Asbjørn Engmark Espe

# Real-Time Ray Tracing of Animated Sparse Voxel Octrees on FPGA

Master's thesis in Cybernetics and Robotics

June 2019

Master's thesis

NTNU
Norwegian University of Science and Technology
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of
Science and Technology

# Real-Time Ray Tracing of Animated Sparse Voxel Octrees on FPGA

Asbjørn Engmark Espe

Master of Science in Embedded Systems

Submission date: June 2019
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 rendering. 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 explored animation of SVO data from a theoretical standpoint, and formulated a general technique suitable for hardware implementation to achieve this. In the master's thesis, the student will continue within the same field of research, and:

- Review existing literature on the subject of hardware ray tracing.
- Investigate whether a hardware implementation for real-time ray tracing of SVO data is feasible, and formulate the specification and design of such a system.
- If feasible, demonstrate hardware ray tracing of SVO data. Explore the possibility of extending such a hardware implementation to support animation of SVO data.

# 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 known as the 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. In the project thesis, a general method for animation of such SVO models was derived.

In this master's thesis, a hardware accelerator for ray tracing of animated sparse voxel octrees is designed and largely implemented. The hardware implementation runs on an FPGA, and its design employs the method from the project thesis in in order to facilitate animation. The proposed solution also makes use of two additional established algorithms as foundation: an efficient method for traversal of octrees, and a memory-efficient data structure scheme for storing SVO data.

Initially, an analysis of requirements is conducted and a specification of the desired system is introduced. The system design is then derived, in which the functionality of the system and its constituent modules are determined and discussed. Ultimately, an implementation of this design is presented and examined. Certain parts of the design were not fully implemented due to time constraints, such as the modules that provide the arithmetic operations necessary for animation. The overall result is the system design of a fully-fledged hardware accelerator for real-time ray tracing of animated SVO models, and a partial implementation of this design.

A number of implementation configurations are tested. Running at 100 MHz with 16 SVO traversal cores, the implementation is shown to be capable of rendering static SVO models with a frame rate of 26.91 Hz at $1280 \times 720$ resolution. With a sufficient number of traversal cores, the implementation appears to be limited by external factors—namely the IO performance of the development board. It is concluded that real-time performance rendering animated SVO models is achievable.

# Sammendrag

Ray tracing—eller *strålesporing*—er en teknikk benyttet i datagrafikk med det mål å tegne en virtuell verden bestående av tredimensjonale volumetriske modeller. Disse modellene kan være uttrykt som geometriske objekter, eller som abstrakte datastrukturer slik som den optimaliserte voxel-baserte modelltypen kalt sparse voxel octree (SVO). En av hovedbegrensningene forbundet med strålesporing av slike SVO-modeller i dag er at datastrukturen i utgangspunktet ikke har noen støtte for animasjon. I prosjektoppgaven ble det utviklet en generell metode for animering av slike SVO-modeller.

I denne masteroppgaven blir en hardwareakselerator for strålesporing av animerte SVO-modeller spesifisert, utviklet og, i stor grad, implementert. Implementasjonen kjøres på en FPGA, og spesifikasjonen benytter metoden utviklet i prosjektoppgaven for å legge til rette for animasjon. Som grunnlag benyttes ytterligere to etablerte algoritmer: en effektiv metode for traversering av octree-strukturer, og en minne-effektiv datastruktur for lagring av SVO-data.

I første omgang utføres en analyse av systemets krav, og en spesifikasjon av den ønskede løsningen presenteres. Systemets utforming utledes deretter, hvor dets funksjonalitet og interne moduler diskuteres og bestemmes. Til slutt presenteres en implementasjon av dette systemet. Visse deler av systemets implementasjon ble ikke ferdigstilt grunnet tidsbegrensninger, inkludert deler av funksjonaliteten knyttet til animasjon. Resultatet er en komplett spesifikasjon og utforming av et system for hardwareakselerert strålesporing av animerte SVO-modeller i sanntid, og en delvis implementasjon av denne spesifikasjonen.

En rekke ulike konfigurasjoner av implementasjonen testes. Ved en klokkefrekvens på 100 MHz og med 16 kjerner for SVO-traversering, viser systemet seg i stand til å tegne statiske SVO-modeller med en bildefrekvens på 26.91 Hz og en oppløsning på $1280 \times 720$. Dersom et tilstrekkelig antall kjerner for SVO-traversering implementeres, blir det demonstrert at systemet begrenses av eksterne faktorer—nemlig ytelsen forbundet med IO på utviklingskortet. Det konkluderes at sanntidsytelse under tegning av animerte SVO-modeller er oppnåelig.

# Preface

This master's thesis shares the subject of research—and in some sense builds upon—the work done in the project thesis [1]. Its approach to the subject at hand, however, differs, and the thesis should therefore be regarded as its own work separate from the project thesis. Moreover, while much of the background matter is shared between the theses, the actual work done and the results presented are thoroughly different in nature. The idea behind the project and master's thesis was conceived in the autumn of 2017 during a conversation after class with my lecturer at the time, Øystein Gjermundnes. I presented my idea to him, and he enthusiastically encouraged me to pursue it as a subject for a project and master's thesis. He also agreed to take the role as my co-supervisor when the time came. I formulated the general gist of the problem statements for both theses around this time.

Much thought has been put into determining how the workload should be divided between the project thesis and the master's thesis. After hearing some horror stories of students putting too much work into the project thesis, and leaving too little for the master's thesis, I ended up partitioning the subject matter into two distinct segments. The project thesis was to be an almost purely theoretical venture, in which some general method or technique would be derived. The master's thesis would be of a much more practical nature, and its work would be comprised of a distinctly different set of problems related to implementation of a system which may incorporate the work done in the project thesis in some way. This distribution of work ensures that both the project thesis and master's thesis may be treated as independent works.

The reason that ray tracing has been chosen as the subject matter for both theses is that I have had a fascination for the technique for many years. Ray tracing appeals to me because of the elegance and simplicity of its implementations—qualities that seem to go missing from traditional rasterisation rendering implementations. This thesis has a particular focus on hardware design of a ray tracing system. The process of designing a digital hardware system of substantial size is very different from the software design flow I'm used to. A considerably larger chunk of the time must be set aside for planning and verification of the system, since the turnaround time for trying and failing is significantly longer. Digital hardware design is challenging, but at the same time highly rewarding; it is immensely satisfying to see the module you have been designing for the last week actually working as intended.

My principal supervisor for this thesis, Sverre Hendseth, has assumed a more supplementary role during the design work than he had in the project thesis. This is mostly due to the fact that his field of expertise is not directly relevant to the work. The occasional meetings I have had with him have been jovial check-ups and geared towards how I am doing with the report work. It should be noted, however, that the

tips he has given me regarding general work planning and organisation have helped considerably during the planning phase of the project.

In his capacity as my co-supervisor, Øystein Gjermundnes has had a much more prominent part advising me over the course of this thesis than he had in the project thesis. He has given me many hardware design tips that have helped me along the way. Moreover, he arranged for me to borrow the development board employed for the design implementation, so that a time-consuming application process could be avoided.

To sum up, I feel I have been very independent during both the design and implementation work, and the writing of the thesis itself. That said, whenever I have been stuck—either technically or motivationally—my supervisors have helped me out with good advice and encouraging words. They have also passionately encouraged me to pursue a PhD degree after this thesis, which I very much look forward to. I therefore want to extend my gratitude to my supervisors, Sverre and Øystein. As was the case for the project thesis, I have had the privilege of doing most of the design work on a personal workstation the offices of *Arm Norway*, with access to unlimited amounts of caffeine. I wish to thank the wonderful people at Arm who have given me this opportunity. Lastly, I would like to highlight my appreciation for my family and friends. Especially my girlfriend, Rachel, who has consistently kept my spirits up for the duration of the project.

# Contents

# Chapter 1

# Introduction

> I do think that some form of ray tracing—forward tracing or reversed tracing—rather than forward rendering will eventually win because there's so many things that just get magically better there. (...) And there's a lot of work, lots of smart people, lots of effort, and lots of great results coming out of it. Eventually ray tracing will win, but it's not clear exactly when it's gonna be.
>
> — *John Carmack, 2011* [2]

The idea of real-time hardware-accelerated ray tracing has by many been described as the *holy grail* of computer graphics [3][4]. In the autumn of 2018, Nvidia threw a curve ball and confounded an entire field with the launch of its newest series of graphics cards which featured this elusive technology [5]. The hype surrounding these latest developments in the field seems to have somewhat diminished in the latest months, and many have even outright dismissed ray tracing as a useless "gimmick" [6][7]. It is understandable that from a video game enthusiast's standpoint, the features may not have borne much fruit. They have remained largely unexercised by the many video games that have been released in the meantime.

Nonetheless, the fact that Nvidia has gone all-in on this technology is nothing but exciting for someone who has been following the ray tracing scene closely for many years. In fact, the new and shiny Vulkan graphics API—which may be regarded as the spiritual successor to OpenGL in many respects—actually contains extensions that expose support for ray tracing [8]. This means that any graphics accelerator vendors that wish to fully support the Vulkan API and its applications are required to also support ray tracing. Moreover, in the context of these recent developments, ray tracing should not be regarded as a replacement for rasterisation rendering. Rather, it ought to be viewed as a complementary technology that may be employed in conjunction with rasterisation rendering to improve visual fidelity.

Of course, no one should expect such paradigm shifts of established technology to happen overnight. It may take many years before the rest of the industry decide to follow Nvidia's new direction—it might not even happen at all. But the fact that one of the major players in the industry have put so much on the line for this technology is intriguing. In any case it shall be very exciting to follow the many breakthroughs

that may happen in the field if the technology gains a foothold. Will the other vendors follow in Nvidia's footsteps or reject the technology entirely?

### About this work, and its relation to the project thesis

This master's thesis is inspired by—and, to some extent, builds upon—the work done in the specialisation project during the autumn of 2018. It may be viewed as a continuation of this work, although the reader should keep in mind that its approach to the subject of study varies, and thus the nature of the results and their implications will be entirely different. In the project thesis, a general technique for animation of sparse voxel octrees was introduced. In this master's thesis, the method will be treated as previous work, and employed in a fully-fledged hardware ray tracing system. The project thesis report is included in the attached ZIP file, as described in Appendix A.

Another point worth noting is that there does not appear to be an established and universally agreed-upon name for the specialisation project that is completed in the first half of the fifth year of study at NTNU. In this thesis, however, it will consistently be referred to as the *project thesis*. The project thesis [1] will also be explicitly cited in the text if deemed necessary.

Much of the background matter, as well as the research context and chosen algorithms are useful and relevant for both the project thesis and this thesis. It would be a wasted effort to rewrite all these relevant sections for this thesis. Therefore, certain relevant sections of text have been extracted, copied, or paraphrased from from project thesis and presented here. Whenever this occurs, it will be clearly stated either directly in the section or in an introductory section.

### Thesis outline

This thesis consists of nine chapters and a set of appendices. Chapters 1 to 4 are background and theory chapters. These contain no results, but serve to bring the reader up to speed on the concepts and theory that is required to understand the results. Chapter 2 contains general background theory, while Chapter 3 presents the general fields of research that this thesis is a part of. In Chapter 4, the algorithms upon which the solutions and results are built are presented in detail.

Chapters 5 to 8 are the result chapters, in which the work done as part of this project and its results are presented and discussed. The development methodology in this master's thesis is loosely based on the waterfall model, and these four chapters each represent one or more stages of the model. In Chapter 5, a high-level requirement analysis of the desired features of the system is presented. Chapter 6 introduces the system design and a specification of its modules is formulated and discussed. The hardware implementation of this system is presented in Chapter 7. The results, and a discussion of these, are presented in Chapter 8.

### 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 [9][10][11], video games [12][13][14], computer-aided design (CAD) [15][16][17], graphical user interfaces (GUIs) [18][19], and special effects for motion pictures (SFX) [20][21].

In this chapter, relevant background information will be presented, starting with a short account of the eventful history of real-time computer graphics. The chapter is partly derived from its counterpart in the project thesis by Espe [1], with some sections taken directly from it. Sourced from the project thesis are Sections 2.1, 2.3 and 2.4, as well as Sections 2.2.1 to 2.2.3. These selected sections have been evaluated as very relevant for this master's thesis, and have such been copied and slightly adapted to be a part of this background chapter.

## 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* [22]. 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 [23][24].

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 [25] and Wolfgang Straßer [26], independently. The alternative to rasterisation was *ray tracing*, an image synthesis technique that was

first proposed by Arthur Appel in 1968 [27], but popularised after a paper by Turner Whitted in 1980 [28].

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 [29]. 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, further consolidating the position of rasterisation as the preferred method for image synthesis [30].

While rasterisation as a technique may have been more popular, there were attempts at hardware acceleration of ray tracing as well. In the late 1980s, a series of ray tracing demonstrations was run on a parallel processing architecture developed by the British semiconductor company Inmos [31]. The microprocessor architecture on which these demonstrations were run was known as the *transputer*. The main purpose of the transputer architecture was to allow for the construction of scalable concurrent systems, and the parallel nature of most ray tracing algorithms made ray tracing a prime candidate for the showcase of their technology.

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 power efficiency [32].

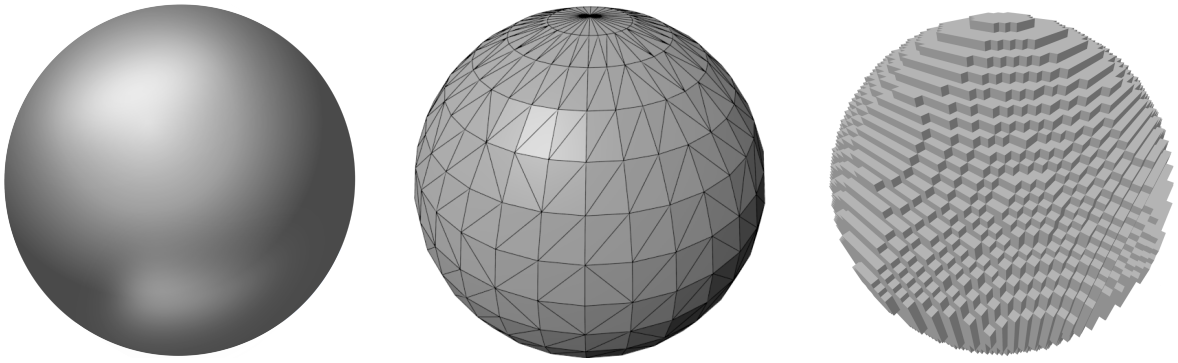
Even though rasterisation is currently the title-holder of the real-time computer graphics race, ray tracing continues to be relevant. In 2018, the popularity of ray tracing experienced a revival after Nvidia unveiled its newest range of GPUs—the *GeForce RTX* series [5]. It should be interesting to see to which degree this recent surge of main-stream attention will impact the future of computer graphics.

## 2.2 Fundamental concepts

The overarching goal of computer graphics is to utilise 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 to be 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 [33, pp. 66, 70][34, p. 176]. For instance, a sphere of radius 1, centred at the origin may be modelled perfectly by its analytical formulation 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 that it may 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 [34, pp. 555–559]. An illustration of a sphere modelled analytically is shown in Figure 2.1a.

### Polygonal models

A more common approach in computer graphics is to describe the object as 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* [34, 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 [14, p. 426][34, p. 177]. As a consequence of their vector graphics nature, polygonal models are particularly well suited for representing large, flat surfaces. A flat plane, for instance, can be modelled by only four vertices describing two triangles. A drawback is that polygonal models cannot accurately represent curved surfaces [33, p. 4]. High-fidelity models quickly become computationally expensive to render, since the level of detail is dependent on the number of polygons.

### Voxel models

Where polygonal models can be considered 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 consisting of a set of voxels. The word *voxel*, short for volumetric element, describes an object that 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 [34, 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 [35].

## 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 [14, 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.

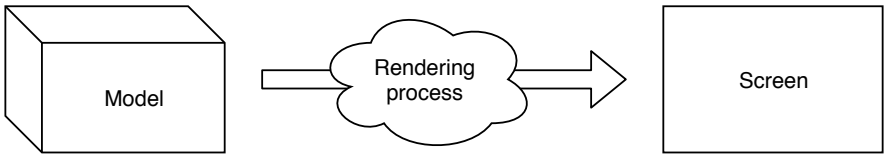



Figure 2.2: The rendering process as a black box.

The rendering process itself is often thought of as a black box process, as shown in Figure 2.2. However, in order to properly understand the different spatial representations, as well as the process of converting between them, there is a need to take a look under the bonnet in search for a more rigorous definition.

### 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* [33, p. 7] or *object space* [14, 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* [33, 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 [14, p. 428][33, p. 7]. All models to be rendered must be placed somewhere in the world space. View space, which is sometimes named *camera space* [33, 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*.

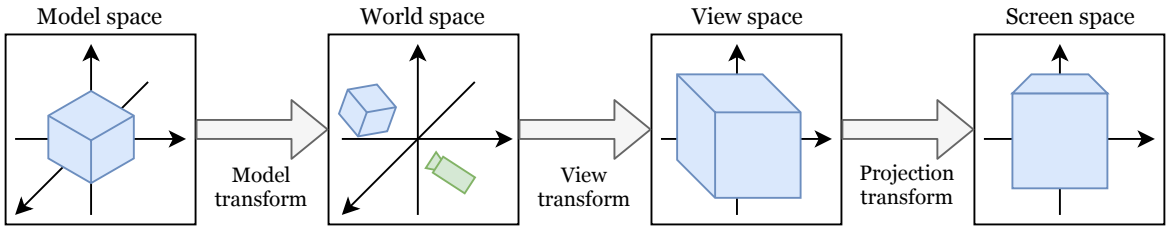



Figure 2.3: Main spaces in rendering. The camera itself is shown in world space.

### 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 each of them may 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*.

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).

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 |

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 [34, 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* [34, 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 [34, p. 615][36, 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 chiefly concerned with a certain type of animation termed *rigid-body animation*. The 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 [34, 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 may not be equal. An illustration highlighting this distinction is shown in Figure 2.4.

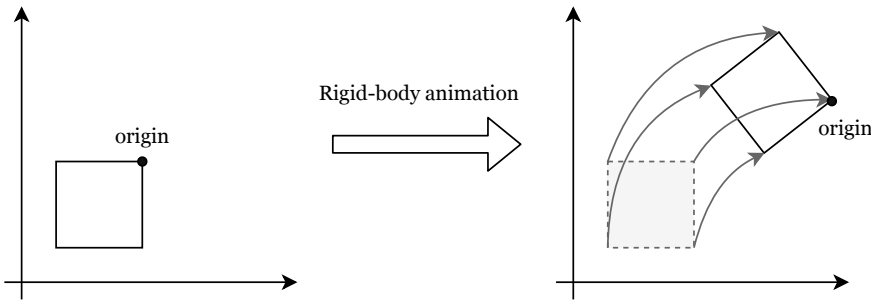



Figure 2.4: Example of rigid-body animation. After rotation and translation of the model, the vertices of the square remain unchanged in model space. In world space, however, their paths are different.

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 [34, p. 632]. The simplifications that are possible as a result of this fact laid the foundation for the work done in the project thesis [1].

### 2.2.4 Number representation

While integers may be sufficient for many applications in computer science, in computer graphics it is often necessary to perform calculations involving real numbers. Indeed, simply observing the linear algebra equations derived in Section 2.2.2, makes it apparent that non-integral numbers are central to the mathematical calculations that facilitate the rendering of three-dimensional scenes.

Representing real numbers accurately in hardware, however, is a complicated endeavour. With a limited number of bits available for number storage, there is a perpetual trade-off between accuracy and size. A larger number of bits set aside for number storage would increase the accuracy, but in turn also increase the size of the number in memory. For some numbers, such as irrational real numbers, total accuracy is not possible regardless of the number of bits employed in representation. This means that the application in which the numbers are used should be considered carefully before determining how numbers are to be represented.

In modern practice there exist two fundamentally different approaches to representing real numbers in computers—*fixed-point* and *floating-point*. Both of these representations are used extensively in hardware applications, and each one has its advantages and drawbacks. A brief introduction of the two representations is laid out in the following, and a comparison between them is presented subsequently.

#### Fixed-point number representation

The first and simplest of the two is the fixed-point number representation. Real numbers are stored in fixed-point by splitting the number into two parts—an integer part and a fractional part. In other words, an $n$-bit fixed-point representation allocates a fixed number of bits, $n_i$, to represent the integer part, and employs the rest of the bits, $n_f$, to hold the fractional part [14, p. 216]. This structure is illustrated in Figure 2.5.

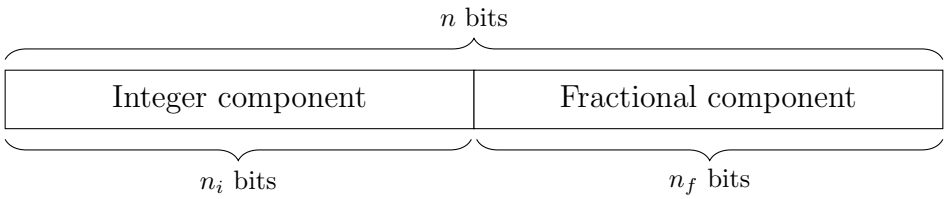



Figure 2.5: An $n$-bit fixed-point number. The representation uses $n_i$ bits to hold the integer component, and $n_f$ bits to hold the fractional component. In order to support negative numbers, two's complement representation is often used.

The number of bits chosen to represent each of the two components in an $n$-bit fixed-point number varies from application to application. In some cases, there is a

need for great precision in terms of the fractional part, which requires that a relatively large number of bits must be set aside to represent this part of the number. In other cases, fractional accuracy is not as important as the need to represent large absolute values. The integer part of the number could then be favoured in terms of allocated number of bits. Making a choice in this regard is up to the designer of the given system. A number of examples of different bit partitions are given in Table 2.2.

Table 2.2: Examples of different subdivisions of a fixed-point number. The numbers shown are in two's complement form in order to support negative values.

| Bit widths | | | Integer range | Fractional precision |
|---|---|---|---|---|
| Total | Integer | Fraction | | |
| 8 | 4 | 4 | −8 to 7 | $6.25 \times 10^{-2}$ |
| 16 | 4 | 12 | −8 to 7 | $2.4 \times 10^{-4}$ |
| 16 | 8 | 8 | −128 to 127 | $3.9 \times 10^{-3}$ |
| 16 | 12 | 4 | −2048 to 2047 | $6.25 \times 10^{-2}$ |
| 32 | 8 | 24 | −128 to 127 | $5.96 \times 10^{-8}$ |
| 32 | 16 | 16 | −32 768 to 32 767 | $1.5 \times 10^{-5}$ |
| 32 | 24 | 8 | −8 388 608 to 8 388 607 | $3.9 \times 10^{-3}$ |

### Floating-point number representation

The second approach, and the most widespread method for binary representation of real numbers today, is the floating-point number representation [37, p. J-13]. Unlike the fixed-point representation, this number format is standardised. It is governed by the IEEE-754 standard, more verbosely known as the *IEEE Standard for Floating-Point Arithmetic* [38]. The original standard dates back to 1985, but the specification upon which this thesis is based is the newer and revised version that was published in 2008.

The floating-point format employs *exponential* notation of numbers. This means that it utilises the concept that any real number $x$ can be written on the exponential form shown in Equation (2.5), where $s$ is a real number in the range $[1, 2\rangle$ and $e$ is an integral number.

$$x = s \times 2^e \tag{2.5}$$

In other words, as long as the base of the exponentiation is agreed upon, the number $x$ can be represented by only storing the significand $s$ and exponent $e$. Moreover, as mentioned above, the significand only needs to hold values in the range $[1, 2\rangle$. This is a consequence of the exponentiation base being 2—if the significand were outside this range, the exponent could be adjusted to bring it back within the range. In binary form, this means that the leading number of the significand will always be 1, and need not be stored explicitly [39, p. 12].

This simplification does, however, raise the question of how to store negative numbers. Two's complement would not really work in this case. The IEEE-754 standard defines the structural layout of a floating-point number as shown in Figure 2.6. The illustration shows that one bit of the number has been reserved to store the sign. The rest of the bits are used to store the exponent and significand.

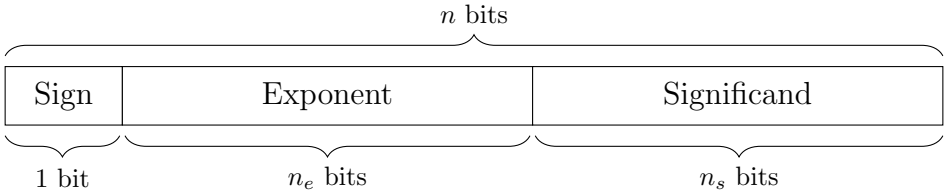



Figure 2.6: An $n$-bit floating-point number. The representation uses 1 bit to signify the sign of the number, $n_e$ bits to store the exponent, and $n_s$ bits to hold the significand.

