NTNU – Trondheim
Norwegian University of
Science and Technology

A Method for Rigid-Body Animation of Sparse Voxel Octrees

And Its Applications in Real-Time Ray Tracing

Asbjørn Engmark Espe

Project Thesis in Embedded Systems

Credits:	15 SP
Submission date:	December 2018
Supervisors:	Sverre Hendseth, ITK
	Øystein Gjermundnes, IES

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem statement

Ray tracing is a rendering technique in which rays are cast from a viewpoint into a scene in order to generate images. Briefly stated, it entails tracing in reverse the paths that physical light particles would take. The technique is viewed as an alternative to rasterisation rendering, which is the *de facto* standard in computer graphics today. The latter is based on the rasterisation of primitives—most commonly triangles.

As a consequence of the technique's physical nature, ray tracing implementations will often produce more realistic images, and will generally result in a more visually and physically correct rendition of a given scene, when compared to rasterisation renders. Caused in part by the fact that the objects to be rendered may be modelled perfectly in a geometric sense, but also because physical effects such as reflection, refraction, and transparency are much easier to model and implement than for rasterisation rendering.

An **octree** is a space partitioning scheme that divides three-dimensional space into a tree structure. It is analogous to a binary tree in three dimensions, meaning that each node in the tree may have up to eight children. A sparse voxel octree—abbreviated SVO—is a specific flavour of octree often used in conjunction with ray tracing. What sets an SVO apart from the generalised octree is that the data structure itself is employed to natively represent volumetric data. Physical objects may be approximated by treating the SVO leaf nodes as either filled or empty (void) voxels. Several solutions have been proposed in the relevant literature for efficient real-time rendering of SVOs using ray tracing.

A drawback of using SVOs to store and render volumetric data is that the data is inherently static; SVOs by themselves do not support any form of animation or efficient data mutation. In order to use this data structure in the rendering of realistic real-time graphics, there is a need for animation—most notably rotation, translation, but also general affine transformations. This is necessary if ray tracing is to be used to render realistic, animated worlds, and not just static scenes.

In the project thesis, the student will:

- Review existing literature on the subject of ray tracing and sparse voxel octrees, and earlier attempts of animation in this context.
- Formulate a method that permits animation of SVO data for use in ray tracing. Evaluate the efficiency of the method in terms of performance and suitability for implementation in hardware.
- If feasible, demonstrate the method by ray tracing animated SVOs in real-time.

Abstract

Ray tracing is a technique used in computer graphics to render virtual scenes consisting of three-dimensional volumetric models. These volumetric models may be formulated as geometric primitives, or as data structures such as the optimised voxel-based model called sparse voxel octree (SVO). One of the main limitations today when using ray tracing to render SVOs is that the octree data structure is inherently static. In other words, efficient animation of a scene to be rendered is challenging to achieve.

Presented in this thesis is a method for animation of models specified on the SVO format. The method is limited to rigid-body animation, which means that certain effects, such as deformation and bending, is not supported. The proposed solution employs two popular established algorithms as foundation: an efficient method for traversal of octrees, and a memory-efficient data structure scheme for storing SVO data. The achieved result is a successful animation technique that—with certain optimising features enabled—does not noticeably slow down the rendering procedure compared to rendering pure, non-animated SVOs. It is also determined that the technique is well-suited for hardware implementation, in part due to its modest memory footprint.

A software implementation of the animation technique is also presented. It is demonstrated on the basis of this implementation that real-time performance is indeed achievable when rendering SVOs in software, both animated and non-animated. The results lend credence to the claim that the animation technique is suited for real-time applications. Alongside the software implementation, an optimisation technique termed hit buffer object (HBO) is introduced. It is shown that the HBO plays a fair part in the performance achieved by the software implementation.

Preface

As far back as I can remember, I always had a personal interest in computer graphics, and in particular, video games. This interest combined with an affinity for programming ensured that I spent a great deal of my free time growing up developing and experimenting with computer graphics software. Again, my most noteworthy endeavours were attempts at creating video games, which were more or (most often) less successful.

I started gaining an interest in the concept of ray tracing in high school, and developed a couple of simple software ray tracers during this time. I was especially attracted to the simplicity and elegance of the ray tracing model in comparison to traditional rasterisation. Already at this stage, I felt that the technology behind ray tracing was vastly underappreciated; to me, ray tracing appeared to be the holy grail of computer graphics.

The choice of topic for the project thesis was therefore rather simple. I still feel ray tracing is the future of computer graphics, and hence want to explore the possibilities of employing ray tracing in a real-time context. After the project thesis work, I hope to continue my research as a part of my master's thesis. I also aim to continue my academic career as a Ph.D. student, and hope that this work might give me valuable experience to this end.

I would like to extend my gratitude to my supervisors, who have aided me over the course of writing this thesis by keeping my motivation high during the process. They have also prompted many of my ideas by asking the right questions at the right times. I would also like to thank my girlfriend for keeping my spirits up during this project, as well as my father for showing an interest in my work and for assisting me in the final stages of writing this thesis. Lastly, special thanks go to *Arm Norway* for letting me use a desk in their offices. By also giving me access to (virtually) unlimited amounts of free coffee, they should be recognised with part of the honour that this finished thesis yields.

Contents

Chapter 1

Introduction

> I do think that there is a very strong possibility—as we move towards next generation technologies—for a ray tracing architecture that uses a specific data structure. (...) There is a specific format I have done some research on that I am starting to ramp back up on for some proof of concept work for next generation technologies. It involves ray tracing into a sparse voxel octree.
>
> — *John Carmack, 2008* [1]

It seems that in recent time, the topic of real-time ray tracing has been on everyone's lips. In the autumn of 2018, the graphics card manufacturer Nvidia promised to "reinvent graphics" with their new *GeForce RTX* series featuring hardware acceleration of ray tracing [2]. The response to their showcase of what the technology has to offer suggests that, this time, ray tracing has really caught the public's interest. But ray tracing is far from a new concept, and throughout its history, the interest in the technology has had both ups and downs.

The main issue concerning ray tracing as a technology appears to be that real-time operation is very difficult to achieve. And, even though it may seem that the technology has finally caught up to enable such performance, the resistance to adoption of ray tracing compared to sticking to regular rasterisation approaches has been too large. Ray tracing is not utilised as a technology in real-time rendering because there exists very little hardware to accelerate it, and there is a sparse selection of hardware because ray tracing is not widely used.

The issue of adoption appears to be a classic example of *path dependence* [3], and escaping or correcting such a dependence will in many cases require some form of disruptive innovation. However, in this case, the solution may be an approach that has recently gained attention—using ray tracing in conjunction with traditional rasterisation. The idea is to try to achieve a sort of *best-of-both-worlds* scenario, where the realism of ray tracing is combined with the speed of rasterisation. This approach is a central part of what Nvidia demonstrates with their new series of graphics cards, and is also reinforced in newer papers written on the topic [4][5].

Interpretation of problem statement

The problem statement details the current state of affairs in the field, and briefly explains a few relevant concepts. The task itself is split into three main points. The first is to perform a thorough literature review in the field and explore earlier attempts. The second point concerns the formulation of a method for animation of sparse voxel octree data. The third point entails implementing the method in some way. Since the problem text is written precisely, there is not a lot of room for interpretation. However, some choices must be made and justified.

The third point states that an implementation of the method should be demonstrated, but does not specify what kind of demonstration should be made. As a consequence of the limited time window, the choice is to implement the demonstration in software. A hardware implementation would frankly result in a project with too large a scope for a project thesis.

Another decision that must be justified is what kind of animation that will be supported by the solution. The problem text states that animation should be permitted, but does not define the nature of said animation. The background text in the problem statement does, however, discuss the nature of the animation to some degree; affine transformations are mentioned. In order to limit the scope of the work, this thesis will focus on rigid-body animation. This distinction is made on the basis that animation of rigid bodies is in general easier to achieve than other forms of animation—for instance mesh animation, which is more suited when using a rasterisation approach.

Thesis outline

In this thesis, a method for animation of sparse voxel octrees will be presented. A brief introduction to relevant concepts can be found in Chapter 2. The research context as a whole is discussed in Chapter 3, and the algorithms and schemes upon which this thesis is based are detailed in Chapter 4. Starting with Chapter 5, the results are presented. In this chapter, the method for animation is explained and discussed. In Chapter 6, an actual implementation of this method is analysed. Finally, the thesis is concluded in Chapter 7.

A small note about figures

Unless otherwise specified, all figures presented in this text are designed and created by the author. It is not an insignificant workload that has been laid down in their creation, and as such, they should be regarded as original (or, in some cases, improved derivative) works and hopefully contribute to the credit of this thesis.

Chapter 2

Background

Images generated by computers permeate the lives of humans today in ways one could not imagine merely half a century ago. In the current age of information, there are numerous areas of which computer graphics is a central part, for instance the fields of numerical computing visualisations [6][7][8], video games [9][10][11], computer-aided design (CAD) [12][13][14], graphical user interfaces (GUIs) [15][16], and special effects for motion pictures (SFX) [17][18].

In this chapter, relevant background information will be presented, starting with a short account of the eventful history of real-time computer graphics.

2.1 A brief history of real-time computer graphics

Computer graphics as a field has only existed for about 60 years. The term first appeared around 1960, and was used to describe early works such as Ivan Sutherland's groundbreaking computer program *Sketchpad* [19]. Sutherland's program is considered the antecedent to modern computer graphics in that it was the first graphical solution enabling *human-computer interaction* (HCI).

The history of real-time computer graphics is defined by breakthroughs. The rendering capabilities at a given point in time was generally constrained by capacity of the underlying hardware. For instance, once transistor-based memory was available in the 1970s, the creation of efficient *frame buffers* was possible. Frame buffers are to this day central, as the technology simplifies and speeds up computation by allowing the decoupling of rendering logic from display logic [20][21].

By the late 1970s, three-dimensional computer graphics had left its infancy. And as the field matured, it became apparent that two main methods of real-time rendering were to dominate the scene. The earliest attempts had used *rasterisation*, a technique which gained popularity after the introduction of the *Z-buffer* in 1974, developed and proposed by both Edward Catmull [22] and Wolfgang Straßer [23], independently. The alternative to rasterisation was *ray tracing*, an image synthesis technique that was first proposed by Arthur Appel in 1968 [24], but popularised after a paper by Turner Whitted in 1980 [25].

It would become clear, through the conception of dedicated graphics processing

hardware in the 1980s and early 1990s, that rasterisation was to be the main-stream technique for real-time computer rendering. One of the earliest single-chip display controllers was the *NEC 7220*, released in 1982 [26]. By incorporating this chip in their designs, manufacturers could construct dedicated *graphics processing units* (GPUs), such as a range of products released by Number Nine Visual Technology through the 1980s. In these early GPUs, rasterisation of primitives—rather than ray tracing—was employed to produce output, consolidating rasterisation as the preferred method for image synthesis [27].

Through the 1990s and 2000s, real-time computer graphics would continue to increase in popularity, as new graphics acceleration hardware would be released on a regular basis. Once the *personal computer* (PC) became a household item, high performance computing, and with it real-time computer graphics, was no longer reserved for the specialist user. In the early 2010s, a new field of research emerged—embedded computer graphics, or mobile graphics. While the main constraints in desktop hardware are performance and price, mobile devices introduce an additional concern: as a consequence of their battery-powered nature, there is a desire to deliver real-time computer graphics while simultaneously maintaining a certain degree of power efficiency [28].

Even though rasterisation is currently the title-holder of the real-time computer graphics race, ray tracing continues to be relevant. In the latest months, the popularity of ray tracing experienced a revival after Nvidia unveiled its newest range of GPUs—the *GeForce RTX* series [2]. It should be interesting to see to which degree this recent main-stream attention will impact the future of computer graphics.

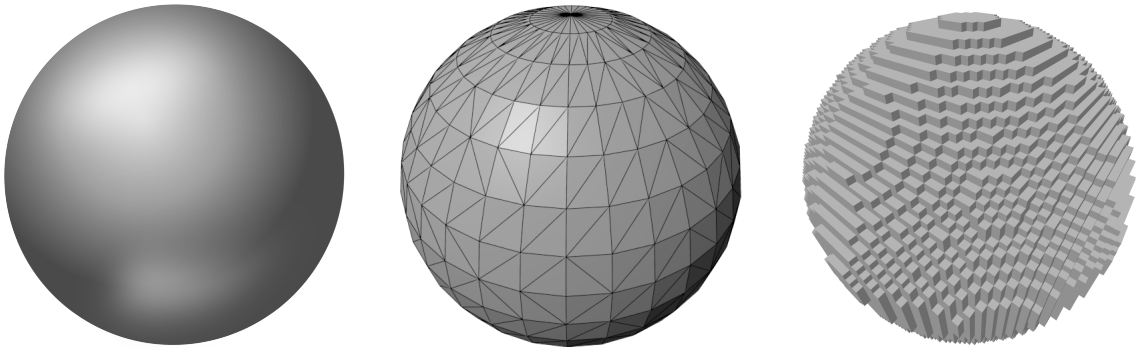
2.2 Fundamental concepts

The overarching goal of computer graphics is to use a computer to deterministically render an image based on a specification of some form. The images generated may be stored for later consumption, or they may be presented in real-time on a display as part of a graphics pipeline. Regardless of what the end goal of the rendering process is, and which techniques were employed to reach this goal, there are some fundamental concepts that are used almost universally. These fundamental concepts are presented in this section.

2.2.1 3D models

In three-dimensional computer graphics, the *model* can be viewed as a digital description of what is to be rendered. Models are a central part of most computer rendering pipelines and contain the data that is interpreted by the rendering process when generating a scene. Ordinarily, a model is a digital representation of a single, distinct object; a scene is a collection of models in some configuration. A model may be sorted into one of three main categories based on way it represents physical objects digitally. Each category has its drawbacks and advantages, which

will be outlined in the following. Illustrations of the different categories are shown in Figure 2.1.



(a) Analytic model (b) Polygonal model (c) Voxel model

Figure 2.1: Three types of models.

Analytic models

Models that can be completely described by some mathematical equation are known as *analytic models*. Such models are often constructed from analytic, geometric primitives, for instance cubes, spheres, curves, and lines. These models have the advantage of supporting arbitrary accuracy [29, pp. 66, 70][30, p. 176]. For instance, a sphere of radius 1, centred at the origin can be modelled perfectly by describing it analytically as all the points $\mathbf{x}$ that satisfy Equation (2.1).

$$||\mathbf{x}|| = 1 \tag{2.1}$$

A drawback of analytic models is that they are in general more computationally expensive to render compared to alternative methods. Another shortcoming is the fact that it can be very difficult to model complex shapes mathematically. Fortunately, for a given level of detail, one hardly ever needs the fidelity of mathematically perfect models. As such, they are rarely used in real-time three-dimensional computer graphics. The main use of this category of models is in the technique called *constructive solid geometry* (CSG), where multiple geometric primitives are combined with Boolean logic to produce more complex models [30, pp. 555–559]. An illustration of a sphere modelled analytically is shown in Figure 2.1a.

Polygonal models

A more common approach is to describe the object by a polygonal surface mesh. These models, known as *polygonal models*, only store the boundary or shell of the object, and are therefore a type of *surface representation* [30, p. 176]. A polygonal model, such as the sphere shown in Figure 2.1b, can be regarded as a vector graphics representation of an object, in the sense that a series of points are employed to define the vertices of a surface. The points are connected by a mesh comprised of

polygons—in almost all practical cases, triangles [11, p. 426][30, p. 177]. By their nature, polygonal models are very well suited for representing large, flat surfaces. As an example, a flat plane can be modelled by four vertices describing two triangles. A drawback is that polygonal models cannot accurately represent curved surfaces [29, p. 4]. Fine detail quickly becomes computationally expensive to render, since the level of detail is dependent on the number of polygons.

Voxel models

If polygonal models are a form of vector graphics representation of objects, *voxel models* may be regarded as a pixel or raster graphics representation. A voxel model, exemplified by Figure 2.1c, is a model which is made up of a set of *voxels*, short for volumetric elements. A voxel can be thought of as the three-dimensional analogue of the two-dimensional pixel. In other words, a voxel represents a single data sample in a three-dimensional grid. As a consequence of their volumetric nature, voxel models are a type of *volume representation*, therefore contrasting polygonal models [30, p. 177]. However, certain optimisations of the model data structure can be made so that the model appears as a type of surface representation, for instance by employing tree-based structures [31].

2.2.2 Spaces and transforms

After being sculpted and exported from some form of graphics modelling software, a model will usually be stored in a normalised, axis-aligned format, with all co-ordinates being relative to a single point called the model's origin [11, p. 428]. In the other end of the rendering process, the final image produced is in most cases going to be displayed on a digital screen. Specifically, the end result of the graphics pipeline is a set of pixels positioned in an ordered two-dimensional grid. The rendering process itself is often thought of as a black box process, as shown in Figure 2.2. However, in order to implement these different spatial representations, and the process of converting between them, there is a need for a more rigorous definition.

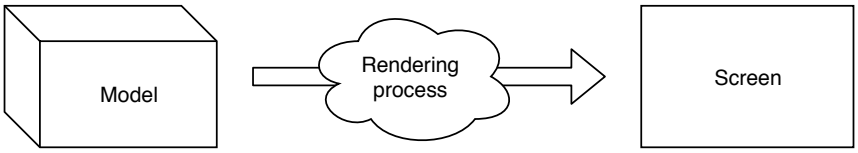


Figure 2.2: The rendering process as a black box.

Vector spaces

By introducing the concept of spaces, the process can be simplified quite extensively. As it turns out, all the different co-ordinate systems normally encountered in computer graphics may be represented mathematically as vector spaces.

The first of the co-ordinate systems mentioned in the introduction to this section—the one which is local to the model—is termed the *model space*, or alternatively *local space* [29, p. 7] or *object space* [11, p. 428]. As for the screen, each individual pixel in this grid will have a unique set of co-ordinates that describes its position. The pixels are said to reside in *screen space* [29, p. 10].

Since the screen on which to display the final image is two-dimensional, while the models are (in most cases) three-dimensional, there is a need to convert between the two spaces. Further, it is desirable in almost all cases to rotate, scale, or move the model around in the scene. To facilitate these requirements, two new vector spaces are introduced: *world space* and *view space*. World space is the global space in which all models are positioned, rotated, and scaled as desired. It is the space that describes the larger world, and is what one usually thinks of as the scene in computer graphics [11, p. 428][29, p. 7]. All models to be rendered must be placed somewhere in the world space. View space, which is sometimes named *camera space* [29, p. 7], is the space that places the camera (or *eye*) in the origin, looking down the negative z-axis, and orients all objects in the scene such that they are placed correctly relative to the camera's point of view. This space is an intermediate step that is needed to correctly perform the final conversion to screen space.

As shown in Figure 2.3, there are four main spaces in the rendering pipeline. For illustration purposes, the camera is shown as an entity in world space. Some method for deterministically transforming co-ordinates in one space to another space is desired. Luckily, since the spaces can be mathematically defined as vector spaces, the transformations between each space can be represented as mathematical *transformations*.

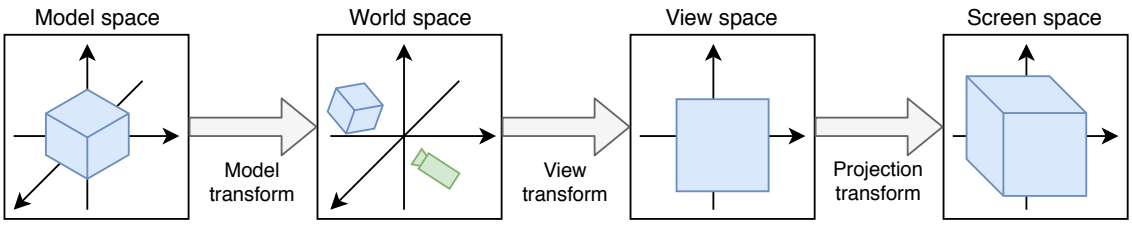


Figure 2.3: Main spaces in rendering. The camera itself is shown in world space.

