octree structure populates a subsidiary structure `IsectInfo` that contains:

- `Shape* hitObject`: a pointer to the shape hit
- `float tNear`: the distance from ray origin to hit point
- if the hit object was a triangle mesh:
    - `int index`: index of the triangle hit
    - `Vec2<float> uv`: parametric point of intersection on the triangle

The trace algorithm then loops through the list of lights and creates a shadow vector for each one. This shadow vector's origin is **not** the hit point on the shape as described in Section 4.3.2, but actually the hit point plus a portion of the hit normal at that point. This is due to floating point arithmetic errors. If we use the hit point as is, the shadow ray ends up intersecting with the shape it's reflecting off, creating speckled shading (see Figure 22), so we must first move the origin slightly away from the shape first before ray-tracing again.

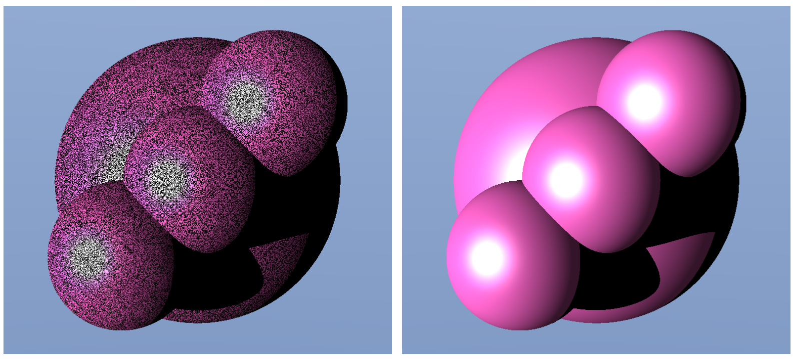

Figure 22: Ray-tracing a simple scene with four spheres: left, without a shadow ray bias; right, with a shadow ray bias

To obtain a portion of the hit normal to add to our hit point we multiply our hit normal by a bias: `const float BIAS = 0.04`. The number value for the bias was chosen through experimentation, and there is no widely accepted default.

In order to save time we start our shadow ray's trace using the octree leaf node that contains the start position of our shadow ray (this octree node would be found by the root octree eventually, but would perform more auxiliary functions and checks first that are unneeded).

Once the trace algorithm has computed a light's shadow ray, it checks its intersection with the octree: if the ray doesn't hit a shape within the scene, then the scene uses the colour contribution algorithm described in Section 4.3.3 to generate a colour. All colours generated this way are then summed (their red values are added to create the new red value; ditto with blue and green). To prevent colour value overflow we cap the colour at its maximum and minimum values (this stops a colour from increasing past white).

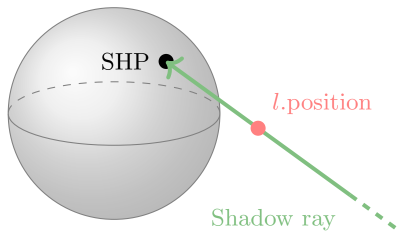

Figure 23: A sphere that intersects with the shadow ray, but doesn't sit between the light source and the hit point, therefore shouldn't prevent the light from illuminating the shadow ray's origin point

If the ray does hit a shape, we must check that the shape hit is actually in between the light source's position and the shadow ray origin before disregarding this light source's contribution.

To do this we find the intersection point of the shadow ray with the shape SHP and check if its x, y and z values lie between the x, y and z values of the hit point and light source origin (see Figure 23). If SHP does not lie within this range then we continue with the colour contribution algorithm. This applies only to light sources that can be placed within the shapes of the scene (point lights and spot lights).

5.4 Octree class

The `octree` type is a recursive data structure: interior nodes contain a list of eight pointers to `octree` children and leaf nodes contain a list of pointers to contained shapes. Pointers are used here to decrease memory usage and time complexity, as passing pointers around is much faster than entire shape or octree structures. A `boolean` member variable identifies whether a node is subdivided (an interior node) or not. During the optimisation process we must "un-subdivide" interior nodes; because of this, interior nodes continue to hold the pointers to their contained shapes, so they can be easily reverted back to being leaf nodes.

After the root node of the octree is instantiated it is populated with shapes by the `scene` class; we must then resize the root octree node to encapsulate all contained shapes. This is done with a member function named `resize`, which simply loops through all the objects saving the minimum and maximum point the octree needs to contain, then changing the midpoint and widths of the octree to satisfy these limits. Once the root node is populated and correctly sized we can begin subdividing (as described in Section 4.4.2).

During ray-tracing the octree uses a global list of pointers to all octrees to find the leaf node voxel. Instead of using a standard `std::vector`, we instead use a hash map (`unordered_map<unsigned long,Octree*>`) which maps the integer names of the octrees to a pointer to that octree. When we want to find the octree we find the hash of its name and retrieve its pointer (almost) instantly. Collisions in a hash map are unavoidable, and as the map grows in size, so do the number of collisions. With a growing number of collisions, the overhead of the hash map (hashing) compared to a list can become significant. To reduce these overheads the hash maps parameters (load factor) is given a maximum value, in theory limiting the number of collisions that the hash map will encounter.

5.4.1 Computing the SAH

The octree subdivisions for uniform and non-uniform are guided by the SAH. In order to compute the SAH for the entire octree we require some information about the octree in its current state. First we need to compute the cost variables. The SAH calls for:

- `double traverseICost`: cost of traversing an interior node
- `double traverseLCost`: cost of traversing a leaf
- `double intersectCost`: cost of testing an object for intersection

These values are computed by a member function named `calculateCosts`. This function went through quite a few iterations and testing (see Section 4.8) before we found the calculations that produced the most optimal octrees. The calculations for these costs are as follows:

`traverseICost = numVoxels * numVoxels`: this interior traverse cost was chosen because the time cost of traversing any interior node is proportional to the number of voxels we've created so far (as we must loop through the vector holding all the voxels). This value is squared because otherwise the octrees were not punished enough for creating more and more voxels, and would create many pointless (and slow) subdivisions because of it.

`traverseLCost = numShapes / numVoxels`: the cost of traversing a leaf node is proportional to the average number of shapes in each voxel, because when we traverse a leaf node we must first check all its intersections before moving on to the next voxel.

`intersectCost = numLights * numShapes * numShapes`: the cost of checking any intersection is proportional to the number of shapes and the number of lights in the scene (as we must calculate a new shadow ray per light source). We found that squaring the number of shapes produced smaller octrees and was beneficial to overall ray-trace times.