The floating-point specification additionally defines two special values that may be encoded as a floating-point number. These values are signalled by setting all bits of the exponent $e$ to one. The first of the two special values is the *not-a-number* value, referred to as NaN. The NaN value is used to represent uninitialised variables or for signalling invalid operations. The second value is the infinity value, which is used to represent results that are numerically unbounded. The specification states that if the significand $s$ is zero, the floating-point number signifies $\pm\infty$—where the sign bit determines sign. If the significand is not equal to zero, the number represent a NaN.

In Table 2.3, the most common floating-point formats are listed along with their specifications and range. Most relevant for this thesis is the *single-precision floating-point specification*, which has the standardised name `binary32`.

Table 2.3: The most common of the IEEE-754 formats. Numbers are taken from the *IEEE Standard for Floating-Point Arithmetic* [38, p. 13].

| Common name | Standard name | $n$ | $n_s$ | $n_e$ | Magnitude range | |
|---|---|---|---|---|---|---|
| | | | | | Min | Max |
| Half-precision | `binary16` | 16 | 10 | 5 | $2^{-14}$ | $2^{15}$ |
| Single-precision | `binary32` | 32 | 23 | 8 | $2^{-126}$ | $2^{127}$ |
| Double-precision | `binary64` | 64 | 52 | 11 | $2^{-1022}$ | $2^{1023}$ |
| Quadruple-precision | `binary128` | 128 | 112 | 15 | $2^{-16\,382}$ | $2^{16\,383}$ |

### Comparison of fixed-point and floating-point

While the floating-point number representation is the most widely used in computers today, the fixed-point representation does have a few conveniences that makes it

preferable in certain contexts. As will be further detailed in Chapters 5 to 8, both number formats are to be used in the solution in order to circumnavigate their individual weaknesses.

The main drawback of floating-point numbers when compared to fixed-point numbers, is that the former is a much more expensive and complex format to implement in hardware. Quite a bit of chip area is required in order to efficiently facilitate operations such as addition and multiplication of floating-point numbers. Fixed-point numbers, however, do not present a significant overhead in implementation. In fact, in hardware one may almost exclusively treat fixed-point numbers as integer numbers. This means that operations such as addition and multiplication of fixed-point numbers can be implemented at a much lower cost than for the floating-point representation. The differences in area can be seen in Table 2.4.

Table 2.4: Differences in chip area requirements between fixed-point and floating-point implementations. Numbers are in $\mu m^2$, and taken from [37, p. 29].

| Type | 16-bit addition | 32-bit addition | 32-bit multiplication |
|---|---|---|---|
| Fixed-point | 67 | 137 | 3495 |
| Floating-point | 1360 | 4184 | 7700 |

On the other hand, an advantage of floating-point numbers is that they have an enormous range compared to fixed-point numbers of the same bit width. This fact is evident by inspecting and comparing Tables 2.2 and 2.3, and is a huge contributing factor to their widespread use. In addition, the exponential nature of floating-point numbers makes them well-suited for many numerical applications. As an example, it is rare that one needs the precision of millionths when dealing with numbers in the order of a billion. There are, however, some situations in which these precisions are needed, and employing floating-point numbers may then lead to rounding errors.

### 2.2.5 Task scheduling

In computer graphics, parallelised systems are commonplace. Whenever a parallelised system is encountered, either in hardware or software, a question is raised concerning how one might best utilise this parallelisation. And a decision regarding the method by which tasks should be ordered and distributed in the parallelised system must be made. The study of how tasks should be ordered, and in which manner they should be distributed among execution units is often termed *task scheduling*.

Normally, the term refers to a collection of methods used to determine the order of execution of tasks on a single or multiple execution units, as well as the analysis of such methods—for instance the study of which methods that yield the highest utilisation. In this thesis, the term is used to mean a slightly different concept. As will be further detailed in the later chapters—specifically Chapters 6 and 7—the system in this thesis is made up of several duplicated execution cores that perform a similar function on a set of data. The term scheduling is then used to describe the process of selecting which

cores that should be assigned new tasks. This distinction is subtle, but should become apparent through the examples presented in the following sections.

Another point to note is that the field of scheduling is very broad, and there exists a host of different scheduling algorithms. Most are not relevant to this thesis, so only a few algorithms will be presented and discussed. Introduced in the following are the round-robin algorithm and the first-available algorithm.

### Round-robin

The simplest form of scheduling is to order all execution units, and simply wait for the next execution unit in line to be ready before assigning the next task. This approach is called *round-robin* scheduling, and is illustrated in Figure 2.7. In the figure, the `select` signal cycles which core is selected in a predetermined order. If the currently selected core is ready, it is assigned a task—here represented by sending it a data packet on the `data bus`.

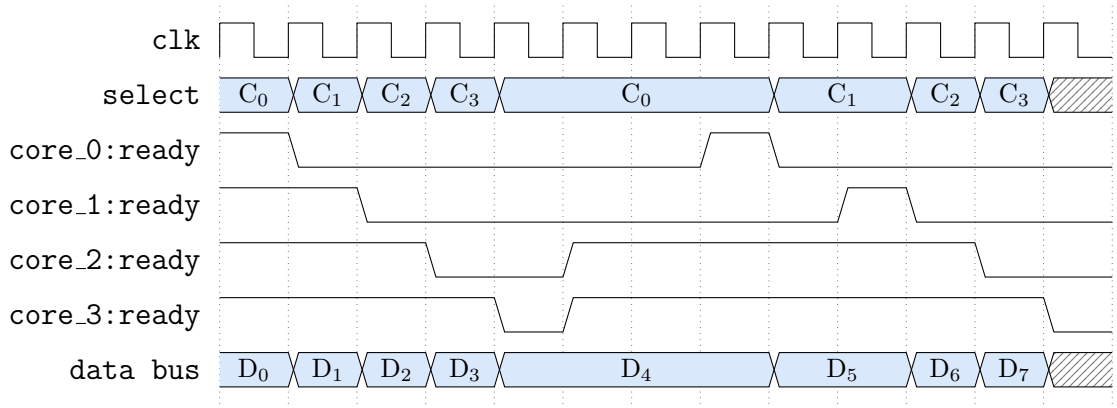



Figure 2.7: Example of round-robin scheduling.

Round-robin scheduling has the advantage that it is exceptionally simple to implement into a system. The select signal may be realised as a counter that is incremented for each job, and reset to 0 whenever it reaches the maximum index. A drawback of this method is that a single execution unit may hold up the entire system if it is busy with a job while next in line. The issue is highlighted by the diagram above, where the system has to wait for core $C_0$ to be ready for several clock cycles before continuing.

### First-available

Another method of scheduling tasks among execution units is to assign the next task to the first execution unit that becomes available. This method is demonstrated in Figure 2.8, where it is shown that the `select` signal always selects the first core that is ready and assigns the next data packet to it.

The scheme is a bit more involved to implement efficiently, especially in hardware. However, it solves the issue raised by round-robin scheduling, where a single execution unit may impede the rest of the system. This is substantiated by the diagram, where the next task is assigned to core $C_2$ when it becomes ready before $C_0$ and $C_1$. In a comparison of the two diagrams, one can see that the first-available scheduling method

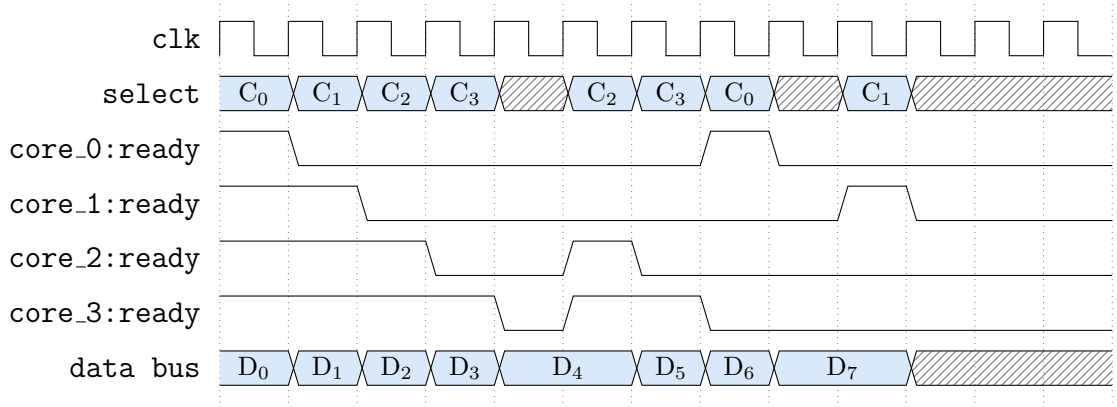



Figure 2.8: Example of first-available scheduling.

is finished two clock cycles before the round-robin method in the same setting—evidence to the claim that it is more efficient and leads to higher utilisation.

### 2.2.6 Memory caching

In the early years of digital hardware design, computation cores and memory performance increased in lockstep; clock speeds of both processors and memories were improved by a significant figure each generation. As the field matured, it became apparent that memory clock speed could not be increased indefinitely, and that further improvements would not match the performance increase of the computation cores. In later years, a considerable gap has been established between processor core speed and main memory speed [37, p. 78]. This gap, illustrated in Figure 2.9, leads to many situations where the computation cores of a hardware design will have to await data from memory before computations requiring such data can continue [40, p. 82].

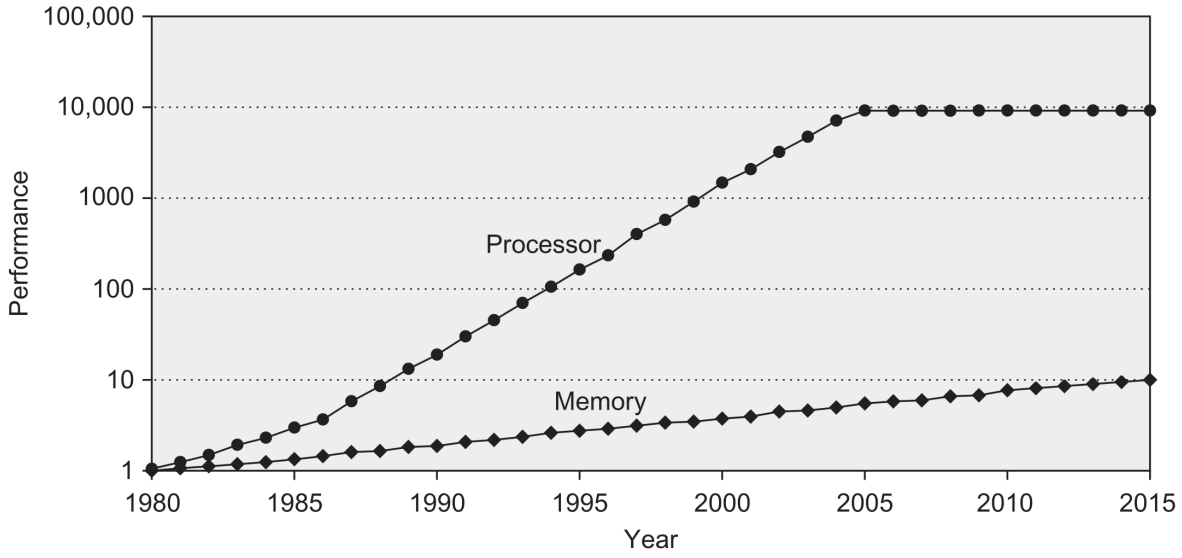



Figure 2.9: The gap in performance between single-core processors and main memory. Image taken from [37, p. 80].

In computer graphics, performance is of the essence. It is therefore crucial that

memory is implemented as efficiently as possible, and that the effects of this performance gap are minimised. The *memory cache* is a digital circuit designed to this end. In essence, the cache module will try to predict which memory addresses that may be used in the future, and subsequently store these addresses in a smaller, faster memory. The processor may then fetch the data from the cache instead of the main memory, which in turn leads to an overall reduction of latency. The selection of which memory addresses that are to be kept in the cache has spawned a whole field of study in itself. However, since the processor will often reuse the same memory addresses multiple times in short order, most policies entail keeping track of addresses that have been recently used [40, p. 83].

There is a trade-off between cache size and cache speed; the access latency of a cache generally increases with its size. A small and fast cache will often result in many *cache misses*, while a larger albeit slower cache will result in a higher degree of *cache hits*. Since the desired behaviour is a fast cache with as many cache hits as possible, many systems have multiple levels of caches between the main memory and the processor, where each tier is smaller and faster than the previous. This tiered organisation of caches is termed the *cache hierarchy*, and the general idea is shown in Figure 2.10.

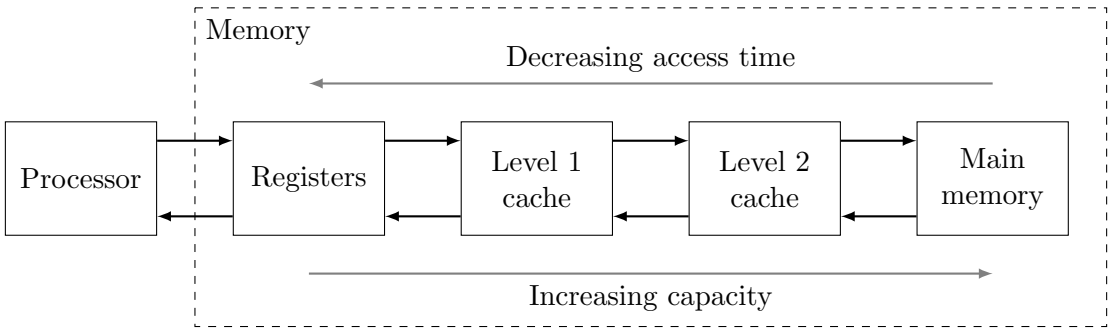



Figure 2.10: An example of a cache hierarchy. Level $n$ caches may be abbreviated L$n$.

One may divide caches into three main categories on the basis of their operation. These categories and their advantages and drawbacks are detailed in the following.

### Direct-mapped cache

A *direct-mapped* cache is a cache in which every memory address may only be placed at a specific position [37, p. 81][40, p. 306]. In other words, the processor will only have to check a single position in the cache in order to determine if the word it is trying to access is available, or if it must be fetched from a higher level memory. This has the advantage that the processor very quickly can determine if the cache has the memory it is requesting. A downside is the fact that multiple words share the same position in the cache, and that a cache line may be prematurely exchanged with another.

### Fully-associative cache

In contrast to the direct-mapped cache, a *fully-associative* cache lets any memory address reside in any location in the cache [37, p. 81]. This reduces the issue of multiple cache lines contesting for the same location in the cache substantially. However, since the processor now has to search through the entire cache in order to determine if the

data it requests is available, this scheme does make the overhead for retrieving words from the cache a bit larger.

#### Set-associative cache

The most popular scheme today is a combination of two extremes detailed above—the *set-associative* cache [37, p. 81][40, p. 308]. Caches belonging to this category are divided in to a number of *sets*, and each memory address is associated with a single set. When a newly fetched cache line from memory is to be placed in the cache, it may reside anywhere within its associated set.

This cache type improves upon the fully-associative cache since the processor only has to search through a single set in order to determine whether the requested memory resides in the cache. Another advantage is that the contesting issue associated with the direct-mapped cache is reduced. On a cache miss, a newly fetched cache line may be exchanged with a number of different cache lines. As such, an appropriate *replacement policy* should be implemented [37, p. J-9][40, p. 309].

## 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 these 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 [34, p. 8]. Shown in Figure 2.11 is the general idea behind the scheme.

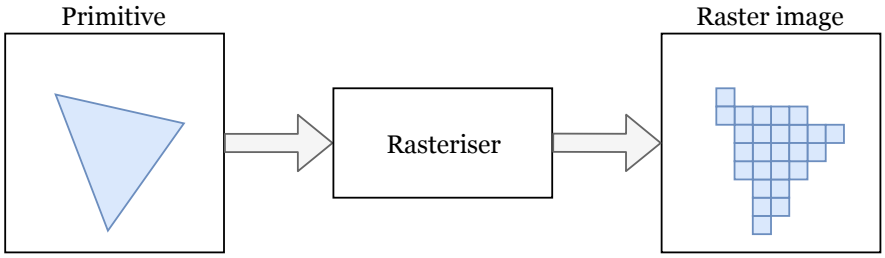



Figure 2.11: Rasterisation used for sampling a primitive.

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 additionally use some form of depth sorting in order to determine which primitive is in front of the other in the case of overlap [14, pp. 467–468][33, p. 19][34, 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 [34, pp. 176–177].

#### Scan line rendering

The most popular method facilitating the implementation of rasterisation is by 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 [34, 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 [27]. 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 [34, p. 548][33, p. 220]. The basic concept is illustrated in Figure 2.12.

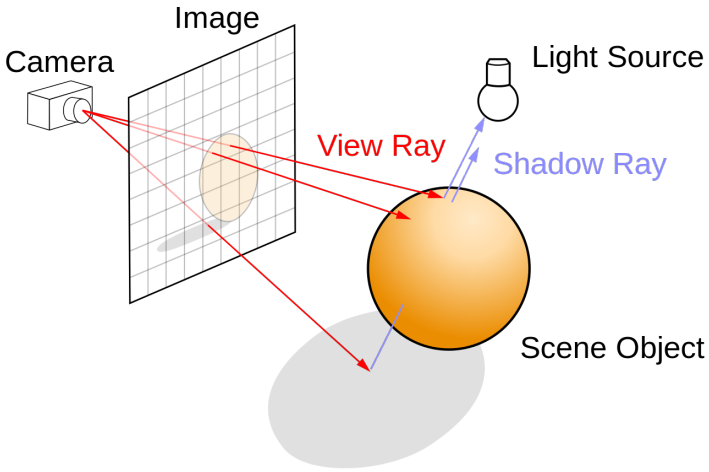



Figure 2.12: Ray tracing process. Image taken from [41].

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 introducing 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 [42].

### 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 2.13.

Figure 2.13: Examples of recursive ray tracing. The scenes are rendered with shadow, reflection, and refraction effects. Images taken from [43] and [44].

The general principle behind recursive ray tracing was pioneered by Whitted [28], 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.

- 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 the rendering of 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 [28].

## 2.4 Space partitioning

In the field of mathematics, *space partitioning* is the study of methods and techniques for efficient subdivision of space into partitions. As a branch of geometry, it 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.

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. These partitioning schemes are based on a common form of partitioning termed *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 [14, p. 436]. In this thesis, two BSP-derived structures will be presented. 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 [45]. 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 $N = 3$.

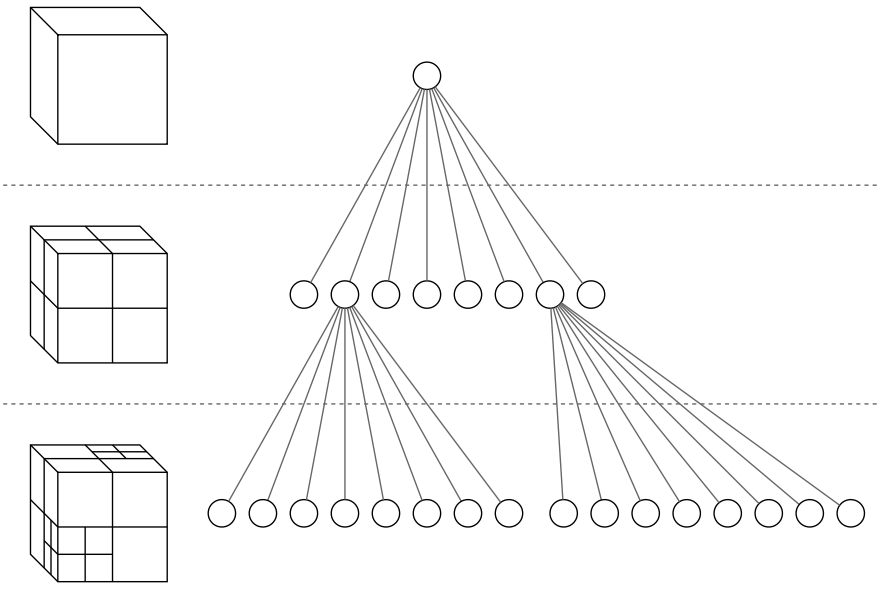


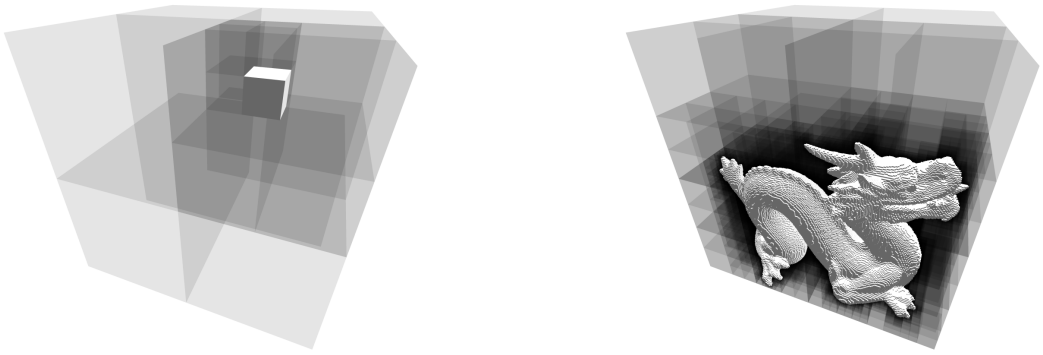
Figure 2.14: An illustration of the hierarchical structure of an octree.

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 space 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 [14, pp. 436–437]. An illustration is provided in Figure 2.14. In the illustration, a three-level octree is visualised, showing both the theoretical hierarchy, and the physical layout.

According to Brönnimann and Glisse [46], 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 [45][47], LoD optimisations [14, p. 439][48][49], and robotics [45][50]. Although the octree in itself is a very efficient format for subdivision of space, this does not automatically make it very 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 may consist of an octree where each terminal node is categorised as either filled or empty. Shown in Figure 2.15 are two voxel models encoded as sparse voxel octrees.



(a) Single cube (b) Stanford dragon [51].

Figure 2.15: Voxel models encoded in SVOs.

Sparse voxel octrees can be extended to contain almost any relevant data. The next step from the simple filled/empty scheme described above could be to include surface

normal vectors and colours for lighting calculations. Other attempts at expanding the format include adding data such as contour descriptions [35] and LoD optimisations [52] to further improve visual fidelity or optimise traversal.

There exist many established algorithms for traversal of octrees, and most of these methods translate well to traversal of 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 backtrack 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 node [53]. Specific algorithms for traversal are highly relevant will be presented and detailed in Chapters 3 and 4.

### Memory requirements

Compared to a three-dimensional array of voxel values, the SVO data structure will in many cases reduce the memory usage of a voxel model by several orders of magnitude. In the majority of models, there are vast regions of either filled or empty space. By utilising 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.

The merging of nodes is visualised in Figure 2.16. In the given case, the number of nodes after the reduction is less than half of the original count. Do note that the illustration shows a two-dimensional case—a quadtree—instead of a three-dimensional octree. The principle, however, is fully analogous to the three-dimensional case, and may in fact yield even greater reduction of node count.

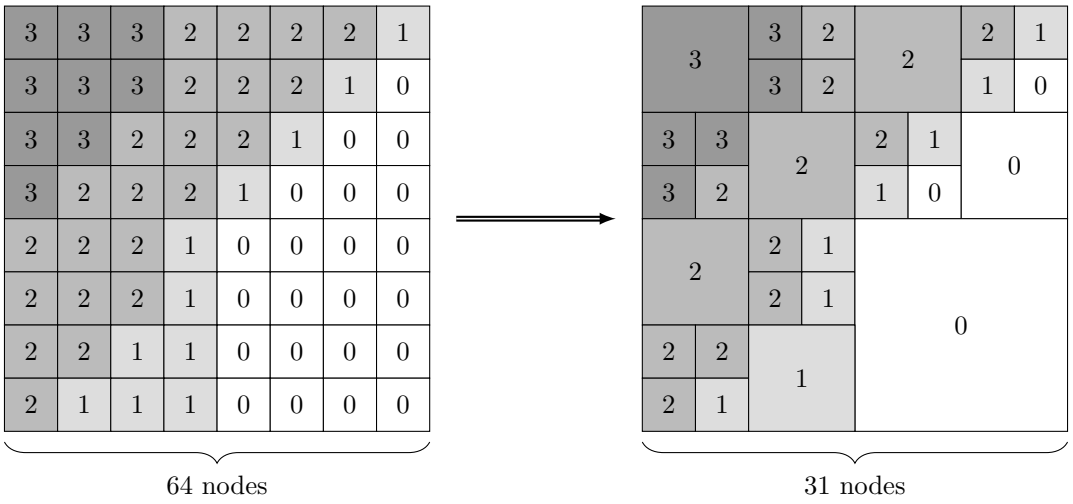



Figure 2.16: Merging neighbouring nodes sharing the same value leads to a significant reduction in number of nodes.