Transforms and transformations

In mathematics, a transformation, also known as a *map*, is a generic function that maps one space to another. The mathematical formulation of a map $\mathbf{T}$ from $\mathbb{R}^n$ to $\mathbb{R}^m$ is shown in Equation (2.2).

$$\mathbf{T} : \mathbb{R}^n \to \mathbb{R}^m \tag{2.2}$$

Many mathematical maps, and all linear maps, can be written on matrix form. The map function $\mathbf{T}$ may be written as shown in Equation (2.3), where $\mathbf{A}$ is a matrix of

size $m \times n$ and $\mathbf{x}$ is a column vector with n elements.

$$\mathbf{T}(\mathbf{x}) = \mathbf{A}\mathbf{x}\ , \qquad \mathbf{x} \in \mathbb{R}^n, \mathbf{A}\mathbf{x} \in \mathbb{R}^m \tag{2.3}$$

As it turns out, all the transformations needed in the graphics pipeline, except one, are *affine transformations*, which means that they can be represented as a homogeneous matrix. In computer graphics, such a matrix is called a *transform*, and the function it provides is called a *transformation*.

Table 2.1: The main transforms of the computer graphics pipeline.

Initial space	Result space	Transform name
Model space	World space	Model transform
World space	View space	View transform
View space	Screen space	Projection transform

Since there are four spaces of concern, it stands to reason that there are three main transforms that are needed to convert between the spaces. These transforms and their names are shown in Table 2.1. The transform which converts from model space to world space is aptly known as the *model transform*, and the transform from world space to view space is called the *view transform*. The last transform, converting view space to screen space, is unique in that it is not a general affine transform. It is known as the *projection transform*, since it serves the function of projecting a three-dimensional world onto a two-dimensional plane (the pixel grid).

In practice, the three transforms are represented by four-dimensional matrices. A complete transformation of a point from model space to screen space can be written mathematically as shown in Equation (2.4).

$$\mathbf{q}' = \mathbf{PVMp} \qquad \mathbf{q} = \frac{\mathbf{q}'}{q'_w} \tag{2.4}$$

Where the vectors $\mathbf{p}$ and $\mathbf{q}$ are the initial and final positions, respectively. The matrices $\mathbf{M}$, $\mathbf{V}$, and $\mathbf{P}$ are the model, view, and projection transforms. Notice that in order to obtain the final point, the result of the matrix transformations has to be divided by its own fourth co-ordinate, the *homogeneous co-ordinate*. This is called the *perspective division* and is a consequence of the fact that the projection transform is not an affine homogeneous transform [30, p. 122]. The perspective division is the step responsible for the perspective effect that results in objects farther away appearing smaller than objects closer to the observer.

2.2.3 Computer animation

It is usually desirable to render not only static scenes, but also dynamic worlds containing movement, rotation, and deformation of models. The way this is achieved

in computer graphics is usually through the stepwise alteration of 3D models, either by changing their associated model transforms, or by changing the model data itself. The models are in most cases altered slightly between each successive rendered frame, so that their movements may be perceived as smooth motion. This practice of presenting a series of still, computer generated images in rapid succession with the goal of giving the appearance of motion is termed *computer animation* [30, p. 615].

The computer generated frames must be presented at a certain frequency in order to properly provide the visual continuity required for the human eye to perceive them as motion. The exact threshold frequency is a topic of much debate, but most studies appear to agree that it lies in the region of 12 to 16 frames per second [30, p. 615][32, p. 24]. If a rendering process is able to synthesise and present a series of images at or above this threshold rate, it is known as a *real-time* rendering process. It is worth noting that the usage of the term real-time in the field of computer graphics differs somewhat from the precise definition of real-time processes found in the field of embedded systems.

Rigid-body animation

This thesis is generally concerned with a certain type of animation termed *rigid-body animation*. This term finds its roots in physics, where a *rigid body* is defined to be a stiff body for which deformation may be disregarded. By directly adopting this definition to the field of computer animation, one ends up with the definition of rigid-body animation—an animated rigid body. In other words, a model animated by rigid-body animation is an animated model which does not support deformation [30, p. 632].

Rigid-body animation can be regarded as a simple form of animation that does not alter the internal data of the model. The case of a polygon model is considered in the following to serve as an example. By treating the polygon model as a rigid body, the model as a whole must be regarded as a stiff, *undeformable* body. This means, in turn, that any animation applied to the model must be applied equally to all the model's internal vertices. And during the animation sequence, the positions of all the vertices in model space must remain unchanged. This does not mean, however, that every vertex will always be translated by the same amount in world co-ordinates. For instance when a model is rotated, each vertex is rotated around the model's origin, and their paths in world space will not be equal. An illustration highlighting this distinction is shown in Figure 2.4.

A consequence of this definition of rigid-body animation is that only certain affine transformations are permitted to be applied to models. It turns out that the only allowed transformations are rotation and translation—both representable by simple affine matrices [30, p. 632]. The simplifications that are possible as a result of this fact will lay the foundation for the proposed solution presented in Chapter 5.

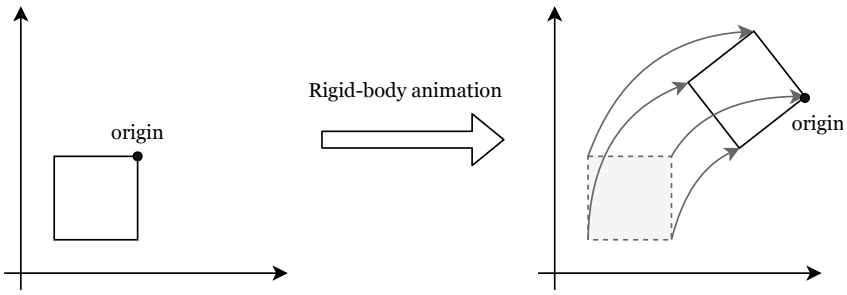


Figure 2.4: Example of rigid-body animation, where the vertices of the square remain unchanged in model space. In world space, however, their paths are different.

2.3 Rendering techniques

The final space encountered in the computer graphics pipeline is the screen space. In this space, the objects to be rendered are projected onto a two-dimensional plane. However, the screen is a discrete grid of values, while the mathematically defined screen space is continuous. This raises the question of how one would go about sampling the continuous screen space in order to render the discrete frame. There are two main methods one may use to this end: rasterisation and ray tracing. In the following sections the two methods will be presented in more detail.

2.3.1 Rasterisation

Rasterisation is a general technique for transforming data specified in vector graphics format into a grid of pixels—a *raster image*. Most graphics pipelines in use today employ some form of rasterisation [30, p. 8]. Shown in Figure 2.5 is the general idea behind the scheme.

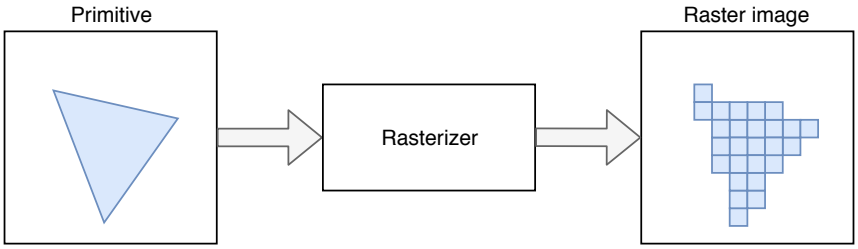


Figure 2.5: Rasterisation used for sampling a primitive.

More specifically, rasterisation concerns itself with taking a set of primitives, for each primitive calculating which set of pixels the primitive is projected onto, and lastly, sampling the primitives to determine which colour the resulting pixel should have. Most implementations of rasterisation use some form of depth sorting in order to determine—in the case of overlapping primitives—which primitive is in front of the other [11, pp. 467–468][29, p. 19][30, p. 27].

The type of models used in rasterisation is, almost exclusively, polygonal models. This is because the models are natively in vector graphics format, which alleviates the need for translating the model into such a format before rendering [30, pp. 176–177].

Scan line rendering

The most popular method of implementing rasterisation is using the technique known as *scan line rendering*. The method works by inverting the problem; instead of working on each polygon, it works on each row of the raster image. By posing the problem this way, the implementation may take advantage of certain properties (such as scan line and edge coherence) in order to reduce the workload [30, p. 42].

2.3.2 Ray tracing

As an alternative to rasterisation, ray tracing is a method of sampling volumetric models, such as voxel models or analytic models. The method of ray tracing is based on modelling the physical properties of light. The idea is to trace the path light would take in reverse, starting from the camera, or eye, into the scene. If the ray hits an object, the colour of this object determines the colour of the pixel the ray passes through on its way [24]. In order to simulate the effects of lighting—such as shadows, reflections, and refractions—secondary rays may be spawned whenever the primary rays hit an object [30, p. 548][29, p. 220]. The basic concept is illustrated in Figure 2.6.

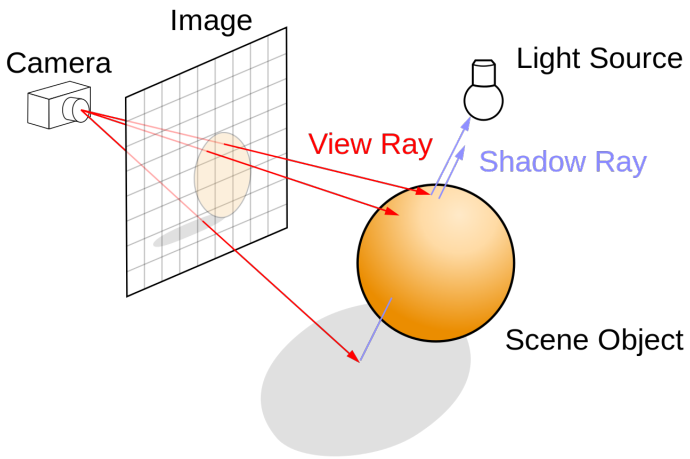


Figure 2.6: Ray tracing process. Image taken from [33].

Since effects of light can be simulated with relative ease, images produced by ray tracing often have a higher degree of realism than those resulting from traditional rasterisation-based methods. However, without heavy optimisations, this improved visual fidelity generally also comes at a higher computational cost. As a result, ray

tracing is widely used in applications which do not require real-time rendering, such as special effects for motion pictures [34].

Recursive ray tracing

Ray tracing as a concept easily lends itself to recursive implementations. One such implementation is termed *recursive ray tracing*. This method—or some derivative of it—is usually central in the generation of the hyper-realistic images one often associates with ray tracing. Examples of such images can be seen in Figure 2.7. The general principles behind recursive ray tracing were pioneered by Whitted [25], and consists of recursively generating new rays whenever a ray is terminated by hitting a surface. For every primary ray hit, one or more secondary rays of the following types may be spawned: shadow rays, reflection rays, or refraction rays. Each of these types of rays are responsible for modelling a single effect of lighting.

Figure 2.7: Examples of recursive ray tracing. The scenes are rendered with shadow, reflection, and refraction effects. Images taken from [35] and [36].

The first type, **shadow rays**, consists of rays that have the purpose of simulating the effects of shadows. The rays are traced from the hit point of the primary ray in the direction towards light sources. If a shadow ray hits an object on its way, the primary hit point is occluded from the light source in question, and hence lies in its shadow.

Secondly, **reflection rays** are, as the name states, rays traced from the hit point on reflective surfaces. Such rays are traced with the direction reflected as calculated by some law of reflection, and allow rendering the mirror effect that appears on reflective objects in the scene.

Finally, **refraction rays** are rays that follow the laws of refraction. Whenever a primary ray hits a (partially or fully) transparent object, a new ray is traced entering into the object. The direction of this ray is often calculated using Snell's law, which is dependent on the refractive indices of the materials it exits and enters [25].

2.4 Space partitioning

In the field of mathematics, *space partitioning* is the study of effectively subdividing regions of space into partitions. As a branch of geometry, space partitioning is most often concerned with Euclidean space. There are several areas of application for the concept—chief among them, perhaps, is computer graphics. In computer graphics, performance is critical; being able to sort or organise objects in a scene by utilising space partitioning, means that certain optimisations are possible. For instance, hidden surface elimination or ray-object intersection can be greatly sped up by maintaining a sorted scene.

Many partitioning schemes are based on some form of *binary space partitioning* (BSP). It is a general style of space partitioning in the form of a tree structure, where each node in the tree may have zero or more children. Every node in the tree can be said to describe a portion of space, but policy of how space is divided among the nodes is dependent on the scheme used [11, p. 436].

A great deal of research has been conducted into the field space partitioning over the years, but some proposed structures are more relevant than others in the context of this project. In this thesis, the octree will be discussed initially, and subsequently, the sparse voxel octree.

2.4.1 Octree

The *octree* is a type of BSP that was first introduced in a 1980 technical paper by Donald Meagher [37]. In it, he describes how binary trees and, especially, quadtrees (two-dimensional binary trees) are established data structures with many areas of applications. He continues by proposing the octree as a data structure which makes use of an N-dimensional binary tree in the representation of N-dimensional objects. In most cases, and especially in this thesis, three-dimensional space is used, so $N = 3$.

Specifically, the octree is a tree-like structure contained within a root node. The root node serves as the *parent node* of 8 *child nodes*. Each of the child nodes can be in one of two states. If the region it covers can be completely described by the child node, the node is a terminal node, or *leaf node*. If the region cannot be completely described by the child node, the region will be further subdivided, and the child node will function as the parent node of 8 new child nodes. The data structure will continue to be subdivided in this fashion until all paths down the tree structure are terminated by leaf nodes [11, pp. 436–437]. An illustration is provided in Figure 2.8. In the illustration, a three-level octree is visualised, showing both the theoretical hierarchy, and the physical layout.

According to Brönnimann and Glisse [38], octrees are theoretically the most efficient space partitioning scheme for three-dimensional space in terms of the number of traversal steps. The octree as a data structure has in itself many applications in computer graphics, such as image processing [37][39], level of detail optimisations [11, p. 439][40][41], and robotics [37][42]. Although the octree in itself is a very efficient format for subdivision of space, this does not automatically make it very

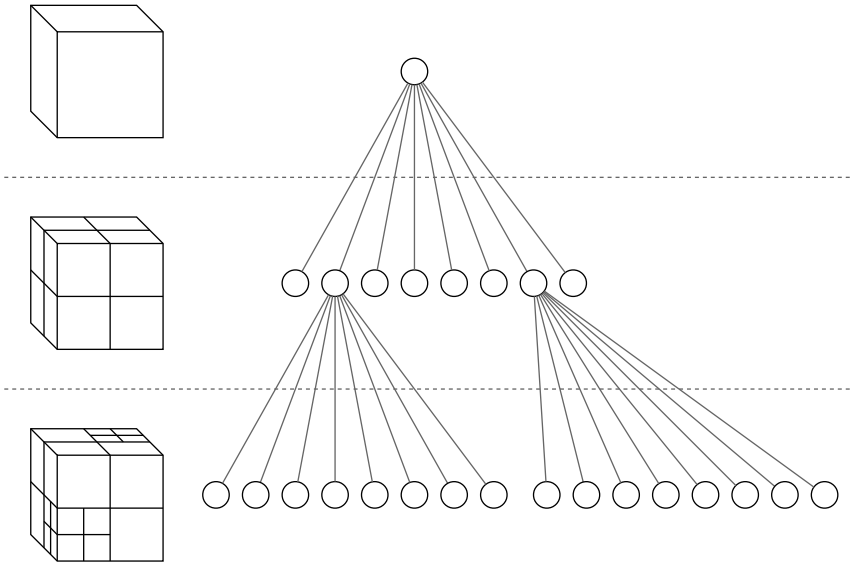


Figure 2.8: An illustration of the hierarchical structure of an octree.

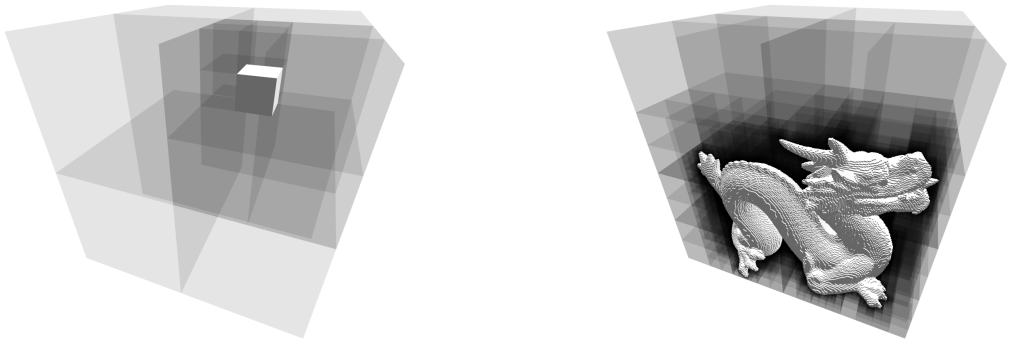
convenient for ray tracing volumetric data. Some minor modifications can be made to the tree structure, however, to make it particularly well-suited for ray tracing.

2.4.2 Sparse voxel octree

A specific flavour of octree developed with applications such as ray tracing in mind is the *sparse voxel octree* (SVO). The scheme is based on the octree, but differs in that instead of using the data structure to subdivide or sort objects, the octree itself directly encodes the volumetric data. By associating certain characteristics with the nodes themselves, the tree structure can be used to natively describe voxel models. As an example, a basic SVO can consist of an octree where each terminal node is categorised as either filled or empty. Shown in Figure 2.9 are two voxel models encoded as sparse voxel octrees.

Sparse voxel octrees can be extended to contain almost any relevant data. The next step from a simple filled/unfilled scheme would be to include normals and colours for lighting calculations. Other attempts at expanding the format include adding data such as contour descriptions [31] and level of detail (LoD) optimisations [44] to further improve visual fidelity.

There exists many established algorithms for traversal of octrees, and most of these methods translate well to traversal sparse voxel octrees with few or no modifications. The algorithms may be divided into two main categories; *top-down* and *bottom-up* approaches. Algorithms belonging to the first category will start at the root node, and move recursively into the tree until a leaf node is reached. If the leaf node does not meet the requirements for algorithm termination (for instance, if it is empty), the algorithm will trace its path back up the tree and enter the next leaf node it encounters. Bottom-up algorithms work by locating the initial leaf node, and traversing the tree by a process called *neighbour-finding* to obtain the next leaf



(a) Single cube (b) Stanford dragon [43].

Figure 2.9: Voxel models encoded in SVOs.

node [45]. The specific algorithms will be presented and discussed in Chapters 3 and 4.

Memory requirements

Compared to a three-dimensional array of voxel values, the SVO data structure can reduce the memory usage of a voxel model by many orders of magnitude. In the vast majority of models, there are large regions of either filled or empty space. By using the property of octrees where related nodes that hold the same data may be collapsed into a single, larger node, the size of SVO models can be drastically reduced compared to explicitly storing the grid of values. In Figure 2.10, the merging of nodes is visualised. The illustration shows a two-dimensional case—a quadtree—instead of an octree, but the principle is analogous to the three-dimensional case.

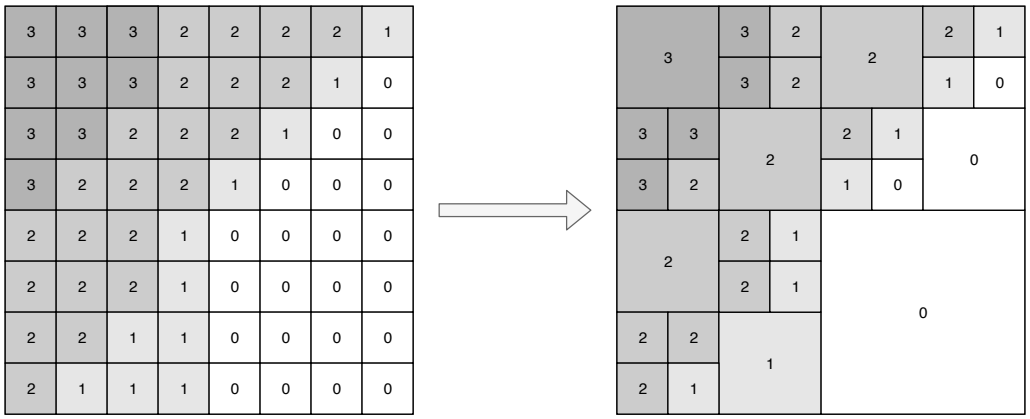


Figure 2.10: Merging neighbouring nodes sharing the same value.

In Table 2.2, the sizes of a selection of models are listed in three distinct formats. Each model has a resolution of 1024 along each dimension, for a total of about 1 billion (2^{30}) data samples. The first format is the raw, uncompressed format in which each voxel is stored as a byte in an array. The second format is based on the raw format, but uses *run-length encoding* (RLE) [46] to reduce the size. The third format is a memory scheme for SVOs based on the results published by Laine and Karras [31]. This format is used throughout the thesis and will consequently be thoroughly described in Chapter 4. The table illustrates the efficiency of SVOs in terms of memory usage. The SVO yields a reduction in size of a factor of 300 and 3 compared to uncompressed and RLE-encoded formats, respectively.