We then need to calculate the summations in the SAH algorithm. This is done using a `struct` whose reference is passed around to every `octree` instance which then adds its own values to the running total. This `struct` is as follows:

```
struct SAHInfo {
    \\ Summation of all interior nodes surface areas
    double SAInterior = 0;

    \\ Summation of all leaf nodes surface areas
    double SALeaf = 0;

    \\ Summation of all leaf nodes surface areas multiplied by the number of
        objects contained in the leaf node
    double SALeafObj = 0;
};
```

We then use the gathered information to calculate the SAH for the current octree structure.

5.4.2 Non-uniform octrees: computing the optimal midpoint

To perform our non-linear optimisation we are using the library **NLopt**[29]. **NLopt** provides access to many different non-linear optimisation algorithms and allowed us to test, swap out and change the parameters of different non-linear optimisation algorithms easily and quickly.

Before we use **NLopt** we must first calculate the spatial average location and object average location: the first is just the midpoint of the current voxel (because when voxels are created they are created with spatially centred midpoints) and the second is computed simply by taking the median.

Because of the nature of non-uniform octrees, they can sometimes find an optimal midpoint that creates voxels that are so small that they cause infinite loops when ray-tracing occurs. The infinite loop is present because the smallest voxel width is used to find the next voxel in the octree. If when the smallest voxel width multiplied by any normalised vector gives the null vector $(0, 0, 0)$, then the octree will try to find the next voxel, but only find itself. This is obviously a floating point arithmetic issue.

We alleviate this problem by limiting how close these averages are to the edge of the voxel. If either average is within some small distance of the edge of the voxel we're subdividing, then we return a null pointer.

Once the reduced search space has been found, we then use **NLopt** to find the optimal midpoint. To use **NLopt** the objective function that we pass to the library must be statically defined; however, our function relies on current octree data (object's positions) which cannot be accessed by any static member function directly. **NLopt** provides the ability to pass relevant data through to the objective function as a void pointer. We then use `std::reinterpret_cast<T>()` to access the data inside the objective function.

NLopt allows us to modify parameters of the optimisation. These are mainly terminating parameters, that is, parameters that decide when the optimisation is finished. Because this process is highly time-based, the two parameters we found most useful were:

- `maxeval`: the maximum number of evaluations the optimiser is allowed to make of the objective function
- `maxtime`: the maximum time the optimiser is allowed to take

Even if the optimiser is halted from these parameters it will still return the most optimal inputs it has found for the objective function at termination. Since the midpoint is then checked by the SAH we can be sure the octree will never be subdivided non-optimally no matter the midpoint the optimiser returns (because if the midpoint doesn't improve the SAH, the octree will remove the subdivision). For this reason we don't mind constricting the effectiveness of the optimiser for the sake of octree generation speed-up.

The number of evaluations we allow the optimiser is proportional to the number of objects in the current node (because if there are fewer objects, finding the optimal midpoint is less important than just executing the midpoint search quickly). This is then capped at one hundred evaluations, which is a contingent value we found, as the best value was extremely dependent on scene input: larger scenes wanted a higher evaluation cap and smaller scenes wanted a lower evaluation cap.

We found during testing that even setting a maximum time for the evaluations increased the time the octree generation took. This is not because the optimiser was taking the full maximum time allowed, but because simply the act of the optimiser timing itself was increasing the overall time taken. We suggest the adjustment and optimisation of these and the other available parameters are avenues for further research.

5.4.3 Tracing over the octree

Our octree contains the member function `trace` which works as described in Section 4.4.4. If the node calling the function is subdivided, then we use the function `next` to find the leaf node that corresponds with the closest node to the ray's origin with which the ray intersects.

If the node calling the function is not subdivided (a leaf node), then we loop through its contained shapes and run its intersection test with the ray, returning the closest intersection. If there's no intersection, the function `next` is used to find the next leaf node that corresponds with the *next* leaf node (note that this is not the closest) with which our ray intersects.

The `next` function uses a subsidiary function named `intersectsWith` to return the distance from the ray origin to the current voxel's bounds. Because the ray will always intersect the bounds twice (once when entering, once when leaving), there are two possible values this function can return. If the calling voxel is subdivided, then we need the distance to the *closest* intersection, because we're presuming the ray is coming from outside the octree and needs the initial leaf node it reaches. However, if the calling voxel is a leaf node, we can presume the ray is already tracing through the octree and we need the *furthest* intersection (i.e. we need the coordinates where the ray intersects the splitting plane between the current voxel and next voxel).

To prevent the infinite loop problem outlined in Section 5.4.2 from crashing the program, we check the the `findleaf` function's returned node is not the same as the node calling the `findleaf` function. If these are identical we simply return a null pointer (i.e. there were no intersections). This fix is never needed as the program currently stands (as we limit how small the octrees can subdivide to), but if the program were to be altered, a crashing program is non-ideal (and can be difficult to debug).

The `next` function then uses this value to find a point within the voxel it wants to return (by adding the small amount), then uses a second subsidiary function `findleaf` to return this voxel. Just looping through all the shapes and returning the closest causes incorrect behaviour in a specific scenario: when calculating the trace of ray r, shape s_1 spans a splitting plane of leaf node A and B, with a smaller object s_2 partially in front of it not spanning the splitting plane between A and B, but instead residing only in node B. When our intersection ray should intersect with both objects s_1 and s_2 but passes through A before B, the intersection between the ray and s_2 will never be checked and our function will return the collision information for s_1, which is incorrect.

To prevent this we must check that every intersection we find inside the trace algorithm is inside the bounds of our current octree node. If not, we ignore the intersection.

5.5 Floating point comparison

Throughout the project we are comparing floating point or double precision variables to floating point or double precision constants. Due to the nature of these types, previous arithmetic performed on these variables can cause them to become off the true value by a small amount. If this happens when the true value is extremely close to the constant, then a comparison that should be true can return false. This can occur in ray-tracing in many different parts of the implementation, but primarily we found it affected the ray–voxel bounds' checking function `intersectsWith`. When a ray passed into a voxel for a very small distance, the function would return that it did not intersect the voxel (when in fact it does). Then an incorrect colour would be returned.

When working with very exact mathematical formulae we must account for the proportion these floating point variables may be off when performing comparisons. This is commonly done internally in three-dimensional software packages by using an "epsilon" value: this is the amount by which the floating point is allowed to be "off" in a comparison operation. For instance, the comparison:

```
float var;
.. \\ operations with var
if (var < 4.2573) {
    ... \\ more code
}
```

becomes:

```
float var;
const float epsilon = 0.000001;
.. \\ operations with var
if (var - 4.2573 < epsilon) {
    ... \\ more code
}
```

The choice of epsilon is non-exact: it should be small enough not to impact calculations (by allowing incorrect calculations into the controlled section), but large enough to catch floating point errors. Because of this the value of epsilon is largely dependent on what calculations precede it, and is discovered through experimentation (although starting with a value around 0.0000001 is the standard).