In Table 2.5, 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 uses *run-length encoding* (RLE) [54] to reduce model size. The third format is a memory scheme for SVOs based on the paper published by Laine and Karras [35]. The latter is used throughout the thesis and will be thoroughly described in Chapter 4. Illustrated by the table is 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 2.5: A selection of models and their file sizes when stored in different formats. All models originate from *Stanford 3D Scanning Repository* [51].

| 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

As a result of the data structure's recursive nature, the computational costs associated with the traversal of an SVO increase with the depth of the tree structure. In some cases, however, the need for spatial resolution is reduced. As an example, consider a case where a deep SVO model is rendered at a far distance. The octree traversal algorithm will at some point reach a depth where one voxel of the octree is smaller than a pixel on the screen. In this situation, the fine spatial resolution of the SVO model goes wasted, and will only tax the performance without yielding any tangible gain in image quality. In fact, it might even lead to worse quality as a result of adverse effects such as aliasing.

Fortunately, sparse voxel octrees may readily be extended to support *level of detail* (LoD) optimisations. A simple method would be to store the average colour in each node. This means that terminal nodes would still remain either completely filled with a colour, or completely empty, while intermediate nodes would store the average colour of all their children. An octree traversal algorithm may then stop its descent into the tree structure when a desired fidelity is reached.

More sophisticated solutions employing LoD optimisations to mitigate this problem may also be developed. One such method is presented by Jabłoński and Martyn [52] in a 2016 paper. Their method is a bit more involved, as they extend the sparse voxel octree format to include a set of redundant nodes, termed *helper nodes*. These nodes are then employed in the interpolation of model's the volumetric data. Their method is ubiquitous in that it is suited for both rasterisation and ray tracing of SVOs. Two illustrations of their LoD transition algorithm in action are shown in Figures 2.17 and 2.18.

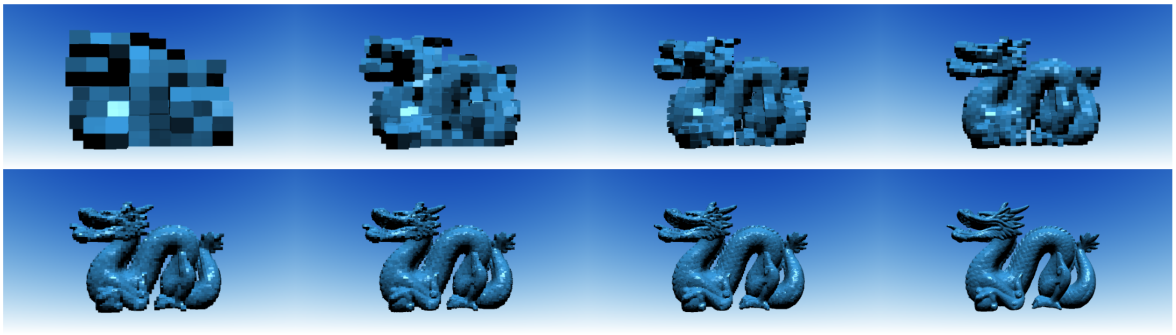


Figure 2.17: LoD optimisations of SVO data. Image taken from [52].

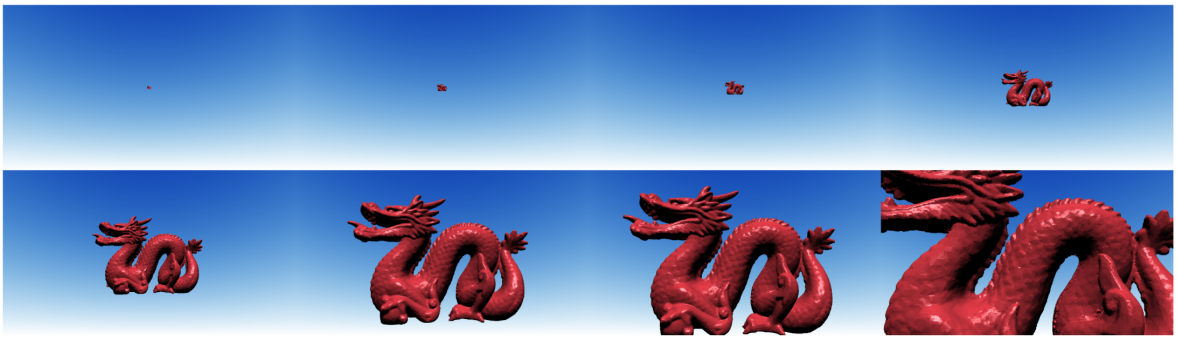


Figure 2.18: The LoD optimisations when viewed at different distances. The perceived quality is invariant, but performance gain may be significant. Image taken from [52].

### SVO model generation

The SVO models to be used with a hardware design may be generated in a very similar manner to the approach that was presented in the project thesis [1]. As such, the description in the following paragraphs is adapted from this source. The toolchain for model generation is illustrated in Figure 2.19 and will be discussed in the following.

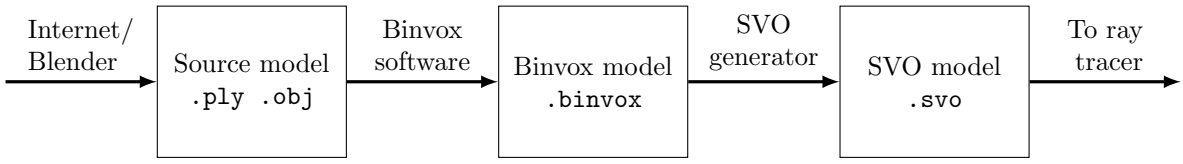



Figure 2.19: Toolchain for SVO model generation.

The polygonal source models are initially obtained from online resources such as *Stanford 3D Scanning Repository* [51], or created in the modelling software *Blender* [55]. These models are subsequently processed by the software *binvox*, which is a freely-distributed program written by Min [56]. The software reads a 3D model file in a widely-used format—in this case `.ply` or `.obj`—and from it produces a voxel model by means of three-dimensional rasterisation. The output voxel model is stored in a custom file format (`.binvox`), which is essentially a raw voxel format that employs the simple compression scheme *run-length encoding* (RLE) to reduce the file size [54][57].

The last step of the SVO generation toolchain is to generate and store the models on the format that will be described in Section 4.2. A custom-made program to perform

this conversion was developed as part of the project thesis work; its core functions are written in C++ and included in the attached ZIP file as specified in Appendix A. Briefly stated, 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.

## 2.5 Digital hardware design

Digital systems are at the core of modern life. Going back just 60 years, digital systems were only found in expensive computing systems and other niche applications [58, p. 31]. However, once the design and production of integrated circuits became cheaper and more viable, digital systems started appearing in wide variety of products. Today, everything from mobile phones and laptop computers to fridges and washing machines contain digital electronic hardware.

The discipline of designing and developing digital electronic hardware architectures is known as digital hardware design, or simply hardware design. The field encompasses everything that might be considered part of the formulation and development of digital integrated circuits. According to Mano and Ciletti [59], the process of digital design can be described as the translation of a functional specification or description of a circuit into a physical specification or description [59, p. 67]. The central stages of this process is shown in Figure 2.20, with the initial specification step highlighted.

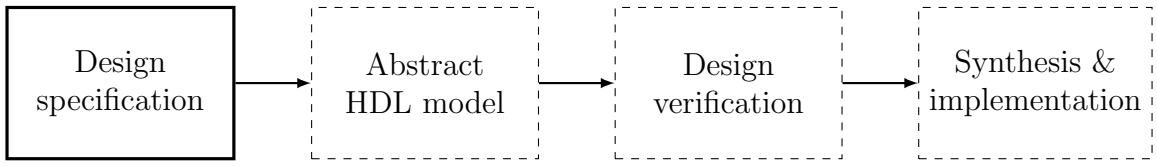



Figure 2.20: The main stages in the digital design process, simplified from [59, p. 363]. The specification step is highlighted.

When compared to the software design process, hardware designs often require a significantly more detailed specification, and considerably stricter and more rigorous forms of verification. Since bugs and errors can be very hard to discover in the implementation, in a typical design process today, more than half the effort may be devoted to testing and verification [58, p. 65]. However, the increased efforts are not for naught—a design realised in hardware will in most cases yield significant improvements in terms of performance when contrasted with a comparable design in software.

### 2.5.1 FPGA

A prevalent platform for prototyping hardware designs, and very relevant to this thesis, is the field-programmable gate array (FPGA). The FPGA is a packaged integrated circuit designed to be configured by the customer at some stage after manufacturing—hence bearing the name "field-programmable" [60, p. 15]. Its configuration is chiefly performed by the design and formulation of digital circuits in a hardware description language (HDL). The logical structure of a typical FPGA is shown in Figure 2.21.

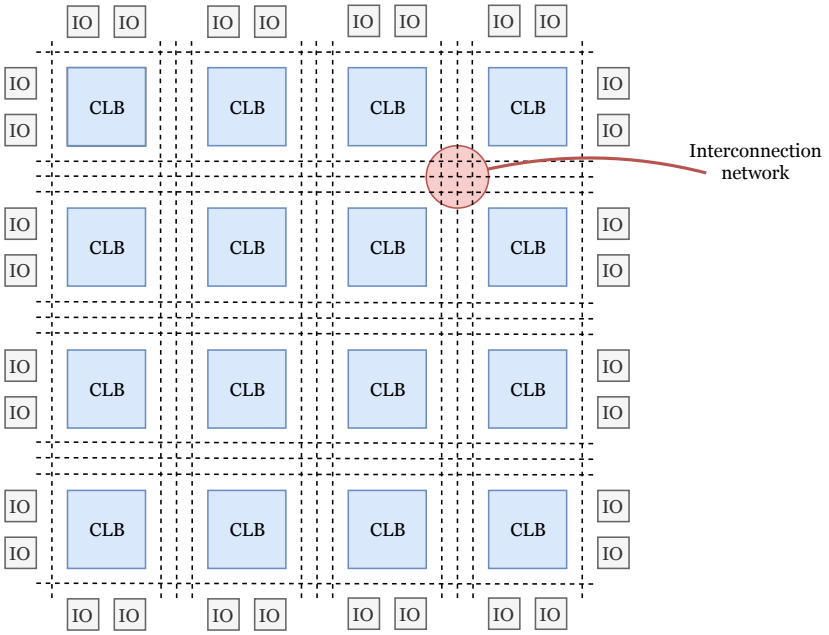



Figure 2.21: The logical structure of an FPGA. The CLBs may be connected to each other and to the IO blocks by programming the interconnection network.

Internally, an FPGA contains thousands of programmable units, termed *configurable logic blocks* (CLB), that may be configured and connected in order to compose digital circuits. Each of these logic blocks consists of several components such as look-up tables (LUTs), multiplexers, gates, and registers. Surrounding the CLBs is a programmable interconnection network that is employed to create electrical connections and thus build digital circuits. Running along the perimeter of the platform is a set of input and output blocks that may be coupled to the CLBs, so that the digital circuits programmed onto the FPGA may communicate with their surroundings [59, p. 332].

The FPGA is often viewed as an alternative to the application-specific integrated circuit (ASIC) [59, p. 68]. Early in the stages of system development, architects are often presented with a choice between these two implementation media. In general, an ASIC has higher performance and is more efficient in terms of area compared to an FPGA [61]. However, the latter allows for much easier prototyping and often increased effectiveness of verification, in turn leading to a shorter time to market. In some cases, especially in low-volume applications, the minimal cost of an FPGA may be preferable to investing in tooling for the manufacture of ASICs [58, p. 69]. Still, the two implementation media need not be mutually exclusive—an FPGA can be used in the development of applications that are to be manufactured as an ASIC at a later stage.

The FPGA often plays a central part in the design verification effort. Since a design synthesised and implemented on an FPGA can run at much higher speeds than a mere software simulation of the same design, verification can be done at a faster rate. However, compared to software simulation, the FPGA has little debugging capability [60, p. 28]. This means that, in an ideal setting, most major bugs and design errors would be rooted out before verification on FPGA begins.

### PYNQ-Z1

Alongside software modelling and simulations, the main development platform employed in this thesis is the *PYNQ-Z1: Python Productivity for Zynq-7000 ARM/FPGA SoC* [62] by Digilent Inc. This development board, shown in Figure 2.22, is a prototyping platform that contains a Xilinx FPGA alongside an Arm CPU, in addition to a wide variety of peripherals such as HDMI, Ethernet, and analogue audio.

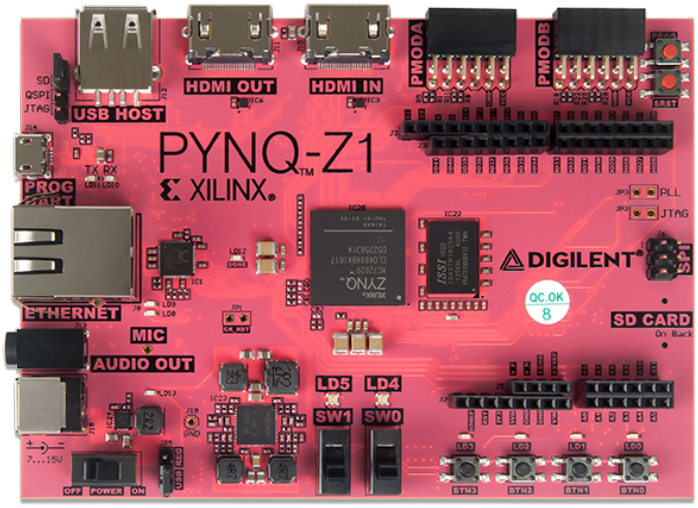



Figure 2.22: The Pynq-Z1 development platform. Image taken from [62].

Running on the board's Arm processor is a fully-fledged Linux system which hosts a Python 3.6 environment [63] including custom Python libraries specifically created for the Pynq [64]. In addition to the basic communication facilities provided by the Linux system, the Python environment may also be directly interacted with through the Jupyter interactive computing environment [65]. This interactive environment can be reached through a web interface and allows the user to easily run Python code on the system. The software driver which will be introduced as part of the system implementation in Chapter 7 makes use of the Jupyter interface. Its source code can be found attached in Appendix F.

The simplicity and elegance of the Pynq platform makes the development and verification of digital systems an untroublesome process. Hardware designs may be downloaded to the FPGA as binary files and subsequently instantiated as objects in Python scripts. Through the provided Pynq Python libraries, scripts may communicate with any hardware module on the FPGA system that is using the appropriate communication protocols. This means that a hardware design can, for instance, be employed as an accelerator for specific operations as a part of software code. Since the topic of this thesis is to create a hardware accelerated design for ray tracing, the Pynq system appears to be very well suited for the project.

Apart from the argument that the development board fits the use case of this project excellently, there is yet another reason for its selection: the board was physically available at the start of this thesis. In fact, co-supervisor Øystein Gjermundnes kindly lent out a Pynq development board—free of charge—to be used as part of this work, thus relieving the need for a cumbersome and time-demanding application process.

### 2.5.2 HDL modelling

It was briefly mentioned that hardware designs are usually formulated in a special machine-interpretable language known as a hardware description language (HDL). After a digital circuit has been modelled in the HDL, the code is read and synthesised by a computer program in order to realise the design as digital circuits and implement it on a physical medium. Moreover, the HDL description of a circuit's functionality can be abstract and implementation media agnostic—that is, without reference to specific hardware—and thereby freeing a designer to devote their attention to higher level functional detail rather than transistor-level detail [59, p. 68]. Revisiting the digital design process diagram introduced at the beginning of this section, the HDL model's can be found as shown in Figure 2.23.

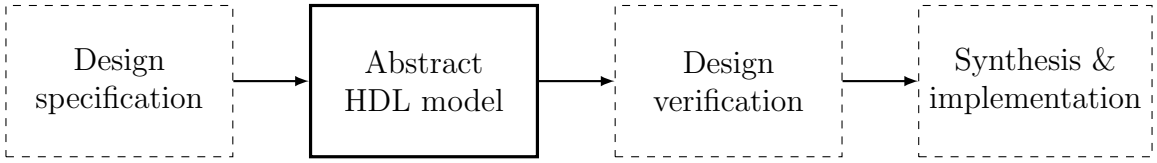



Figure 2.23: The main stages in the digital design process, simplified from [59, p. 363]. The HDL step is highlighted.

The hardware systems developed as part of this thesis are written in the SystemVerilog HDL. Initially introduced in 2002 as a set of extensions to the popular Verilog HDL, SystemVerilog shares the same C-like syntax as its precursor [60, p. 123]. Today, SystemVerilog is regulated as its own distinct language by the *IEEE Standard for SystemVerilog* [66]. Because the syntaxes of Verilog and SystemVerilog are akin, the latter should not present a huge challenge to learn for a hardware engineer already knowing Verilog.

Due to the limited language constructs to facilitate the process, verification environments in plain Verilog have to be developed manually. This means that verification of complex designs can be strenuous and time-consuming [60, p. 123]. SystemVerilog was introduced to yield a productivity boost in design and validation, and hence includes many extensions to the Verilog language with the overarching goal of allowing more efficient verification of designs [66, p. 8]. Because of this approach, the language is often referred as a type of hardware description and verification language (HDVL) [66, p. 8][60, p. 123]. In its capacity as a HDVL, SystemVerilog introduces several new data types, it allows dynamic memory allocation, it has more explicit constructs for separating sequential and combinational logic, and it includes native support for concurrent assertions and functional coverage—to list a few of its features.

SystemVerilog was chosen as the design, development, and verification language for this thesis mainly because of the author's and supervisor's familiarity with it. If a different language were to be employed, the process of learning how to use it would require a significant amount of time that could otherwise have been spent on development. As reference works, the books by Mano and Ciletti [59], Mohamed [60], and Cerny et al. [67]—as well as the SystemVerilog specification [66]—were used throughout the development process whenever needed.

### 2.5.3 Verification

Hardware designs are much more prone to errors than software designs. Partly because of the increased complexity brought on by the concurrent nature of hardware designs, meaning that the designer will have to consider multiple operations happening in parallel. But also because the synthesis tool—whose function is to turn an abstract HDL model into an explicit RTL definition—may make incorrect assumptions about the design. For instance, the synthesis tool can infer latches that make the entire design malfunction. These kinds of bugs introduced by the logic synthesiser are often not detectable through software simulations, and might only appear after synthesising and implementing the design.

Verification encompasses all methods used to verify the correctness of a hardware design. While the design to be validated—often called the *design under test* (DUT)—is assumed to be a synthesisable design written in RTL code, the verification of it can take many forms. Its place in the digital design process is shown in Figure 2.24.

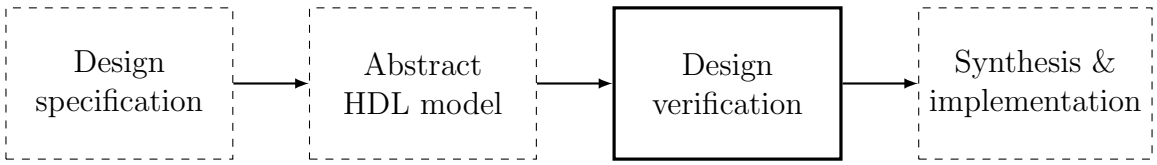



Figure 2.24: The main stages in the digital design process, simplified from [59, p. 363]. The verification step is highlighted.

According to Mohamed [60], there are two main categories of verification in hardware design: functional verification and formal verification. Most of the regularly encountered types of verification are sorted under functional verification. In this category, one can find simulation-based verification, accelerator-based verification, emulation-based verification, and FPGA prototyping [60, p. 28]. In this thesis, however, only simulation-based verification and FPGA prototyping will be considered, as the other two types are essentially various combinations of these.

One example of formal verification is assertion-based verification (ABV) [60, p. 29]. Formal verification methods such as ABV are extremely useful in digital hardware design, since they can be used to mathematically and formally prove the correctness of a DUT, and thus through exhaustive analysis locate design errors that may be missed in simulation. However, for relatively small designs such as the one in this thesis, such verification may be considered somewhat excessive. In addition, licensing for tools that support this kind of design validation are solicited towards businesses, and typically much too expensive for single users.

Mano and Ciletti [59], on the other hand, use slightly different classifications. They divide verification into functional verification and timing verification, where the former comprises all validation that concerns a circuit's logical operation. This means that both formal verification and the previous definition of functional verification are sorted under this broader view of functional verification. In their usage, timing verification concerns analysis of a circuit's timing characteristics, which means that requirements and constraints are put on the circuit's temporal performance [59, p. 180].

### 2.5.4 Logic synthesis

In early hardware design, most systems were designed using hand-drawn circuit diagrams. Today, however, software tooling has all but replaced the traditional circuit diagram [59, p. 68]. This use of software tools in digital hardware design is known as electronic design automation (EDA)[68, p. 23]. The process of transforming an HDL model of a logic circuit into an optimised implementation of gates that behaves identically is termed *logic synthesis*, or—in many cases—just synthesis [59, p. 361]. The output from the synthesis process is said to be at the register-transfer level (RTL), and HDL code that can be successfully realised as digital circuits through synthesis is called RTL code [60, p. 5][59, p. 352]. The place of the logic synthesis step in the design flow is highlighted in Figure 2.25.

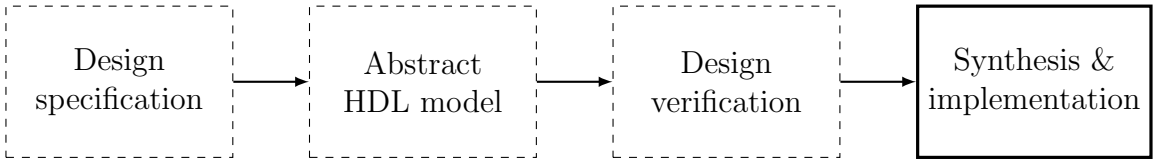



Figure 2.25: The main stages in the digital design process, simplified from [59, p. 363]. The logic synthesis step is highlighted.

In all practical cases, the synthesis of a model written in an HDL such as SystemVerilog involves an EDA computer program. There exists a wide variety of logic synthesis tools, each with their advantages and drawbacks. For this project, the *Vivado Design Suite* [69] was the logical choice. The software is a comprehensive digital hardware design application developed by Xilinx Inc. Its main appeal is its logic synthesis functionality, but it may also be used to view elaborated designs, run simulations, and create *block designs*—a modular design approach where hardware modules are placed on a canvas and wired together visually.

The main reason why this software was chosen is that it is the recommended hardware design tool for the synthesis and implementation of designs on the Pynq development board. The *system on a chip* (SoC) on this board is the *Zynq XC7Z020-1CLG400C*, which is designed by Xilinx. This SoC contains the FPGA that is to be used in development, and it is natural to assume that synthesis and implementation for the target technology will be best supported through the logic synthesis software maintained by the same company.

Another argument in Vivado's favour is the fact that it is available for free through Xilinx's WebPACK licence. In the author's experience, most logic synthesis tools are often far too expensive to license for single users. Alternatively, they may provide a cumbersome application process that is required in order to obtain an educational licence. Vivado, however, can be downloaded and installed free of charge, without such an application process. Furthermore, the author and supervisor are already very familiar with the tool from previous projects. This alone makes using it much preferable to starting anew and learning a different tool.

### 2.5.5 Communication protocols

In modular hardware designs, the communication interfaces between modules is a major design concern. Usually, the signals used for communication follow established patterns or specifications known as communication protocols. While there exists a host of different protocols in the field of hardware design, some are more relevant than others in this thesis. The communication protocols that are to be used in the design and implementation are the ready-valid protocol, as well as a flavour of the AMBA AXI protocol.

#### Ready-valid

Among the simplest communication interfaces is the *ready-valid* handshake protocol, which is sometimes called the FIFO interface [70]. The ready-valid protocol is an elementary handshake protocol between a master (source) and a slave (sink) that is often used for applications such as inter-module communication in larger systems. The protocol works as shown in Figure 2.26.