Table 2.2: A selection of models and their file sizes when stored in different formats. All models originate from *Stanford 3D Scanning Repository* [43].

Model	Uncompressed	RLE	SVO
Stanford Dragon	1024 MB	11 MB	2.8 MB
Stanford Bunny	1024 MB	12 MB	3.9 MB
Happy Buddha	1024 MB	11 MB	2.3 MB
Armadillo	1024 MB	11 MB	2.7 MB

Level of detail

Sparse voxel octrees can be extended to natively support *level of detail* (LoD) optimisations. When traversing an SVO, the computational costs increase with the depth of the tree. In some cases, however, the need for spatial resolution is reduced. Consider for instance the case where the tree is at a distance, and the tracing algorithm has reached a depth where one voxel of the tree is smaller than a pixel on the screen. In this setting, the spatial resolution goes wasted, and will only tax the performance without yielding any real gain in image quality. It may even lead to worse quality stemming from effects such as aliasing.

Some solutions employing LoD optimisations exist. A simple method is to store the average colour in each node. In this case, terminal nodes would either be completely filled with a colour, or completely empty. Intermediate nodes could then store the average colour of all their children. An octree traversal algorithm may then stop its descent into the tree structure when a desired level of detail is reached.

A more involved method is presented by Jabłoński and Martyn [44]. In their 2016 paper, they extend the sparse voxel octree format with redundant nodes, termed *helper nodes*. These nodes are employed in the interpolation of the volumetric data. Their method is suited for both rasterisation and ray tracing of SVOs. An illustration of their LoD transition algorithm is shown in Figure 2.11.

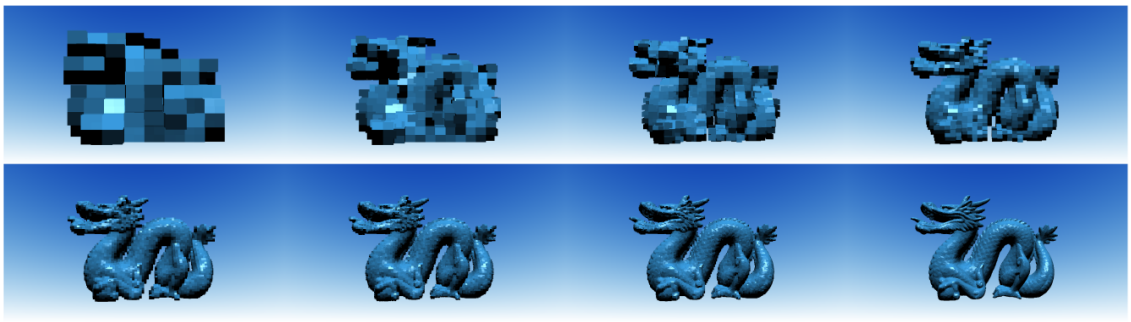


Figure 2.11: LoD optimisations of SVO data. Image taken from [44].

2.5 Rendering and computation APIs

There is a host of application programming interfaces (APIs) available to the developer when starting out with 3D computer graphics. In this section, a few of the software libraries relevant to this thesis will be presented and discussed. The implementation using these libraries is discussed in detail in Chapter 6.

2.5.1 OpenGL

OpenGL is an API for rendering in 2D and 3D. It is a software library that contains functions for communicating with graphics processing hardware, and can be used to ease the process of rendering computer graphics. It provides a hardware-independent interface, which means that the same program can be compiled and run on any configuration, as long as the OpenGL API itself is supported [47].

While there are several options when it comes to rendering APIs, OpenGL is often preferred due to the sheer magnitude of supported hardware and software. Since its release in 1992, it has grown to become the *de facto* standard in cross-platform computer graphics [48]. As a result of its popularity, a plethora of resources exists for learning how to use OpenGL. On the basis of the availability of resources, and the fact that the author has previous experience with the library, OpenGL was chosen as the API for rendering the result of the ray tracing processes in this thesis. Do note that due to the minor role this library plays in this project, this introduction has been kept brief.

2.5.2 Nvidia CUDA

The *Compute Unified Device Architecture* (CUDA) framework is an API developed by Nvidia [49]. It is a C-like language and programming environment that allows software to utilise both the CPU and GPU to run code. The reason one might want to use the graphics processing hardware in calculations is because of the exceptional multithreading capabilities of GPUs. Graphics processing units are a type of *single instruction, multiple data* (SIMD) processors, when classified by Flynn's taxonomy [50][51]. Nvidia themselves classify the model on which CUDA is based as *single*

instruction, multiple thread (SIMT) [51]. Using the GPU to do work in this manner is often called *general-purpose computing on GPU*, or GPGPU for short [47][51][4].

By the conventions introduced by CUDA, the code running on the CPU is the entry point of the software. It is termed the *host* code, and essentially controls the entire program. The host code may spawn thousands of identical, parallel threads that run on the GPU. The code run by these threads is called the *device* code. In order to spawn threads running device code, an entry point function on the device, called the *kernel*, is defined [49]. The kernel function is called from the host code in order to spawn threads on the device. When calling a kernel function from the CPU, some configuration parameters are supplied which describe how many threads should be spawned for the specific kernel, and how these threads should be grouped. The threads are grouped into two-dimensional units called *blocks*, and these blocks are again grouped into a larger two-dimensional unit called the *grid*. This logical structure is shown in Figure 2.12.

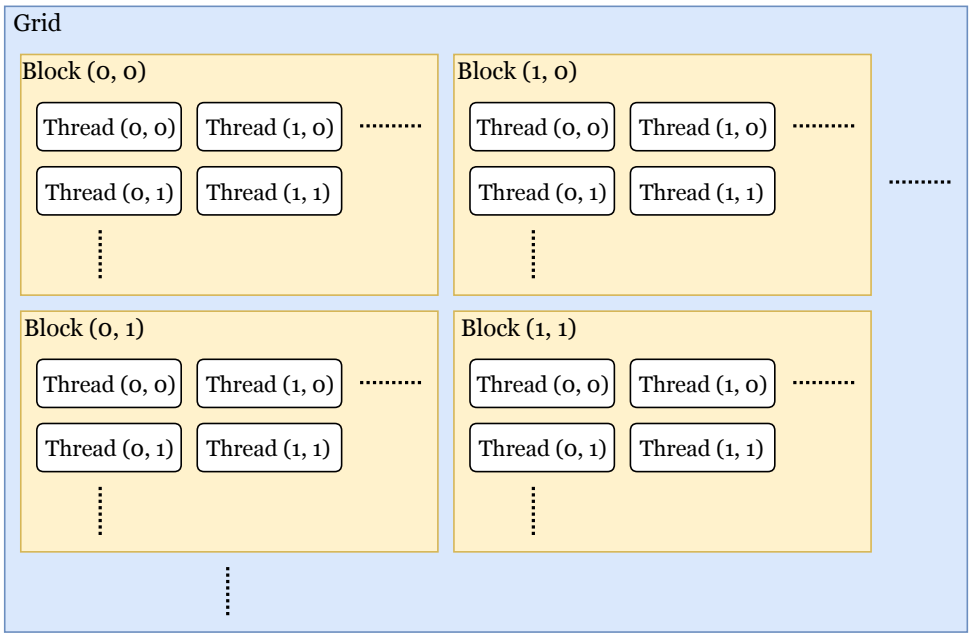


Figure 2.12: The logical structure of CUDA threads.

The size of the blocks and grid are defined when calling the kernel. However, in order to do something useful with these threads spawned on the GPU, some way to differentiate them by uniquely identifying them in the software is needed. To ease this process, the CUDA framework gives each thread a unique index and exposes this index through a set of special variables in the kernel. Using these variables, the position of the current thread in its block, and the position of the block in the larger grid, may be retrieved.

Justification for choice of compute library

There are alternative, non-proprietary compute libraries, such as the open source *OpenCL* platform that could be used instead of CUDA in this project. The reason that CUDA was chosen for this thesis is mainly the availability of resources. Tutorials and help forums appear to be much more accessible compared to similar resources for alternative libraries. In other words, it seems that CUDA is the most widely used compute library in the field, and that the availability of support reflects this.

Another reason for its selection is the raw performance of CUDA versus OpenCL. In this project, the Nvidia GPU *GeForce GTX 680* [52] is employed for accelerating the software. And, as it has been demonstrated by Du et al. [53], the CUDA API yields higher performance than alternatives such as OpenCL when run on Nvidia GPUs.

As a side note, in parallel with the project thesis work, the author was also enrolled in the course *TTK8*, supervised by Geir Mathisen. The purpose of this course is to serve as support work for the thesis. In this course, the author completed a minor project using Nvidia CUDA in order to learn how to use the framework.

Chapter 3

Research context

In this chapter, the research field as a whole will be explored. The specific papers and algorithms upon which this thesis is based is detailed in Chapter 4. After a thorough literature review, few attempts at animation of sparse voxel octrees were found. However, there is a vast body of published papers on the topic of efficient ray tracing and traversal of sparse voxel octrees.

3.1 Algorithms for octree traversal

Since the extent of the research done on the subject is very large, only the most prominent works, and those of the highest relevance to this thesis, will be discussed in this section.

The earliest method for traversal of octrees found in the literature was authored by Glassner [54] in 1984. The paper states that over 95 percent of the total rendering time may be spent on ray-object intersection calculations. Hence, there is a huge potential for performance gain by optimising this process. Glassner then suggests sorting the scene into an octree and presents an algorithm for traversal of such an octree. Another method was introduced by Levoy [55]. In the paper he introduces two methods for enhancing the performance of ray tracing of volumetric data, the first of which employs octrees to encode spatial coherence in the data.

Many subsequent attempts at improving the performance of octree traversal exist. They can generally be grouped into two main categories based on how they solve the traversal problem: bottom-up and top-down schemes. The algorithm by Glassner [54], as well as other, similar schemes [56][57], are instances of bottom-up octree traversal algorithms. The method by Levoy [55], as well as the *HERO algorithm* presented by Agate, Grimsdale, and Lister [58], and a host of other algorithms [59][60][61][62][63] provide examples of a top-down parametric traversal algorithms. From the number of papers alone, it appears that top-down traversal algorithms are most popular in the field.

An efficient algorithm for octree traversal was presented by Revelles, Ureña, and Lastra [45] in their 2000 paper. The article introduces a top-down parametric method that is very well documented. After this work was published, there seems

to be few new algorithms developed that contest its speed and simplicity. In 2006, a paper published by Knoll et al. [63] describes an algorithm based on the work of Gargantini and Atkinson [61], however this algorithm is not as well-documented as [45], and seems more complicated while not revealing any tangible performance increase. Indeed, perhaps the opposite, as the algorithm is recursive, and therefore does not translate readily for implementation on a GPU without modification. The algorithm by Revelles, Ureña, and Lastra [45] is also recursive, but due to its simplicity is bound to translate well into an iterative method. In fact, this is already demonstrated by Wilhelmsen [4].

In conclusion, the algorithm elected for traversal of sparse voxel octrees is [45]. Because the algorithm is simple and fast, and also improves upon the performance of earlier algorithms, this algorithm will serve as the starting point of this thesis. The algorithm will be presented in detail in Chapter 4.

3.2 Parallelising the workload

The algorithms presented in the previous section are sequential algorithms, and do not deliberately exploit the fact that ray tracing is highly parallelisable by nature. There are several papers that discuss advantages of using the parallel computing capabilities of GPUs to distribute the workload. A few of the most notable are presented in the following.

A paper that does not describe a method of ray tracing, but is still noteworthy is Gobbetti and Marton [64]. They discuss the rendering of very large surface models using *out-of-core* data management, meaning that the approach supports data sets too large to fit in working memory. In their paper, they use hardware acceleration in the form of a GPU to parallelise the workload of rendering the data sets, while implicitly supporting different levels of detail. A similar method could perhaps be utilised in ray tracing hardware to support very large, highly detailed models.

In their 2009 paper, Crassin et al. [65] propose a new approach for rendering large volumetric data sets by ray tracing on a GPU. Their impressive result is that the system achieves interactive to real-time rendering performance of several billion voxels. The method takes care to avoid using a stack—and therefore no recursion—in order to increase GPU optimisation, and utilises mipmapping as a LoD-technique in order to hide visual noise. The algorithm supports on-the-fly loading of chunks of data from CPU memory to GPU memory whenever the ray tracer encounters missing data, which means that the size of the models is not limited by GPU memory.

The most relevant work for this thesis is the article by Laine and Karras [31]. They use the GPGPU capabilities provided by Nvidia CUDA to trace SVOs in parallel. Also introduced through their work is a compact SVO memory structure that will form the basis of the SVO specification used in this thesis. The memory structure presented in this thesis can be viewed as a simplification of the structure presented in their paper. The data structure will be detailed in Chapter 4.

3.3 Methods for animation of SVOs

Only one significant attempt at animation of sparse voxel octrees was found in the literature. The bachelor's thesis by Bautembach [5] outlines a general method of animating sparse voxel octrees during the rendering process. In his paper, he describes a method for animation of SVOs based on the idea that each leaf node of the tree is an individual *atom* that may be animated. The method presented can be regarded as a *bottom-up* solution to the problem of animation. Bautembach's proposition contrasts what will be presented here in that the solution introduced in this thesis may be viewed as a *top-down* approach.

Note that Bautembach presents a general method for animation of sparse voxel octrees, but that this method is not suitable for ray tracing. As is explicitly stated by Bautembach himself, his method destroys the hierarchical structure of the SVO to be rendered. Consequently, most ray tracing algorithms will no longer work, since the method in turn prohibits efficient intersection tests. In his work, he therefore resorts to rasterisation in order to render the animated sparse voxel octrees.

3.4 Other works of significance

Another thesis of interest is a master's thesis written by Wilhelmsen [4]. In it he implements a hardware ray tracer for sparse voxel octree data. The thesis is very well written, and many of the choices made are relevant to this project. For instance, his choice of algorithms closely match the algorithms selected for this thesis, and the justifications for the choices are similar, since both this thesis and his work are attempts at tracing SVOs in a manner suited for hardware acceleration.

Chapter 4

Established algorithms chosen as foundation

In this chapter a set of established algorithms and schemes that are to be used in the thesis, will be presented. The workings of the algorithms will be explained in detail, as well as the reasoning behind their selection.

4.1 An efficient parametric algorithm for octree traversal

The chosen algorithm for traversal of octrees, and in the case of this thesis, sparse voxel octrees, is the parametric algorithm proposed by Revelles, Ureña, and Lastra [45] in their 2000 paper. After the literature review, this algorithm seemed the most efficient and well documented. Many newer papers—such as works on global illumination [66], virtual X-ray imaging [67], and the very relevant works of Laine and Karras [31] and Wilhelmsen [4]—employ this algorithm in some shape or form in their projects. This lends credence to the claim that the algorithm is still both relevant and competitive.

The algorithm is a recursive, top-down algorithm with neighbour-finding. As will be further discussed in Chapter 6, recursive algorithms are usually problematic in a GPGPU context, however this algorithm can without much effort be modified to be iterative instead of recursive. This is demonstrated by Wilhelmsen [4] in his hardware approach.

In this section, the algorithm will be reviewed like it is presented in [45], albeit with some minor modifications to the naming of parameters and variables. In addition, the algorithm for the three-dimensional case had to be converted to use a right-handed co-ordinate system.

4.1.1 Simplified algorithm for the 2D case

To present the algorithm in as straightforward a manner as possible, a simplified version for traversal of two-dimensional quadtrees is considered. The algorithm is

parametric, which in this context means that all computations are centred around a parameter t, such that t is a positive number describing a point $\mathbf{p}$ along a ray R as shown in Equation (4.1). The ray R is defined by its origin $\mathbf{r}_o$ and direction $\mathbf{r}_d$.

$$\mathbf{p}(t) = R_t(\mathbf{r}_o, \mathbf{r}_d) = \mathbf{r}_o + t \cdot \mathbf{r}_d \ , \quad t \geq 0 \tag{4.1}$$

For now, all rays are assumed to have directions with strictly positive components. The algorithm will at a later stage be extended to allow rays with arbitrary directions.

First phase: node boundaries

The first concept to be introduced is the characterisation of node boundaries. In the two-dimensional case, each node has four boundaries. These boundaries represent the four edges of the node, and are given the names x_0, x_1, y_0, and y_1. Since the algorithm is parametric, the boundary of a node may also be defined by the value t at which the ray crosses the boundary. Therefore, the following values of t are declared: t_{x0}, t_{x1}, t_{y0}, and t_{y1}. The values and how they relate graphically to the ray R can be seen in Figure 4.1

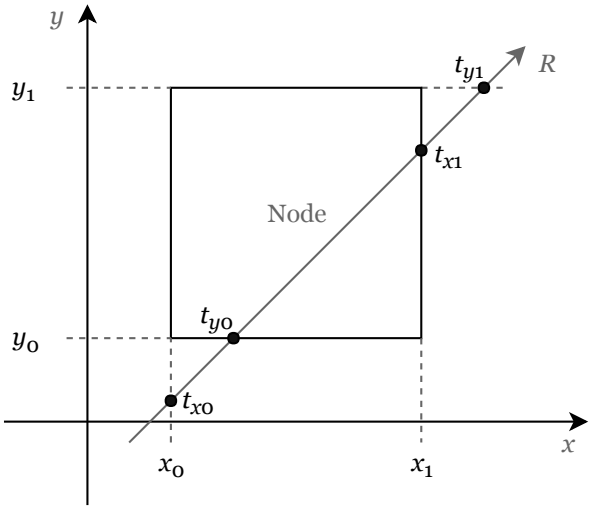


Figure 4.1: The node boundaries.

The first phase of the algorithm therefore consists of calculating the values of t at which the ray crosses each of the boundaries. A simple expression for each boundary is derived in [45] and given in Equation (4.2).

$$\begin{aligned} t_{x0} &= (x_0 - r_{ox})/r_{dx} \quad \wedge \\ t_{x1} &= (x_1 - r_{ox})/r_{dx} \quad \wedge \\ t_{y0} &= (y_0 - r_{oy})/r_{dy} \quad \wedge \\ t_{y1} &= (y_1 - r_{oy})/r_{dy} \end{aligned} \tag{4.2}$$

Second phase: intersection test

Once these values of t are calculated, the second phase of the algorithm begins. The goal of this phase is to determine if the ray intersects the node at all. Since the ray direction is assumed positive, it is evident that $t_{x0} < t_{x1}$ and $t_{y0} < t_{y1}$. This means that if the ray intersects the node, the boundaries t_{x0} and t_{y0} will be the boundaries through which the ray enters the node, and conversely, t_{x1} and t_{y1} are the boundaries through which the ray leaves the node.

It can be shown that if the maximum t-value of all the enter boundaries is less than the minimum t-value of all the exit boundaries, the ray intersects the node. Therefore, two more values of t are calculated as shown in Equation (4.3).

$$\begin{aligned} t_{\min} &= \max(t_{x0}, t_{y0}) \quad \wedge \\ t_{\max} &= \min(t_{x1}, t_{y1}) \end{aligned} \tag{4.3}$$

Hence, the intersection test consists of checking the expression $t_{\min} < t_{\max}$. In addition, $t_{\min}$ and $t_{\max}$ are the t-values at which the ray enters and exits the node itself. The proof for these statements can be found in [45].

Third phase: recursion

If the intersection test passes, the algorithm moves onto the third phase. In this phase, the algorithm will recurse into the children of the node. First, two additional t-values are defined. These are the values of t at which the ray crosses the centre—or middle—of the node. The values are defined mathematically as shown in Equation (4.4).

$$\begin{aligned} t_{xm} &= (t_{x0} + t_{x1})/2 \quad \wedge \\ t_{ym} &= (t_{y0} + t_{y1})/2 \end{aligned} \tag{4.4}$$

Selecting the child nodes that are intersected by the ray is a matter of systematically testing the different parameters introduced so far. Pseudocode for the process is shown in Figure 4.2.

```
1.  if t_min < t_max then
2.      if t_ym < t_xm then the ray crosses q_2 and
3.          if t_x0 < t_ym then the ray crosses q_0
4.          if t_xm < t_y1 then the ray crosses q_3
5.      else if t_xm < t_ym then the ray crosses q_1 and
6.          if t_y0 < t_xm then the ray crosses q_0
7.          if t_ym < t_x1 then the ray crosses q_3
```

Figure 4.2: Selecting the correct child node.