5.6 Ray-tracer validation tests

To validate that the basic elements of the ray-tracer are working as intended, we've created some relatively simple test scenes that we can validate through observation of the input and output of the ray-tracer. In this section we aim to test the three types of light source, the two types of shape and the changes to camera and light parameters. In all tests the camera is placed at position $(0, 0, 10)$ facing towards the centre of the scene.

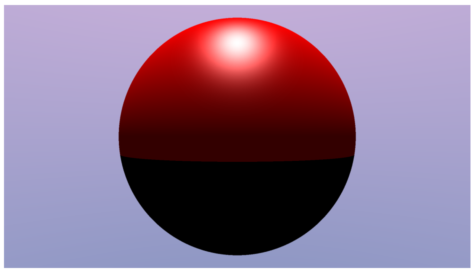

Figure 24: Output for sphere and directional light test

5.6.1 Test: sphere and directional light

Here we test the simplest scene possible, containing a sphere of radius 4 of colour red (`rgb (255,0,0)`) and a directional light with direction vector $(0, -1, 0)$ (i.e. coming from above) with colour white (`rgb(255,255,255)`). The scene *.x3d* code can be found in Appendix E.

The output shown in Figure 24 corresponds directly with expectations. The parts of the sphere not illuminated by the light source are displayed in shadow. The light reflection and specular highlights are at the correct point on the sphere in relation to the light source, and the rest of the sphere is the correct colour.

5.6.2 Test: spot light parameters

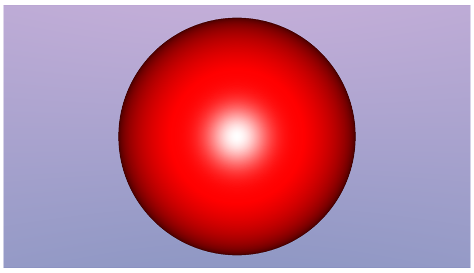

Figure 25: Output for first test with larger spot light cut-off angle

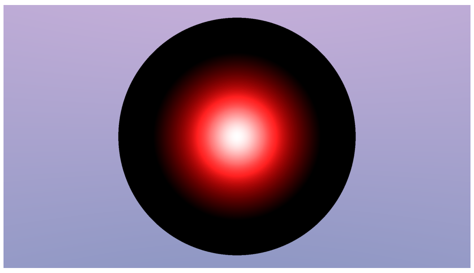

Figure 26: Output for second test with smaller spot light cut-off angle

Here we test the above scene but with a spot light at the camera's position facing towards the sphere instead of a directional light. We present two variations of this test: the first with the default cut-off angle and beam width parameters for the spot light; the second with both parameters reduced. The second variation scene *.x3d* code can be found in Appendix F.

From comparing Figure 25 to Figure 26, we can see that reducing the spot light's beam width and cut-off angle reduces the spread of the light source correctly.

5.6.3 Test: colour interactions and camera field of view

Here we test that the colour of the light is correctly interacting with the colour of the shapes as well as testing that the field of view is correctly changing. We do this by altering the scene test to contain four spheres arranged in a square structure, three with one of the primary pixel colours (red, green or blue) and a fourth that is grey (`rgb(127,127,127)`. We then present four variations of this test, each one altering the light's projected colour to one of the four sphere colours. Finally we slowly increase the field of view for the camera for each test. The code for the final test in Figure 30 can be found in Appendix G.

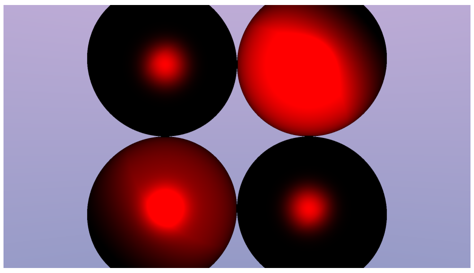
Figure 27: Projecting the colour red with a camera field of view value of 0.7

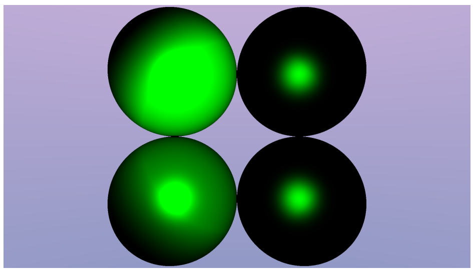
Figure 28: Projecting the colour green with a camera field of view value of 0.8

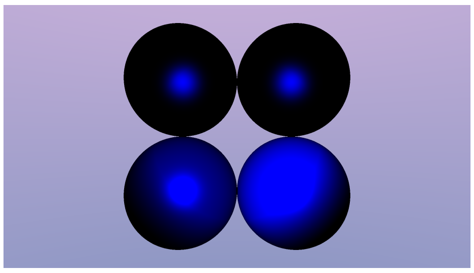
Figure 29: Projecting the colour blue with a camera field of view value of 0.9

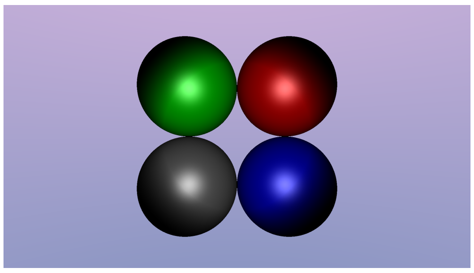
Figure 30: Projecting the colour grey with a camera field of view value of 1.0

We can tell that our tests produce accurate outputs because of the following: in all the three output images where the light corresponds to a single `rgb` colour (Figures 27, 28 and 29), each sphere that corresponds to the light's colour is fully illuminated and spheres of different colours to the light source only have specular highlights. The grey sphere is only partly illuminated by all three different `rgb` light sources. Finally in Figure 30 the grey light illuminates all spheres evenly, showing their true colours.

Additionally we can see that as the camera field of view increases, so does the viewing angle of the image.

5.6.4 Test: triangle meshes, point lights and shininess

To test that triangle meshes are working correctly we triangulated and exported a geometric primitive (an icosphere) from the three-dimensional modelling software **blender**[30]. We placed this icosphere four times in mirrored positions, giving the two left icospheres a shininess of 1 and the right icospheres a shininess of 0.1. Finally we replaced our spot light from the previous tests with a point light in the centre of the placed shapes. The source code for this test is omitted from the appendix for its length, but can be found in the `testfiles` directory of the attached code named `icosphere.x3d`.