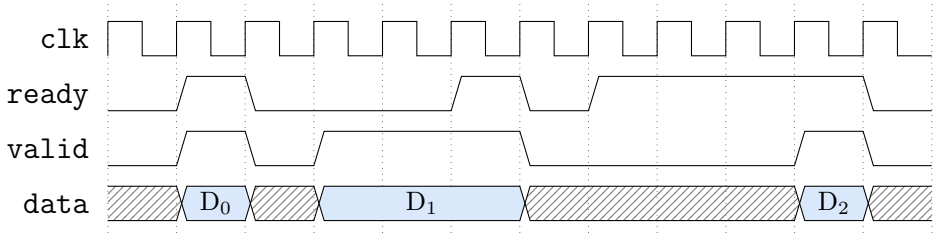



Figure 2.26: Timing diagram of a simple *ready-valid* protocol.

When the master wants to initiate a transfer, it puts the data on the bus and asserts the `valid` signal. The slave asserts its `ready` signal whenever it is ready to accept a transfer. A data transfer occurs whenever both `valid` and `ready` are asserted on the same clock cycle. Do note that the assertion of the handshaking signals `ready` and `valid` may happen in any order [70, p. 2]. As shown in the timing diagram, the slave may be ready to accept a transfer before the master puts valid data on the bus, and vice versa.

The ready-valid protocol is attractive because of its simplicity. However, it is not standardised, so its implementation is up to the designer. It is also very general, and other, more special-case protocols may be more efficient for specific applications.

#### AMBA AXI

The AXI (Advanced eXtensible Interface) interface is a suite of protocols. It is the third generation of the AMBA (Advanced Microcontroller Bus Architecture) specification, which is a registered trademark of Arm Ltd. [71]. Nonetheless, since the protocols are well-documented and royalty-free, they are widely used in the industry.

The AXI interface is a collection of protocols suited for many different purposes. Most relevant to this thesis is the AMBA 4 AXI4-Stream protocol. The protocol is a standardised interface that is used to connect components that wish to exchange a

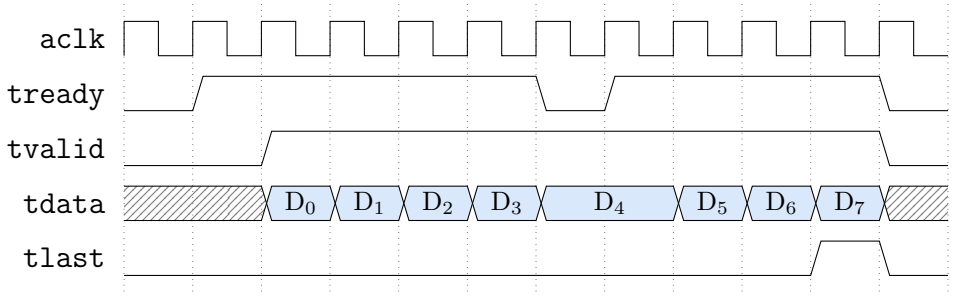



Figure 2.27: Streaming 8 bytes with the AMBA 4 AXI4-Stream protocol.

stream of data. While the protocol may be extended with many additional signals, the basic handshake and data transfer procedure is shown in Figure 2.27.

The handshake can be thought of as an extended version of the ready-valid handshake described above. Data is transferred whenever both the signals `tready` and `tvalid` are asserted at the same clock edge. An additional signal, `tlast`, is used to mark the end of a data transfer. The protocol is governed by the *AMBA 4 AXI4-Stream Protocol Specification* [72], which lists a number of additional signals which may be implemented if necessary. These signals—such as `tstrb`, `tkeep`, and `tuser`—are optional and not relevant to this thesis, and will therefore not be detailed further.

# Chapter 3

# Research context

In this chapter, the research field as a whole will be explored. In other words, the most important findings and results in the fields relevant to this thesis will be introduced. The specific papers and algorithms upon which this thesis is based are introduced and the reasoning behind their selection is discussed. The methods themselves, however, will be described in further detail in Chapter 4. Since this thesis is a continuation of the work done in the project thesis by Espe [1], the following chapter will be partly adapted from the corresponding chapter there. Section 3.1 is directly sourced from the project thesis, and Sections 3.2 and 3.4 are partly sourced from the project thesis. The other parts of the research field—such as the section on hardware ray tracing—were not presented in [1].

## 3.1 Algorithms for octree traversal

Since the extent of the research done on the subject of octree traversal 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 [73] 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 [74]. 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 [73], as well as other, similar schemes [75][76], are instances of bottom-up octree traversal algorithms. The method by Levoy [74], as well as the *HERO algorithm* presented by Agate, Grimsdale, and Lister [77], and a host of other algorithms [78][79][80][81][82] 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 [53] 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. [82] describes an algorithm based on the work of Gargantini and Atkinson [80], however this algorithm is not as well-documented as [53], 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 in hardware without modification. The algorithm by Revelles, Ureña, and Lastra [53] is also recursive, but due to its simplicity is bound to translate well into an iterative method suited for hardware implementation. In fact, this has already been demonstrated in the master's thesis by Wilhelmsen [83] and was further explored in the project thesis [1].

In conclusion, the algorithm elected for traversal of sparse voxel octrees is [53]. The algorithm is simple and fast, and also improves upon the performance of earlier algorithms. In addition, the traversal algorithm was employed with great success in the project thesis work. This alone makes it very relevant for this thesis, since the author already has a deep understanding of its implementation intricacies. On the basis of these considerations, the algorithm will serve as the starting point of this thesis, and will be presented in detail in Chapter 4.

## 3.2 Animation of sparse voxel octrees

Introduced in the project thesis by Espe [1] is a method for efficient animation of sparse voxel octrees. This master's thesis will build upon the results from the project thesis, and the method will be employed as a foundation for the work done as part of this thesis. The technique will be detailed in Chapter 4.

Apart from the work done in the project thesis, only one significant attempt at animation of sparse voxel octrees was found in the literature. The bachelor's thesis by Bautembach [84] 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 the method introduced in the project thesis [1] in that the latter 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.

## 3.3 Hardware ray tracing

There have been many approaches to hardware-accelerated ray tracing through the years, and a great deal of these have employed existing SIMD and MIMD architectures. Examples of such solutions are the works by Crassin et al. [85], Laine and Karras [35], the transputer demonstrations by Packer [31], as well as the relevant project thesis by Espe [1].

However, there have also been several notable attempts at pure hardware ray tracing utilising application-specific hardware. Some approaches have been general purpose solutions for ray tracing—the most well-known of which may be the *AR350* by *Advanced Rendering Technologies*. This solution is employed directly by some, such as in the paper by Cassagnabère, Rousselle, and Renaud [86], but due to its ubiquity is also used by many subsequent works as a comparative basis. For instance, Fender and Rose [87] claim that their hardware implementation of a well-known software algorithm could outperform the *AR350* by a factor of up to three.

Other attempts at pure hardware ray tracing include application-specific hardware solutions, such as the *VIZARD* system by Knittel and Straßer [88]. In 2004, Schmittler et al. [89] introduce a new solution for real-time ray tracing of dynamic scenes on an FPGA chip, which is further extended to a programmable ray tracing unit in 2005 by Woop, Schmittler, and Slusallek [90]. In a PhD thesis authored by Collinson [91] in 2014, further elaboration on the feasibility of hardware-accelerated ray tracing on FPGAs is conducted. Collinson investigates the viability of using parallelism on FPGAs to efficiently ray trace scenes, and concludes that FPGA technology has great potential to rival the efficiency of both GPGPU and CPU implementations, especially in terms of bandwidth and power. However, Collinson mainly explores traditional ray tracing and not ray tracing of SVO models, which means that his specific results are of limited relevance to this master's thesis.

While there are many previous solutions for hardware ray tracing, few of these are suited for ray tracing of sparse voxel octrees. In fact, only one attempt was found in the literature. The master's thesis by Wilhelmsen [83] presented in 2012 showcases a hardware solution for ray tracing of SVOs on FPGA. Wilhelmsen's results are highly relevant for the work done in this thesis, as many of the design choices made by Wilhelmsen were echoed in the project thesis—upon which this thesis builds.

## 3.4 Other works of significance

A very relevant work for the project thesis, as well as this master's thesis is the article by Laine and Karras [35], published in 2011. Their work includes employing the GPGPU capabilities provided by Nvidia CUDA to trace SVOs in parallel. The results presented in their article proved immensely helpful during the project thesis work [1]. Especially useful is the compact SVO memory structure introduced in their work, which will form the basis of the SVO specification used in this thesis. As was the case for the project thesis, the memory structure used in this master's thesis can be viewed as a simplification of the structure presented in their paper. The data structure will be detailed in Chapter 4.

# Chapter 4

# Established algorithms chosen as foundation

In this chapter a set of established algorithms and schemes that are to be used in this work, will be presented. The workings of the algorithms will be explained in detail, as well as the reasoning behind their selection. It should be noted that two of the methods presented in this chapter are the same that were detailed in the project thesis by Espe [1]. Since this master's thesis is a continuation of the project thesis, the same algorithms are described here by employing the same description as was written for the project thesis.

In addition to these two algorithms, the results from the project thesis itself will be presented. The method for animation of sparse voxel octrees will be summarised in the following so that the reader is brought up to speed on where this work left off. Like the preceding two sections, this section is also adapted from the relevant sections in project thesis.

## 4.1 An efficient parametric algorithm for octree traversal

The chosen algorithm for traversal of octrees, and in this case, sparse voxel octrees, is the parametric algorithm proposed by Revelles, Ureña, and Lastra [53] in their 2000 paper. This algorithm was the one employed in the project thesis [1], and it is only logical that it should be used in the further work that builds upon the results from the project thesis. As explained in the literature review, this algorithm seems to be the most efficient and well documented. Many newer papers—such as works on global illumination [92], virtual X-ray imaging [93], and the very relevant works of Laine and Karras [35] and Wilhelmsen [83]—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 can be classified as a recursive, top-down algorithm with neighbour-finding. Exactly what this entails should be apparent after the following sections. A point to note, that was discussed in detail in the project thesis, is that recursive algorithms are usually problematic in a GPGPU context. The same can be said for

algorithms to be implemented in hardware. However, it was shown by Espe [1] that this algorithm could without much effort be modified to be iterative instead of recursive. This was also demonstrated by Wilhelmsen [83] in his hardware approach.

In this section, the algorithm will be reviewed like it is presented in [53], 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 in order to be compatible with the rest of the 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 initially. 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 simplicity, all rays are at this point in time 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

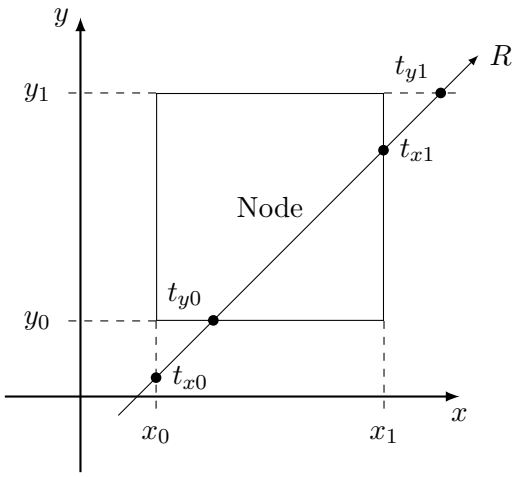



Figure 4.1: The node boundaries.

The first phase of the algorithm 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 [53] and presented 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 $t_{y0} < t_{y1}$. From this it can be deduced 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 $t_{y1}$ 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).

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

The node 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 [53].

**Third phase: recursion**

If the intersection test at the end of the second phase passes, the algorithm moves onto the third and final phase. In this phase, the algorithm will—if applicable—recurse into the children of the node. To facilitate recursion 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 simply defined by taking the arithmetic average of the enter and exit planes in each dimension. Their mathematical definition is shown in Equation (4.4).

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

At this stage, all the required values of $t$ have been calculated, and these may now be used to determine which child nodes that are intersected by the ray. The process is simply a matter of systematically testing the different parameters introduced so far. Pseudocode for the process of obtaining child nodes is shown in Figure 4.2.

After the intersected child nodes have been determined, the algorithm will process them one after the other, ordered by increasing indices. For each child node to process, if the node has further children, the algorithm will recurse into it. The values of $t$ for a child node will differ from those of the parent node. The $t$-values to use for each of the

**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.

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. Since all the required parameters are calculated already, the algorithm may skip directly to the third and final phase when it recurses.

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 shows the case where the ray enters through boundary $x_0$, while Figure 4.4 highlights the situation that $y_0$ is the initial boundary.

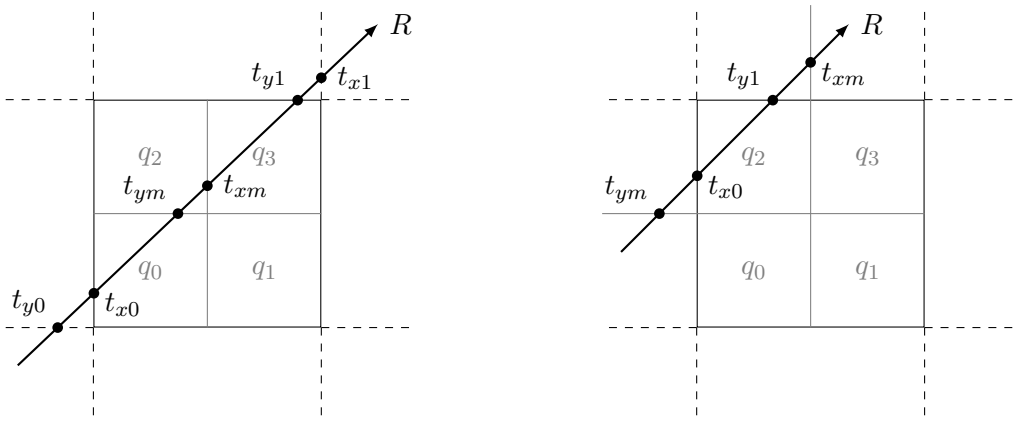



Figure 4.3: Sub-nodes crossed when $t_{x0} > t_{y0}$. In the right figure, $t_{y0}$ and $t_{x1}$ lie outside the diagram.

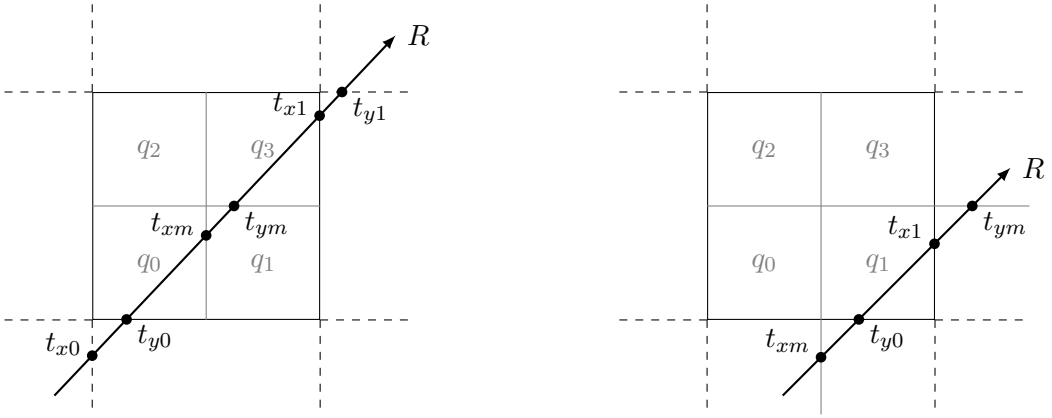



Figure 4.4: Sub-nodes crossed when $t_{y0} > t_{x0}$. In the right figure, $t_{x0}$ and $t_{y1}$ lie outside the diagram.

### 4.1.2 Extending the algorithm to octrees

In order to allow 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 as 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 test that determines whether the ray intersects the node or not is unchanged from the two-dimensional case; the simple expression $t_{\min} < t_{\max}$ 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. In the following sections, the process of selecting the initial and next child nodes in an octree is described.

#### 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.

#### 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

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 |

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.

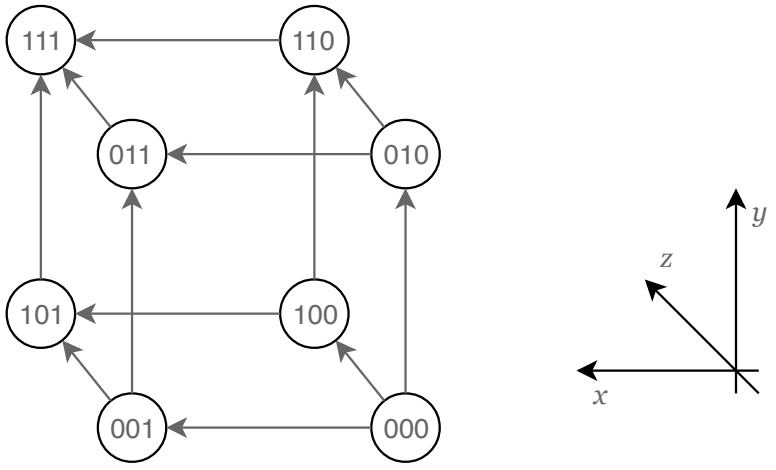



Figure 4.5: The traversal state machine.

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 applicable child nodes of the current parent have 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 [53] 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 hardware.

Another solution which only resulted in a slight modification of the algorithm was introduced in the project thesis [1]. It 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 mathematically result in a slight distortion of the rendered image, but as long as the value is small enough, it should not be noticeable. By solving the issue this way, the algorithm remains simple and the need to handle special cases is eliminated.

### 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 [53] 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 symbol "$\oplus$" is here used to mean a bitwise *exclusive or* operator, 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 the same that was chosen for the project thesis by Espe [1], which is a simplified version of the scheme authored by Laine and Karras [35] in 2011. The focus of said paper was to devise an efficient data structure for the storage of voxel data. The resulting scheme was also designed to minimise memory bandwidth requirements and therefore has a modest memory footprint.

As was the case for the project thesis, the data structure will be presented here 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 [83].

### 4.2.1 Scheme overview

The data structure 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 and increasing cacheability, 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 bits wide, its layout on the form shown in Figure 4.6.

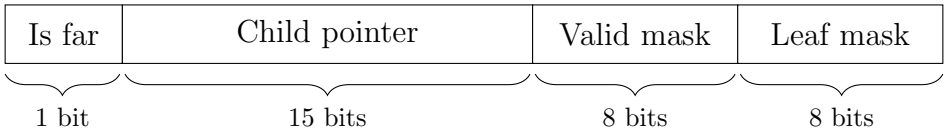



Figure 4.6: A single node entry.

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 [35] 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 [83], 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 briefly 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 using the software implementation presented in the project thesis [1].

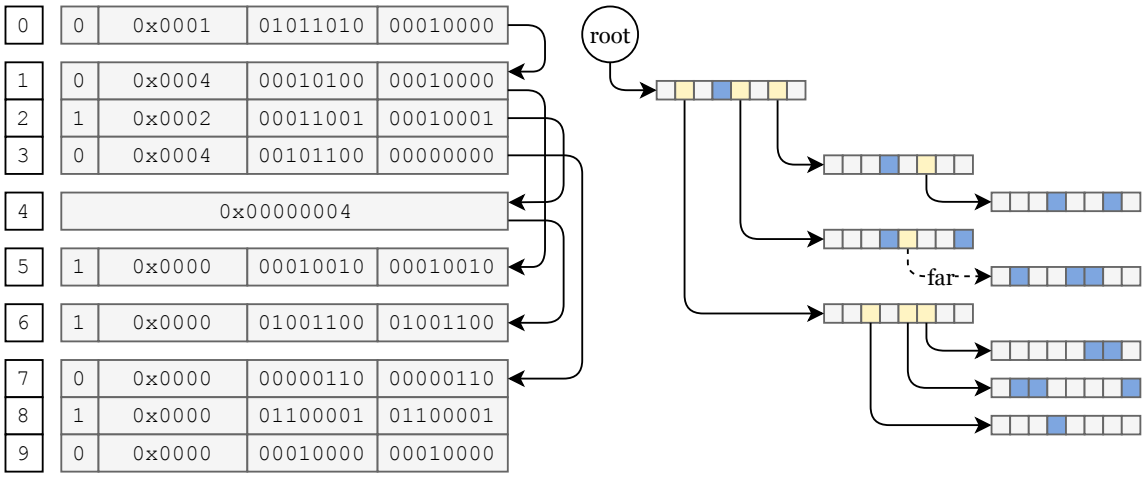



Figure 4.7: An example SVO. Both the structure in memory and hierarchical layout are shown.

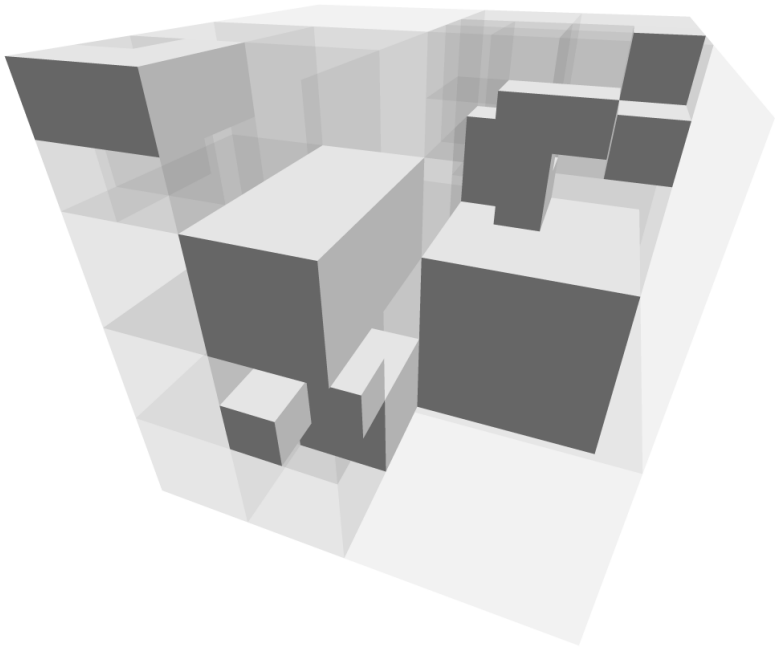


Figure 4.8: The octree specified in Figure 4.7 rendered.

## 4.3 A method for rigid-body animation of sparse voxel octrees

The project thesis by Espe [1] forms the starting point for this master's thesis, which will employ the work done in the project thesis as part of a larger project of hardware implementation. In the project thesis, a general method for rigid-body animation of SVOs was developed and presented. The method will be detailed in this chapter since it is to be implemented as part of the hardware solution. The following sections are heavily sourced from the project thesis, with some minor adaptions.

### 4.3.1 Method overview

The main idea behind the method is to take 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. 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.

It is not feasible to apply the transformation to every data point contained in the model in the rendering stage as one would do when animating polygonal models. The solution authored by Espe [1] 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 4.9 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.

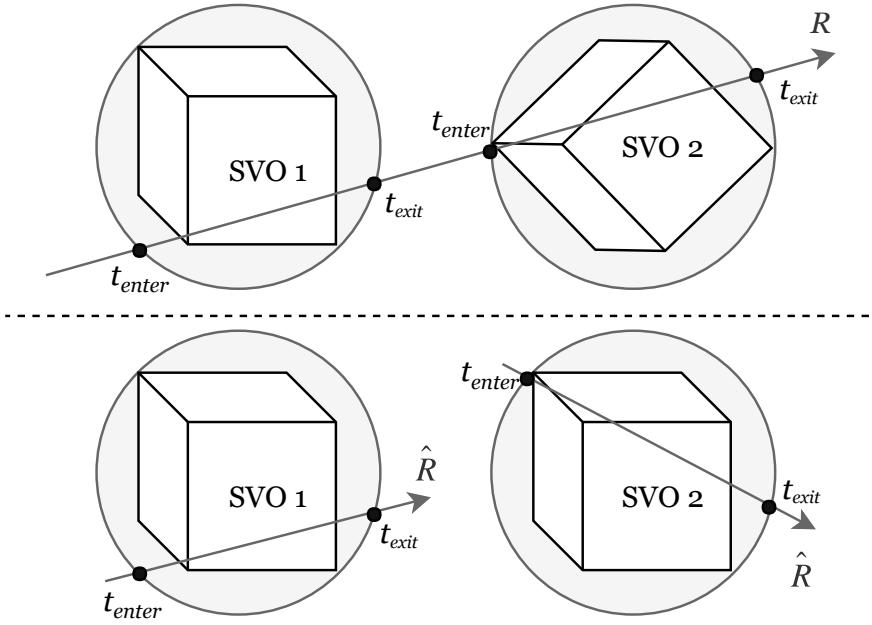



Figure 4.9: Demonstrating the ray transformation process.