When the intersected child nodes are determined, the algorithm will process them in turn, ordered by increasing indices. For each child node to process, the algorithm will recurse, but skip directly to the third and final phase. Further, the values of t will differ from those of the parent node. The t-values to use for each of the child nodes q_0, q_1, q_2, and q_3, can be defined simply as some selection of the t-values of the parent. The rules can be seen in Table 4.1.

Table 4.1: Which parameters to use when recursing into child node q_i.

Child node	New t_{x0}	New t_{y0}	New t_{x1}	New t_{y1}
q_0	t_{x0}	t_{y0}	t_{xm}	t_{ym}
q_1	t_{xm}	t_{y0}	t_{x1}	t_{ym}
q_2	t_{x0}	t_{ym}	t_{xm}	t_{y1}
q_3	t_{xm}	t_{ym}	t_{x1}	t_{y1}

Illustrations of how the ray attributes affect the different parameters introduced are shown in Figures 4.3 and 4.4. The figures may also serve to reinforce the understanding of the process of selecting child nodes.

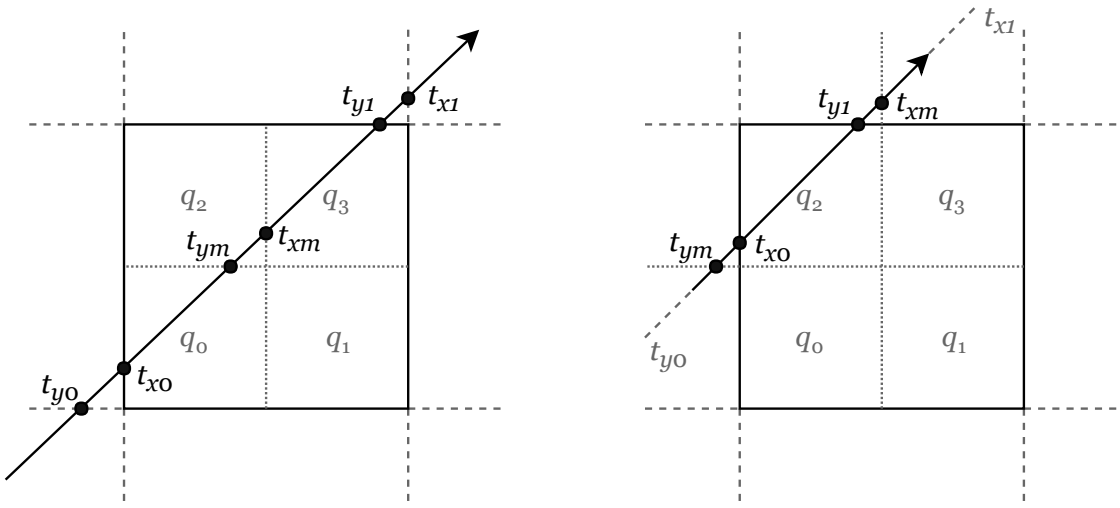


Figure 4.3: Sub-nodes crossed when $t_{x0} > t_{y0}$.

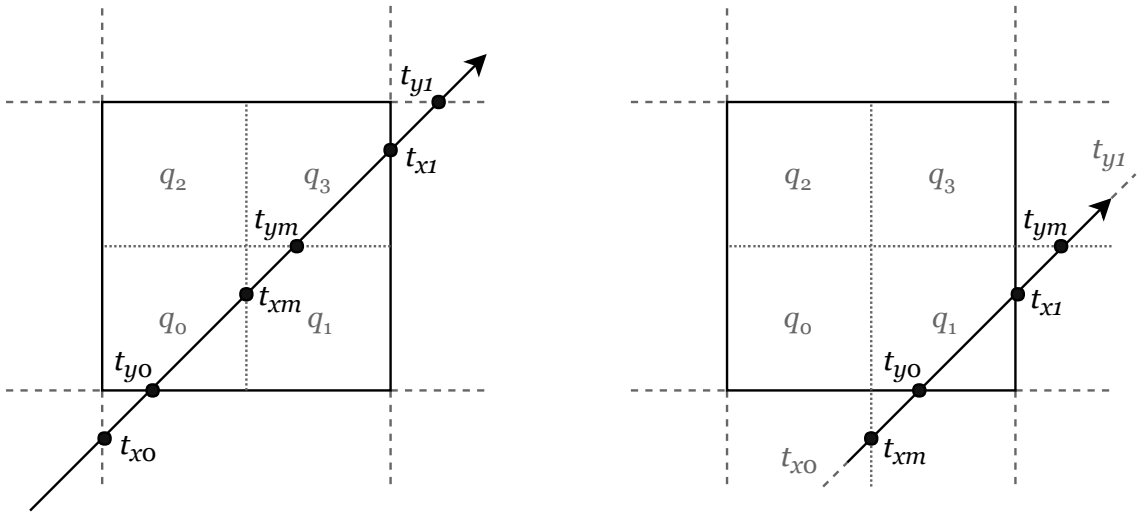


Figure 4.4: Sub-nodes crossed when $t_{y0} > t_{x0}$.

4.1.2 Extending the algorithm to octrees

In order to extend the algorithm to traverse octrees and not quadtrees, a third co-ordinate is introduced where needed. The parameters t_{z0}, t_{z1}, and t_{zm} are defined in a similar manner to their x and y counterparts in Equations (4.2) and (4.4). There is also a need to redefine $t_{\min}$ and $t_{\max}$. These must now be defined as shown in Equation (4.5).

$$\begin{aligned} t_{\min} &= \max(t_{x0}, t_{y0}, t_{z0}) \quad \wedge \\ t_{\max} &= \min(t_{x1}, t_{y1}, t_{z1}) \end{aligned} \tag{4.5}$$

The check that determines whether the ray intersects the node or not is unchanged from the two-dimensional case; a simple test of $t_{\min} < t_{\max}$ is evaluated. This means that—apart from introducing the third co-ordinate—the first and second phase of the algorithm are unchanged. The third phase, however, will differ somewhat.

Obtaining the initial child node

In the event of a successful intersection test, the process of selecting the first child node crossed by the ray becomes more involved compared to the quadtree case described by pseudocode in Figure 4.2. The first step is to obtain the entry boundary of the octree node. This boundary is a plane in three-dimensional space, and is determined by retrieving the maximum value among t_{x0}, t_{y0}, and t_{z0}. Next, a set of tests are performed. These tests have the potential to assert a bit in a variable which is subsequently used to index the child node. The whole process of determining the index of the child node is described in Table 4.2.

Table 4.2: Determining index of initial child node.

Maximum value	Entry boundary	Conditions to examine	Index bit affected
t_{x0}	YZ-plane	$t_{ym} < t_{x0}$	1
		$t_{zm} < t_{x0}$	2
t_{y0}	XZ-plane	$t_{xm} < t_{y0}$	0
		$t_{zm} < t_{y0}$	2
t_{z0}	XY-plane	$t_{xm} < t_{z0}$	0
		$t_{ym} < t_{z0}$	1

Obtaining the next child node

After traversing into the initial child node, there needs to be some functionality for the obtaining of the next sibling node. For instance, if the ray crosses into the first child node of the root node, but does not terminate within this node—i.e. it hits nothing—the algorithm needs to continue the traversal by entering the next direct child node of the root node intersected by the ray.

The algorithm implements this functionality by defining a simple state machine with a set of transitions. Each child node of the current parent node is a state, and the traversal through the set of child nodes is described by transitions. These states, as well as the allowed transitions between them, are shown in Figure 4.5. In the illustration, the index of the states correspond to the index calculated in the previous step.

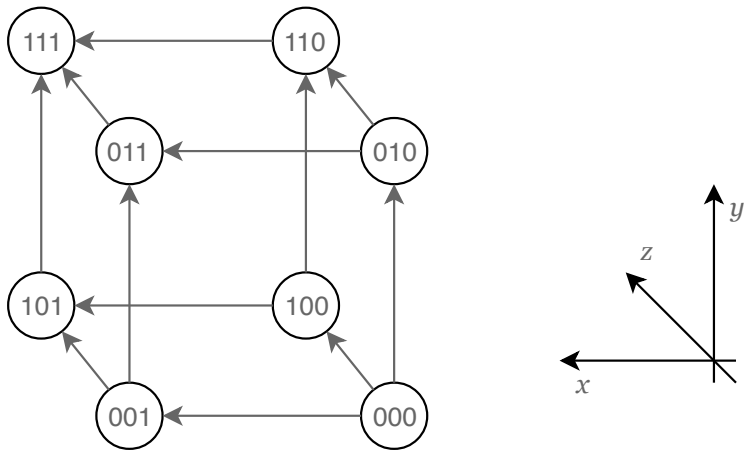


Figure 4.5: The traversal state machine.

The rules for determining the next node to visit can be summarised as shown in Table 4.3. In the table, the exit plane is determined in the same manner as the

entry plane (in Table 4.2), apart from using the exit boundaries of the current child instead. The transition to the next state in the state machine is then established based on the exit boundary and current state. If the END state is reached, all relevant child nodes of the current parent has been traversed.

Table 4.3: Determining next child node.

Current child node	If t_{x1} is max, exit-plane YZ	If t_{y1} is max, exit-plane XZ	If t_{z1} is max, exit-plane XY
000	001	010	100
001	END	011	101
010	011	END	110
011	END	END	111
100	101	110	END
101	END	111	END
110	111	END	END
111	END	END	END

4.1.3 Supporting arbitrary ray directions

The algorithm presented so far requires that the components of the ray direction be strictly positive. To allow arbitrary ray directions—i.e. positive, negative, or zero components—some modifications must be made.

Allowing zero-valued direction components

Since the calculation of the entry and exit boundaries in Equation (4.2) includes an expression where each parameter is divided by the components of the ray direction, issues arise whenever the ray direction includes zero-valued components. There are two main ways to mitigate this problem.

The first solution would be to include a special case that, whenever the direction is used in some manner, checks if one or more components are zero. Further, the algorithm must be modified to handle this special case. This means that all the proposed rules for selecting first and next nodes must include special checks, and would in turn lead to the algorithm becoming much more complex.

The solution proposed in [45] is to allow infinite values, and perform the check by simply setting the parameters to infinity if the direction component is zero, then including a couple of checks when calculating the t_m-values. This solves the problem mathematically, but may still lead to issues in implementation, mostly because infinities can be hard to model in software. A slight modification to the proposed solution which translates well into software is discussed in Chapter 6.

Allowing negative direction components

The algorithm currently only supports non-negative directions. This is ingrained in the rules for choosing the first and subsequent child nodes. Fortunately, Revelles, Ureña, and Lastra [45] have an elegant solution to the problem. A few modifications are made during the initialisation phase of the algorithm, leaving the rest of the algorithm unchanged.

Firstly, if a given component of the direction is negative, the direction and origin of the ray are flipped around the centre of the root node for this component. This is formulated mathematically in the following fashion: for every negative component i of the ray direction, the ray is modified as shown in Equation (4.6), where c_i is component i of the centre of the root node.

$$\begin{aligned} r_{di} &= -r_{di} \\ r_{oi} &= c_i - r_{oi} \end{aligned} \tag{4.6}$$

Secondly, the order in which child nodes are traversed must be modified to account for the modified ray direction. A new coefficient a is defined such that after the next node index i has been calculated, i is modified as shown in Equation (4.7).

$$\begin{aligned} i' &= a \oplus i \\ a &= 4s_z + 2s_y + s_x \end{aligned} \tag{4.7}$$

The operator $\oplus$ is a bitwise XOR, and the values s_i are set to 1 if the original ray direction is negative for component i, and 0 otherwise.

4.2 Efficient sparse voxel octrees

The sparse voxel octree data structure itself also deserves to be a topic of discussion. The chosen data structure for this thesis is a simplified version of the scheme presented by Laine and Karras [31] in 2011. The focus of said paper was to devise an efficient data structure for storing of voxel data. The resulting scheme was also designed to minimise memory bandwidth requirements and therefore has a modest memory footprint.

In this thesis, the data structure will be presented with certain simplifications. These simplifications will be justified in the following, and stem from the fact that some specific features of the original data structure are unneeded, and would only increase complexity without yielding any benefits. Many of the simplifications draw inspiration from the implementation found in the master's thesis by Wilhelmsen [4].

4.2.1 Scheme overview

The data structure described by the scheme is in essence a large table of node descriptors. Each entry in the table corresponds to a specific node in the tree, and stores information about its children. This means that leaf nodes do not have their

own entry, as all information is stored in their parent. In order to support large models while reducing memory bandwidth requirements, the data structure is split into *blocks* of contiguous memory. Within a single such block, all memory references are relative.

Node descriptor

The basic node descriptor outlined in the paper is 64 bits wide, consisting of a 15-bit pointer field, a single-bit field describing the nature of the pointer field, two eight-bit fields that hold the metadata about the children of the current node, and 32 bits of contour information. However, the contour information stored in the descriptor is not of interest in this thesis, so a simplified version omitting this data is used. Hence, the descriptor employed in this thesis is 32-bit wide, its layout on the form shown in Figure 4.6.

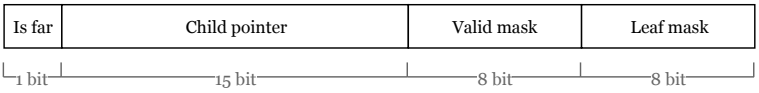


Figure 4.6: A single node entry.

Each of the fields shown in Figure 4.6 serves a specific purpose. Starting with the right-most fields: the *valid mask* and *leaf mask* are bit masks that describe the children of the current node. The masks each have eight entries (bits) that describe each of the eight child slots by the rules stated in Table 4.4. Next, the 15-bit field—the *child pointer*—generally stores a pointer to the entry of the first child of the current node. The rest of the children are stored sequentially after the first child. However, if the single-bit *is far* field is asserted, the child pointer is an indirect pointer to a *far pointer*. This means that the child pointer field will, instead of pointing directly to the first child, point to a 32-bit pointer entry that holds a relative pointer to the child entry. The far pointer is utilised whenever the child pointer field is too small to hold the pointer, and will split the tree data structure into separate, contiguous blocks of memory.

Table 4.4: Bit mask interpretation for child slots.

Valid mask	Leaf mask	Interpretation
0	0	The child slot is empty.
1	0	The child slot is not a leaf and has data.
1	1	The child slot is a leaf, and is filled.
0	1	Invalid combination.

Page headers and info sections

The scheme presented by Laine and Karras [31] also includes a set of special descriptors that will not be utilised in this thesis. As a simplification of the voxel data structure, they will be omitted from the implementation. This decision draws inspiration from Wilhelmsen [4], and generally results in a simpler layout as well as a smaller memory footprint. However, since they are part of the original scheme, they will be presented in this section.

The first omitted descriptor is the *info section*. This type of entry contains a pointer to the first node descriptor, and describes the information available in the octree structure. In other words, it contains metadata about the child descriptors, and states which features are in use; for instance if contour information, colours, or normals are employed. Since none of these features are used, the info section is not implemented at all. The second omitted descriptor, the *page header*, is spread among the child descriptors at a set spacing. These descriptors contain a relative pointer to the current memory block's info section. Since the info section itself is not implemented, it stands to reason that the page header is unneeded.

Example voxel data structure

An example voxel data structure is shown in Figure 4.7. In the hierarchical illustration, grey nodes are empty, blue nodes are filled, and nodes highlighted yellow are non-terminal, meaning that they have children with data. In Figure 4.8, the same entry table is rendered as an SVO.

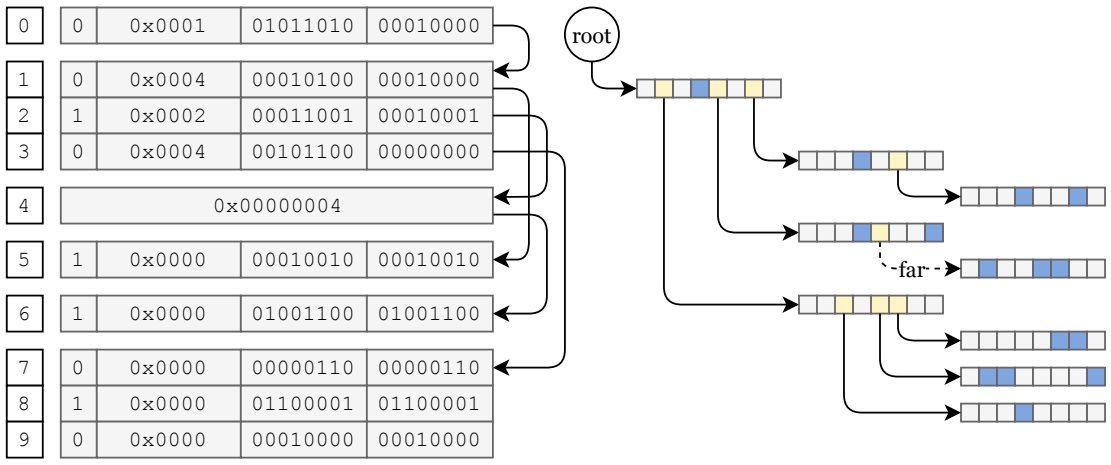


Figure 4.7: An example SVO. Both the structure in memory and hierarchical layout are shown.

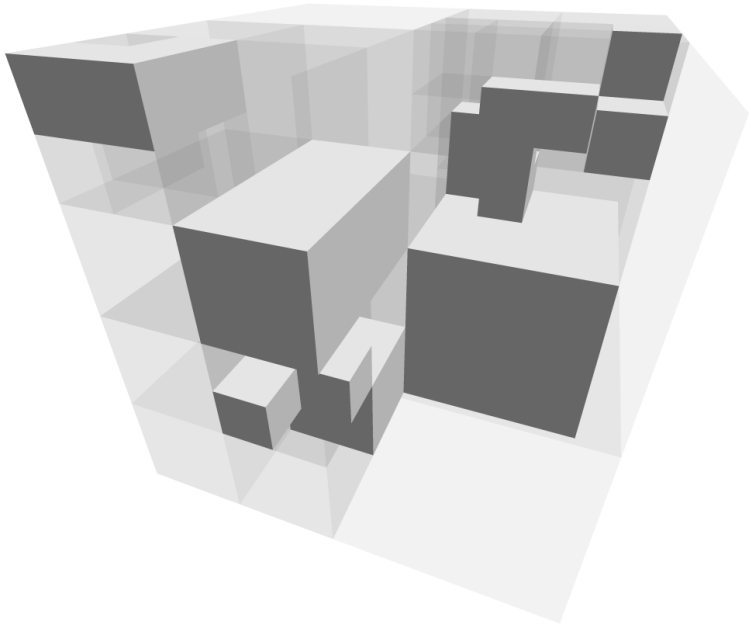


Figure 4.8: The octree specified in Figure 4.7 rendered.

Chapter 5

The proposed method for SVO animation

Sparse voxel octrees are static data structures [31]. Compared to polygonal models, which are inherently easy to manipulate, octrees often need to be extensively modified, perhaps even restructured from the ground up, to respond to even minor changes. As an example, animation of a polygonal model may entail nothing more than translating a single vertex of the model by some distance. This operation is computationally cheap, and will result in a deformed, and hence animated, model. As for a sparse voxel octree, in order to properly animate the volumetric data, one would need to traverse the tree and process every voxel that should be altered as a result of the animation. In addition, the octree structure will in most cases need to be rebuilt as a result of the animation—some leaf nodes might be merged and substituted by a larger leaf node, while other leaf nodes may have to be split into smaller nodes. Compared to polygonal animation, this operation is computationally expensive, and not suited for real-time applications.

The method of SVO animation discussed above results in a scheme where each frame of a given animation sequence in essence produces a distinct sparse voxel octree. This naïve method may be described as the bottom line, or *basis method*. In the following sections, the basis method is referred to in order to establish a comparative foundation for a new method.

5.1 Objectives

The purpose of this thesis is to formulate an efficient method for animation of sparse voxel octrees, that may be more relevant in a real-time setting than the basis method described above. The problem statement requires the solution to be efficient, but leaves open for interpretation the metrics by which the efficiency should be measured. In order to properly measure the efficiency—and thus validity—of a proposed method, three goals, or objectives, are formulated and discussed in the following.

Execution time

A proposed solution should yield an improvement upon the basis method of animation in terms of execution time. This improvement should be to such a degree that the solution may be suitable for real-time applications.

Memory requirements

Compared to the basis method, a solution should not require any significant increase in memory usage. It should neither present a considerable increase in memory bandwidth requirements.

Hardware suitability

In order to provide any discernable benefits for future work, a solution must be applicable in a hardware—or low-level software—setting. This means that special considerations should be made in the design phase to ensure that a solution is suited for implementation in hardware.