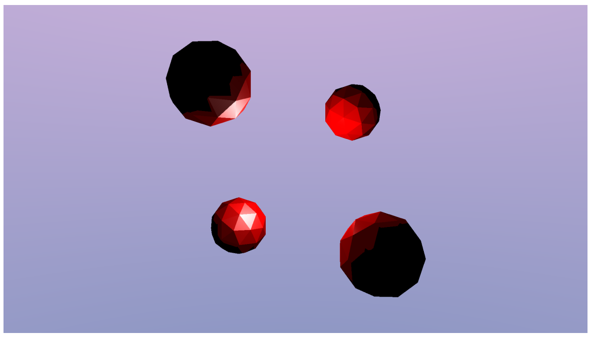

Figure 31: Scene of four icospheres exported from blender. The left icospheres have a shininess of 1, whereas the right icospheres have a shininess of 0.1

We can see from the ray-tracer output (Figure 31) that the difference in shininess is directly apparent: the two leftmost icospheres produce much brighter light reflection points than the right icospheres. Additionally it's clear that the point light illuminates each triangle mesh correctly.

5.6.5 Other features

It's clear from the above tests that at the basic level the ray-tracer is working as intended. However, there are other features included in the program that have not been tested in this section. For example: background colour mixing (described in Section 4.3.2), light attenuation (described in Section 4.3.3) and face weighted vertex normal generation (described in Section 5.2.2). These features are not tested directly because testing these minor features is not a priority to the project's main goal.

5.7 Octree tests

In this section we aim to test the difference between the regular and non-uniform octree structures by timing how long each one takes to generate and subsequently ray-trace over a number of test files. We take a couple of additional measurements to aid understanding in why one may improve over the other; these measurements are the generated octree's depth and total number of voxels, as well as the number of calls that the program makes to the intersection and find-next-voxel functions.

5.7.1 Care in testing

In order to obtain accurate test results, it's important to try to only time things that are deterministic in their process time. For this reason we moved all input and output of the program to outside the timed section. This includes, most importantly, file access. For this reason our program reads the file into a buffer before the timing section, and the octree generation then builds from that buffer every time.

Even after moving all I/O, a process's running length is still inherently stochastic. Because of this we have taken a timing of a large number of octree builds and subsequent ray-traces, and averaged them. It's important when timing multiple repeats not to start and restart the timer each time, as starting/stopping the timer is not necessarily instant.

A process can normally always be interrupted by another process (say, from the operating system); for this reason we ran all our test's processes with a much higher than average priority. A process with a higher priority is less likely to be interrupted, or more crucially, swapped out of memory for another process. We ran the tests giving them a priority of -10 on our Linux system using the command `sudo nice -n -10`. The default priority of a process is 0, and the highest priority (mainly only used by critical operating system processes) is -19.

When timing a process it's important to know the difference between the CPU's wall clock and the CPU's native clock. The wall clock will try to measure real time, whereas the CPU clock will measure the number of CPU ticks the process takes to complete and multiply it by its own measure of real time per tick. This CPU time gives a much more accurate description of how long our process is spending in user space, especially when working with small units of time. `C++` offers some standard library objects and methods for timing processes, but these are relatively inaccurate for our purpose. For this reason we're using **boost**'s `process_user_cpu_clock` to gather timing values instead.

Finally, to make sure the generated images are identical from both octrees, we added a flag to our testing program that would compute both and return the differences between the two files. This is run by adding `-compare` to the end of the command line argument.

All tests were run as a single thread on an Intel i5-2500K CPU running at 3.5GHz. Images were generated at 1080p, which creates 2,073,600 primary rays. In order to count the number of intersection tests and voxel searches each octree performed, we used `valgrind`'s profiler (`valgrind --tool=callgrind`).

5.7.2 Test file design

The main difference in the octree structures is how they split up many shapes in the scene, in order to properly test the differences between the two types of structure; the test files should organise the shapes in the scene in ways that challenge or aid each structure.

The uniform structure is aided when shapes do not cross the central splitting lines of the scene. On the other hand, the non-uniform structure shouldn't be aided by anything in particular, but theoretically would struggle more on scenes where the spatial centre and the shape average are extremely close. Giving the optimiser a very small reduced search space area that has identical midpoint scores all over wastes octree generation time with no benefit. In each test file we designed them to either challenge or aid a particular octree type.

5.7.3 Test files

This section describes each of the test files we created or generated and how we thought the octree structures would handle them. All tests (bar the base test) contain a point light placed in the spatial midpoint between the shapes, as well as a directional light that is aligned with the camera direction to better illuminate the faces of the shapes the camera is directed at.

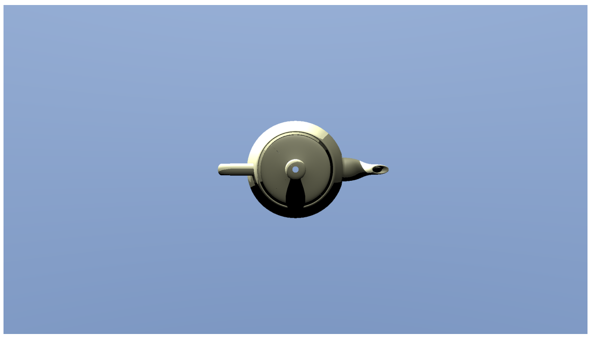

Figure 32: The Utah teapot; 25,352 triangles

Base test: Utah teapot This test produces no subdivisions for either type of octree. We are using this test to calculate any inherent bias towards either type of octree, potentially because of poor or unfair implementation. Because of the duration of this test, it was only repeated fifteen times. We chose the Utah teapot model as it contains many triangles, and if there are any biases a long test will accumulate a time difference. The generated output for this test is shown in Figure 32.