### 4.3.2 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 (4.8), 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 (4.9).

$$R_t(\mathbf{r}_o, \mathbf{r}_d) = \mathbf{r}_o + t \cdot \mathbf{r}_d \ , \quad t \geq 0 \tag{4.8}$$

$$\hat{R}_t(\hat{\mathbf{r}}_o, \hat{\mathbf{r}}_d) = T\left[R_t(\mathbf{r}_o, \mathbf{r}_d)\right] \tag{4.9}$$

A graphical description of the desired transformation is shown in Figure 4.10. 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.

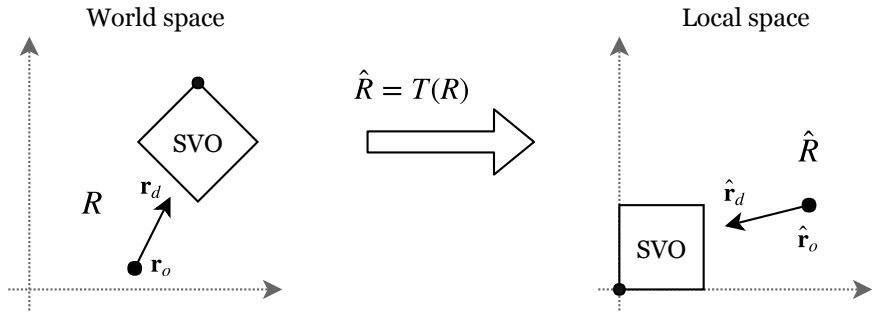



Figure 4.10: The co-ordinate system transformation of a ray in world space to local space.

The resulting mathematical definition of $T$ is shown in Equation (4.10). The ray direction is determined by simply premultiplying 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 (4.10).

$$\begin{aligned} T : R_t(\mathbf{r}_o, \mathbf{r}_d) &\mapsto \hat{R}_t(\hat{\mathbf{r}}_o, \hat{\mathbf{r}}_d) \\ \text{such that } &\begin{cases} \hat{\mathbf{r}}_d = \mathbf{M}_R^{-1}\mathbf{r}_d \\ \hat{\mathbf{r}}_o = \mathbf{M}_R^{-1}\mathbf{M}_T^{-1}\mathbf{r}_o \end{cases} \end{aligned} \tag{4.10}$$

### 4.3.3 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 master's 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 its bounding sphere—also need to be updated to take the scale of the octree into account.

### 4.3.4 Optimisations

The original method also introduces a set of optimisations that increase the general performance of the algorithm. These optimisations will not be utilised in this master's thesis due to its scope already being quite substantial. However, since they are a part of the original method, they will be presented briefly.

#### Bounding-sphere 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 [73]. 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 was presented in the project thesis 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 4.11 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.

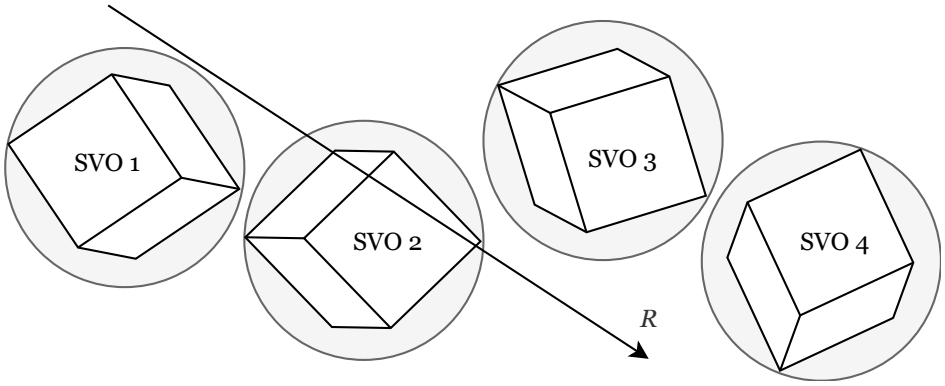



Figure 4.11: Using the bounding sphere of an SVO to avoid traversing octrees that will be missed. SVO 2 is the only octree that will be traversed in this case.

### 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. An example of this is illustrated in Figure 4.12.

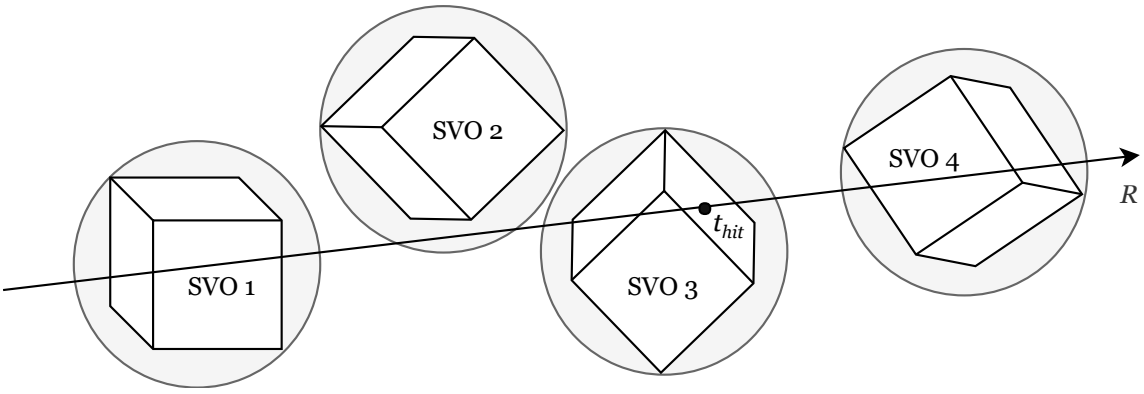



Figure 4.12: Tracing a sorted list of SVOs.

In the illustration, 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.

### Hit buffer algorithm

A simple buffering mechanism was developed as part of the project thesis work. 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 4.13.

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

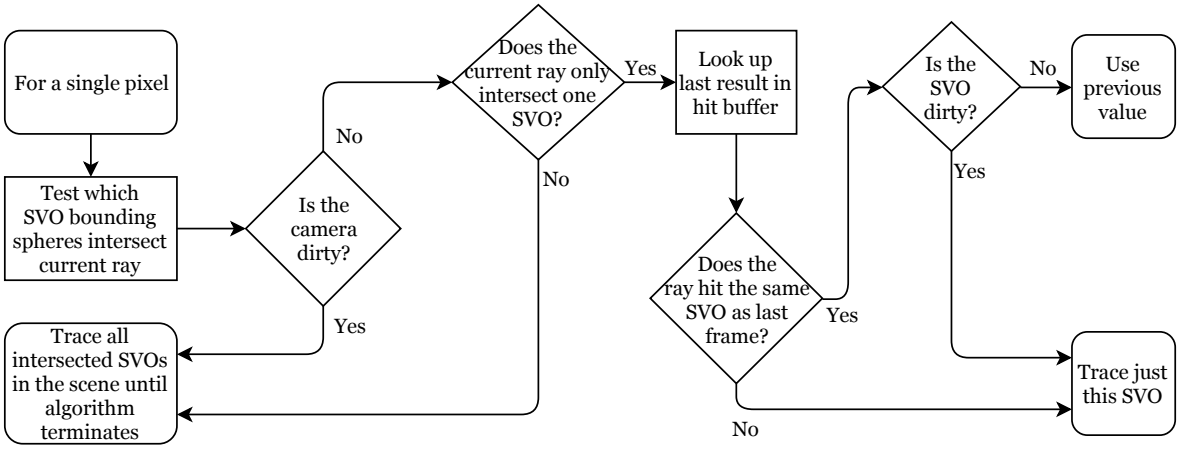



Figure 4.13: The hit buffer algorithm.

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 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.

# Chapter 5

# Requirement analysis

In this chapter, an analysis of the system requirements will be conducted, and a general specification will be formulated. The ambition is to end up with an overarching specification for a system that will meet the requirements posed by the problem statement. To this end, the problem statement is discussed initially, and a few points concerning its interpretation are raised. Subsequently, the requirement specification of the system is formulated.

The *requirement specification* of a system is used to describe the functional and non-functional requirements that are put on the system [58, p. 60]. The specification may be viewed as an agreed-upon description of expected behaviour, and as a set of well-defined criteria that will be revisited during the verification and validation effort in order to assert a system's correctness [37, p. D-10]. Mano and Ciletti [59] regard the functional specification as the entry point to the digital design process [59, p. 129], which places further emphasis on the significance of a proper system specification.

According to the methodology detailed by Dally, Harting, and Aamodt [58], a system specification should open with a concept development and feasibility investigation. The concept development includes a functional specification, as well as an interface specification and an overall block diagram. Pursuant to their methodology, this chapter is laid out by first presenting the primary functional and non-functional requirements. This is followed by more specific requirements, such as an interface specification and a feasibility evaluation, which is elaborated in a throughput requirement discussion. Nextly, Dally, Harting, and Aamodt [58] recommend moving on to a more detailed specification by partitioning the design [58, p. 65]. These considerations are discussed in Chapter 6, where the system's internal design is specified.

## 5.1 Interpretation of the problem statement

The work done in this project should fundamentally be grounded in—and may be considered a direct response to—the problem statement included in the front matter of this thesis. Moreover, it should be self-evident that the interpretation of this problem statement must be as correct as possible if the work done in this master's thesis is to be relevant. While the problem statement itself is quite precise, it will be examined point for point in this section. This is done to make sure that the solutions presented in this thesis are indeed a direct response to the points in the problem statement. Additionally,

if decisions must be made regarding its interpretation, these will be highlighted in the following.

The problem statement opens with a general presentation of the field and exposes the problem at hand. Following this introduction, three distinct points are presented. These points may be regarded as the actual requirements for this thesis, as they briefly state what one should expect of the solution.

> **First point:** *Review existing literature on the subject of hardware ray tracing.*

The first of these points states that a thorough examination of relevant previous work should be carried out. This point has already been covered by the literature review presented in Chapter 3 and the detailed discussion of chosen algorithms in Chapter 4, and as such will not be part of the requirement specification.

> **Second point:** *Investigate whether a hardware implementation for real-time ray tracing of SVO data is feasible, and formulate the specification and design of such a system.*

The second point of the problem statement states that an investigation should be launched into the feasibility of hardware ray tracing of SVOs. The point subsequently requests that a specification of such a system be formulated. The feasibility of hardware ray tracing was briefly discussed in the literature review in Chapter 3, where multiple previous examples were presented. Hardware applicability have also been kept in mind and brought up whenever relevant during the detailed algorithm presentations in Chapter 4. The point will be further substantiated by the investigations done in this chapter and Chapter 6. The specification and design of such a system will be discussed in this chapter, and further detailed in Chapter 6.

> **Third point:** *If feasible, demonstrate hardware ray tracing of SVO data. Explore the possibility of extending such a hardware implementation to support animation of SVO data.*

The third and last point states that a hardware implementation should be demonstrated, and that the possibility of animation of SVO data should be discussed. A hardware implementation is the core topic of Chapter 7. The possibility of SVO animation is discussed throughout this thesis—the method by which this can be achieved was introduced in the project thesis and presented in Section 4.3. The topic is brought up whenever relevant in this chapter and, especially, Chapters 6 and 7.

## 5.2 Primary functional and non-functional requirements

From the problem statement, three initial requirements may be formulated. These three requirements can be regarded as the primary functional and non-functional requirements that are placed on the system, as they are essentially a summary of the problem statement. They will be revisited in the discussion found in Chapter 8 so that the validity of the implemented system may be evaluated.

#### First requirement: ray tracing of SVO models

The first and chief requirement of the system is the functional requirement that it must be able to trace SVO models. Particularly, it must be able to traverse an SVO model on the format described in Section 4.2, employing the algorithm detailed in Section 4.1.

#### Second requirement: SVO model animation

The second functional requirement is that the system must support animation of SVO models. The method for rigid-body animation of SVO models to be used was the main topic of the project thesis [1] and is detailed in Section 4.3.

#### Third requirement: real-time performance

The third and last requirement is the non-functional requirement that the system must be able to trace animated SVO models in real-time. In Section 2.2.3, a real-time rendering process was defined as having a frame rate of at least 12 to 16 Hz. The requirement is therefore that system in this thesis must perform above the upper bound of 16 Hz.

## 5.3 Further requirement specification

Building upon the system's primary functional and non-functional requirements, a more detailed analysis of its requirements can be performed. In the following sections, a set of additional considerations and requirements will be presented, and further elaboration of the system's requirements will be conducted.

### 5.3.1 System scalability

To genuinely exploit the parallelisability of ray tracing, the system should be of a scalable nature. More to the point, the system should have some sort of central computation core which may be duplicated to increase the *throughput*—the amount of work that can be done by the system per time unit [37, p. 48].

A simple scalable system is illustrated in Figure 5.1. The cores within the system are identical and designed to perform a task that may limit system throughput. By duplicating the cores, the throughput may be increased, as tasks sent to the system are distributed among the computation cores and processed in parallel. This distribution of tasks would be performed by a scheduler in a manner that maximises utilisation.

Amdahl's law states that the speed-up achieved from parallelising a system is limited by the serial portion of the computation load [94]. Amdahl was chiefly concerned with parallelisation of software in his 1967 paper, but the same reasoning may be applied to hardware design—the larger section of the system that is parallelised, the better the overall throughput. For instance, a floating-point multiplication unit may be duplicated to increase throughput of this specific operation, while an SVO traversal module may be duplicated separately to achieve higher efficiency in this part of the algorithm.

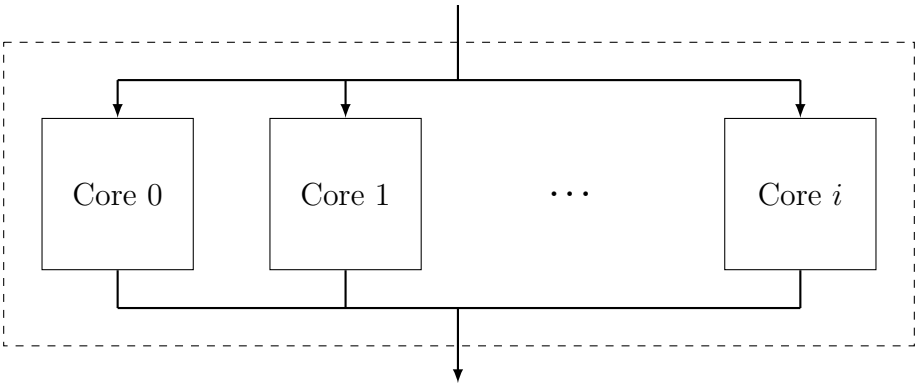



Figure 5.1: A scalable system with duplicated computation cores. The dashed rectangle represents the system, while the individual computation cores are shown within.

A decision must be made concerning the selection of which units of the system that should be part of this scalability effort. Naturally, this is selection will depend on which sections of the underlying algorithms that lend themselves to parallelisation. Furthermore, the system must from a design standpoint be modularised in a fashion that allows such duplication. As a general rule, computationally heavy operations that must be done for all inputs may benefit from parallelisation.

In some situations, multiple distinct types of cores could be duplicated separately to parallelise different tasks. For instance, both a floating-point multiplication unit and a SVO traversal module could be duplicated to achieve higher overall throughput. The specifics for the solution presented in this thesis will be discussed as part of the module definitions in Chapter 6.

Another decision that must be made is to which degree modules of the system should be duplicated. As an example, it would certainly be a waste of hardware resources if the system were to contain 32 SVO traversal cores spending most of their time in an idle state, because they are limited by the throughput of a single floating-point multiplication unit. Evaluating these considerations is in many cases a matter of testing an implemented system with the intention of determining which parts of the system that act as *bottlenecks*. In other words, one must establish which sections that act as limiting factors when maximising throughput. These concerns will be revisited, and related parameters determined empirically, in Chapter 8.

### 5.3.2 Pipelining

In many situations it may not be necessary to duplicate entire modules in order to achieve a higher throughput. If an operation requires a known, constant number of clock cycles to complete—and especially if each distinct part of the operation is only performed once—one may employ *pipelining* as an alternative approach to increase the throughput.

As illustrated by Figure 5.2, pipelining is achieved by splitting up a complex operation into multiple simpler stages [40, p. 66]. The pipeline is set up so that each of these stages performs a simple operation which takes one clock cycle to complete. By coupling each stage's output to the next stage's input and placing a register between them, every stage of the operation will complete its function and hand the result over

to the next stage within one clock cycle. In performance terms, this means that if all the stages are run in parallel, the pipeline is fully utilised and will produce a result each clock cycle.

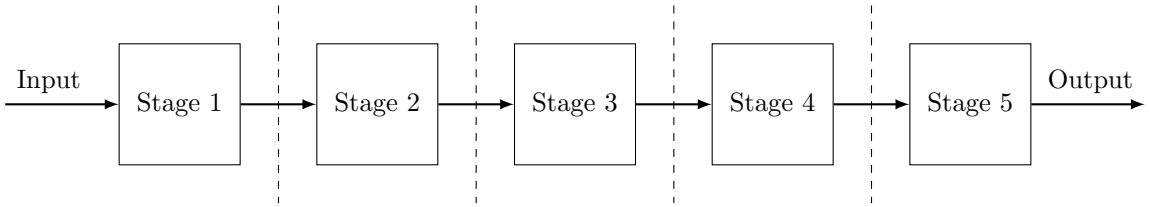



Figure 5.2: A pipelined operation with five stages. The discrete stages of the pipeline are shown connected by a data path. The dashed lines represent intermediate registers that store the results between clock cycles.

While the latency of the whole operation is unchanged, the throughput of the pipelined system is increased to the point where one operation is completed each time step. This is further illustrated by an example in Table 5.1, in which a pipelined process with five stages is described. In the example, the latency of the operation is five time steps—in other words, it takes five time steps to complete each operation. However, since the system is pipelined, all of these stages are computed in parallel, and one operation is finished every time step. The throughput is therefore increased fivefold through a simple pipelining setup.

Table 5.1: A set of operations moving through the pipeline. Once an operation has been processed in stage 5, its final result is output.

| Time $t$ | Operation | | | | | | |
|---|---|---|---|---|---|---|---|
| | $O_1$ | $O_2$ | $O_3$ | $O_4$ | $O_5$ | $O_6$ | $O_7$ |
| 1 | Stage 1 | — | — | — | — | — | — |
| 2 | Stage 2 | Stage 1 | — | — | — | — | — |
| 3 | Stage 3 | Stage 2 | Stage 1 | — | — | — | — |
| 4 | Stage 4 | Stage 3 | Stage 2 | Stage 1 | — | — | — |
| 5 | Stage 5 | Stage 4 | Stage 3 | Stage 2 | Stage 1 | — | — |
| 6 | — | Stage 5 | Stage 4 | Stage 3 | Stage 2 | — | — |
| 7 | — | — | Stage 5 | Stage 4 | Stage 3 | Stage 1 | — |
| 8 | — | — | — | Stage 5 | Stage 4 | Stage 2 | Stage 1 |
| 9 | — | — | — | — | Stage 5 | Stage 3 | Stage 2 |

The pipeline is fully utilised whenever every stage of the pipeline is active. This happens at time step 5 in the table. At time step 6, $O_6$ is not yet ready to be submitted to the pipeline. The result is that from time step 6 and onward, a *bubble* is introduced

in the pipeline [37, p. C-17]. The effect if this is that for a number of subsequent time steps, the pipeline will not be fully utilised since one stage will always be inactive. Pipeline bubbles are undesirable, since they propagate through the system, and lower the overall throughput.

Determining which modules of the system that are suited for pipelining is a matter of establishing which operations that take a constant number of clock cycles to complete. This is required to allow the operation to be unfolded into a set of discrete stages. For instance, since an SVO traversal module may run for an arbitrary number of clock cycles before returning its result, such a module can not readily be split into a fixed set of stages for pipelining. As will be detailed in Chapter 6, the system will contain a conversion operation from floating-point representation to fixed-point representation. It turns out that this conversion is highly suited for pipelined implementation, and its functionality and design as a pipelined operation will therefore be discussed in detail.

### 5.3.3 Interface specification

The interface specification of the system is part of an elaboration of the system's functional requirements, introduced in the previous as the first and second primary requirements. The interface is initially formulated at the top-most abstraction level. To this end, it makes sense to consider the system as a *black box*—an opaque container with a well-defined interface, but whose contents are hidden from view [95, p. 58]. One may regard this as a description of the system from the point of view of the end user, since he or she is rarely interested in the internal workings of a system.

In this case, the end user of the system would expect it to function as a hardware accelerator for ray tracing—a *ray tracing unit* (RTU). A slightly more detailed description is that the system should await a ray on its input interface, process this ray, and return a result on its output interface. Figure 5.3 shows the system modelled as a black box. The interfaces of the system employ the simple ready-valid protocol described in Section 2.5.5 to accept new rays and return results.

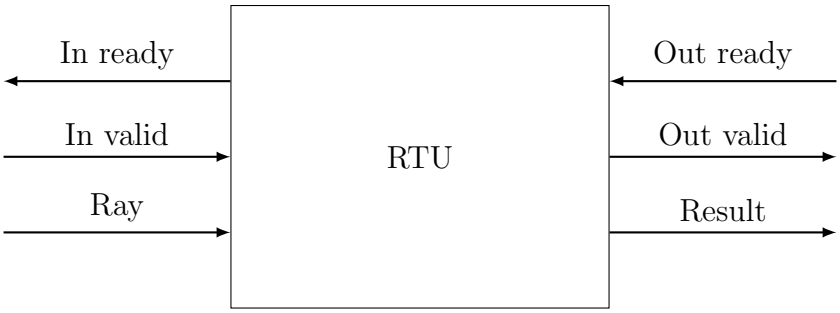



Figure 5.3: The ray tracing system as a black box.

The inputted ray should contain all information necessary for the computation that is to be performed. In other words, it should at least contain the primary attributes of the ray: the ray origin and direction. Since there may be multiple models in the scene, it should additionally contain a pointer to the memory location of octree model that is to be traced. This memory location would also provide any animation data related to the model such as transform matrices. Moreover, as it is desirable to have a

system that can process many rays in parallel, it is also necessary for the input packet to contain a unique ID that can be used to differentiate results. These results may very well be returned out of order due to the parallel nature of the system.

In the other end of the system, the result output should contain the result of the ray tracing procedure. First and foremost, it must contain a value that describes the result. This value could, for instance, be a simple Boolean switch that states whether or not the ray hit a solid voxel. Secondly, it must hold the same unique ID of the ray packet that was submitted so that each result may be identified properly. The rest of the contents should give further information about the nature of the hit. It would be useful to return the parametric $t$-value along the ray of the hit, since this could be employed in a depth buffer. In addition, the surface normal of the hit should be returned so that it can be employed in lighting calculations. A third value which might be very useful for debugging purposes is the number of clock cycles that were spent processing the ray. This value could be used in verification and validation to signify the ray cost.

Based on the above discussion, the data structures of the input and output objects may be formulated. In the following, two data structures are specified, beginning with the input ray structure. This structure has a total of four fields, while the second structure—the output result packet—has five fields. The specific formats of the two structures, such as number representation and bit widths, should become clear through the discussions in Chapters 6 and 7.

**Input ray packet fields**

- `orig`: A three-dimensional vector of numbers that holds the ray origin.
- `dir`: A three-dimensional vector of numbers that holds the ray direction.
- `addr`: A pointer to the memory location of the SVO model to trace. It is assumed that the animation data is reachable in the same location.
- `job_id`: A unique ID that the end user assigns to this job. When the result for this job is returned, it will contain the same ID.

**Output result packet fields**

- `hit`: A Boolean value which states whether the ray hit a solid node in the given SVO model.
- `t_hit`: A number which holds the parametric $t$-value along the ray of the hit. Holds garbage data if the ray missed.
- `normal`: A three-dimensional vector of numbers that holds the surface normal vector of the node that was hit. Holds garbage data if the ray missed.
- `cost`: An integer that states how many clock cycles the SVO traversal core spent processing the ray.
- `job_id`: The same unique ID that was passed in with the input packet.

### 5.3.4 Throughput and feasibility

Since the system is a hardware accelerator to be used in computer graphics, it stands to reason that it should be able to process rays at a speed suitable for rendering. In fact, the third functional requirement of the system states that one should expect real-time performance from the system, which in turn puts certain requirements on the throughput that the system must be able to deliver. As stated in the sections on scalability and pipelining, throughput may be increased by designing the system so that it supports the processing of several rays in parallel. This means that multiple ray tracing jobs may be submitted to the system in short order, while the results of earlier jobs are simultaneously received.