5.2 The method

The proposed method is now to be detailed. It takes advantage of the fact that a rigid-body animation may be modelled as a system of rigid bodies transformed relative to each other. This system of rigid bodies is essentially a set of static models wherein each model has an associated transform. The main idea is that the process of animation then consists of altering the transform associated with each model, and leaving the model data itself unchanged. For a polygonal model, this approach is common. The transform is applied to each and every vertex contained in the model during the rendering phase. The next section will discuss the possibility for adapting this method for use in ray tracing of voxel models.

5.2.1 Adapting the method for ray tracing

As touched upon, the method discussed above is an established technique when animating polygonal models in a rasterisation setting. Nonetheless, the approach may be applicable in a ray tracing context as well. In ray tracing of sparse voxel octrees, a sparse voxel octree can be regarded as a self-contained volumetric model. As such, a rigid-body animation may be formulated as a set of independent SVOs, where each SVO is a static, rigid body model with a corresponding transform. The process of animation is then reduced to simply modifying these transforms in a timely manner. The internal data of each SVO may remain unmodified for the duration of the animation.

This description raises the question of how to utilise the transform in the process of animating the volumetric data. It is not feasible to apply the transformation to every data point contained in the model as one would do when animating polygonal

models. The solution presented here is to invert the problem. Instead of transforming the model data, one may perform an inverse transformation on the ray. In other words, each ray in the ray tracing process may be transformed from world space to the local co-ordinate system of each of the animated SVOs that are to be traced.

Shown in Figure 5.1 is the procedure of transforming rays as a part of the animation process. When a ray enters the boundary of a transformed SVO, the ray itself is transformed inversely in order to facilitate the transformation of the model. To reiterate, in place of transforming the SVO data itself—which may involve transforming billions of data points—the transformation is performed on the ray. In practice, the ray transformation is performed by pre-multiplying the ray origin and ray direction with a set of matrices. This matrix multiplication is a step that must be done in any case when the ray is generated, meaning that the transformation of the ray is computationally cheap compared to the ray tracing algorithm itself. In Figure 5.1, the ray transformation stage is illustrated. SVO 1 represents an SVO with identity transformation, leaving the ray unchanged, while SVO 2 changes the ray origin and direction as a result of its orientation.

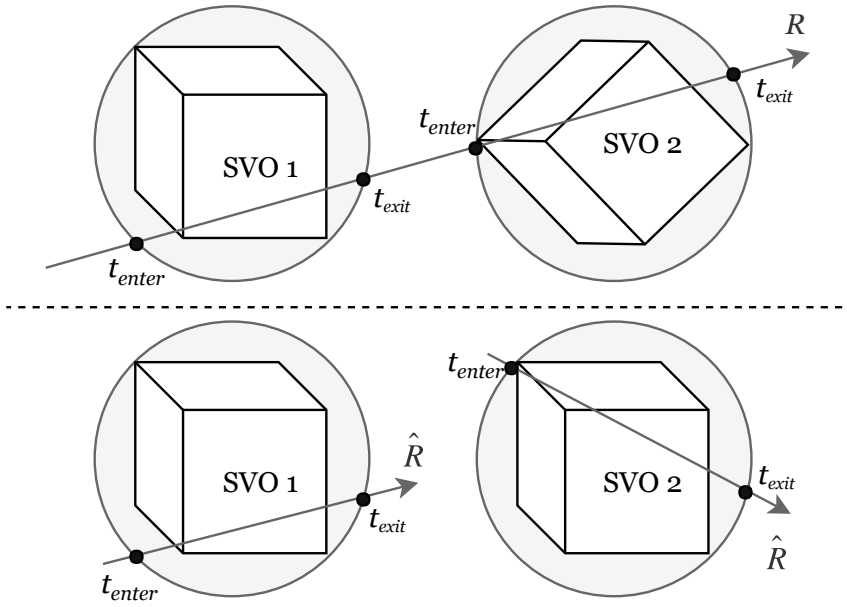


Figure 5.1: Demonstrating the ray transformation process.

5.2.2 Mathematical formulation

In order to implement the solution in software or hardware, a mathematical formulation for the ray transformation is desired. Given the ray definition shown in Equation (5.1), a transformation T from the ray R in world co-ordinates to the ray $\hat{R}$ in local co-ordinates to the SVO must be derived. The desired transformation

should work as shown in Equation (5.2).

$$R_t(\mathbf{r}_o, \mathbf{r}_d) = \mathbf{r}_o + t \cdot \mathbf{r}_d \ , \quad t \geq 0 \tag{5.1}$$

$$\hat{R}_t(\hat{\mathbf{r}}_o, \hat{\mathbf{r}}_d) = T\left[R_t(\mathbf{r}_o, \mathbf{r}_d)\right] \tag{5.2}$$

A graphical description of the desired transformation is shown in Figure 5.2. Formulating the transformation mathematically is, at its core, a matter of deriving two matrices with which to multiply the constituent vectors $\mathbf{r}_o$ and $\mathbf{r}_d$. These vectors represent the ray origin and direction, respectively. By initially only allowing the SVO to be rotated and translated, the linear algebra formulation is straightforward. Given the rotation matrix $\mathbf{M}_R$ and the translation matrix $\mathbf{M}_T$ of the SVO, the transformation function T can be formulated as described in the following. Since it makes no sense translating a directional vector, it should be self-evident that the ray direction only is influenced by the rotation of the SVO. The ray origin, however, is affected by both rotation and translation.

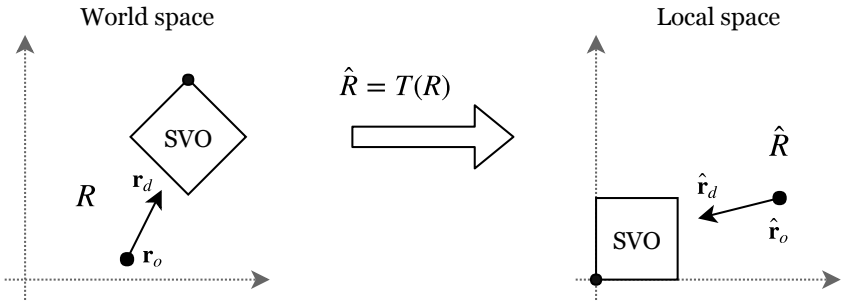


Figure 5.2: The co-ordinate system transformation of a ray in world space to local space.

The resulting mathematical definition of T is shown in Equation (5.3). The ray direction is determined by simply pre-multiplying it with the inverse rotation of the SVO. Transforming the ray origin is a bit more involved, but may be regarded as a two-step process. Firstly, the ray origin is translated so that the origin of its co-ordinate system is at the origin of the octree. Secondly, the vector is rotated around the SVO origin by the same inverse rotation as employed for the ray direction. The mathematical formulation is shown in Equation (5.3).

$$\begin{aligned} T : R_t(\mathbf{r}_o, \mathbf{r}_d) &\mapsto \hat{R}_t(\hat{\mathbf{r}}_o, \hat{\mathbf{r}}_d) \\ \text{such that } &\begin{cases} \hat{\mathbf{r}}_d = \mathbf{M}_R^{-1}\mathbf{r}_d \\ \hat{\mathbf{r}}_o = \mathbf{M}_R^{-1}\mathbf{M}_T^{-1}\mathbf{r}_o \end{cases} \end{aligned} \tag{5.3}$$

5.2.3 Extending the method to allow anisotropic scaling

The transformation so far only accounts for SVO models with transforms consisting of translation and rotation. While these two affine transforms are the only ones

strictly required to provide the functionality of rigid-body animation, there is a third transform that one often needs in animation. In order to fully facilitate animation, there should be a way to animate the size of models as well, by enlarging or shrinking the models along some or all principal axes. In other words, there should be support for non-uniform, or anisotropic, scaling of SVOs. The most convenient way of supporting scaling in the scheme presented above is to implement the support at the traversal stage of the ray tracing process. Luckily, most SVO traversal algorithms, and especially the one this thesis is based upon, already allow the tuning of octree dimensions. By employing this directly in the animation process, the method presented above may remain simple, and only take the rotation and translation into account.

For instance, in the traversal algorithm presented in Section 4.1, one simply has to input the dimensions of the octree as a set of parameters x_0, x_1, y_0, y_1, z_0, and z_1. Do note, however, that any calculation involving the scale of SVOs—such as determining the bounding sphere—also need to be updated to take the scale of the octree into account.

5.3 Measures for improving efficiency

The method described above is an improvement upon the basis method in many ways. Yet, some measures may be taken in order to further improve the efficiency of the method.

5.3.1 Ray-sphere intersection tests

As a result of the sheer number of intersection tests, the most computationally heavy stage of the ray tracing process is the traversal of the SVO [54]. Therefore it is desirable to only traverse SVOs that can lead to a ray tracing hit. For instance, in a situation where one of the SVOs is located some distance away from the origin of the current ray, and the direction of the ray is pointed in the opposite direction of the octree, the SVO can safely be excluded from the process, as the ray will never hit it.

The process of determining which octrees that will never be hit by a given ray may be implemented in a multitude of ways. One method that will be looked further into in this section is to perform an intersection test between the ray and the bounding sphere of the SVO. This intersection test is very fast compared to traversal of the entire SVO, and will in many cases lead to the exclusion of octrees that will never be hit by a given ray. In Figure 5.3 the principle is illustrated. A scene of animated SVOs is shown, where bounding spheres are utilised to determine which octree models that will be missed by the ray. In this case, only SVO 2 will be traversed, as the bounding sphere of the other three octrees in the scene do not intersect the ray.

The rationale behind choosing the bounding sphere to represent the bounds of an octree model, instead of the bounding box—which at first glance may seem

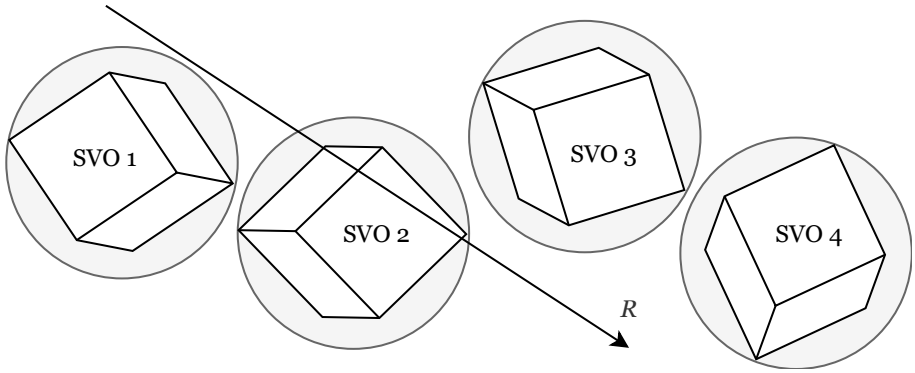


Figure 5.3: Using the bounding sphere of an SVO to avoid traversing octrees that will be missed. SVO 2 is the only octree that will be traversed in this case.

more logical—is that the sphere is invariant under rotation of the corresponding SVO. The only consideration one has to make when constructing the bounding sphere of an SVO is the position of its centre and its radius, both of which are readily obtainable from the SVO; indeed, they are given directly by its translation and scale. This means that the ray-sphere intersection test remains simple even though the underlying SVO may have an arbitrary orientation. A drawback of using bounding spheres instead of bounding boxes is that it will on occasion lead to false positives. In other words, situations may arise where an octree model is processed and traversed even though the ray does not intersect with the SVO.

5.3.2 Depth sorting

The use of bounding spheres means that still further improvements to efficiency can be made. For instance, the SVO models may be traced in a front-to-back order, sorted by the distance along the ray for each intersected bounding sphere. Once an SVO model is traversed and results in a ray hit that is closer than the bounding sphere of the next SVO to be traced, the ray tracing process may be stopped early, as no object can lie in front of the current hit.

In Figure 5.4, a ray tracing scene consisting of an animated set of SVOs is shown. The tracing is performed in a sorted manner, where the order of traversal is by increasing distance to the bounding sphere centre, in this example starting with SVO 1 and ending with SVO 4. The figure shows a situation where SVO 1 is traversed without a hit, SVO 2 is traversed as a false positive, SVO 3 results in a trace hit, and SVO 4 is not traversed. The second octree can be regarded as a false positive because the ray hits the bounding sphere, but not the octree itself. The ray tracing process is terminated after SVO 3 is processed, since it results in a ray hit, and the distance along the ray of this hit is closer than the distance to the boundary of the next octree that would be traversed, SVO 4. In other words, it is mathematically impossible that traversal of the fourth octree will yield a ray hit closer than the hit produced by traversal of the third octree.

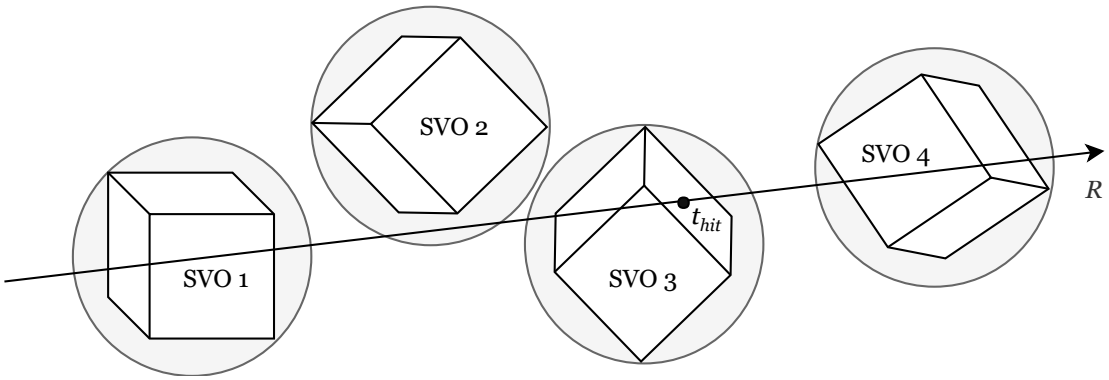


Figure 5.4: Tracing a sorted list of SVOs.

5.4 Discussion

An assessment of the solution detailed in the previous sections is now to be made. A discussion of the efficiency of the solution will be presented, and comparisons to other existing solutions for animation of SVOs will be drawn. Subsequently, the limitations of the solution will be presented.

5.4.1 Efficiency assessment

Firstly, the objectives formulated at the beginning of the chapter will be evaluated. The proposed solution will be compared to the basis method outlined in the introduction to this chapter.

Execution time

Determining how well the solution meets the requirements in terms of execution time is not straightforward. However, since the solution can be compared to the basis method, some conclusions may be drawn.

The proposed solution does not change the internal data of the SVO in any way, as the process of animation is reduced to simply updating a transformation matrix. Hence, it should be self-evident that the execution time will be reduced drastically when compared to the basis method. The basis method must either store each frame of animation as its own SVO, or update the octree data directly and rebuild the hierarchical structure between frames. The former is infeasible in terms of memory bandwidth, as will be detailed below, while the latter is computationally expensive, and presumably impossible in real-time.

An optimisation technique utilising bounding spheres was presented. This technique should increase the performance by limiting the number of SVOs that will be traversed for a given ray in a frame. Since the number of rays processed per frame is in the millions, the gains in execution time yielded by such optimisations quickly add up. As will be seen in Chapter 6, such optimisations are indeed imperative to make real-time performance possible.

Memory requirements

In terms of memory, the proposed solution does not represent a significant increase in space. In practice, only a few more bytes are needed per SVO in order to enable animation. While each octree in theory only needs a transform matrix which specifies its translation, rotation, and scale, it might also make sense to store the bounding sphere and scale separately in order to speed up execution. In any case, the amount of data that needs to be stored is minimal compared to the basis method, where each frame of animation is, in practice, a separate octree.

As for memory bandwidth, when compared to the basis method—where a new octree must be streamed from memory each frame—the proposed solution should perform far better. This is because the only data that needs to be updated between frames is the transform matrix, and in some cases the bounding sphere and scale.

Hardware suitability

For all intents and purposes, the method should be well-suited for hardware. While the method in itself is agnostic with regard to the underlying traversal algorithm and the SVO data structure, the algorithms chosen for the software implementation showcased in Chapter 6 have already been demonstrated to work in hardware by Wilhelmsen [4]. As for the specific features needed by the animation logic, matrix multiplication can easily be accelerated by dedicated circuits. The same can be said for the ray-sphere intersection tests. A consequence of this is that such computations may be kept on-chip, allowing shorter critical paths, and a higher core frequency.

In the discussion of memory bandwidth above, the conclusion was drawn that the proposed solution represents a significant improvement over the basis method, since very little data has to be updated in memory between frames. This also translates well into direct improvements in a hardware setting. By limiting the memory bandwidth, a hardware implementation may avoid performing accesses to external memory altogether, further increasing performance. In addition, since the sparse voxel octree data itself remains unchanged when animated, one or more levels of caching may be employed in order to reduce the latency associated with memory accesses. Lastly, since the amount of data that describes the animation of an octree is very small, these transformation matrices may also be stored directly in very fast register memory close to the core, further reducing memory access latency.

5.4.2 Comparison with other methods

The only alternative animation technique discovered in the literature is the method presented by Bautembach [5]. His solution is very different from the one presented in this thesis. Bautembach treats every node of an SVO as a singular *atom* that may be transformed independently of all other nodes in the octree. His technique can be viewed as a *bottom-up* solution to the problem, while the one presented here is a *top-down* solution.

In a similar manner to the solution presented here, Bautembach's solution does not change the octree data itself. Each node, however, is drawn at a different position that is computed on-the-fly. As is detailed in his paper, the solution is not suitable for ray tracing, since the hierarchical structure of the octree that is relied upon by most ray tracing algorithms is destroyed. The simple deduction that child nodes are no longer necessarily contained within their parent nodes places further emphasis on this fact. Therefore, Bautembach's solution can not really be used for further comparison, since the premise of the solution presented in this thesis is that it should be used in ray tracing.

5.4.3 Limitations

There are some limitations of this method that should be noted. The main limitation is that the only type of animation supported is rigid-body animation. This means that effects such as deformation and other, similar features are not supported. In other words, the animation functionality provided by the solution is mostly suited for stiff, mechanical objects and not mesh objects. The basis method outline in the beginning of this chapter is not restricted by this, and could in theory animate anything, also deformation.

Further, the fact is that each part of the scene that is to be animated must be stored as a separate SVO with its own transform. This may not at first seem like a serious concern, but consider a complex object with hundreds of moving parts. This object would have to be formulated as hundreds of SVOs, and it is not clear how the algorithm would respond to this in terms of performance. A consequence is that each ray would have to be intersection-tested with the bounding sphere of each SVO in the scene, resulting in many millions of extra intersection tests that would have to be performed each frame. As one may realise, in this case it appears that the algorithm will not under any circumstances be able to deliver real-time performance. Future work may include exploring measures that can be taken to mitigate this problem. An idea to this end would be to investigate whether the entire scene could be subdivided—for instance into a large octree—in order to sort the objects so that only relevant SVOs would have to be considered by the ray tracing algorithm.

Chapter 6

Software demonstration

A software demonstration of the scheme detailed in Chapter 5 was prepared for this thesis. The implementation was written in the C programming language, and employs the APIs Nvidia CUDA and OpenGL; the former for the ray tracing itself, and the latter for easing the process of displaying the result. A description of the toolchain, as well as a discussion of the results of testing will be presented in this chapter. The core functions of the software can be found in Appendix A.

6.1 Preliminary design choice assessments

There are several ways in which ray tracing may be implemented. Some methods result in better visual results, and others are more computationally efficient. Since real-time computer graphics is the topic of this thesis, it is natural to assume that at least some aspect of computational efficiency is desired, and that choices to this end should in many cases be preferred. On the basis of these observations, a number of design choices were made when implementing the scheme in software. Justifications for these are presented in the following.

6.1.1 Parallelisation

When computational efficiency is of the essence, it is generally detrimental to limit a software implementation to sequential processing. Luckily, the concept of ray tracing is of such a nature that it easily lends itself to parallelisation. As described in Chapter 2, this is a consequence of the fact that, in and of itself, each ray is independent of every other ray. For every pixel on screen, one or more rays are spawned and traced into the scene. The traversal of the scene by these rays ultimately produces the colour of the individual pixels. Hence, it is logical to try to exploit the parallel nature of ray tracing in some shape or form. While the most efficient solution would presumably be application-specific hardware, there exists readily available solutions that may be used to this end—the most easily available parallel platform being the GPU. More specifically, using the parallel processing capabilities of the graphics processing hardware to run general computational software (GPGPU) may

speed up the computation by many orders of magnitude. As detailed in Chapter 3, Crassin et al. [65] makes a convincing argument for using GPGPU implementations for software ray tracing.