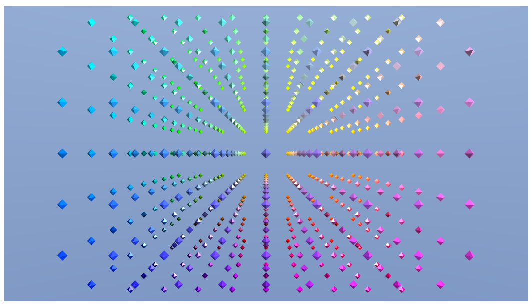

Figure 33: Test 1: each diamond contains 8 triangles, and the scene contains 728 diamonds for a total of 5,824 triangles

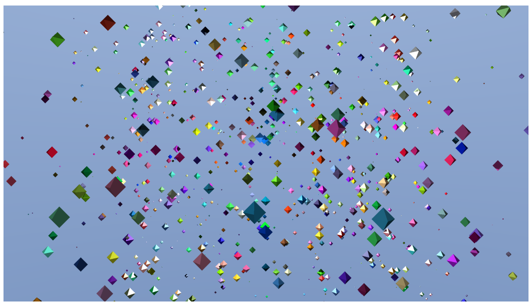

Figure 34: Test 2: each diamond is 8 triangles, and the scene contains 1,000 diamonds for a total of 8,000 triangles

Test 1: Uniform diamonds This test aligns the shapes of the scene directly in the path of the spacial split points of the scene. Theoretically this should be a poor performing scene for the uniform octree. The generated output for this test is shown in Figure 33.

Test 2: Random diamonds This test placed 8,000 of the diamond triangle mesh randomly inside a 1,000 × 1,000 × 1,000 cube within the scene. When placed randomly in such a small area the diamonds align themselves beside each other, creating awkward subdivisions for the uniform octree structure. The non-uniform structure, however, should be able to decipher optimal midpoints within the hectic scene. The generated output for the test is shown in Figure 34.

Test 3: Concentric spheres This test reduces the optimality of the non-uniform octree by placing the shape average point closely to the spatial midpoint. We're trying to force the non-uniform octree to waste time trying to compute the optimal midpoint, when the optimal midpoint is just the spatial centre, hopefully giving an advantage to the uniform octree structure. The generated output for the test is shown in Figure 35.

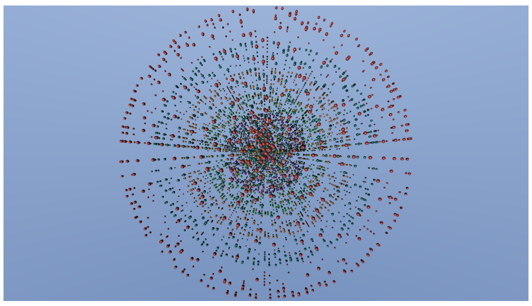
Figure 35: Test 3: this scene contains 5,985 spheres placed on the surface area of 5 larger spheres concentrically placed

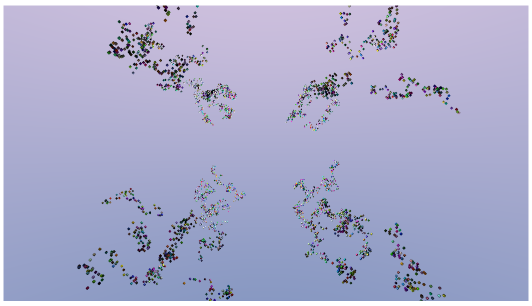
Figure 36: Test 4: this scene contains 3,200 diamond triangle meshes separated into different clusters, leaving empty space for subdivisions

Test 4: Clumped diamonds This test creates 64 spatially separated clumps of randomly placed diamonds. This test is trying to split up the shapes, reducing the number of shapes that span the larger splitting planes. For the smaller subdivisions we let the triangles cluster and overlap, reducing the effectiveness of the subdivisions at this level. The generated output for the test is shown in Figure 36.

5.7.4 Repeating experiments

These experiments can be repeated once the program is downloaded and installed from the git repository (instructions in Appendixes B and C). The base directory of the code contains a script named `alltests.sh`, which runs all the above experiments serially (bar the base test).

5.8 Results and analysis

Here we present and analyse the results of the tests presented in Section 5.7. We'll first analyse each test individually and explain the outcome between the two octree types, then give an overall comparison found between the two types in all tests.

5.8.1 Base test: Utah teapot

Utah teapot	**Uniform**	**Non-uniform**	**% change**
tree depth	1	1	*(-0.0%)*
voxels	1	1	*(-0.0%)*
voxels per shape	1	1	
intersection tests	529,548	529,548	*(-0.0%)*
intersections per ray	0.26	0.26	
voxel searches	284,970	284,970	*(-0.0%)*
searches per ray	0.14	0.14	
time taken (s)	213.763	213.912	*(+0.07%)*

This test ran exactly as expected. The only difference in the times between uniform and non-uniform were from expected random variance in CPU timing. Some variance was expected because this test was only repeated fifteen times (compared to the other tests that were repeated 100 times). This shows us that the random variance that occurs during timing is equal to $\sim \pm 0.1\%$.

5.8.2 Test 1: Uniform diamonds

Uniform diamonds	**Uniform**	**Non-uniform**	**% change**
tree depth	9	6	*(-33.33%)*
voxels	8,653	5,552	*(-32.68%)*
voxels per shape	11.89	7.63	
intersection tests	21,843,910	21,789,021	*(-0.25%)*
intersections per ray	10.53	10.51	
voxel searches	27,018,572	26,713,679	*(-1.13%)*
searches per ray	13.03	12.88	
time taken (s)	10.379	9.899	*(-4.62%)*

This test shows us how the shapes' median position affects the non-uniform octree's effectiveness. Even though the shapes lie directly on the spatially centred splitting point, both types of octree must make their initial subdivision there, as it's also the shapes' median position. With every subdivision made, the shapes' median position differs from the spatial centre more and more, giving the non-uniform octree more chances to subdivide optimally.

We see that the change in number of intersection tests and voxel searches of the non-uniform octree isn't a large improvement over the uniform octree. The change in number of voxels generated decreases the speed of each of the voxel searches done. This follows the theory that even if both octrees have identical voxel search and intersection counts, the tree with a smaller voxel count will see temporal improvements due to having fewer collisions while searching the global hashmap of all the octrees when finding the next octree leaf node.

5.8.3 Test 2: Random diamonds

Random diamonds	**Uniform**	**Non-uniform**	**% change**
tree depth	9	8	*(-11.11%)*
voxels	9,381	7,064	*(-24.70%)*
voxels per shape	9.38	7.06	
intersection tests	17,183,358	11,235,896	*(-34.61%)*
intersections per ray	8.29	5.42	
voxel searches	46,117,742	45,362,845	*(-1.64%)*
searches per ray	22.24	21.88	
time taken (s)	13.850	12.533	*(-9.51%)*

This test is a good example of how the non-uniform octree can reduce intersection tests. This scene's shapes are unevenly spread throughout the space and give the optimiser of the non-uniform octree a large search space to optimise within (causing the reduction in intersection tests). This reduction of intersection tests alongside a reduction in voxels generated gives a notable $\sim 10\%$ decrease in time taken.