Shown in Figure 5.4 is a diagram of how the desired parallel-processing system may behave in terms of its interface signals. The system in the illustration is parallelised and pipelined to such a degree that it outputs one result per clock cycle, which may be viewed as the ultimate throughput goal of the system. It should be remarked that the processing time for each job is highly underestimated for illustrative purposes; the diagram ambitiously shows that it takes about three clock cycles to process each ray job. In addition, note that the subscript indices of the input and output only designate the packets' temporal index, and do not reflect their unique ID. As stated in the previous section, the outputs may be received out of order in relation to the inputs.

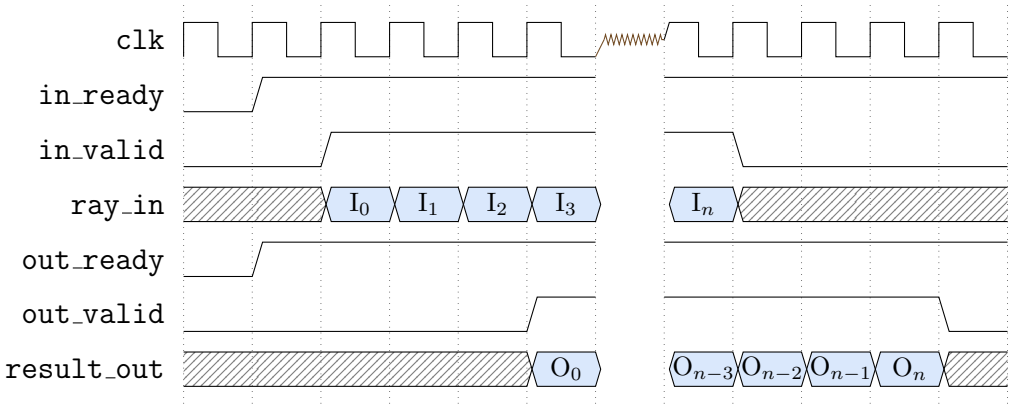



Figure 5.4: Communication interface for top-level module, highlighting the desire for a throughput of one result per clock cycle.

As part of an investigation into the feasibility of such a system, static analysis of the throughput requirements may be performed. It can be calculated how many rays the system is required to process, and at which speed these rays must be processed by a couple of simple calculations. This information may then be analysed and the feasibility of the system evaluated.

Shown in Table 5.2 are the theoretical throughput requirements of the system for a selection of resolutions and frame rates. The table illustrates how many rays the system must be able to process per frame, as well as how many rays must be processed per second for different desired frame rates. The last column shows how much slack the RTU has in terms of ray processing when compared to the desire of one ray per clock cycle outlined in the previous. In other words, the numbers state how many clock cycles the system may wait between outputting each result. These numbers are calculated

under the assumption that the system runs at 100 MHz, and essentially highlight the degree to which the system must be parallelised and pipelined in order to satisfy the requirements.

Table 5.2: Throughput requirements for a selection of resolutions and frame rates. The RTU is presumed to run at 100 MHz.

| Resolution | Rays per frame | Frame rate | Rays per second | Cycles per ray (@ 100 MHz) |
|---|---|---|---|---|
| $320 \times 180$ | $6.84 \times 10^4$ | 15 Hz | $1.03 \times 10^6$ | 115.74 |
| 〃 | 〃 | 30 Hz | $1.73 \times 10^6$ | 57.87 |
| 〃 | 〃 | 60 Hz | $3.46 \times 10^6$ | 28.94 |
| $640 \times 360$ | $2.30 \times 10^5$ | 15 Hz | $3.46 \times 10^6$ | 28.94 |
| 〃 | 〃 | 30 Hz | $6.91 \times 10^6$ | 14.47 |
| 〃 | 〃 | 60 Hz | $1.38 \times 10^7$ | 7.23 |
| $1280 \times 720$ | $9.22 \times 10^5$ | 15 Hz | $1.38 \times 10^7$ | 7.23 |
| 〃 | 〃 | 30 Hz | $2.76 \times 10^7$ | 3.62 |
| 〃 | 〃 | 60 Hz | $5.55 \times 10^7$ | 1.81 |
| $1920 \times 1080$ | $2.07 \times 10^6$ | 15 Hz | $3.11 \times 10^7$ | 3.22 |
| 〃 | 〃 | 30 Hz | $6.22 \times 10^7$ | 1.61 |
| 〃 | 〃 | 60 Hz | $1.24 \times 10^8$ | 0.80 |

Higher numbers in the last column leave the system a lot of slack for processing. The lower the number, the greater the degree of parallel processing and pipelining must be. When the number is less than one, as can be seen in the last row—for $1920 \times 1080$ @ 60 Hz—the system is required to return multiple results per clock cycle. This is not supported in the current specification, since the specified system interfaces can at most deliver one result per clock cycle. The conclusion must therefore be that it is not feasible that the RTU will be able to satisfy this requirement with its current specification. All other configurations, however, are feasible—at least according to the information gathered from this static analysis.

An assumption made in this section is that the RTU only has to process one ray for each pixel, which is only valid if there is a single SVO model in the scene. The validity of the static analysis is therefore somewhat limited since the current specification states that a single ray job may only trace a single model, and multiple ray jobs must be submitted to the system in order to trace a scene with multiple models. Moreover, techniques for improving visual output, such as multisample anti-aliasing or supersampling would require multiple rays per pixel per model. Such techniques are not of interest in this thesis, however, and are consequently disregarded.

### 5.3.5 Memory latency

While efficient memory handling is generally of crucial importance in the design of real-time computer graphics systems, it has been determined that the design and implementation of a fully functioning memory module would make the scope of this master's thesis too large. A complete memory module will therefore *not* be entertained in the system's design and implementation, except as part of peripheral discussion and elaboration on limitations and future work.

Nonetheless, in an effort to paint a complete picture, certain considerations concerning memory latency and caching optimisations will be presented and discussed in the following. These concerns are not directly applicable to the current design, but may be revisited and included a requirement specification for a later revision of the system as part of future work.

It is expected that that memory latency will become a major obstacle in the design of a hardware ray tracing system with proper memory handling. The SVO traversal cores—the main computation cores of the system which will be introduced in Chapter 6—primarily operate on the data contained in the SVO structure, which will at some point have to be fetched from memory. It would not be feasible to fetch the entire model at once, so the individual nodes of the data structure would be fetched incrementally as the SVO is traversed. The issue of memory latency should presumably manifest itself to a great degree, as each and every node in the tree that a traversal core requires must be requested and delivered from main memory. Without mitigating the memory latency problem, the traversal cores may spend a lot of their time blocked, and waiting for memory requests to come through.

Fortunately, there exists a solution that may help minimise this problem. By introducing a multi-level memory architecture in the system, the latency issue might be resolved to some degree. The memory cache, which was detailed in Section 2.2.6, has been part of virtually every processing architecture in the last decades and can be used to this end, especially since the chosen SVO data structure almost exclusively employs relative memory references. Additionally, as was detailed in Section 4.2, the data structure is configured to place sibling nodes contiguously in memory, which should enable the the system to benefit profoundly from memory caching.

# Chapter 6

# System design

Employing the requirement specification presented in Chapter 5 as a basis, the system design is now to be derived. A major part of this design process entails modularising the system to the point where each module serves a specific and well-defined function, and has a simple and clean interface [59, p. 351][58, p. 65]. This modularisation is done not only for the purpose of creating a thorough technical description the system, but also in order to simplify the development and verification process.

The development of a modular system is simpler, more manageable, and more comprehensible than for a large monolithic system. Under the assumption that the functionality of each module and the interfaces between them are well-defined, the developers may concentrate on developing each module one at a time, while in their mind disregarding every other module [58, p. 65]. In larger teams, modularisation has an even greater effect—the modules may be developed concurrently, which significantly speeds up the design process.

A modularised system also aids in reducing the required verification effort. By dividing the system into modules that perform logically discrete functions, the correctness of each module may be tested and verified in isolation. This is known as *unit testing*. In other words, unit tests are special tests written as part of the verification effort to test and verify the functionality of each individual module. [58, p. 65]. The verification of the system and the usage of unit tests is part of the implementation process, and will be described in detail in Section 7.5.

## 6.1 System modularisation

The black box introduced in the requirement specification in Chapter 5 is now to be split into a set of modules. Dally, Harting, and Aamodt [58] recommend partitioning the system to such a degree that each module may be directly realised using a synthesis procedure. These resulting bottom-level modules may be combinational logic blocks that compute a logical function on its inputs, arithmetic modules that manipulate numbers, or finite-state machines that sequence the operation of the system [58, p. 65].

The top-level modularisation of the entire RTU system is visualised in Figure 6.1, with the data flow through the modules highlighted. This diagram introduces many new modules which will be discussed—and some even further modularised—in the following. Note that the job manager module and SVO cores additionally communicate

with the memory module, but in order to increase readability, these interfaces are not drawn in the diagram. At this stage, the scalability of the system is illustrated through duplication of SVO traversal cores.

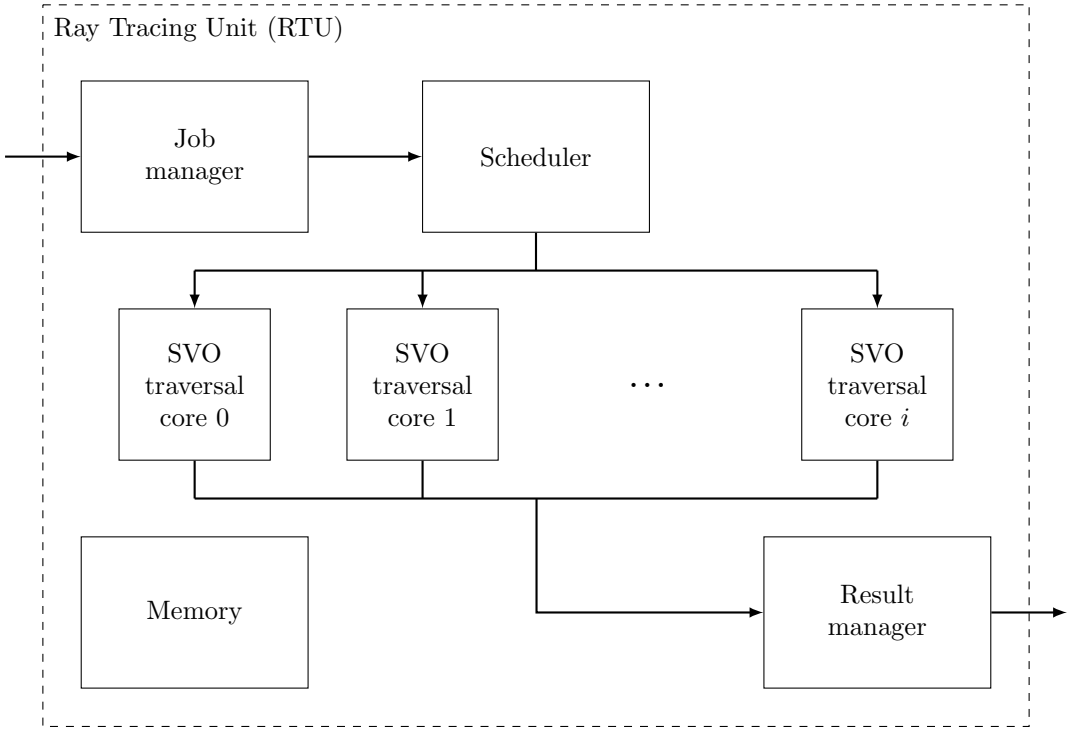



Figure 6.1: The main internal modules of the ray tracing unit, as well as the data flow through the modules. Communication with the memory module is not shown.

### 6.1.1 Method and justification

While there are a number of different approaches when modularising a system, in many cases the logical place to start is by identifying the central computation load of its desired functionality.

As required by the specification, the octree traversal algorithm detailed in Section 4.1 is to be implemented in hardware, and this algorithm will likely represent the majority of the processing load. It would make sense to keep all aspects related to this traversal of SVO models contained within one module, which might then be modularised further. The most important reason for this, perhaps, is that by keeping this process within a module, the module may be duplicated so that the requirement of a scalable system can be satisfied. It is expected that for a strictly sequential system, the SVO traversal would be the main bottleneck. Hence, there is likely much to gain in terms of performance by parallelising this computation load.

The next module that should be part of the system is a direct result of the effort towards system scalability. Since there are now an arbitrary amount of SVO traversal cores in the RTU, a scheduler module to arbitrate between all these cores is needed. In other words, there is a need for a module that will decide which core should do what work at which time. The scheduler will interface directly with all the traversal cores, with the main responsibility of assigning work to each core. In order to keep utilisation

high, the scheduler must keep track of which cores are idle and busy, so that pending jobs may be assigned to idle cores as soon as possible.

In their current configuration, the SVO traversal cores will contain a fair amount of duplicated logic related to the calculation of internal parameters and facilitation of animation. This logic is comprised of operations such as matrix multiplication and floating-point to fixed-point conversions—that is, operations which require a fixed number of clock cycles to complete. As detailed in Section 5.3.2, operations that use a fixed number of clock cycles may be unfolded into a set of discrete stages, and subsequently pipelined. This means that instead of duplicating the logic for each SVO core, the logic may be isolated into a separate module and pipelined instead. This solution should save a considerable amount of hardware resources. The module with these pipelined calculations has been titled the *job manager*.

Once the SVO traversal cores have finished their assigned jobs, they need to be able to return their results somewhere. The simplest solution to this end would be to simply have them wait until the result is retrieved externally. This approach, however, would couple the internal throughput of the system to the performance of the environment, and might quickly make the whole system stall; the cores would be unable to accept new jobs before the result from the last one was retrieved. On account of this, it makes sense to introduce an additional module which handles the results from the traversal cores and buffers them until they are retrieved externally. This module accepts the results, and lets the SVO traversal cores start processing a new job without pause—essentially introducing a pipeline for results on the module-level. The module that handles the results is called the *result manager*.

Lastly, the SVO traversal cores need to fetch data from memory as part of their traversal operation, and the job manager module needs to retrieve data such as transformation matrices related to animation. While a fully-functioning memory module has been largely omitted from the requirement specification, some rudimentary form of memory handling must necessarily be included. Any logic to this end should naturally be formulated as its own module so that this communication may be facilitated and optimised. The memory module is shown in Figure 6.1, but its communication interfaces with the mentioned modules are not drawn in the diagram.

With the exception of memory interfaces, every module interface is based on the *ready-valid* protocol described in Section 2.5.5. A consequence of this that may not be obvious at first glance is that if the surroundings fail to extract the results from the ray tracing system, the entire system will grind to a halt, since no module is ready to accept new results or jobs from the module before it in the data flow. Once a result is fetched, the result manager module will be ready to accept new results from the traversal cores, and the system will run again. This situation highlights the fact that the throughput of the ray tracing system may be limited by external factors. It also means, fortunately, that no result will ever get lost.

## 6.2 Module design

Now that the system has been modularised into a set of discrete modules, a closer look at each of its modules is warranted. In the following sections, all the modules of the system will be presented and their internal design derived and discussed.

### 6.2.1 Job manager

The main entry point in the ray tracer system is the *job manager*. This module accepts an input ray package from the external data lines of the system and from it constructs job in a format more suitable for the internal SVO traversal cores.

#### Interface

The job manager interface is shown in Figure 6.2. The signals on the left-hand side in the diagram are coupled directly with the external signals of the RTU system interface, while the signals on the right-hand side are connected to the next module in the data path—the scheduler.


Figure 6.2: The interface of the job manager module.

#### Internal modularisation

As shown in Figure 6.3, the module can be split into three stages which will be described in detail in the following. Briefly stated, in the first stage the job manager takes the ray parameters of the input ray object and applies any transformation needed to facilitate animation of the SVO model to be traversed. In the second stage, it calculates the traversal parameters required by the SVO traversal algorithm. Finally, in the third stage, it converts floating-point numbers to fixed-point numbers. After the job manager has constructed a job from the calculated parameters, it passes this job on to the scheduler.

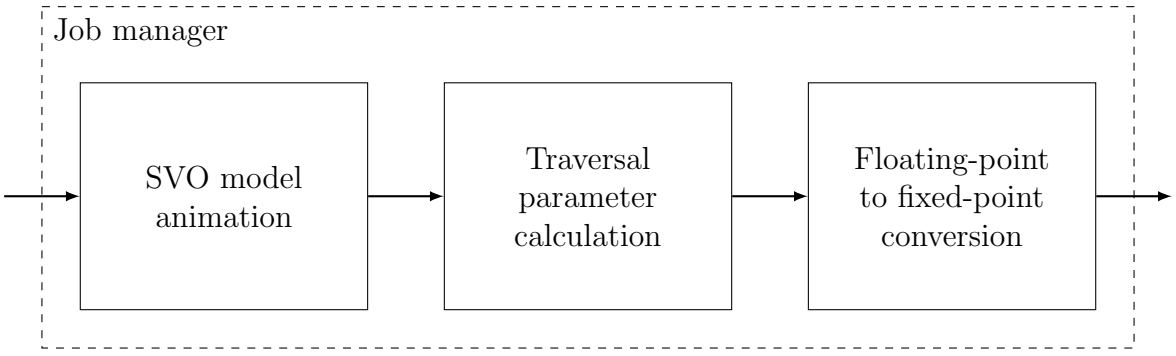



Figure 6.3: Internal modularisation of the job manager.

While it may not have been immediately obvious, the pipelining effort done as part of the system modularisation resulted in a section of the SVO traversal algorithm

described in Section 4.1 being relocated from the SVO traversal module to the job manager module. Specifically, the entire first phase of the algorithm is now calculated by the job manager. This is the phase that entails taking the ray origin and direction, and from it calculating the $t$-values and the direction coefficient $a$ that are employed in the second and third phase of the algorithm.

### First stage: SVO model animation

The first internal stage of the job manager as shown in Figure 6.3 encompasses all the calculations related to SVO model animation. The technique for animation of SVO models was introduced in the project thesis [1], and reiterated in detail in Section 4.3. Briefly stated, in order to achieve efficient animation of the SVO models to be traced, the rays must be transformed inversely corresponding to the orientation of the SVO. The mathematical formulation is shown in Equation (6.1).

$$\begin{aligned} T : R_t(\mathbf{r}_o, \mathbf{r}_d) &\mapsto \hat{R}_t(\hat{\mathbf{r}}_o, \hat{\mathbf{r}}_d) \\ \text{such that } &\begin{cases} \hat{\mathbf{r}}_d = \mathbf{M}_R^{-1}\mathbf{r}_d \\ \hat{\mathbf{r}}_o = \mathbf{M}_R^{-1}\mathbf{M}_T^{-1}\mathbf{r}_o \end{cases} \end{aligned} \tag{6.1}$$

The job manager module is responsible for performing this operation on the rays. It takes the input ray components and applies the transformation of the current SVO model as described by Equation (6.1), before passing these transformed components on to the next step. The transformation of the ray components is performed by matrix multiplication of floating-point numbers, an operation which is often realised in hardware as a set of *multiply-accumulate* (MAC) circuits [37, p. E-5]. In the implementation of the floating-point matrix multiplication circuit, inspiration may be drawn from solutions such as the design presented by Zhuo and Prasanna [96]. This solution employs a parallel systolic structure based on a MAC circuit that they introduce in their paper. Found in the paper by Sajish et al. [97] is a hardware module suited for FPGA implementation which may also be relevant. The co-processor solution introduced by Tertei, Piat, and Devy [98], should also be evaluated as part of this effort, as they claim their FPGA implementation outperforms all other similar modules.

The matrices reside in the memory associated with the SVO model, and are fetched by the job manager in order to perform the transformations. These matrices are updated at most once per frame, which means that they are unchanged for many thousands of rays at a time. In order to minimise the effects of memory latency and presumably improve performance drastically, these matrices only need to be fetched at the beginning of each frame, and can be cached for all subsequent ray calculations in the same frame. It should be mentioned that in the project thesis, additional optimisations were introduced to improve general performance. These optimisations—such as the hit buffer object (HBO), and the sorting of models based on their bounding sphere—will not be considered in the current design, but can certainly be investigated as part of future work.

### Second stage: traversal parameter calculation

The next stage requires the job manager to calculate the traversal parameters that are to be employed by the SVO traversal modules. The parameters are a fundamental

part of the algorithm in Section 4.1, and are determined by the expressions shown in Equation (6.2). For simplicity, only the two-dimensional case is considered here. Any calculations for the third co-ordinate are fully analogous to the first and second.

$$\begin{aligned} t_{x0} &= (x_0 - r_{ox})/r_{dx} \quad \wedge \\ t_{x1} &= (x_1 - r_{ox})/r_{dx} \quad \wedge \\ t_{y0} &= (y_0 - r_{oy})/r_{dy} \quad \wedge \\ t_{y1} &= (y_1 - r_{oy})/r_{dy} \end{aligned} \tag{6.2}$$

The expressions require a circuit that can supply the division functionality. Savas et al. [99] introduce a relevant design in their 2017 paper which is described as having a small size, low latency, and high throughput. Their design is based on a two-stage method where the inverse of the denominator is first calculated. Subsequently, this inverse is multiplied by the numerator. Another design that could be evaluated is the solution by Peng et al. [100]. Alongside a square-root unit, they introduce a division circuit suited for SIMD applications.

Before these traversal parameters are calculated, the direction coefficient $a$ needs to be determined. This is because the calculation of $a$ may alter the ray origin and direction. The calculation involves deriving a coefficient as shown in in Equation (6.3), where the values $s_i$ are set to 1 if the original ray direction is negative for component $i$, and 0 otherwise.

$$a = 4s_z + 2s_y + s_x \tag{6.3}$$

After the direction coefficient $a$ has been determined, the ray direction and origin components are, if negative, modified as shown in Equation (6.4). Following this modification, the traversal parameters shown in Equation (6.2) may be calculated.

$$\begin{aligned} r_{di} &= -r_{di} \\ r_{oi} &= c_i - r_{oi} \end{aligned} \tag{6.4}$$

As a consequence of the SVO animation step, the model to trace will always be located in the origin of the co-ordinate system and have unit size along each dimension. This means that the above calculations may be further simplified. The whole process of deriving the $t$-values and $a$ ends up being as shown in Equation (6.5).

$$\begin{aligned} a &= 4s_z + 2s_y + s_x \\ t_{x0} &= \begin{cases} -(1 + r_{ox})/r_{dx} & \text{if } r_{dx} > 0 \\ (1 - r_{ox})/r_{dx} & \text{if } r_{dx} < 0 \end{cases} \\ t_{x1} &= \begin{cases} (1 - r_{ox})/r_{dx} & \text{if } r_{dx} > 0 \\ -(1 + r_{ox})/r_{dx} & \text{if } r_{dx} < 0 \end{cases} \\ t_{y0} &= \begin{cases} -(1 + r_{oy})/r_{dy} & \text{if } r_{dy} > 0 \\ (1 - r_{oy})/r_{dy} & \text{if } r_{dy} < 0 \end{cases} \\ t_{y1} &= \begin{cases} (1 - r_{oy})/r_{dy} & \text{if } r_{dy} > 0 \\ -(1 + r_{oy})/r_{dy} & \text{if } r_{dy} < 0 \end{cases} \end{aligned} \tag{6.5}$$

There is a symmetry for the calculations above which means that only four values need to be calculated regardless of the sign of the direction. If the sign is negative, the $t$-values are simply flipped so that $t_{x0}$ becomes $t_{x1}$, and vice versa. Moreover, all the calculations may be expressed on the form shown in Equation (6.6), which suggests that they lend themselves to certain optimisations that may be done in hardware. By using a division circuit where the reciprocal is firstly calculated—for instance the one devised by Savas et al. [99]—the entire operation may be realised in hardware as two circuits: a reciprocal and a MAC circuit.

$$t_i = -r_{oi} \cdot \frac{1}{r_{di}} \pm \frac{1}{r_{di}} \tag{6.6}$$

The optimised operation would consist of two steps. Initially, the reciprocal of the ray direction would be determined. Subsequently, by utilising a MAC circuit, the reciprocal would be multiplied with the ray origin, and then accumulated with itself. The $\pm$ notation on the last term would be handled by computing both versions of the result. As was argued in the previous, both these results must be determined anyway, as a result of the symmetry of the calculations.

### Third stage: floating-point to fixed-point conversion

To allow proper matrix multiplication, the job manager receives its input numbers and does all the transformations on floating-point form. As will be substantiated in their section, however, the SVO traversal cores only accept fixed-point numbers. This means that the resulting traversal parameters must be converted from floating-point to fixed-point format if they are to be used by the SVO traversal cores. This conversion between floating-point to fixed-point is quite straightforward, and is illustrated in Figure 6.4.