6.1.2 Memory bandwidth

There is a desire to minimise the memory bandwidth requirements of the software, since this can quickly turn into a bottleneck in performance—especially with regard to the sharing of memory between the CPU and GPU of a system [68]. As shown by Yang et al. [69], in order to fully realise the potential of GPGPU, one must consider efficient utilisation of the GPU memory hierarchy, given its dominant impact on performance. Keeping this in mind during the software development will arguably lead to a more efficient design, both in terms of performance and execution time predictability.

6.1.3 Model data

The scene to be traced is made up of a set of octrees employing the SVO format described in detail in Chapters 4 and 5. Each of these SVOs can be thought of as a single, self-contained model. An effort should be made to avoid having to generate these models in a hard-coded manner on application start-up. Models could be generated in advance by processing a polygonal model in a widely used format, and storing it as a binary file in the correct SVO format. There are three main arguments in support of this solution:

Application performance: Offloading the generation of the models to an earlier stage would mean an increase in application performance—at least in the context of start-up time. The models could then be loaded directly into memory as a binary object in the chosen SVO data structure format on application start-up.

Model fidelity: By not generating the models from hard-coded data structures, the model resolution can be increased massively. Models can be generated with arbitrary fidelity, in practice only limited by the capacity of available memory.

Basis of comparison: Generating the voxel models from existing polygonal models will provide a basis of comparison. This is advantageous since useful comparative data results may be gathered and analysed.

6.2 Rendering pipeline description

The ray tracing core was written as a GPGPU program using the Nvidia CUDA API. Based on the traversal algorithm detailed in Chapter 4, the CUDA kernel traces the scene based on a description of the scene itself, as well as the raw SVO data initially copied to graphics memory. The image generated by the ray tracing procedure is placed in a buffer located in graphics device memory. The data is stored in pixel buffer object (PBO) format—essentially as a block of raw pixel data.

Running concurrently alongside the ray tracing kernel is a simple OpenGL program, the purpose of which is to display the data generated by the ray tracer. This OpenGL application reads the PBO data directly from graphics memory and uses the data as a texture. The texture is applied to a simple quad spanning the entire screen of the application, and thus displays the result of the ray tracing process to the user.

The pipeline is illustrated in Figure 6.1. As detailed in Section 6.1.2, care should be taken into minimising the required memory bandwidth of the application, since this may lead to a bottleneck in performance. In the PBO solution described, the data is generated in graphics memory and read directly from the same memory location. This means that the data stays within device memory, and does not have to go via the CPU in order to be displayed, alleviating the concerns for memory bottlenecks.

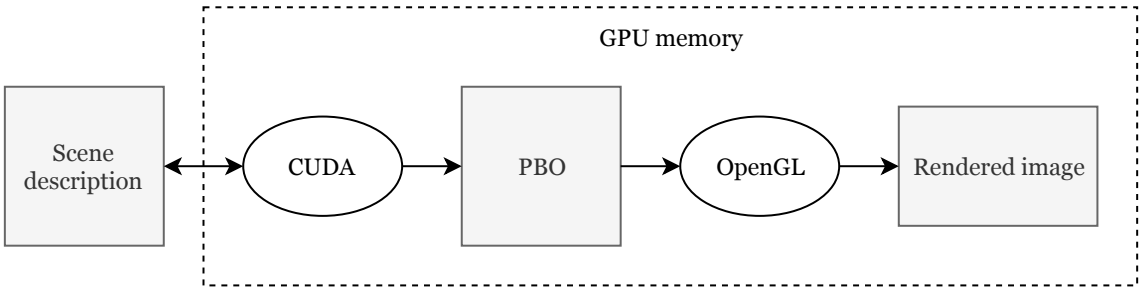


Figure 6.1: The pipeline used when solving the problem.

6.3 Implementation details

The following sections describe details of the implementation, and provide justifications where needed for choices made during development.

6.3.1 Sparse voxel octree generation

The sparse voxel octrees used in the software demonstration was generated from polygonal models in a process similar to the one described in Section 6.1.3. The toolchain for model generation is illustrated in Figure 6.2.

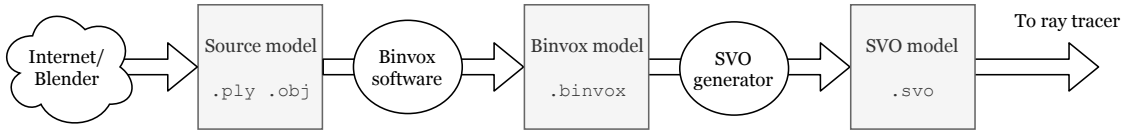


Figure 6.2: Toolchain for generating SVO models.

The polygonal source models were obtained from online resources such as *Stanford 3D Scanning Repository* [43], or created in the modelling software *Blender* [70].

The models were then processed by the software *binvox*, which is a freely-distributed program written by Min [71]. It reads a 3D model file in a widely-used format—in this case `.ply` or `.obj`—and produces a voxel model by means of three-dimensional rasterisation. The voxel model is stored in a custom file format (`.binvox`), which is a raw voxel format that employs the simple compression scheme *run-length encoding* (RLE) to reduce the file size [72].

The last step of the SVO generation toolchain is to generate and store the models on the format specified in Section 4.2, so that they can quickly be loaded into memory as a binary object on application start-up. A custom-made program to perform this conversion was developed; its core functions are written in C++ and included in Appendix C. The program takes a raw voxel model in the `.binvox` format, and initially constructs a raw, inefficient octree data structure from it. Subsequently, the octree is converted to the efficient format used in the solution, and stored to disk in a `.svo` file.

6.3.2 Tracing sparse voxel octrees

The algorithm presented by Revelles, Ureña, and Lastra [45], and described in detail in Section 4.1, is very applicable to the use case presented by this thesis. As such, this algorithm forms the base of the ray tracer core in the software demonstration. The implementation is generally very true to the original algorithm as described in the paper, however, some minor modifications must be made to the algorithm in order for it to work properly in this particular application. Further, certain changes are made to make it even more efficient.

Co-ordinate system

A modification that has to be made to the algorithm during its implementation is to translate it from a left-handed co-ordinate system to the more widely-used right-handed co-ordinate system. The justification for this conversion is that the rest of the application, including the SVO model format employed, assumes a right-handed co-ordinate system, and a discrepancy here would not be desirable. The conversion is achieved by inverting the direction of the z-axis in the algorithm description, and making the corresponding changes to the algorithm state machine. The detailed description of the algorithm in Section 4.1 already accounts for this modification.

Stack usage reduction

Since the algorithm is to be run on a GPU, care should be taken when programming to avoid excessive function calls, and recursion in particular. The reason for this is that the threads that run in parallel on the GPU have very small stacks, and function calls quickly eat up available stack capacity. This poses a problem, since the algorithm as it is defined in the original paper is recursive, and early implementation attempts revealed that the stack placed a hard limit on the maximum octree traversal depth of the algorithm. However, as has already been demonstrated by Wilhelmsen

[4], it is indeed possible to convert the algorithm to use an iterative approach instead. This was done during implementation by defining a small data structure in static, compile-time allocated memory for each thread which essentially functions as a stack, holding all traversal state information at the current stage. The rest of the algorithm runs in a loop and utilises this data structure instead of the native stack.

Allowing zero-valued directions

The algorithm description in Section 4.1 introduces a set of special cases in order to handle rays with zero-valued directions. A simpler approach to this end was discovered during development, and entails checking whether components of the ray direction are zero on algorithm start-up. If one or more components are zero, they are instead assigned a very small number. This will result in a slight distortion of the rendered image, but as long as the value is small enough, it will not be noticeable. By solving the issue this way, the algorithm remains simple and the need to handle special cases is eliminated.

6.3.3 Additional features and optimisations

There is a set of additional features and optimisations that were included in the application. These features or optimisations are not directly related to the algorithms and schemes which form the basis of the implementation, but are part of the software demonstration. As such, they are presented and discussed in this section.

Hit buffer algorithm

A simple buffering mechanism was developed in order to improve the general performance of the application. The buffer—termed *hit buffer object* (HBO)—stores the ray tracing result for each pixel in the last rendered frame. In other words, the buffer contains a data structure for each pixel describing the last result. The idea is that if, for a given pixel, the scene is unchanged *enough* since the last frame, the traversal of SVOs for this pixel may be skipped, and the value from the last frame may be reused. The algorithm is illustrated in Figure 6.3.

Stored for each pixel in the HBO data structure are: the colour, the normal, the t-value of the hit along the ray (i.e. the depth), the index of the SVO that was hit, and the nature of the hit. Most of these are explanatory, except, perhaps, the last entry. The field describing the nature of the hit provides information such as whether nothing was hit, or if the ray passed through multiple bounding spheres before arriving at the hit point.

The HBO is then used in conjunction with a set of state variables to determine if the value of a pixel is unchanged. Each SVO in the scene, as well as the camera, has a switch that specifies if the object has moved since the last frame (if it is *dirty*). The application can then look up the hit buffer data for the current pixel and, if the camera is unchanged, and the SVO object hit by the ray the last frame is unchanged, simply use the last value and avoid tracing the SVO again. It is also

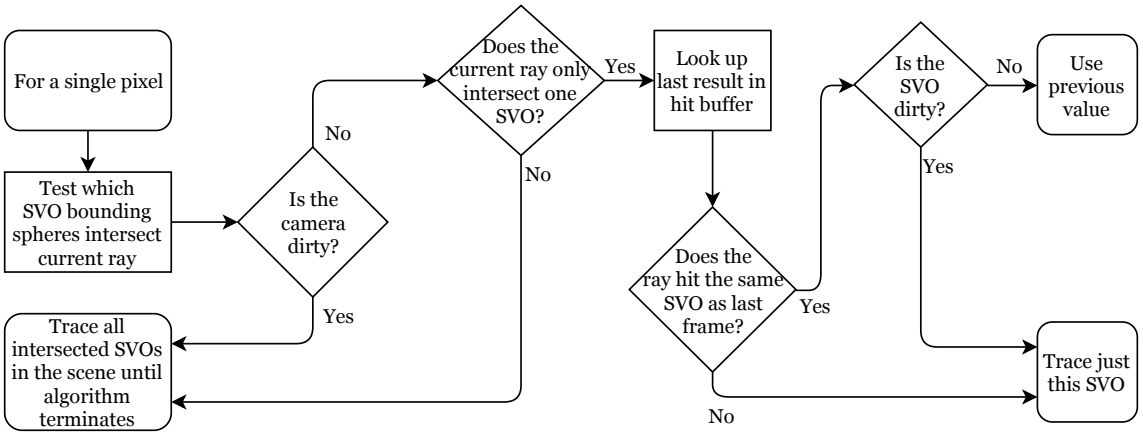


Figure 6.3: The hit buffer algorithm.

required that the ray did not pass through multiple bounding spheres before the hit for this optimisation to take place. The reason for this requirement is that if the ray passes through other SVOs, these might have changed in the meantime, and there is no way of determining if the ray would hit data in these SVOs without traversing them again.

6.4 Results

An animated model of a car rendered with the software implementation is shown in Figure 6.4. The model consists of 2048^3 data points and is a digital representation of an *AC Cobra 269 1963* car, taken from [73]. In this scene, the car body, the wheels, the doors, and the steering wheel are each realised as separate SVO models with their own transforms. The animation sequence itself is included in Appendix B, and the figure shows three snapshots of this sequence.

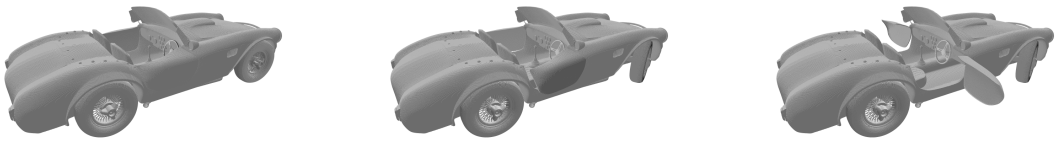


Figure 6.4: Three snapshots of an animated car using the solution. The wheels are rolling, the doors open and close, and the steering wheel and front wheels turn left and right.

In Figure 6.5, a static, non-animated model is shown side-by-side with an animated model for comparison. Summarised in Table 6.1 is the average render time and frames per second for a selection of model types. The first model type is a static, non-animated model of a car, as shown in Figure 6.5a. The second model type is a simple animated car where each part of the body is its own SVO, as shown in Figure 6.5b. The last model type is an animated car consisting of the same parts as the second type, but with the bounding sphere optimisation techniques mentioned in Section 5.3, and the hit buffer mechanism described in Section 6.3.3 enabled. The bounding spheres used in optimisation are highlighted in Figure 6.6.

(a) Non-animated car

(b) Animated car

Figure 6.5: A static, non-animated car compared to an animated car. Both models have a resolution of 2048^3 voxels.

Table 6.1: Average frame rendering time for different model types. The render time is averaged over 10 seconds. Frames per second are calculated as the inverse of the average render time.

Model type	Render time	Frames per second
Non-animated	51.82 ms	19.30 FPS
Animated	71.38 ms	14.01 FPS
Animated (with optimisations)	52.55 ms	19.03 FPS

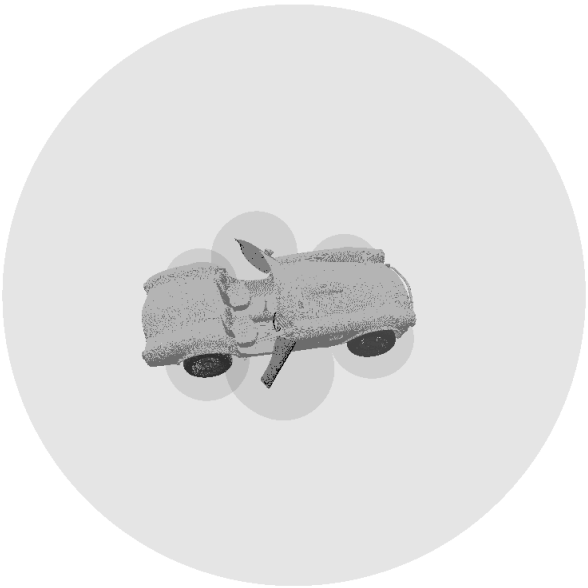


Figure 6.6: Car model with the bounding spheres used for optimisation purposes highlighted.

6.5 Discussion

A discussion of the results gathered from the software implementation is presented in the following. The majority of the discussion regarding the method itself has already been presented in Chapter 5.

6.5.1 Performance

As is apparent from the numerical values shown in Table 6.1, the difference in raw performance between the static model and the animated model with optimisations enabled is almost negligible. This result testifies to the claim that the solution presented in this thesis is suitable for animation of sparse voxel octree data. And since the animation technique presented in this thesis does not introduce a noticeable overhead, further improvements in performance may be feasible with even more optimisation in the traversal algorithm itself. As indicated by the performance counters, most execution time is spent in this section of the software, not as a part of the animation process itself.

It can be argued whether the performance shown should be classified as animation in real-time. This will first require a definition of the term *real-time*. The concept is here taken to have a somewhat different meaning than it does in the field of embedded systems. While the term has a rigorous definition there, it is here used to judge the performance of an animation technique. Therefore it is taken to mean any performance that will be perceived as motion by the human eye. The measure

of frames per second in relation to human motion perception, and at what stage a series of frames is perceived as animation, is a topic outside the scope of this thesis. However, according to Read and Meyer [32, p. 24], 16 frames per second is enough to adequately provide visual continuity so that a set of frames may be perceived as motion. Nonetheless, they go on to explain that with anything below 24 frames per second, a flicker may be perceived, for instance in changes in brightness.

Based on these numbers, the performance shown in the results may be classified as real-time. It should also be noted that the GPU employed when gathering the data, the *GeForce GTX 680* [52], was released in 2012. The graphics card is over 6 years old at the time of writing, so performance of the software demonstration is likely to have improved substantially if run on newer hardware with better CUDA specifications.

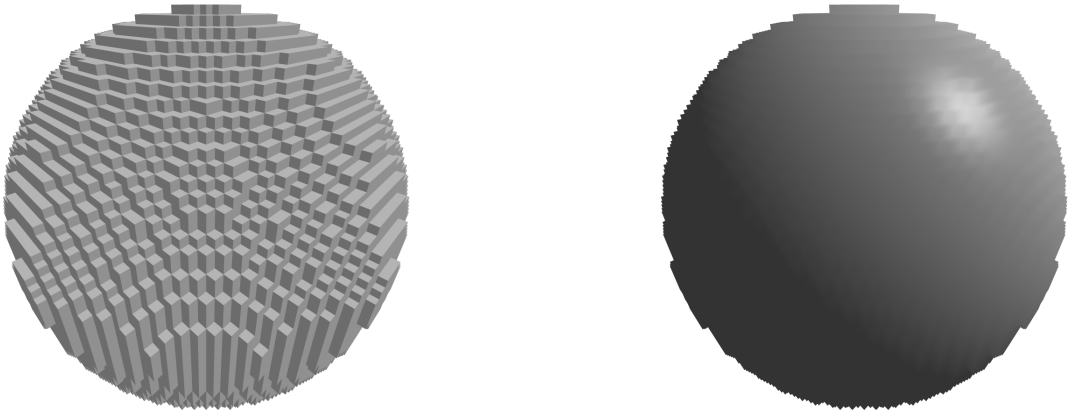
6.5.2 Visual fidelity

The models rendered with the software implementation and shown in the figures earlier in this chapter have a resolution of 2048 voxels in each dimension. In other words, if the models were scaled so that each side had a length of 1 metres, the smallest detail that could be discerned would be around 0.5 mm. For most models in a given scene, this resolution ought to be enough in terms of detail.

However, due to the way the normals are currently calculated, some erroneous visual effects appear. These artefacts can for instance be spotted in Figure 6.5, where one may notice certain aliasing effects on the model. The artefacts stem from the fact that the software implementation currently does not use normal vectors stored in the model itself, but simply calculates a surface normal vector at run-time based on which face of a given voxel that is hit by the ray. If the SVO model format were extended to include surface normals on a per-voxel basis, the visual fidelity would increase dramatically, as the aliasing effects would all but disappear. Including continuous normals would also allow for more complex lighting calculations such as refraction, reflection, and specular effects.

A simple demonstration was written in support of this argument, and the results are shown in Figure 6.7. In the illustration, a sphere is rendered with the two different normal configurations. The sphere only has a resolution of 64 voxels in each dimension, so one would expect the visual fidelity to be even better with higher resolution models, such as the car rendered earlier in this chapter. The sphere in Figure 6.7b also demonstrates one of the more sophisticated lighting models that may be implemented. In this case, ambient light, diffuse light, and specular highlights are modelled, as described by the *Phong reflection model*, formulated by Phong [74].

This normal vector feature is not currently implemented in the software demonstration, and could thus be a part of future work on the results of this thesis. Indeed, the specification of the SVO data structure format by Laine and Karras [31] does present a way for normals to be included as an extension to the SVO format. This might even increase the general performance of the algorithm since the normal vec-



(a) Face-based normals (b) Continuous normals

Figure 6.7: Normals calculated based on hit face compared to continuous normals stored in the model. Both models have a resolution of 64^3.

tors could be fetched directly from the model instead of being calculated explicitly for each ray.

6.5.3 Limitations of the optimisation techniques

There is a limit to the region of applicability for most optimisation techniques. In this case, the hit buffer optimisation introduced in Section 6.3.3 is limited in that becomes ineffective once the camera moves. This is a consequence of the fact that once the camera moves in some fashion, most rays of the algorithm have also necessarily been changed since the last frame. As such, the previous results stored in the hit buffer are no longer applicable, and the entire scene must be retraced. The bounding sphere optimisation introduced in Section 5.3 is still effective, however, as it is an optimisation in world space, not in screen space. In other words, it is a part of the ray tracing algorithm itself, and not dependent on the camera. Thus, it should provide a performance gain regardless of camera movement.

Further investigation into optimisation techniques such as the bounding sphere optimisation, as well as improving the HBO optimisation technique are topics that could be part of future research.

6.5.4 Suitability for hardware implementation

The method's suitability for hardware implementation was discussed in detail in Chapter 5, where it was argued that the general idea is well-suited for hardware acceleration. Since borderline real-time performance has already been demonstrated when running the method in software, it is very exciting to think of the possible increase in performance that may come about if one were to implement the method in application-specific hardware. It is not far-fetched to draw the conclusion that

ray tracing may have a central role in the future of real-time computer graphics. Implementing the method in hardware is certainly something that could be explored in future work.