The regular octree struggles with subdividing this scene well in comparison as the irregularity of the shape's positions causes its subdivisions to often place the shapes along the splitting planes. This explains the uniform octree's larger number of intersection tests in this scene, because when shapes span the splitting planes their intersections must be calculated whenever a ray occurs inside any of the containing voxels.

5.8.4 Test 3: Concentric spheres

Concentric spheres	**Uniform**	**Non-uniform**	**% change**
tree depth	8	8	*(-0.0%)*
voxels	9,178	8,422	*(-8.24%)*
voxels per shape	1.53	1.41	
intersection tests	33,225,320	24,876,977	*(-25.13%)*
intersections per ray	16.02	12.00	
voxel searches	25,603,151	25,089,438	*(- 2.01%)*
searches per ray	12.35	12.10	
time taken (s)	8.607	8.382	*(-2.61%)*

While the number of intersection tests is far fewer for the non-uniform octree structure in this test, the time taken for these intersection tests compared to the other test scenes is far smaller. This is because this is the only test scene that uses spheres over triangle meshes; a sphere intersection test is around an eighth of the speed of the intersection test of the diamond triangle mesh used in the other tests.

Overall, this scene's non-uniform octree has the smallest time improvement compared to all other test scenes. This can be attributed to the relatively small reduction in the number of voxels the non-uniform octree generates, when compared to the other test files.

5.8.5 Test 4: Clumped diamonds

Clumped diamonds	**Uniform**	**Non-uniform**	**% change**
tree depth	9	9	*(-0.0%)*
voxels	22,807	15,401	*(-32.47%)*
voxels per shape	7.13	4.81	
intersection tests	5,954,439	6,190,599	*(+3.97%)*
intersections per ray	2.87	2.99	
voxel searches	23,818,331	21,231,578	*(-10.86%)*
searches per ray	11.49	10.24	
time taken (s)	21.539	14.386	*(-33.21%)*

This is the only test with the non-uniform octree having an increase in the number of intersection tests done, despite being the test with the percentage change decrease in time taken. The reason for this behaviour is that the huge number of voxels generated causes the average time taken for the voxel search algorithm to become extremely large for both octree types. Note that while this voxel search algorithm may take longer than the intersection test in this scene, the sheer number of intersection tests it's reducing far outweighs the time saved by not subdividing.

The reason both octrees have a much larger number of voxels generated in this test compared to the other tests, is that this scene contains a large amount of blank space between the different groups of objects. This is especially devastating to the uniform octree, as it requires many more subdivisions than the non-uniform octree for any of the leaf node voxels to come close to encapsulating the scene's shapes.

The percentage change in voxels generated by the non-uniform structure combined with the larger than average percentage change of calls to the voxel search algorithm account for the huge time decrease between the two structures.

5.8.6 Comparing tests

We can notice some patterns in how the generated octrees and the changes in number of intersection and voxel searches affect the final time taken. The only value recorded that has no direct effect on the time taken is the depth of the octree; this can be related to how we're searching for the next voxel during ray-tracing, since we're using a hash map of all the octree nodes' names.

Any reduction in the number of intersection tests improves the time taken considerably, and we can see from the results that the non-uniform octrees can (usually) reduce these (the exception being the **Clumped diamonds** test). However, the non-uniform octree's greatest performance enhancement is its ability to reduce the number of generated voxels in comparison to the regular octree. Once the number of generated voxels in the octree is reduced, every single voxel search computation becomes faster. When the non-uniform octree additionally reduces the number of these voxel computations, further time improvements can be seen. This is specifically corroborated by the **Clumped diamonds** test.

Additionally the generation times for both octree types were noticed to be proportional to the number of voxels generated (this is as intuition suggests). Because of this the non-uniform octree was found to actually decrease generation times in comparison to the uniform octree.

6 Appraisal

In this section we review the ray-tracer built for the project, as well as the tests performed and any improvements that should be investigated further.

6.1 Comparison of uniform and non-uniform octree

The created uniform and non-uniform octree structures are both effective optimisations to a simple ray-tracing program. Here we give the advantages and disadvantages of creating and using a non-uniform octree structure over a uniform one.

One specific change in how we tested the two structures would be to separate the octree generation time and ray-tracing time as currently we time both together. This is not vital, but may give increased understanding in the major differences between the structures (the system *does* currently time one octree generation of the scene separately before performing any test; however, only one timing is not sufficient for analysis due to the inaccuracies pointed out in Section 5.7.1).

6.1.1 Advantages

As shown in Section 5.8 the speed-up gained from the non-uniform structure ranges from $\sim$ 2–30% depending on scene complexity and structure.

6.1.2 Disadvantages

The major disadvantage of the non-uniform octree is the difficulty of implementation. One example is the non-uniform octree's desire to create indiscriminately small voxels (described in Section 5.4.2). The fix to this issue (limiting the smallest size any voxel can become) limits how well the non-uniform octree can potentially encapsulate the shapes contained in the scene. A more complex solution may be possible by altering how the next leaf node is found, but is beyond the scope of this current project.

The second noticeable disadvantage to the non-uniform octree is the additional memory overhead. For small to medium scenes (where the number of shapes is below 100,000) this overhead is largely ignorable (we found no issues running multiple of these scenes at once with 8GB of RAM). However, for larger scenes (where the shapes exceed one million), this overhead becomes a noticeable boon on the effectiveness of the optimisation. This sets a limit on how effective the optimisation can be in relation to the running computer. We suggest potentially alleviating this by creating a hybrid uniform–non-uniform structure.

Additionally, there could be potential changes to the program to make more optimal use of `C++`'s memory management tools.

6.2 Further development

6.2.1 Potential ray-tracer improvements

To further improve the ray-tracer developed in this project there are almost endless features and optimisations that can be implemented. However, only some would improve the octree experiments. The most important of these is GPU parallelisation. Being able to compare our experimental results to other modern optimisation results directly would improve how we evaluate the effectiveness of our optimisation. Additionally, parallelising the generation of both octrees would have proved an interesting challenge, as the non-linear optimisation performed by the external library would have to be re-written to run on the GPU as well. This was too large a task for this project's undertaking, but is suggested as one potential avenue for further investigation, as some non-linear optimisation algorithms (specifically MLSL[19]) would benefit greatly from parallelisation.