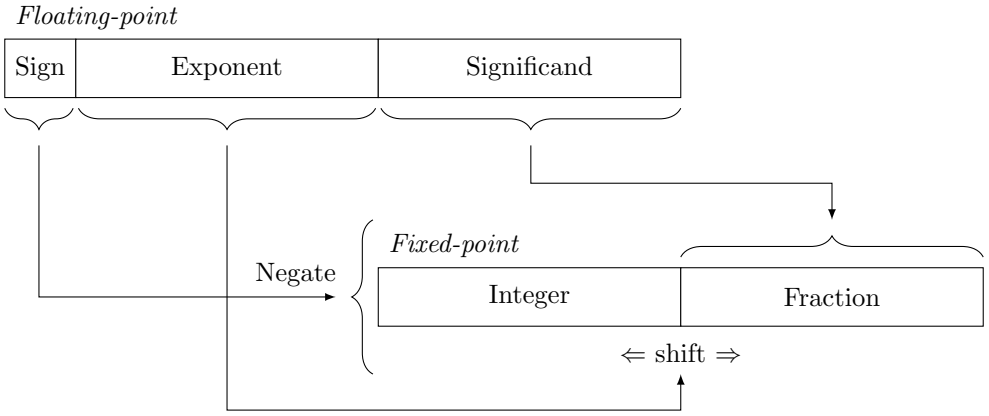



Figure 6.4: The process of converting floating-point numbers to fixed-point representation.

Firstly, the significand of the floating-point number is read and stored in an intermediate register. This number is then prepended the implicit leading 1 which is not present in the floating-point format. Secondly, the exponent is read, and for each of the following clock cycles the significand is shifted according to the value of the exponent. Lastly, the sign bit is read and the number is negated by taking it's two's complement

if the sign is negative. This entire operation may be pipelined so that one conversion is performed each clock cycle.

As will be substantiated in Chapter 7, the fixed-point numbers used in this thesis are 32 bits wide, with 16 bits set aside to store the fractional component. However, in order to not lose any bits of information during the shifting stage, the intermediate register in which the significand is stored must be sufficiently large. For the single-precision floating-point representation used in this thesis, the significand is 23 bits wide. As a consequence, the fractional component of the intermediate register must be at least this size even though the resulting fixed-point number may only use 16 bits for the fraction. After the shifting step is finished, the resulting number may be truncated to its nominal bit width.

Since number used in this thesis is 32 bits wide and uses 16 of these bits to hold the fractional component, it stands to reason that the significand may only be shifted a maximum of 15 places in each direction before the number over- or underflows. This means that the entire number can be set to the minimum or maximum value if the exponent is outside the range $[-15, 15]$. If the exponent is within this range, it can be shown that an entire shifting operation requires no more than four distinct steps. The key is to evaluate the exponent's absolute value as a binary number. Each of the bits of this binary number will then correspond to a shifting operation by $2^n$ steps, where $n$ is the zero-indexed position of the bit. Pseudocode for the evaluation is shown in Figure 6.5.

```
1.  if e > 0 then
2.      if e_3 = 1 then shift the significand 8 places left.
3.      if e_2 = 1 then shift the significand 4 places left.
4.      if e_1 = 1 then shift the significand 2 places left.
5.      if e_0 = 1 then shift the significand 1 places left.
6.  else if e < 0 then
7.      if e_3 = 1 then shift the significand 8 places right.
8.      if e_2 = 1 then shift the significand 4 places right.
9.      if e_1 = 1 then shift the significand 2 places right.
10.     if e_0 = 1 then shift the significand 1 places right.
```

Figure 6.5: The significand shifting process. The variable $e$ is the exponent itself, while $e_n$ signifies bit $n$ of the *absolute value* of the exponent.

Each tier of the shifting process takes one clock cycle and only needs to be performed once for each number. This means that the entire conversion operation is very suitable for pipelining. Shown in Figure 6.6 is an example of such a pipelined implementation. Upon entry to the pipeline, the input significand is sorted into one of two paths based on the sign of the exponent. If the exponent is positive, the significand will potentially be left shifted by some amount each clock cycle. Conversely, if the exponent is negative, it may be right shifted. Each of the shifting stages in the pipeline is completed within one clock cycle, and the intermediate result stored in registers placed between each shifting tier. Once a significand has passed through the entire pipeline, it is truncated and outputted as the fractional part of a fixed-point number.

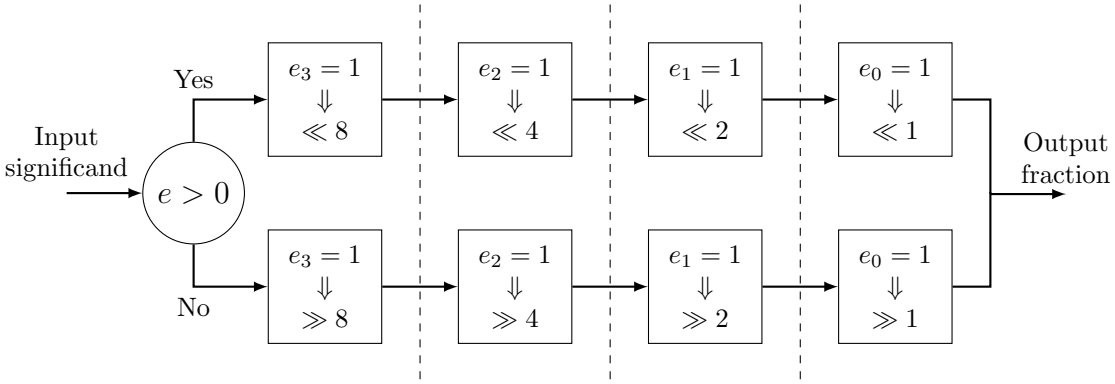



Figure 6.6: A pipelined version of the shifting process.

### 6.2.2 Scheduler

When the job manager is finished processing a job, it is passed on to the *scheduler*. The scheduler module is responsible for delegating jobs to the SVO traversal cores in an efficient manner. This involves keeping track of which cores are busy processing another job, or idle and ready to accept a new job.

#### Interface

The module interface of the scheduler is shown in Figure 6.7. The left-hand side signals are connected directly to the job manager, while the right-hand signals are connected to the SVO cores. Since only one job can be assigned to a single core each clock cycle, the job bus is scalar and connected to all cores. The `ready` and `valid` signals from the cores are treated as a bus of signals. The scheduler utilises the `ready` signals from the cores to keep track of which cores are idle and which are busy, and the `valid` signals as a form of chip-select signal in order to choose a core to assign the next pending job to.

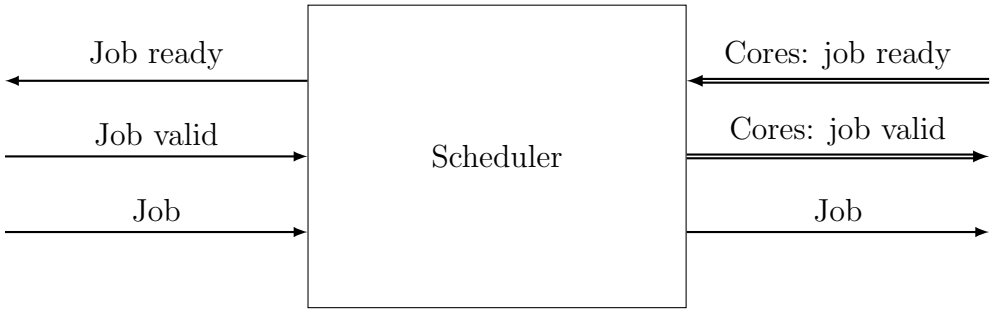



Figure 6.7: The scheduler module.

#### Scheduling algorithm

There is a desire to maximise the utilisation of all the SVO traversal cores. To this end, there should be a discussion about different scheduling algorithms and which ones that might be suited for this use case. In Section 2.2.5, two different scheduling

algorithms were presented, and it would seem that both round-robin and first-available are applicable solutions for this module.

Intrinsic to the round-robin algorithm is the notion that it may make the system stall for a number of clock cycles whenever the next core in line is busy. This situation may arise even though there are idle cores after it ready to accept new jobs. The first-available scheme alleviates this problem, but is more involved to implement efficiently in hardware. An account of these issues, a simplified variant of the first-available scheme has been designed. The scheme, illustrated in Figure 6.8, will be referred to as the *cyclic first-available* scheme, and works as described in the following.

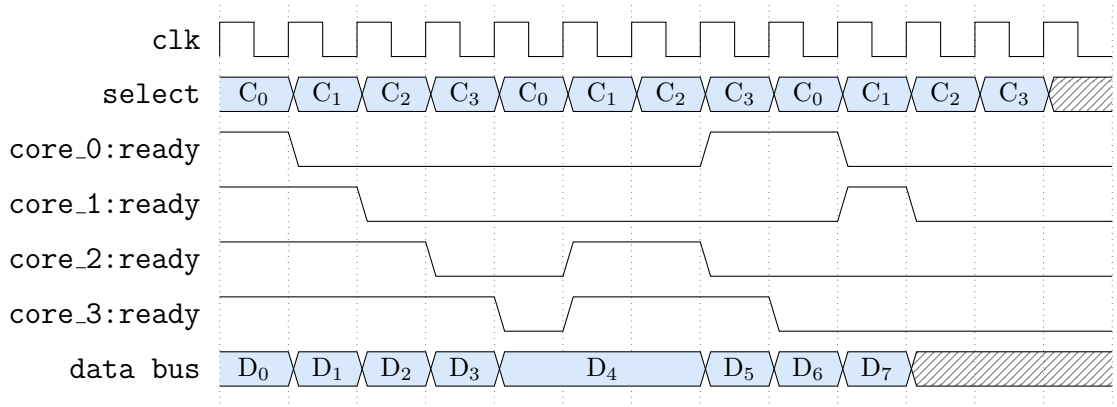



Figure 6.8: Example of the cyclic first-available scheduling.

The scheduler continuously loops through all cores, selecting a new core each clock cycle. When a core is selected, the scheduler checks it if is ready by testing its `ready` signal. If a job is pending and the currently selected core is ready, the pending job is assigned to the selected core. A job is said to be *pending* whenever the job manager has finished processing a job and has passed it on to the scheduler so that may be assigned to some core.

As a consequence of the scheduler continuously looping through all the cores, one core cannot hold up the entire system. Using the cyclic first-available scheme, there may be worst-case situations where a core will never be assigned a job, since no job is ready to be assigned whenever it is selected by the scheduler. These situations, however, should presumably only arise very rarely, and would suggest that there are an excessive amount of cores implemented in the system.

Looking at the worst-case delay before a job is assigned in a system with at least one ready core should highlight the compromise this scheme presents. These worst-cases are listed in Equation (6.7), in which the symbol $n_c$ represents the number of cores in the system, and $W_{\text{J}}$ is the worst-case processing time for a job.

$$\begin{aligned} W_{\text{RR}} &= W_{\text{J}} \\ W_{\text{CFA}} &= n_c - 1 \\ W_{\text{FA}} &= 1 \end{aligned} \tag{6.7}$$

The worst-case delay of the cyclic first-available scheme, denoted $W_{\text{CFA}}$, is demonstrated in Figure 6.8 when the job $\text{D}_4$ is delayed three clock cycles. Assuming $n_c < W_{\text{J}}$, the scheme improves upon the round-robin algorithm, for which a worst-case delay $W_{\text{RR}}$

would manifest itself as a situation where the ready core is blocked by every other core processing a worst-case job in parallel. The number of cores have no impact here, since every other core has already been processing the job for at least $n_c$. As expected, CFA performs worse than the more complex first-available algorithm that was presented in Section 2.2.5, for which the worst-case delay $W_{\text{FA}}$ is 1 clock cycle.

### 6.2.3 SVO traversal core

The main computation happens in the *SVO traversal core*, whose function is to take a job, and traverse an SVO memory structure based on the setup information contained in this job. The core should follow the algorithm by Revelles, Ureña, and Lastra [53], presented in Section 4.1, in order to traverse an SVO model on the data structure specified by Laine and Karras [35], which was presented in Section 4.2. When the core is done, it should return a result which describes the outcome of this traversal procedure. According to the requirement specification the result should contain information about whether a solid node was hit, and if so, the parametric value along the ray of the hit (the depth), the cost of the traversal process in number of clock cycles spent, and the normal of the surface that was hit. The core should also pass on the unique ID of the ray tracing job so that this ID can be outputted as part of the result.

The module should be designed so that it may be duplicated to cater to the ambition of a scalable system. This duplication will enable the system to process multiple rays in parallel, and help alleviate the predicted bottlenecks associated with the traversal process. Furthermore, the module may contain memory caches as part of its communication with the memory module. These caches will then help mitigate the expected issues related to memory latency.

#### Interface

The SVO traversal module's interface is shown in Figure 6.9. On one end, the signals are connected to the scheduler. Through these input signals, the core accepts jobs and signals whether it is busy or idle. On the other end the signals are connected to the result manager. The core will through this output interface return the results from its assigned jobs.

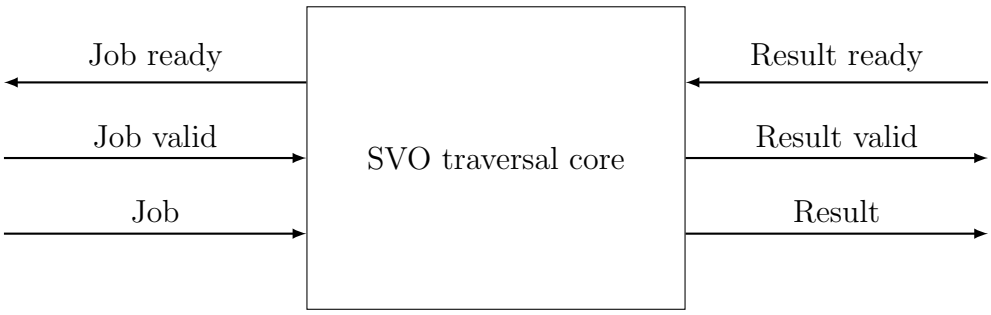



Figure 6.9: The SVO traversal core module.

### Adaption of the algorithm

The main difference between the original algorithm and a hardware adaption of it is that a switch from a recursive approach to a flat, iterative approach should be done. The original algorithm was described as a recursive, top-down method with neighbour-finding, but this kind of recursion is generally undesirable in a hardware design. The reason for this is that in order to support recursion, a global function-call stack must be maintained on which the current execution context—such as the program counter, parameters, and local variables—is stored [40, p. 407]. Luckily, the original algorithm is very adaptable, and can easily be converted to use an iterative approach. This was demonstrated with great success in the project thesis [1], as well as in the master's thesis by Wilhelmsen [83].

Accordingly, the design presented here incorporates an iterative state machine. Still, while the algorithm implementation itself is not recursive, the traversal of the SVO data structure is bound happen in a recursive manner. This is because the chosen data structure can only be efficiently traversed in a top-down manner. To facilitate such recursive traversal, each traversal core in the implemented design features a very rudimentary stack on which the current state can be pushed when recursing into the data structure. The state can then be retrieved when backtracking up so that previously calculated variables do not have to be recalculated.

### Number representation

Before implementing the module, a choice was made as to which number representation that should be used. The choice fell on fixed-point, and will be substantiated in the following.

As detailed in Section 2.2.4, the main advantage of fixed-point numbers compared to floating-point numbers is that calculations involving the former are much easier to implement in hardware. And the fact of the matter is that the original SVO traversal algorithm only requires two basic arithmetic operations: addition and division by two. These two operations are very easy to implement for fixed-point numbers—as a simple adder and right shift operation—but comparatively costly in terms of throughput and hardware resources to implement for floating-point numbers.

### State machine

The algorithm is implemented in hardware by defining a set of distinct states and the transitions between them. These states were determined by taking a functioning software model of the algorithm and reducing it down to a state machine. The software model, which will be discussed further in Section 7.2, was reduced until the point where dependent calculations made further reduction impossible. The result is a set of seven states in which the traversal core may be, as well as a set of calculations and evaluations describing the transitions between them. The states and their meaning are listed in the following.

- `IDLE`: The initial state in which the system is brought after reset. In this state, the core is idle and ready to accept a job assigned to it by the scheduler.

- `INIT`: The scheduler has assigned a job to this core, and it has just been accepted. In this state, the core is determining which direct child of the root that is initially pierced by the ray based on the parameters of the job.
- `EVAL`: The core is now evaluating the current node. The validity of the ray is assessed, which means that it is determined whether or not the ray hits the SVO at all. If it is valid, it is established whether or not the ray hits a solid direct child of the current node.
- `NEXT`: The ray did not hit anything in the current node. The core is moving on to determine which sibling node to evaluate next.
- `PUSH`: The current node is not a leaf node, which means that it must be recursed into. The core is pushing its current variables onto the stack, and fetching the child nodes of the current node from memory.
- `POP`: The ray did not hit anything in the current node or any of its siblings. The core is now backtracking one step up in the SVO tree to continue the traversal.
- `OUT`: The core has finished evaluating the tree and is ready to report a hit or miss. It will stay in this state until the result has been fetched by the result manager module.

In Figure 6.10, a state machine diagram is shown. This diagram illustrates graphically each of the states and the transitions between them. As explained above, the initial state is `IDLE`. The traversal core will spend one clock cycle in each state, which means that the minimum processing time for a ray is four clock cycles. In other words, at least four SVO traversal cores must be present in the system to achieve the throughput of one ray per clock cycle.

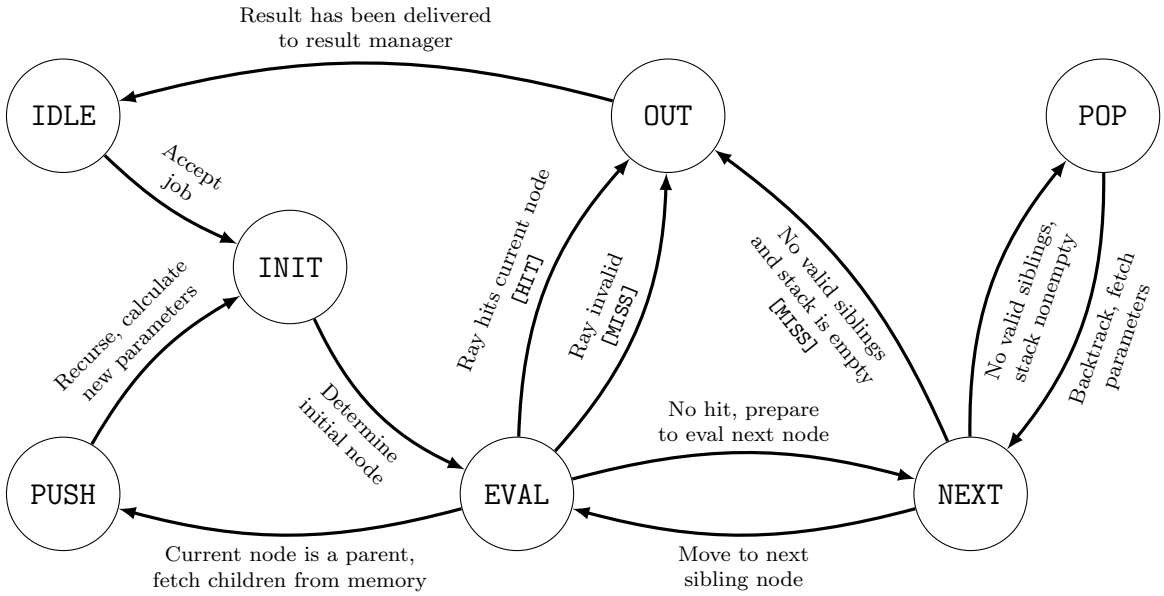



Figure 6.10: The internal state machine of the SVO traversal core.

### Memory interface

The SVO traversal module interfaces the memory module in order to read the SVO data structure. However, since the requirement specification does not include a fully-functioning memory module, support for stalls in relation to memory latency has not been incorporated in the design. This means that the cores currently assume that every memory access happens immediately, without delay. One approach that would extend the module and offer support for traversal cores being blocked waiting for memory is outlined in the following. An eight state could be introduced to the system—for instance named `BLOCKED`. This new state would be inserted between `EVAL` and `PUSH`, as this is where new nodes are fetched from the memory. The system would simply remain in this `BLOCKED` state until the data from memory becomes available. Extending the state machine in this manner is a topic for future work.

As discussed in the requirement specification in Chapter 5, it is expected that memory latency will play a huge role in design performance once implemented. The latency of the memory accesses could be improved by utilising memory caches. For instance, an L2 cache may be included in the memory module, while placing L1 caches in every SVO traversal core. Furthermore, the initial node of the model to be traced will necessarily always be a direct child node of the octree root node. Therefore, this node data could be cached in a register in the core to avoid having to fetch identical data from memory for every job.

## 6.2.4 Result manager

In order to multiplex the many SVO traversal cores that may be instantiated in the system, a final module is needed. The *result manager* keeps track of all results, and exposes an external interface for the retrieval of the result of each submitted ray.

### Interface

The interface of the result manager is displayed in Figure 6.11. As was the case for the scheduler module, the result manager is connected to all the traversal cores and monitors their output lines awaiting results. The result manager's output lines are coupled directly to the external lines of the RTU so that results may be delivered to the environment.


Figure 6.11: The result manager module.

#### Result buffering mechanism

The result module must return results on the format specified in Section 5.3.3 of the requirements. It should additionally contain a buffering mechanism so that it can accept results from SVO traversal cores and ready these for transmission to the environment.

The chief function, perhaps, of the result manager is this buffering of the results from the SVO traversal cores. The mechanisem will help decouple the traversal cores from the environment and pipeline the result output so that the traversal cores do not have to wait for results to be retrieved before starting to process new jobs. In other words, the cores may then immediately begin processing new jobs, not needing to wait for the results to be retrieved by the environment. To this end, the result manager contains an array of buffers—one for each SVO core. In order to demultiplex this array of results to the environment, a scheme similar to that of the scheduler is be used. The result manager will continuously loop through the buffers, exposing each of them to the environment in turn.

An idea for future work is to investigate whether adding multiple layers of buffers will increase performance. These buffers could be configured so that they sort the output by the jobs' unique ID before delivering them externally. Doing this ensures that the results are retrieved in raster order—the same order that they were submitted to the RTU. It might be possible to achieve a significant performance gain by sorting the results in hardware instead of offloading this to the software driver.

### 6.2.5 Memory

The last module included in the system is the memory module, whose main responsibility is to communicate with any module that requires data from memory. Specifically, the memory module needs to interface the SVO traversal cores and the job manager, as these are the two types of modules that currently need data from memory. The traversal cores must fetch the SVO models to be traced from memory, while the job manager must retrieve any data associated with animation.

#### Interface

The interface of the memory module is as shown in Figure 6.12. Each of the modules that communicate with the memory module has their dedicated address and data lines. Once an address is put on the address bus, the data is immediately returned on the data bus.

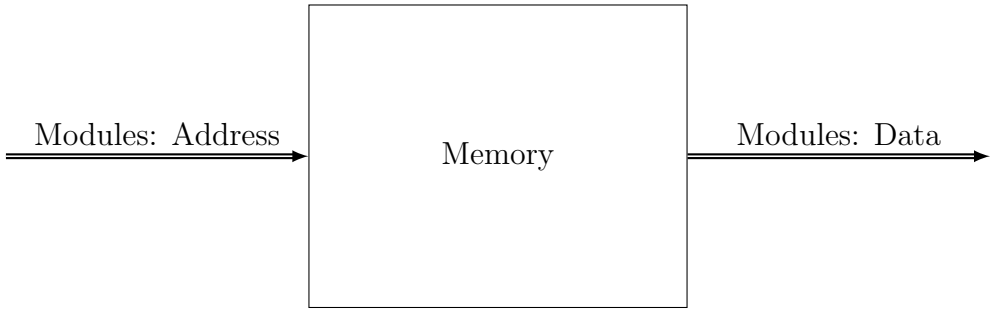



Figure 6.12: The simplified memory module.

### Implications of thesis scope reduction

As was explained in the requirement analysis in Chapter 5, the memory module has not been the main focus of this thesis. Since including a fully-fledged memory module with caches would make the scope significantly larger, the memory module is currently specified and designed as a read-only hard-coded data store. A consequence of this reduced memory module is that the size of the models that can be rendered is considerably reduced compared to what would be possible with a full memory module. The limitation stems from the fact that any model would have to be hard-coded and synthesised as part of the memory module design.

The reduced memory module also negates the need for memory caches. However, a proper memory module would in all likelihood require memory caching to be implemented to further reduce latency issues. As discussed in the section describing the SVO traversal module, the traversal cores may contain an L1 cache to this end. Latency issues could be further alleviated by introducing an L2 cache in the memory module consisting, for instance, of fast block RAM.