6.5.5 Additional data

Please note that the results presented in this chapter are gathered from a single animated model, and may not represent the general case. It would be advantageous to run the implementation on additional models such that more data can be gathered. This could be used to further investigate the claim that the animation does not tax the performance noticeably, and help gain understanding of how the algorithm responds to the rendering of different data sets.

Chapter 7

Conclusions

In this thesis a method for animation of sparse voxel octrees was presented. The method is designed for real-time ray tracing, and is therefore optimised for this application. It employs the works of Revelles, Ureña, and Lastra [45], and Laine and Karras [31] as foundation, where the former paper provided the octree traversal algorithm, and the latter contains the formulation upon which the SVO format is based. No previous attempts of animating sparse voxel octrees for use in ray tracing were found in the literature. The method by Bautembach [5] does specify a technique for animation of sparse voxel octrees, but—as he states explicitly himself—his work is not suited for ray tracing applications.

The proposed solution was presented in Chapter 5, where it was subsequently detailed and discussed. A comparative foundation was established, termed the *basis* method, which naïvely reconstructs the octree data structure for each frame. In the discussion, it was argued that the performance of the proposed solution should be far superior to that of the basis method. In addition, its memory requirements and suitability for hardware implementation were evaluated. The conclusions were drawn that an animated SVO requires a negligible increase in memory usage and bandwidth compared to a non-animated, static model, and that the method itself is indeed suitable for hardware implementation. Moreover, some general optimisation techniques relevant to the solution were detailed, for instance the usage of bounding spheres to enable a form of lazy traversal.

A software implementation employing the Nvidia CUDA API was prepared for the sake of gathering empirical data in the evaluation of the proposed solution. In Chapter 6, this implementation was studied and evaluated. The chapter culminated in the demonstration of a car model animated with borderline real-time performance. In the subsequent discussion, its efficiency and visual fidelity were evaluated. It was argued that one could expect further improvements in performance if the solution were to be run on application-specific hardware. An optimisation technique termed *hit buffer object* (HBO) was also presented in this chapter.

In conclusion, the method presented in this thesis does indeed appear to satisfy most, if not all, the requirements stated in the problem text. Whether it is flexible enough to meet the needs of the computer graphics industry remains to be seen. Its

applicability will also depend on what the future holds for ray tracing itself. Due to the dominance of the rasterisation technique in the field of image synthesis, it is uncertain whether ray tracing will ever be widely used. However, certain new and interesting developments in the field—such as Nvidia's newest line of GPUs—do indicate a brighter future for ray tracing, and it should be very exciting to follow what might unfold in the industry over the next few years.

7.1 Limitations

The limitations of the animation technique were detailed in Chapter 5, but the main points will be reiterated here—for completeness' sake. The algorithm only supports rigid-body animation, which means that it is only relevant for certain kinds of animation sequences. For instance, it cannot easily be used to model deformation and mesh animation such as a flexing muscle or a waving flag. Another limitation of the method is that each animated part of the scene has to be formulated as its own SVO. This can lead to a situation where a given scene may consist of a huge number of objects that have to be considered by the traversal algorithm.

Limitations of the software implementation were also detailed. Currently, the implementation does not support surface normals to a satisfactory degree. The normals are at this time determined in the traversal algorithm by looking at which face of a voxel a given ray hits. As a consequence, models appear "blocky", and this may in turn give rise to certain aliasing artefacts. Certain parts of the optimisations implemented are also limited to some degree. For instance, the HBO optimisation technique becomes ineffective once the camera moves.

7.2 Future work

The solution presented in this thesis opens up for quite a few possibilities in terms of continued research. The main area that should be explored is further optimisation. In particular, an attempt to implement the solution in hardware would be very exciting, as it would presumably lead to even better performance. This section of future work is a part of what the author expects he will focus on when writing his master's thesis.

Other kinds of optimisations that may be explored is to extend the bounding sphere optimisation detailed in Chapter 5 even further. One may, for instance, investigate the possibility of sorting the SVOs that make up a scene into another data structure, such as another octree, to increase support for a larger number of models.

It would also be interesting to see how high-resolution models, such as the car rendered in Chapter 6, would look with proper surface normals. These normals could be used in an implementation of a superior lighting model, such as the phong illumination model [74]. As such, extending the SVO model format to include normals is certainly part of future work.

Further evaluation should be performed, for instance by running the implementation on several different scenes. Since the current results only stem from one animated model, there is simply not enough empirical data to fully support the claim that the technique does not significantly slow down the rendering process.

Bibliography

[1] R. Shrout. *John Carmack on id Tech 6, Ray Tracing, Consoles, Physics and more*. 2008. URL: https://www.pcper.com/reviews/Graphics-Cards/John-Carmack-id-Tech-6-Ray-Tracing-Consoles-Physics-and-more (visited on 08/31/2018).

[2] *GeForce RTX - Graphics Reinvented*. 2018. URL: https://www.nvidia.com/en-us/geforce/20-series/ (visited on 09/28/2018).

[3] M. J. Roe. "Chaos and Evolution in Law and Economics". In: *Harvard Law Review* 109.3 (Jan. 1996), p. 641. DOI: 10.2307/1342067.

[4] A. Wilhelmsen. "Efficient Ray Tracing of Sparse Voxel Octrees on an FPGA". M.S. thesis. 2012. URL: http://hdl.handle.net/11250/2370620 (visited on 08/22/2018).

[5] D. Bautembach. *Animated Sparse Voxel Octrees*. B.S. thesis. 2011. URL: http://masters.donntu.org/2012/fknt/radchenko/library/asvo.pdf (visited on 08/22/2018).

[6] *Maple*. Version 2018. Maplesoft, Mar. 21, 2018. URL: https://www.maplesoft.com/products/maple/ (visited on 09/28/2018).

[7] *MATLAB*. Version R2018b. MathWorks, Sept. 12, 2018. URL: http://mathworks.com/products/matlab (visited on 09/28/2018).

[8] *Wolfram Mathematica*. Version 11.3.0. Wolfram Research, Mar. 8, 2018. URL: http://www.wolfram.com/mathematica (visited on 09/28/2018).

[9] E. Adams. *Fundamentals of Game Design (3rd Edition)*. New Riders, 2013. ISBN: 978-0321929679.

[10] M. Wolf. *The Video Game Explosion: A History from PONG to PlayStation and Beyond*. Greenwood, 2007. ISBN: 978-0313338687.

[11] S. Rabin. *Introduction to Game Development, Second Edition*. Course Technology PTR, 2009. ISBN: 978-1584506799.

[12] *SketchUp*. Version 18.0.16975. Trimble Inc., Nov. 14, 2017. URL: https://www.sketchup.com/ (visited on 09/28/2018).

[13] *AutoCAD*. Version 2019. Autodesk, Mar. 22, 2018. URL: https://www.autodesk.com/autocad (visited on 09/28/2018).

[14] *SolidWorks*. Version 2018 SP3. Dassault Systèmes, May 21, 2018. URL: http://www.solidworks.com/ (visited on 09/28/2018).

[15] *GTK+*. Version 3.24.1. The GNOME Project, Sept. 19, 2018. URL: https://gtk.org/ (visited on 09/28/2018).

[16] *Qt*. Version 5.11.2. Qt Project, Sept. 20, 2018. URL: https://www.qt.io/ (visited on 09/28/2018).

[17] R. Rickitt. *Special Effects: The History and Technique*. Billboard Books, 2007. ISBN: 0-8230-8408-6.

[18] T. Porter and T. Duff. "Compositing digital images". In: *Proceedings of the 11th annual conference on Computer graphics and interactive techniques - SIGGRAPH '84*. ACM Press, 1984. DOI: 10.1145/800031.808606.

[19] I. E. Sutherland. "Sketchpad: A Man-Machine Graphical Communication System". In: *Proceedings of the May 21-23, 1963, spring joint computer conference on - AFIPS '63 (Spring)*. ACM Press, 1963. DOI: 10.1145/1461551.1461591.

[20] A. M. Noll. "Scanned-display computer graphics". In: *Communications of the ACM* 14.3 (Mar. 1971), pp. 143–150. DOI: 10.1145/362566.362567.

[21] B. W. Jordan and R. C. Barrett. "A scan conversion algorithm with reduced storage requirements". In: *Communications of the ACM* 16.11 (Nov. 1973), pp. 676–682. DOI: 10.1145/355611.362537.

[22] E. Catmull. "A Subdivision Algorithm for Computer Display of Curved Surfaces". AAI7504786. PhD thesis. Dec. 1974.

[23] W. Straßer. *Schnelle Kurven- und Flächendarstellung auf grafischen Sichtgeräten*. Heinrich-Hertz-Institut für Nachrichtentechnik Berlin, West: Technischer Bericht. Technische Universität Berlin, 1974.

[24] A. Appel. "Some techniques for shading machine renderings of solids". In: *Proceedings of the April 30–May 2, 1968, spring joint computer conference on - AFIPS '68 (Spring)*. ACM Press, 1968. DOI: 10.1145/1468075.1468082.

[25] T. Whitted. "An improved illumination model for shaded display". In: *Communications of the ACM* 23.6 (June 1980), pp. 343–349. DOI: 10.1145/358876.358882.

[26] T. Oguchi et al. "A single-chip graphic display controller". In: *1981 IEEE International Solid-State Circuits Conference. Digest of Technical Papers*. IEEE, 1981. DOI: 10.1109/isscc.1981.1156160.

[27] C. Machover and J. Dill. "New products". In: *IEEE Computer Graphics and Applications* 5.10 (Oct. 1985), pp. 67–75. DOI: 10.1109/mcg.1985.276240.

[28] Daecheol You and K.-S. Chung. "Dynamic voltage and frequency scaling framework for low-power embedded GPUs". In: *Electronics Letters* 48.21 (2012), p. 1333. DOI: 10.1049/el.2012.2624.

[29] A. Watt and M. Watt. *Advanced animation and rendering techniques*. Addison-Wesley Professional, 1992. ISBN: 978-0201544121.

[30] T. Theoharis et al. *Graphics and Visualization: Principles & Algorithms*. A K Peters/CRC Press, 2008. ISBN: 978-1-4398-6435-7.

[31] S. Laine and T. Karras. "Efficient Sparse Voxel Octrees". In: *IEEE Transactions on Visualization and Computer Graphics* 17.8 (Aug. 2011), pp. 1048–1059. DOI: 10.1109/tvcg.2010.240. URL: https://www.nvidia.com/docs/IO/88972/nvr-2010-001.pdf (visited on 05/16/2013).

[32] P. Read and M.-P. Meyer. *Restoration of Motion Picture Film (Butterworth-Heinemann Series in Conservation and Museology)*. Butterworth-Heinemann, 2000. ISBN: 0-7506-2793-X.

[33] Henrik. *Ray trace diagram*. 2008. URL: https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg (visited on 11/05/2018).

[34] Per H. Christensen et al. "Ray Tracing for the Movie *Cars*". In: *2006 IEEE Symposium on Interactive Ray Tracing*. IEEE, Sept. 2006. DOI: 10.1109/rt.2006.280208.

[35] T. Babb. *Recursive raytrace of a sphere*. 2008. URL: https://commons.wikimedia.org/wiki/File:Recursive_raytrace_of_a_sphere.png (visited on 11/05/2018).

[36] G. Tran. *Glasses 800 edit*. 2006. URL: https://en.wikipedia.org/wiki/File:Glasses_800_edit.png (visited on 11/05/2018).

[37] D. Meagher. "Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer". In: IPL-TR-80-111 (Oct. 1980).

[38] Hervé Brönnimann and Marc Glisse. "Cost-Optimal Trees for Ray Shooting". In: *LATIN 2004: Theoretical Informatics*. Springer Berlin Heidelberg, 2004, pp. 349–358. DOI: 10.1007/978-3-540-24698-5_39.

[39] D. S. Bloomberg. *Color quantization using octrees*. Sept. 2008.

[40] D. Luebke et al. *Level of Detail for 3D Graphics*. Morgan Kaufmann, 2003. ISBN: 1-55860-838-9.

[41] M. R. Schmid et al. "Dynamic level of detail 3D occupancy grids for automotive use". In: *2010 IEEE Intelligent Vehicles Symposium*. June 2010, pp. 269–274. DOI: 10.1109/IVS.2010.5548088.

[42] J. Elseberg et al. "Comparison on nearest-neigbour-search strategies and implementations for efficient shape registration". In: 3 (Jan. 2012), pp. 2–12.

[43] *Stanford 3D Scanning Repository*. 2012. URL: https://graphics.stanford.edu/data/3Dscanrep/ (visited on 09/24/2018).

[44] S. Jabłoński and T. Martyn. "Real-Time Rendering of Continuous Levels of Detail for Sparse Voxel Octrees." In: *Computer Graphics, Visualization and Computer Vision WSCG 2016. Short Papers Proceedings*. 2016, pp. 79–88.

[45] J. Revelles, C. Ureña, and M. Lastra. "An Efficient Parametric Algorithm for Octree Traversal". In: *Journal of WSCG* 8.1-3 (2000), pp. 212–219.

[46] A.H. Robinson and C. Cherry. "Results of a prototype television bandwidth compression scheme". In: *Proceedings of the IEEE* 55.3 (1967), pp. 356–364. DOI: 10.1109/proc.1967.5493.

[47] D. Shreiner et al. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3 (8th Edition)*. Addison-Wesley Professional, 2013. ISBN: 978-0-321-77303-6.

[48] J. Peddie. *Who's the Fairest of Them All?* July 2012. URL: http://www.cgw.com/Publications/CGW/2012/Volume-35-Issue-4-June-July-2012/Who-s-the-Fairest-of-Them-All-.aspx (visited on 09/26/2018).

[49] *CUDA Toolkit Documentation*. Jan. 2017. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (visited on 09/26/2018).

[50] Michael J. Flynn. "Some Computer Organizations and Their Effectiveness". In: *IEEE Transactions on Computers* C-21.9 (Sept. 1972), pp. 948–960. DOI: 10.1109/tc.1972.5009071.

[51] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach (The Morgan Kaufmann Series in Computer Architecture and Design)*. Morgan Kaufmann, 2017. ISBN: 978-0-12-811905-1.

[52] *GeForce GTX 680*. 2012. URL: https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-680 (visited on 10/31/2018).

[53] P. Du et al. "From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming". In: *Parallel Computing* 38.8 (Aug. 2012), pp. 391–407. DOI: 10.1016/j.parco.2011.10.002.

[54] A. S. Glassner. "Space subdivision for fast ray tracing". In: *IEEE Computer Graphics and Applications* 4.10 (1984), pp. 15–24. DOI: 10.1109/mcg.1984.6429331.

[55] M. Levoy. "Efficient ray tracing of volume data". In: *ACM Transactions on Graphics* 9.3 (July 1990), pp. 245–261. DOI: 10.1145/78964.78965.

[56] H. Samet. "Implementing ray tracing with octrees and neighbor finding". In: *Computers & Graphics* 13.4 (Jan. 1989), pp. 445–460. DOI: 10.1016/0097-8493(89)90006-x.

[57] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley Pub (Sd), 1995. ISBN: 0-201-50300-X.

[58] M. Agate, R. L. Grimsdale, and P. F. Lister. "The HERO Algorithm for Ray-Tracing Octrees". In: *Eurographics Workshop on Graphics Hardware*. Ed. by R. Grimsdale and W. Strasser. The Eurographics Association, 1989. ISBN: ISBN 3-540-53473-3. DOI: 10.2312/EGGH/EGGH89/061-073.

[59] F. W. Jansen. "Data structures for ray tracing". In: *Data Structures for Raster Graphics*. Springer Berlin Heidelberg, 1986, pp. 57–73. DOI: 10.1007/978-3-642-71071-1_4.

[60] D. Cohen and A. Shaked. "Photo-Realistic Imaging of Digital Terrains". In: *Computer Graphics Forum* 12.3 (Aug. 1993), pp. 363–373. DOI: 10.1111/1467-8659.1230363.

[61] I. Gargantini and H. H. Atkinson. "Ray Tracing an Octree: Numerical Evaluation of the First Intersection". In: *Computer Graphics Forum* 12.4 (Oct. 1993), pp. 199–210. DOI: 10.1111/1467-8659.1240199.

[62] R. Endl and M. Sommer. "Classification of Ray-Generators in Uniform Subdivisions and Octrees for Ray Tracing". In: *Computer Graphics Forum* 13.1 (Feb. 1994), pp. 3–19. DOI: 10.1111/1467-8659.1310003.

[63] A. Knoll et al. "Interactive Isosurface Ray Tracing of Large Octree Volumes". In: *2006 IEEE Symposium on Interactive Ray Tracing*. IEEE, Sept. 2006, pp. 115–124. DOI: 10.1109/rt.2006.280222.

[64] E. Gobbetti and F. Marton. "Far Voxels – a multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms." In: *ACM Transactions on Graphics* 24.3 (2005), pp. 878–885. DOI: 10.1145/1186822.1073277.

[65] C. Crassin et al. "Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering". In: *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*. 2009, pp. 15–22.

[66] S. Thiedemann et al. "Voxel-based global illumination". In: *Symposium on Interactive 3D Graphics and Games on - I3D '11*. ACM Press, 2011. DOI: 10.1145/1944745.1944763.

[67] N. Li et al. "Virtual X-ray imaging techniques in an immersive casting simulation environment". In: *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms* 262.1 (Aug. 2007), pp. 143–152. DOI: 10.1016/j.nimb.2007.04.262.

[68] Yuri Torres, Arturo Gonzalez-Escribano, and Diego R. Llanos. "Using Fermi Architecture Knowledge to Speed up CUDA and OpenCL Programs". In: *2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*. IEEE, July 2012. DOI: 10.1109/ispa.2012.92.

[69] Yi Yang et al. "A GPGPU compiler for memory optimization and parallelism management". In: *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation - PLDI '10*. ACM Press, 2010. DOI: 10.1145/1806596.1806606.

[70] *Blender*. Version 2.79b. Blender Foundation, Mar. 22, 2018. URL: http://blender.org/ (visited on 10/29/2018).

[71] P. Min. *Binvox*. Version 1.26. Oct. 22, 2017. URL: https://www.patrickmin.com/binvox/ (visited on 10/08/2018).

[72] Michael Kazhdan. *BINVOX voxel file format specification*. 2015. URL: https://patrickmin.com/binvox/binvox.html (visited on 10/30/2018).

[73] alex38. *AC Cobra 269 3d model*. 2016. URL: https://free3d.com/3d-model/ac-cobra-269-83668.html (visited on 10/30/2018).

[74] Bui Tuong Phong. "Illumination for computer generated pictures". In: *Communications of the ACM* 18.6 (June 1975), pp. 311–317. DOI: 10.1145/360825.360839.