Other improvements to the ray-tracer that are a standard for ray-tracing software but would not directly effect our octree tests are:

- rotations
- transparency
- reflections
- back-face culling (the process of ignoring triangles whose normal is facing the same direction as the intersecting ray)

All of these optimisations are not incredibly involved to implement; however, they would have all taken time out of the more crucial aspects of the project, and wouldn't have improved our experimentations in any meaningful way.

6.2.2 Other areas for further investigation

One major avenue for experimentation and investigation is in the parameters passed to the non-linear optimisation algorithm and choice of algorithm used. While we did perform our own simple experimentation between the different algorithms, we suggest a more complex and thorough study of the algorithms be done to improve the non-uniform octrees' generation.

Finally the use of animation is vital in areas where ray-tracing is found in industry. We suggest that alongside parallelisation, a study into the real-time ray-tracing of animated scenes using non-uniform octrees could give a more realistic use-case of this type of octree generation.

7 Conclusion

In this paper we have created a ray-tracer including octree optimisations in `C++`. We have shown that splitting the octree structure non-uniformly through the use of non-linear optimisation gives a temporal performance increase of 2–30% in comparison to regular spatially divided octrees. These non-linear optimisations minimise a midpoint score heuristic, created by modification of the kD-tree splitting heuristic, to guide each subdivision to be as optimal in relation to the surface area of the voxels it creates. Proposals for improvements on the created ray-tracer have been made; we suggest that other projects take these further to make non-uniformly subdivided octrees a more realistic optimisation.

References

[1] Scratchapixel. `https://www.scratchapixel.com/index.php?redirect`, 4 2016. Accessed: April 8, 2018.

[2] Andreas Dietrich, Abe Stephens, and Ingo Wald. Exploring a Boeing 777: Ray tracing large-scale CAD data. *IEEE Computer Graphics and Applications*, 27(6):36–46, November 2007.

[3] Hugo Pacheco. Ray tracing in industry. Technical report, Universidade do Minho, Braga, Portugal, 2015.

[4] Bounding volume hierarchy image. `https://en.wikipedia.org/wiki/Bounding_volume_hierarchy/media/File:Example_of_bounding_volume_hierarchy.svg`, October 2017. Accessed: November 9, 2017.

[5] Christer Ericson. *Real-Time Collision Detection.* Elsevier Science, 2004.

[6] Andrew S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–24, 1984.

[7] Michal Hapala and Vlastimil Havran. Review: kD-tree traversal algorithms for ray tracing. *Computer Graphics Forum*, 30:199–213, 2011.

[8] Marke Vinkler, Vlastimil Havran, and Jiri Bittner. Performance comparison of bounding volumne hierarchies and kD-Trees for GPU ray tracing. *Computer Graphics Forum*, 35:68–79, 2016.

[9] Arsene Perard-Gayot, Javor Kalojanov, and Philipp Slusallek. GPU ray tracing using irregular grids. *Computer Graphics Forum*, 37:477–486, May 2017.

[10] Carsten Wachter and Alexander Keller. Instant ray tracing: The bounding interval hierarchy. In *Eurographics Symposium on Rendering*, pages 139–149, 2006.

[11] John Amanatides and Andrew Woo. A fast voxel traversal algorithm for ray tracing. Master's thesis, University of Toronto, 1987.

[12] J. David MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6:153–166, 1990.

[13] Turner Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23:343–349, June 1980.

[14] Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM*, 18:311–317, June 1975.

[15] Ulises Olivares, Hector G. Rodriguez, Arturo Garcia, and Felix F. Ramos. Efficient construction of bounding volume hierarchies into a complete octree for ray tracing. *Computer Animation and Virtual Worlds*, 27:358–368, 2016.

[16] Kyu-Young Whang, Ju-Won Song, Ji-Woong Chang, Ji-Yun Kim, Wan-Sup Cho, Chong-Mok Park, and Il-Yeol Song. Octree-r: an adaptive octree for efficient ray tracing. *IEEE Transactions on Visualization and Computer Graphics*, 1:343–349, December 1995.

[17] James T. Klosowski, Martin Held, Joseph S.B. Mitchell, Henry Sowizral, and Karel Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4:21–36, 1998.

[18] Thomas Philip Runarsson and Xin Yao. Search biases in constrained evolutionary optimization. *IEEE Transactions on Systems Man and Cybernetics*, 35(2):233–243, 2005.

[19] Sergei Kucherenko and Yury Sytsko. Application of deterministic low-discrepancy sequences in global optimization. *Computational Optimization and Applications*, 30:297–318, 2005.

[20] E. G. Birgin and J. M. Martinez. Improving ultimate convergence of an augmented lagrangian method. *Optimization Methods and Software*, 23(2):177–195, 2008.

[21] M. J. D. Powell. Direct search algorithms for optimization calculations. *Acta numerica*, 7:287–336, 1998.

[22] T. Rowan. *Functional Stability Analysis of Numerical Algorithms.* PhD thesis, Department of Computer Sciences, University of Texas, 1990.

[23] Myeong Won Lee, Charlse Whitlock, Limei Yu, and Alexandra Gigant. Information technology – computer graphics, image processing and environmental data representation – extensible 3d (x3d) – part 1: Architecture and base components. Technical Report 3, Information Technology Task Force, 11 2013.

[24] Tomas Möller. Fast, minimum storage ray/triangle intersection. *Journal of Graphics Tools*, 2:21–28, 1997.

[25] Doug Baldwin and Michael Weber. Fast ray–triangle intersection by coordinate transformation. *Journal of Computer Graphics Techniques*, 5(3):39–49, 2016.

[26] Olinde Rodrigues. Des lois géometriques qui regissent les déplacements d'un systéme solide dans l'espace, et de la variation des coordonnées provenant de ces déplacement considérées indépendant des causes qui peuvent les produire. *Journal de mathématiques pures et appliquées*, 1(5):380–440, 1840.

[27] George Watson. Phys345 introduction to the right hand rule. `http://www.physics.udel.edu/~watson/phys345/Fall1998/class/1-right-hand-rule.html`, 1998.

[28] Scikit-optimize. `https://scikit-optimize.github.io/`. Accessed: April 8, 2018.

[29] Steven G. Johnson. The NLopt nonlinear-optimization package. `http://ab-initio.mit.edu/nlopt`.

[30] Blender: Free and open source 3D creation suite. `https://www.blender.org/`. Accessed: November 30, 2017.

Appendixes

A Structure of zip file attached

The zip file attached contains the following:

- README.md: explains the compilation and running of the program (also located in appendix section C)
-
- The source code for the project: this is split between multiple different class files, the main file is in the bottom-most directory named main.cpp. The headers for all used classes are in the **include** directory and the implementation of these headers are in the **src** directory.
- The test files for the project: these are all contained within the **testfiles** directory.
- The callgrind output files: these are the profiler files that count the number of calls to each function in the program for the specific runs used in the results section of the report. These are located in the the **callgrinds** directory.
- Script that runs all octree tests from the report: this is located in the base level directory and is named *alltests.sh*
- Makefile: Located in the base directory, called using the `make` command compiles with `-Og`, whereas `make fast` compiles with `-Ofast`
- Test generator programs: these generated many of the tests for the project, these are written in `C++` and are located in the **testfileGenerator** directory.

B Compiling the code

The project builds with the external library flags `-lboost_system`, `-lboost_chrono`, `-lnlopt` and `-lm`, from the libraries **NLopt** and **Boost**. These libraries must be installed before the project will compile.

These libraries dowload and install instructions can be found at:

- **NLopt**: `https://nlopt.readthedocs.io/en/latest/#download-and-installation`
- **Boost**: `https://www.boost.org/doc/libs/1_66_0/more/getting_started/unix-variants.html`

Additionally the extra boost library `boost_chrono` must be built, the instructions to do so are also included in the above link.

C Running the code

The project is a command line based application, with several parameters you can alter (if not filled they will revert to working defaults). For a list of available parameters the application can be run with the flag `-h` or can be seen below:

- `--file=path/to/file` : sets the *.x3d* file to be read
- `--rep=15 : sets the` number of repetitions to 15

- `--res=840,350` : sets the resolution of the output image
- `--type=uniform` : sets the octree type to be used to uniform
- `--type=nonuniform` : sets the octree type to be used to non-uniform
- `-showoctree` : before running the tests displays the generated octree as a text output (extremely verbose)
- `-compare` : runs both types of octree on the given file and compares the two output arrays of colours to confirm they generate the same image
- `-h` : displays this command list

The program will the generate the *.bmp* output image in the same directory as the source image. While running the program will output the built octrees depth and voxel number to the command line. It will additionally output the average time it takes to generate and ray-trace over the number of repetitions given to the program.

The default arguments as specified by the user looks like so: `./main --file=testfiles /trianglemesh --rep=5 --type=uniform --res=1920,1080`

D Subvoxel creation algorithm

```
split(type) {
    // If we're splitting nonuniformly: first compute a new optimal centre
    if (type == nonuniform)
        this.midpoint = this.compute_new_midpoint()

    // Create all the subvoxel octrees
    for int newname = 1 to 8 {
        //Find maximum point and minimum point directed by the name
        //If the name is 1 to 4 we know the y lies above the midpoint,
            therefore:
        if (newname < 5) {
            minP.y = this.MP.y
            maxP.y = this.MP.y + this.positive_width.y
        } else {
            //Otherwise we know the y lies below the midpoint, therefore:
            minP.y = this.MP.y - this.negative_width.y
            maxP.y = this.MP.y
        }

        //If the name is even we know the x lies on the positive side of the
            midpoint:
        if (newname mod 2 == 0) {
            minP.x = this.MP.x
            maxP.x = this.MP.x + this.positive_width.x
        }
        else {
            //Otherwise it lies on the negative side
            minP.x = this.MP.x = this.negative_width.x
            maxP.x = this.MP.x
        }

        //Finally we check the names required for the subnodes z value to be
            positive
        if (newname == {3,4,7,8}) {
            minP.z = this.MP.z
            maxP.z = this.MP.z + this.positive_width.z
        } else {
         //Otherwise the z lies on the negative side of the midpoint
            minP.z = this.MP.z - this.negative_width.z
            maxP.z = this.MP.z
        }
        // The midpoint is then between the maximum point and minimum point
            found
        newMP = (minP + maxP) / 2
        // and the widths can then be calculated
        newpositive_width = maxP - newMP
        newnegative_width = newMP - minP
        //  We can now create the sub voxel
        newvoxel = Octree(this.name.concat{newname}, newMP, newpositive_width
            , newnegative_width)
        this.leaves.add(newvoxel)
    }
    // Then we place the objects into each subvoxel
    ...
}
```

E Validation test: sphere and directional light

```
<Scene>
    <Viewpoint position='0 0 10'/>
    <Background groundColor="0.43 0.54 0.72" skyColor="0.9 0.74 0.9"/>
    <DirectionalLight direction='0 -1 0'/>
    <Transform translation='0 0 0'>
        <Shape>
            <Sphere radius='4'/>
            <Appearance>
                <Material diffuseColor='1.0 0 0'/>
            </Appearance>
        </Shape>
    </Transform>
</Scene>
```

F Validation test: spot light parameters

```
<Scene>
 <Viewpoint position='0 0 10'/>
 <Background groundColor="0.43 0.54 0.72"
             skyColor="0.9 0.74 0.9"/>
 <SpotLight direction='0 0 -1' location='0 0 10' cutOffAngle='0.3' color='1 1
     1' beamWidth='0.1'/>
 <Transform translation='0 0 0'>
  <Shape>
   <Sphere radius='4'/>
   <Appearance>
    <Material diffuseColor='1.0 0 0'/>
   </Appearance>
  </Shape>
 </Transform>
</Scene>
```

G Validation test: colour interactions and field of view

```
<Scene>
    <Viewpoint position='0 0 10' fieldOfView='1'/>
    <Background groundColor="0.43 0.54 0.72" skyColor="0.9 0.74 0.9"/>
    <SpotLight direction='0 0 -1' location='0 0 10' cutOffAngle='0.45'
        color='0.5 0.5 0.5' beamWidth='0.3'/>
    <Transform translation='2 2 0'>
        <Shape>
            <Sphere radius='2'/>
            <Appearance>
                <Material diffuseColor='1.0 0 0'/>
            </Appearance>
        </Shape>
    </Transform>
    <Transform translation='-2 2 0'>
        <Shape>
            <Sphere radius='2'/>
            <Appearance>
                <Material diffuseColor='0 1 0'/>
            </Appearance>
        </Shape>
    </Transform>
    <Transform translation='2 -2 0'>
        <Shape>
            <Sphere radius='2'/>
            <Appearance>
                <Material diffuseColor='0 0 1'/>
            </Appearance>
        </Shape>
    </Transform>
    <Transform translation='-2 -2 0'>
        <Shape>
            <Sphere radius='2'/>
            <Appearance>
                <Material diffuseColor='0.5 0.5 0.5'/>
            </Appearance>
        </Shape>
    </Transform>
</Scene>
```