Appendix A

Ray tracer functions

A.1 Ray tracer core

```
typedef struct {
    svo_entry_t* entry;
    float3 t0;
    float3 t1;
    float3 tm;
    uint8_t cur_child;

} traversal_stack_frame;

inline __device__ bool traverse_svo(
    svo_t svo, float* t_hit, uint8_t a,
    traversal_stack_frame* stack, uint8_t stack_idx,
    float4* hit_color, float3* hit_normal
) {
    if (stack[stack_idx].t1.x < 0.0 ||
        stack[stack_idx].t1.y < 0.0 ||
        stack[stack_idx].t1.z < 0.0
        ) {
        return false;
    }

    while (stack_idx < MAX_STACK_DEPTH) {
        uint8_t child_transformed = a ^ stack[stack_idx].cur_child;
        float t_enter = fmaxf(
                            fmaxf(stack[stack_idx].t0.x, stack[stack_idx].t0.y),
                            stack[stack_idx].t0.z
                        );
        float t_exit = fminf(
                            fminf(stack[stack_idx].t1.x, stack[stack_idx].t1.y),
                            stack[stack_idx].t1.z
                        );

        if (
            t_exit > 0.0 &&
            get_valid_mask(*stack[stack_idx].entry) & (1 << child_transformed)
```

```
) {
    float3 t0_next, t1_next;
    get_child_entry_coords(
            stack[stack_idx].cur_child,
            stack[stack_idx].t0,
            stack[stack_idx].tm,
            stack[stack_idx].t1,
            &t0_next,
            &t1_next
            );

    if (get_leaf_mask(*stack[stack_idx].entry) & (1 << child_transformed)) {
        *t_hit = fmaxf(fmaxf(t0_next.x, t0_next.y), t0_next.z);
        if (*t_hit == t0_next.x) {
            *hit_normal = make_float3(1.0, 0.0, 0.0);
        } else if (*t_hit == t0_next.y) {
            *hit_normal = make_float3(0.0, 1.0, 0.0);
        } else {
            *hit_normal = make_float3(0.0, 0.0, 1.0);
        }
        *hit_color = make_float4(1.0, 1.0, 1.0, 1.0);
        return true;

    } else {
        stack_idx++;
        stack[stack_idx].entry = get_child_entry(
                                     stack[stack_idx - 1].entry,
                                     child_transformed
                                 );
        stack[stack_idx].t0 = t0_next;
        stack[stack_idx].t1 = t1_next;
        stack[stack_idx].tm = 0.5 * (t0_next + t1_next);
        stack[stack_idx].cur_child = get_first_node(
                                         stack[stack_idx].t0,
                                         stack[stack_idx].tm
                                     );
    }
} else {
    uint8_t next_child = get_next_child(
                             stack[stack_idx].cur_child,
                             stack[stack_idx].tm,
                             stack[stack_idx].t1
                         );

    while (next_child == 8) {
        if (stack_idx == 0) {
            return false;
        }

        stack_idx--;
        next_child = get_next_child(
                         stack[stack_idx].cur_child,
                         stack[stack_idx].tm,
```

```
                                stack[stack_idx].t1
                            );
            }

            stack[stack_idx].cur_child = next_child;
        }
    }

    return false;
}

inline __device__ void intersect_svo(
    ray_t ray, float scale,
    float3* t0, float3* t1,
    float* t_enter, float* t_exit, uint8_t* a
) {
    if (ray.dir.x == 0.0) {
        ray.dir.x = 0.0001;
    }
    if (ray.dir.y == 0.0) {
        ray.dir.y = 0.0001;
    }
    if (ray.dir.z == 0.0) {
        ray.dir.z = 0.0001;
    }

    *a = 0;
    float dim = scale * 0.5;

    if (ray.dir.x < 0.0) {
        ray.orig.x = -ray.orig.x;
        ray.dir.x = -ray.dir.x;
        *a |= 1;
    }

    if (ray.dir.y < 0.0) {
        ray.orig.y = -ray.orig.y;
        ray.dir.y = -ray.dir.y;
        *a |= 2;
    }

    if (ray.dir.z < 0.0) {
        ray.orig.z = -ray.orig.z;
        ray.dir.z = -ray.dir.z;
        *a |= 4;
    }

    *t0 = make_float3(
              (-dim - ray.orig.x) / ray.dir.x,
              (-dim - ray.orig.y) / ray.dir.y,
              (-dim - ray.orig.z) / ray.dir.z
          );
```

```
    *t1 = make_float3(
              (dim - ray.orig.x) / ray.dir.x,
              (dim - ray.orig.y) / ray.dir.y,
              (dim - ray.orig.z) / ray.dir.z
          );

    *t_enter = fmaxf(fmaxf(t0->x, t0->y), t0->z);
    *t_exit = fminf(fminf(t1->x, t1->y), t1->z);
}

inline __device__ bool trace_single_ray(
    svo_t svo, ray_t ray,
    float* t_hit, float4* hit_color, float3* hit_normal
) {
    float3 t0, t1;
    float t_enter, t_exit;
    uint8_t a;

    intersect_svo(ray, svo.scale, &t0, &t1, &t_enter, &t_exit, &a);

    if (t_enter < t_exit) {
        traversal_stack_frame stack[MAX_STACK_DEPTH] = {0};
        stack[0].entry = svo.entries;
        stack[0].t0 = t0;
        stack[0].t1 = t1;
        stack[0].tm = 0.5 * (t0 + t1);
        stack[0].cur_child = get_first_node(stack[0].t0, stack[0].tm);

        return traverse_svo(svo, t_hit, a, stack, 0, hit_color, hit_normal);
    }

    return false;
}

inline __device__ void trace(scene_t scene, ray_t ray, hit_t* hit, camera_t camera)
{
    float4 color = CLEARCOL;
    float3 normal;
    float t_hit;

    hit_t new_hit;

    new_hit.type = HIT_NONE;
    new_hit.color = color;
    new_hit.t = 100000000.0;

    for (uint8_t idx = 0; idx < scene.svo_count; idx++) {
        if (intersection(ray, get_bounding_sphere(scene.svo[idx]), &t_hit)) {
            if (new_hit.type == HIT_NONE) {
                new_hit.type = HIT_SIMPLE;
            } else {
                new_hit.type = HIT_MULTIPLE;
            }
```

```
        if (t_hit < new_hit.t) {
            new_hit.t = t_hit;
            new_hit.svo_idx = idx;
        }
    }
}

if (
    new_hit.type == HIT_SIMPLE && new_hit.svo_idx == hit->svo_idx &&
    !scene.svo[hit->svo_idx].dirty && !camera.dirty
) {
    return;
}

new_hit.t = 100000000.0;

if (new_hit.type == HIT_SIMPLE) {
    ray_t ray_tr = transform_ray(
                        ray,
                        scene.svo[new_hit.svo_idx].position,
                        scene.svo[new_hit.svo_idx].rotation
                    );
    if (trace_single_ray(
            scene.svo[new_hit.svo_idx], ray_tr, &t_hit, &color, &normal
        )
    ) {
        new_hit.t = t_hit;
        new_hit.color = color * abs(dot(
            LIGHT_DIR,
            transpose(make_rotation(scene.svo[new_hit.svo_idx].rotation)) * normal
        ));
        new_hit.normal = normal;
    }
} else if (new_hit.type == HIT_MULTIPLE) {
    ray_t ray_tr = transform_ray(
                        ray,
                        scene.svo[new_hit.svo_idx].position,
                        scene.svo[new_hit.svo_idx].rotation
                    );
    if (trace_single_ray(
            scene.svo[new_hit.svo_idx], ray_tr, &t_hit, &color, &normal
        )
    ) {
        new_hit.t = t_hit;
        new_hit.color = color * abs(dot(
            LIGHT_DIR,
            transpose(make_rotation(scene.svo[new_hit.svo_idx].rotation)) * normal
        ));
        new_hit.normal = normal;
    }

    for (uint8_t idx = 0; idx < scene.svo_count; idx++) {
```

```
            if (idx == new_hit.svo_idx) {
                continue;
            }

            if (intersection(ray, get_bounding_sphere(scene.svo[idx]), &t_hit)) {
                if (new_hit.t < t_hit) {
                    continue;
                }
                ray_tr = transform_ray(
                             ray,
                             scene.svo[idx].position,
                             scene.svo[idx].rotation
                         );
                if (trace_single_ray(
                        scene.svo[idx], ray_tr, &t_hit, &color, &normal
                    )
                ) {
                    if (t_hit < new_hit.t) {
                        new_hit.t = t_hit;
                        new_hit.color = color * abs(dot(
                            LIGHT_DIR,
                            transpose(make_rotation(scene.svo[idx].rotation)) * normal
                        ));
                        new_hit.normal = normal;
                    }
                }
            }
        }
    }
    *hit = new_hit;
}
```

A.2 Ray tracer helper functions

```
typedef uint32_t svo_entry_t;

typedef struct {
    svo_entry_t* entries;
    uint32_t size;
    float3 position;
    float3 rotation;
    float scale;
    bool dirty;
} svo_t;

inline __device__ uint8_t get_first_node(float3 t0, float3 tm)
{
    uint8_t child = 0;

    if (t0.x > t0.y && t0.x > t0.z) {
        // YZ-plane
        child |= ((tm.y < t0.x) << 1);
        child |= ((tm.z < t0.x) << 2);
    } else if (t0.y > t0.x && t0.y > t0.z) {
        // XZ-plane
        child |= ((tm.x < t0.y) << 0);
        child |= ((tm.z < t0.y) << 2);
    } else {
        // XY-plane
        child |= ((tm.x < t0.z) << 0);
        child |= ((tm.y < t0.z) << 1);
    }
    return child;
}

inline __device__ void get_child_entry_coords(
    uint8_t child,
    float3 t0, float3 tm, float3 t1,
    float3* t0_next, float3* t1_next
) {
    t0_next->x = (child & 1) ? tm.x : t0.x;
    t0_next->y = (child & 2) ? tm.y : t0.y;
    t0_next->z = (child & 4) ? tm.z : t0.z;

    t1_next->x = (child & 1) ? t1.x : tm.x;
    t1_next->y = (child & 2) ? t1.y : tm.y;
    t1_next->z = (child & 4) ? t1.z : tm.z;
}

inline __device__ uint8_t get_next_child(uint8_t cur_child, float3 tm, float3 t1)
{
    float3 vec;
    vec.x = (cur_child & 1) ? t1.x : tm.x;
    vec.y = (cur_child & 2) ? t1.y : tm.y;
    vec.z = (cur_child & 4) ? t1.z : tm.z;
```

```
    uint3 next;
    next.x = (cur_child & 1) ? 8 : cur_child + 1;
    next.y = (cur_child & 2) ? 8 : cur_child + 2;
    next.z = (cur_child & 4) ? 8 : cur_child + 4;

    if (vec.x < vec.y && vec.x < vec.z) {
        return next.x;
    } else if (vec.y < vec.x && vec.y < vec.z) {
        return next.y;
    } else {
        return next.z;
    }
}

__device__ uint8_t get_valid_mask(svo_entry_t entry)
{
    return (entry >> 8) & 0xFF;
}

__device__ uint8_t get_leaf_mask(svo_entry_t entry)
{
    return entry & 0xFF;
}

__device__ svo_entry_t* get_child_entry(svo_entry_t* entry, uint8_t idx)
{
    svo_entry_t* child_ptr = entry + get_child_table_pointer(*entry);

    if (get_is_far(*entry)) {
        child_ptr = entry + (uint32_t)(*child_ptr);
    }

    uint8_t valid_mask = get_valid_mask(*entry);
    uint8_t leaf_mask = get_leaf_mask(*entry);

    uint32_t offset = 0;
    for (uint8_t i = 0; i < idx; i++) {
        if ((valid_mask >> i) & 0x01) {
            child_ptr++;
        }
    }

    return child_ptr + offset;
}

__device__ sphere_t get_bounding_sphere(svo_t svo)
{
    sphere_t sphere;
    sphere.center = svo.position;
    sphere.radius = sqrt(svo.scale * svo.scale * 0.25 * 3);
    return sphere;
}
```

Appendix B

Car animation sequence

```
__global__ void animation_kernel(scene_t scene, double time)
{
    for (uint8_t i = 0; i < scene.svo_count; i++) {
        scene.svo[i].dirty = false;
    }

    // Wheels
    scene.svo[1].position = make_float3(0.28, -0.07, -0.17);
    scene.svo[1].rotation = make_float3(0.0, M_PI, -time);
    scene.svo[1].scale = 0.17;
    scene.svo[1].dirty = true;

    scene.svo[2].position = make_float3(-0.31, -0.07, 0.17);
    scene.svo[2].rotation = make_float3(0.0, sin(time) * 0.5, time);
    scene.svo[2].scale = 0.17;
    scene.svo[2].dirty = true;

    scene.svo[3].position = make_float3(-0.31, -0.07, -0.17);
    scene.svo[3].rotation = make_float3(0.0, M_PI + sin(time) * 0.5, -time);
    scene.svo[3].scale = 0.17;
    scene.svo[3].dirty = true;

    scene.svo[4].position = make_float3(0.28, -0.07, 0.17);
    scene.svo[4].rotation = make_float3(0.0, 0.0, time);
    scene.svo[4].scale = 0.17;
    scene.svo[4].dirty = true;

    // Car doors
    float t = (sin(time / 2) + 1.0) / 2 * M_PI / 2;
    float st = sin(t);
    float ct = cos(t);
    scene.svo[5].position = make_float3(0.0205, -0.022, 0.167) +
                            make_float3(0.06 * cos(t), 0.0, 0.11 * sin(t));
    scene.svo[5].rotation = make_float3(0.0, -t, 0.0);
    scene.svo[5].scale = 0.195;
    scene.svo[5].dirty = true;

    scene.svo[6].position = make_float3(0.0205, -0.022, -0.167) +
```

```
                            make_float3(0.06 * cos(t), 0.0, -0.11 * sin(t));
    scene.svo[6].rotation = make_float3(0.0, t, 0.0);
    scene.svo[6].scale = 0.195;
    scene.svo[6].dirty = true;

    // Steering wheel
    scene.svo[7].position = make_float3(0.0, 0.0, -0.1);
    scene.svo[7].rotation = make_float3(-0.3, M_PI / 2, sin(time) * 3.0);
    scene.svo[7].scale = 0.08;
    scene.svo[7].dirty = true;
}
```

Appendix C

SVO model generation

```
class octree_node {
    public:
        uint32_t max_depth;
        uint32_t posx;
        uint32_t posy;
        uint32_t posz;
        uint32_t dim;
        uint8_t leaf_mask;
        uint8_t valid_mask;
        octree_node* children[8] = {nullptr};
        bool leaves[8] = {false};
        octree_node* parent = nullptr;

    octree_node(
        uint8_t* raw_data,
        uint32_t root_dim, uint32_t dim,
        uint32_t ox, uint32_t oy, uint32_t oz,
        octree_node* parent
    ) {
        this->dim = dim;
        this->posx = ox;
        this->posy = oy;
        this->posz = oz;

        valid_mask = 0xFF;
        leaf_mask = 0x00;
        max_depth = 0;
        this->parent = parent;
        for (int dz = 0; dz < 2; dz++) {
            for (int dy = 0; dy < 2; dy++) {
                for (int dx = 0; dx < 2; dx++) {
                    uint32_t idx = dx + dy * 2 + dz * 2 * 2;
                    if (dim > 2) {
                        children[idx] = new octree_node(
                                            raw_data,
                                            root_dim,
                                            dim / 2,
                                            ox + dx * dim / 2,
```

```
                                        oy + dy * dim / 2,
                                        oz + dz * dim / 2,
                                        this
                                    );

                        if (children[idx]->leaf_mask == children[idx]->valid_mask) {
                            if (children[idx]->valid_mask == 0xFF) {
                                leaf_mask |= 1 << idx;
                                valid_mask |= 1 << idx;
                                delete children[idx];
                                children[idx] = nullptr;
                                leaves[idx] = true;
                            } else if (children[idx]->valid_mask == 0x00) {
                                leaf_mask &= ~(1 << idx);
                                valid_mask &= ~(1 << idx);
                                delete children[idx];
                                children[idx] = nullptr;
                                leaves[idx] = false;
                            }
                        }

                        if (
                            children[idx] != nullptr &&
                            children[idx]->max_depth + 1 > max_depth
                        ) {
                            max_depth = children[idx]->max_depth + 1;
                        }
                    } else {
                        uint8_t unfilled = get_raw_value(
                                                raw_data,
                                                root_dim,
                                                ox + dx,
                                                oy + dy,
                                                oz + dz
                                            ) == 0;
                        valid_mask &= ~(unfilled << idx);
                        leaf_mask = valid_mask;
                        leaves[idx] = true;
                    }
                }
            }
        }
    }

    ~octree_node()
    {
        for (int idx = 0; idx < 8; idx++) {
            if (children[idx] != nullptr) {
                delete children[idx];
            }
        }
    }
```

```
uint32_t get_descendant_count()
{
    int count = 1;
    for (int idx = 0; idx < 8; idx++) {
        if (children[idx] != nullptr) {
            count += children[idx]->get_descendant_count();
        }
    }
    return count;
}

uint32_t get_child_count()
{
    int count = 0;
    for (int idx = 0; idx < 8; idx++) {
        if (children[idx] != nullptr) {
            count++;
        }
    }
    return count;
}

uint32_t get_leaf_count()
{
    uint8_t count = 0;
    for (int idx = 0; idx < 8; idx++) {
        if (leaves[idx]) {
            count++;
        }
    }
    return count;
}

uint32_t get_depth()
{
    if (parent != nullptr) {
        return parent->get_depth() + 1;
    }
    return 0;
}

uint8_t get_raw_value(
    uint8_t* raw_data,
    uint32_t dim,
    uint32_t x,
    uint32_t y,
    uint32_t z
) {
    uint64_t idx = (uint64_t)x * (uint64_t)dim * (uint64_t)dim +
                   (uint64_t)z * (uint64_t)dim +
                   (uint64_t)y;
    return raw_data[idx];
}
```

```
    octree_node* get_first_child()
    {
        for (int idx = 0; idx < 8; idx++) {
            if (children[idx] != nullptr) {
                return children[idx];
            }
        }

        return nullptr;
    }
};

inline uint32_t get_svo_entry(
        bool is_far,
        uint16_t child_table_pointer,
        uint8_t valid_mask,
        uint8_t leaf_mask
        )
{
    uint32_t entry = 0;
    entry |= is_far ? (1 << 31) : 0;
    entry |= (uint32_t)(child_table_pointer & 0x7FFF) << 16;
    entry |= (uint32_t)(valid_mask) << 8;
    entry |= (uint32_t)leaf_mask;

    return entry;
}

void generate_svo_bank(std::vector<uint32_t>* bank_entries, octree_node* root)
{
    std::queue<octree_node*> bfs_queue;

    for (int idx = 0; idx < 8; idx++) {
        if (root->children[idx] != nullptr) {
            bfs_queue.push(root->children[idx]);
        }
    }

    while (!bfs_queue.empty()) {
        octree_node* node = bfs_queue.front();

        if (node == nullptr) {
            bfs_queue.pop();
        } else {
            bank_entries->push_back(get_svo_entry(
                                        false,
                                        bfs_queue.size(),
                                        node->valid_mask,
                                        node->leaf_mask
                                    )
            );
            bfs_queue.pop();
```

```
            for (int idx = 0; idx < 8; idx++) {
                if (node->children[idx] != nullptr) {
                    bfs_queue.push(node->children[idx]);
                }
            }
        }
    }
}

uint32_t svo_bank_depth(octree_node* root)
{
    if (root->get_descendant_count() < 0x1FFF) {
        return 0;
    }

    uint32_t max = 0;

    for (int idx = 0; idx < 8; idx++) {
        if (root->children[idx] != nullptr) {
            uint32_t cur = svo_bank_depth(root->children[idx]);
            if (cur > max) {
                max = cur;
            }
        }
    }

    return max + 1;
}

void generate_svo_entries(uint32_t* svo_size, uint32_t** svo_entries, octree_node* root)
{
    std::vector<uint32_t> entries;

    uint32_t bank_depth = svo_bank_depth(root);

    if (bank_depth > 0) {
        std::queue<octree_node*> bfs_queue, bank_queue;
        bfs_queue.push(root);

        while (bfs_queue.size() > 0) {
            octree_node* node = bfs_queue.front();

            if (node->get_depth() < bank_depth) {
                entries.push_back(get_svo_entry(
                                    false,
                                    bfs_queue.size(),
                                    node->valid_mask,
                                    node->leaf_mask
                                )
                );
                for (int idx = 0; idx < 8; idx++) {
                    if (node->children[idx] != nullptr) {
```

```
                        bfs_queue.push(node->children[idx]);
                    }
                }
            } else {
                entries.push_back(get_svo_entry(
                                    true,
                                    bfs_queue.size() + bank_queue.size(),
                                    node->valid_mask,
                                    node->leaf_mask
                                )
                );
                bank_queue.push(node);
            }

            bfs_queue.pop();
        }

        uint32_t number_of_banks = bank_queue.size();

        std::vector<uint32_t>::iterator far_ptrs = entries.end();
        std::vector<uint32_t> bank_entries;

        while (!bank_queue.empty()) {
            octree_node* node = bank_queue.front();
            uint32_t far_ptr = bank_queue.size() +
                               number_of_banks +
                               bank_entries.size();
            entries.push_back(far_ptr);
            bank_queue.pop();

            generate_svo_bank(&bank_entries, node);
        }

        entries.insert(entries.end(), bank_entries.begin(), bank_entries.end());
    } else {
        entries.push_back(get_svo_entry(
                            false,
                            0x0001,
                            root->valid_mask,
                            root->leaf_mask
                        )
        );
        generate_svo_bank(&entries, root);
    }

    *svo_size = entries.size();
    *svo_entries = (uint32_t*)malloc(*svo_size * sizeof(uint32_t));
    std::copy(entries.begin(), entries.end(), *svo_entries);
}

int generate_svo(
    uint8_t* buffer,
    uint32_t dim,
```

```
    uint32_t* svo_size,
    uint32_t** svo_entries
) {
    octree_node* octree = new octree_node(buffer, dim, dim, 0, 0, 0, nullptr);

    generate_svo_entries(svo_size, svo_entries, octree);

    delete octree;
    return 0;
}
```