Both a true memory module and memory caches are topics that should be revisited as part of future work. It would be highly interesting to see what effect this might have on system performance, especially if coupled with the introduction of a `BLOCKED` state in the SVO traversal core.

# Chapter 7

# System implementation

The abstract system design of the ray tracing unit (RTU) and its modules presented in Chapter 6 is now to be employed as a foundation for a hardware implementation. This chapter opens with a presentation of the target technology, and follows up with the details of the hardware implementation and verification methodology. The results from the implementation, such as timing and utilisation reports, are presented and discussed in Chapter 8.

## 7.1 Target technology

There are several viable target technologies that may be employed for implementation of a hardware design. The choice of which medium might be best suited for implementation is in many cases a matter of assessing the applications of the end product. For instance, if the end product is to be mass-produced and sold as a packaged circuit, an ASIC would be the logical choice from an economical standpoint. In other cases, however, one might benefit from keeping the development process itself in mind. The ASIC approach, for instance, would require the design to be sent to a manufacturer for production in order to realise it as a circuit. Hence, iterative design exploration on an ASIC would presumably be infeasible.

For the development work in this master's thesis, there is virtually only one practical alternative that has the desired qualities for hardware implementation—the FPGA. When compared to other types of media, such as the ASIC, it is clear that while it may result in a slower and less streamlined implementation, only the FPGA can provide the short turnaround time desired for this project. The development process will in all likelihood entail a fair bit of design exploration and general trial and error, and an FPGA would accommodate such a development methodology well. Furthermore, there is simply not available budget to allow for an ASIC design in this master's thesis, as manufacturing expenses can be in the millions [58, p. 69].

Based on these considerations, the FPGA was elected as the target technology for the hardware implementation. The chosen platform is the *PYNQ-Z1: Python Productivity for Zynq-7000 ARM/FPGA SoC* [62], which has already been presented in detail as part of the background chapter, in Section 2.5.1. One of the supervisors for this master's thesis, Øystein Gjermundnes, has kindly lent the author one such development board to be used in the development work. The physical development

setup in use for this master's thesis is shown in Figure 7.1. Placed on the right-hand side of the desk is the Pynq development board that was provided.

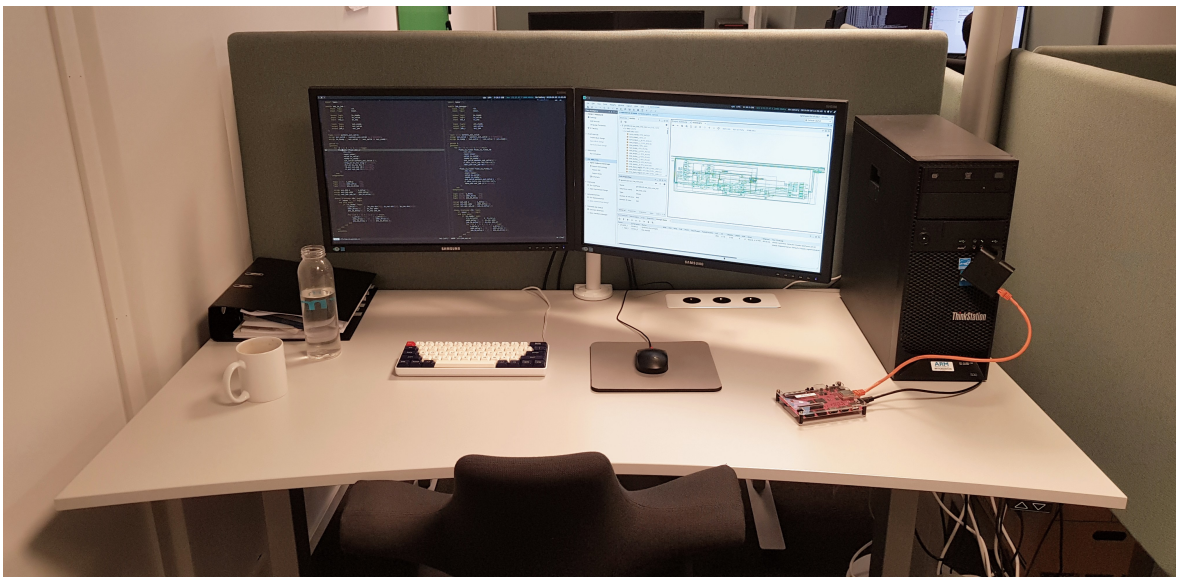


Figure 7.1: The development setup.

The FPGA SoC on the Pynq is the *Zynq XC7Z020-1CLG400C*, which is part of Xilinx's Artix-7 family of programmable logic [62]. The most relevant specifications for this FPGA are provided in Table 7.1. For the design presented in this thesis, the most important figure is the number of available logic slices, and derived from it, the number of slice LUTs and slice registers. These numbers denote how much resources—or *area*—the FPGA can provide when implementing hardware designs. In other words, the higher these numbers are, the larger the designs may be that are to be programmed onto the FPGA.

Table 7.1: Principal specifications of the FPGA. Taken from [62].

| Logic slices (CLB) | Slice LUTs | Slice registers | Block RAM | DSP slices |
|---|---|---|---|---|
| 13 300 | 53 200 | 106 400 | 630 kB | 220 |

The discussion regarding available resources ties in with the scalability consideration that was introduced as part of the requirement analysis in Section 5.3.1. According to the design specification, the system will contain a central computation core—the SVO traversal core—that may be duplicated in order to parallelise the workload and increase throughput. Furthermore, the number of logic slices available on the FPGA will presumably be the limiting factor in terms of the number of SVO traversal cores that will fit on the target technology. This metric will therefore be revisited in the results and discussion found in Chapter 8.

## 7.2 Software model

Before commencing the actual hardware implementation work, a software model of central parts of the design was created. Initially, this model was developed in order to help the author build an understanding the chosen algorithms, and how these may be efficiently adapted to hardware. However, the software model proved itself immensely useful throughout the entire project—not only as a crucial tool to help build understanding, but also as a central part of the verification process that was conducted concurrently with the development.

In the preliminary stages of the development process, the model was employed as an accessory to the design exploration effort. It was used in this manner to allow for quick design exploration with a much shorter turnaround time than for traditional hardware design exploration. In other words, several different approaches to the same problem could be implemented and tested in short order, without having to go through the more cumbersome process of formulating them in RTL code and testing them in hardware. Minor tweaks could also be made without having to simulate or synthesise an entire hardware design repeatedly.

The model is written in Python in a modularised manner that resonates with how the design is implemented in hardware. The model implementation of the SVO traversal core, for instance, contains a state machine in software which runs exactly like its counterpart on the FPGA. By implementing the software model in this fashion, it could conveniently be employed as a reference for comparison during debugging and validation of the hardware design. Specifically, the software model could be treated as the blueprint for how the implementation should behave and function, as it is both *bit-exact* and *cycle-exact* with respect to the hardware implementation.

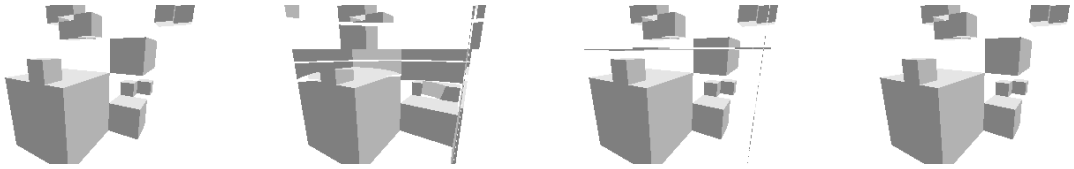
The most relevant components of the finalised software model have been included in Appendix G. Comparing the SVO traversal core model in Appendix G.1 with its actual hardware implementation in Appendix C.4, one can readily see the similarities between the two. The software model's role in the verification process is further detailed in the discussion on verification methodology found in Section 7.5.

### 7.2.1 Fixed-point decimal precision

The software model was employed in a major portion of the design exploration process, which partly consisted of the determination of certain implementation details and parameters. One such configuration parameter is the decimal precision of the 32-bit fixed-point numbers that are employed in the SVO traversal module.

The fixed-point number format was detailed in Section 2.2.4, where it was pointed out that a certain decision must be made when implementing the number representation. This decision concerns how the total 32 bits should be distributed between the integer part, $n_i$, and the fractional part, $n_f$. The trade-off that arises when determining these two components' sizes was subsequently described as part of this background theory section. The software model was therefore employed in an effort to empirically determine the optimal bit distribution for this specific area of application. Several model runs were executed with a selection of configurations, after which the optimal configuration was chosen by visual inspection of the output renders.

In Figure 7.2 the results of these model runs are shown. Firstly, a control image was created, which can be seen in Figure 7.2a. This output was rendered with 32 bits of information set aside for each of the components of the fixed point number. In other words, a total of 64 bits were used to store the fixed-point numbers in this run. Thus, no matter the configuration of a 32-bit fixed-point number, the precision and range can not be better than for this control render.



(a) $n_i = 32; n_f = 32$ (b) $n_i = 4; n_f = 28$ (c) $n_i = 8; n_f = 24$ (d) $n_i = 12; n_f = 20$

(e) $n_i = 16; n_f = 16$ (f) $n_i = 20; n_f = 12$ (g) $n_i = 24; n_f = 8$ (h) $n_i = 28; n_f = 4$

Figure 7.2: Model runs for different configurations of $n_i$ and $n_f$. (a) functions as a control render, with 32 bits allocated to both the integer and fraction parts.

In the runs illustrated by Figures 7.2b and 7.2c, significant visual artefacts appear. These artefacts presumably stem from the fact that not enough bits have been allocated for the integer part of the number. While having a very fine fractional resolution, the fixed-point numbers in these configurations suffer from a tiny integral range. At the other end of the scale, in Figures 7.2g and 7.2h, the edges in the output renders appear to exhibit some sort of oscillation pattern. This "wobbly" visual effect may be attributed to the very limited fractional precision that the fixed-point numbers can provide at these configurations. The result is a very visible quantisation effect that distorts the output image.

Both of the effects witnessed at each end of the scale are undesirable, and their configurations therefore unsuitable for the implementation. Three of the configurations, however—namely the runs illustrated by Figures 7.2d to 7.2f—seem to have none of these artefacts or distortions, and are thus all valid candidates. Ultimately, the decision was to use the configuration shown in Figure 7.2e for the system implementation. This configuration—with both $n_i$ and $n_f$ equal to 16 bits—was chosen for the reason that it is essentially the "farthest" from both of the undesirable effects. In addition, while not a satisfactory argument by itself, it was also noted that this configuration offers an attractive symmetrical elegance by allocating 16 bits for each of the number components, which may ease implementation slightly.

## 7.3 Module implementation

The hardware implementation of the system designed in Chapter 6 is now to be presented. The implementation was written entirely in *SystemVerilog* [66], and synthesised by the *Vivado Design Suite* [69], both of which were presented in Section 2.5.

In Figure 7.3, the top-level schematic of the whole RTU system implemented with four SVO cores is shown. While the text is hardly legible due to limitations of the Vivado rendering options, all the major modules that were described in the specification can be found. Upon closer inspection, it is revealed that the input signals are shown on the bottom left, and are coupled directly to the job manager module on the lower right-hand side. Just above and to the left of the job manager is the scheduler module, which controls the four SVO traversal cores found distributed in the upper left-hand quadrant of the schematic. Centred between the four traversal cores is the memory module. Finally, the result manager is the tall module on the upper right-hand side, which is subsequently connected to the output lines.

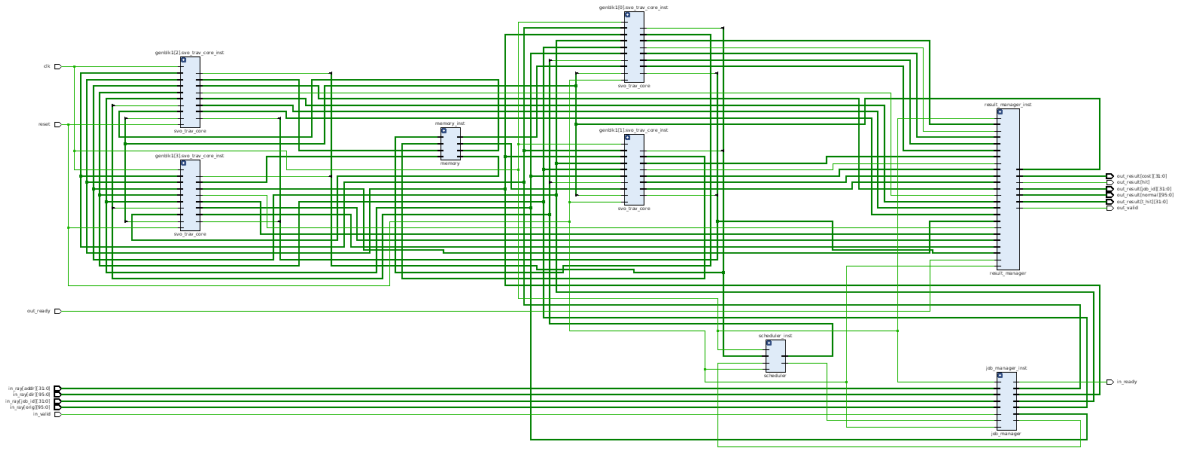


Figure 7.3: The elaborated design for the entire RTU. The schematic is drawn with 4 SVO cores.

The attentive reader may have noticed that the job manager module does not communicate with the memory module in the schematic. This is true for the implementation, and a result of the fact that the job manager module was only partly implemented according to its specification. As will be further explained in the appropriate section in the following, the scope of the implementation work turned out to be too large, and certain features of this module could not be finished in time.

The individual modules that compose the RTU system are now to be presented. The following sections will dive deeper into the implementation details of each module, with justifications given for any design choices that were made. The source code for all implemented modules is included in Appendix C, with the top-level module found in Appendix C.1.

### 7.3.1 Job manager

As stated in its design description in Section 6.2.1, the main responsibility of the job manager module is to transform an input ray, comprised of an origin and a direction,

to an internal job structure that can be passed on to the SVO traversal cores. This entails transforming the ray parameters such that animation of the SVO models can be achieved, before converting the ray components to the set of traversal parameters required by the chosen SVO traversal algorithm. Lastly, the job manager should convert the floating-point numbers to a fixed-point format suitable for the SVO traversal cores. For reference, the source code for this module is attached in Appendix C.2.

### Schematic

The elaborated schematic of this module is shown in Figure 7.4. The schematic is included here to illustrate the general complexity of the module, and is not expected to be readable. Nonetheless, certain information may be gathered from the schematic. The pipelined nature of the module, for instance, is apparent. From left to right, there are layers of registers connected in series that illustrate this pipeline. The six parallel data paths in the right half of the schematic are the floating-point to fixed-point conversion modules.

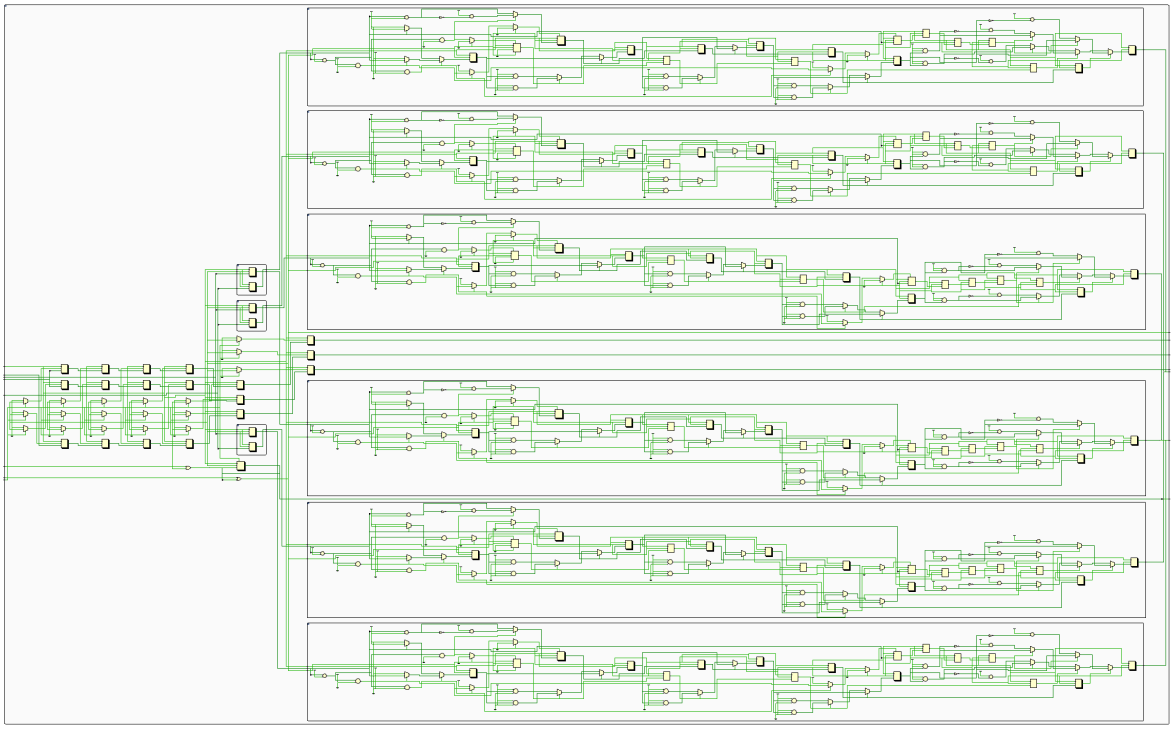


Figure 7.4: Schematic of the job manager module internals. The module has 2844 cells and 12101 nets.

### Comparison to the design specification

As will be outlined in the following, certain aspects of the job manager's specified functionality were not implemented. Their implementation details were explored but, unfortunately, the actual design of these functions could not be completed in time. It would seem that including these in the specification lead to a workload whose scope was simply too large for a master's thesis.

The first functionality that could not be implemented due to time constraints is the matrix multiplication operation needed for SVO model animation. The design of a matrix multiplication unit was explored, and many suitable solutions were referenced in the system design, but their implementation ultimately proved too time-consuming. This functionality is therefore an aspect of the system that should be revisited in future work. Furthermore, this is the reason for the lack of a communication interface between the job manager and the memory module in the top-level schematic.

The second functionality that proved too time-consuming to implement properly is the floating-point division operation used in the traversal parameter generation. Division is generally a very complex operation, and different designs were explored during the implementation work. However, as with the SVO animation, a finished design of a division module could not be finalised in time.

As a result of these limitations in the implemented design, actual animation of SVO models could not be demonstrated. In addition, since the traversal parameter calculation was not fully implemented, the traversal parameters were calculated in the software driver and sent to the RTU instead of the ray origin and direction.

The floating-point to fixed-point conversion stage, however, was implemented and demonstrated to work with huge success. It was implemented as described by the module design in Chapter 6, and pipelined to such a degree that one such operation could be completed every clock cycle.

### 7.3.2 Scheduler

When the job manager is finished processing a job, it is passed on to the scheduler module. This module's main responsibility is to keep track of all the SVO traversal cores and their state. If there are idle cores, the scheduler will assign the next available job to one of them. The source code for this module is found in Appendix C.3.

#### Schematic

The simplicity of the scheduler module is highlighted by its schematic, shown in Figure 7.5. In the figure, one can see that the scheduler is only concerned with the `ready` and `valid` signals, and does not handle the job data. This was a further simplification from the design specification which will be explained shortly.

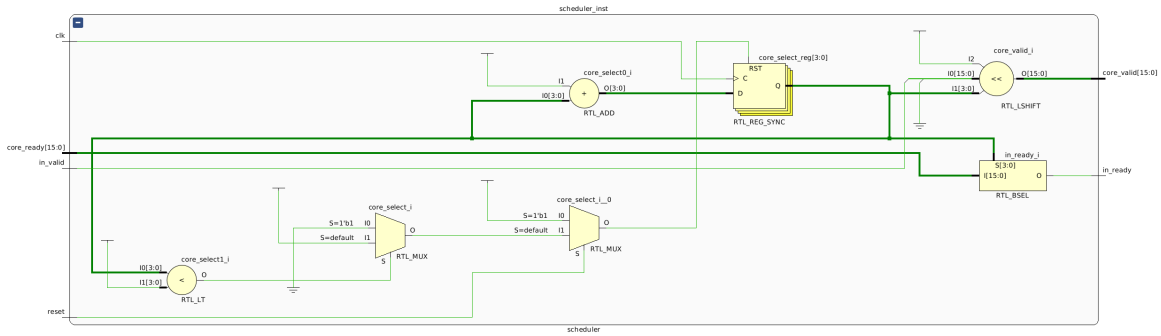



Figure 7.5: Schematic of the scheduler internals. For scheduling 16 SVO cores, this module has 11 cells and 49 nets.

#### Comparison to the design specification

The scheduler was initially implemented exactly as described in the design chapter. It functioned as a simplified implementation of the first-available scheduling algorithm, that passed a pending job from the job manager on to the first available core.

However, in later implementations, the scheduler was further simplified. It was implemented as a simple counting demultiplexer that passes the `ready` and `valid` signals of its currently selected core directly to the job manager. This means that the scheduler does not handle the actual job data at all, and is just concerned with the `ready` and `valid` signals. This effort towards further simplification was made to avoid any overhead associated with the scheduling itself, and can be observed in the source code of the scheduler module, found in Appendix C.3.

### 7.3.3 SVO traversal core

The primary computation logic of the ray tracer happens in the SVO traversal core module. This module is direct hardware adaption of the chosen algorithm described in detail in Section 4.1. When compared, the algorithm implemented in the module remains fairly true to the original algorithm. However, some minor alterations have been made to make it more suitable for hardware design. These alterations were described in Chapter 6. The source code for this module is found in Appendix C.4.

#### Schematic

This module is arguably the most complex and where most time was spent during the design. The schematic in Figure 7.6 can bear witness to the module's complexity. As with earlier dense diagrams, the reader is not expected to be able to decipher the actual schematic—it is only included to serve as an illustration of the general complexity.

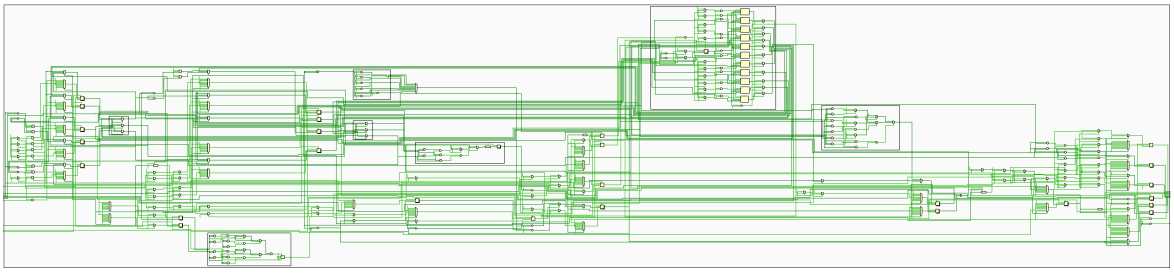


Figure 7.6: Schematic of the SVO traversal module internals. The module has 1047 cells and 5271 nets.

#### Comparison to the design specification

The implementation of the SVO traversal core closely reflects the module design presented in Chapter 6. The interface and function are for all practical purposes implemented identically to their specification. If there is a pending job and the scheduler has selected the SVO traversal module by asserting its `valid` signal, the traversal core can accept the job by signalling its `ready` signal. In other words, a job is accepted by an SVO traversal core whenever both its `ready` and `valid` signals are asserted.

After accepting a job, the core begins the computation by traversing the SVO model referenced in the job according to the given parameters. Once the SVO traversal operation is complete, the core will wait until the result can be passed on to the result manager before it is ready to accept a new job. The communication between the SVO traversal cores and the result manager happens by a similar ready-valid scheme as the job assignment.

In an effort to achieve higher performance and increase the throughput the system, the traversal core module may be duplicated. The scheduler and result manager modules support an arbitrary number of cores running in parallel, which means that the available hardware resources is the factor limiting the number of cores.

### 7.3.4 Result manager

After a SVO traversal core has determined whether a ray results in a hit or miss, the result is transferred to the result manager module. This module is responsible for handling the results from multiple traversal cores, and preparing and demultiplexing these so that they can be fetched through the external interface of the RTU. The source code for this module is found in Appendix C.5.

#### Schematic

The schematic of the module is shown in Figure 7.7. While the module may look complex initially, the following description should clarify that its implementation is actually quite straightforward.

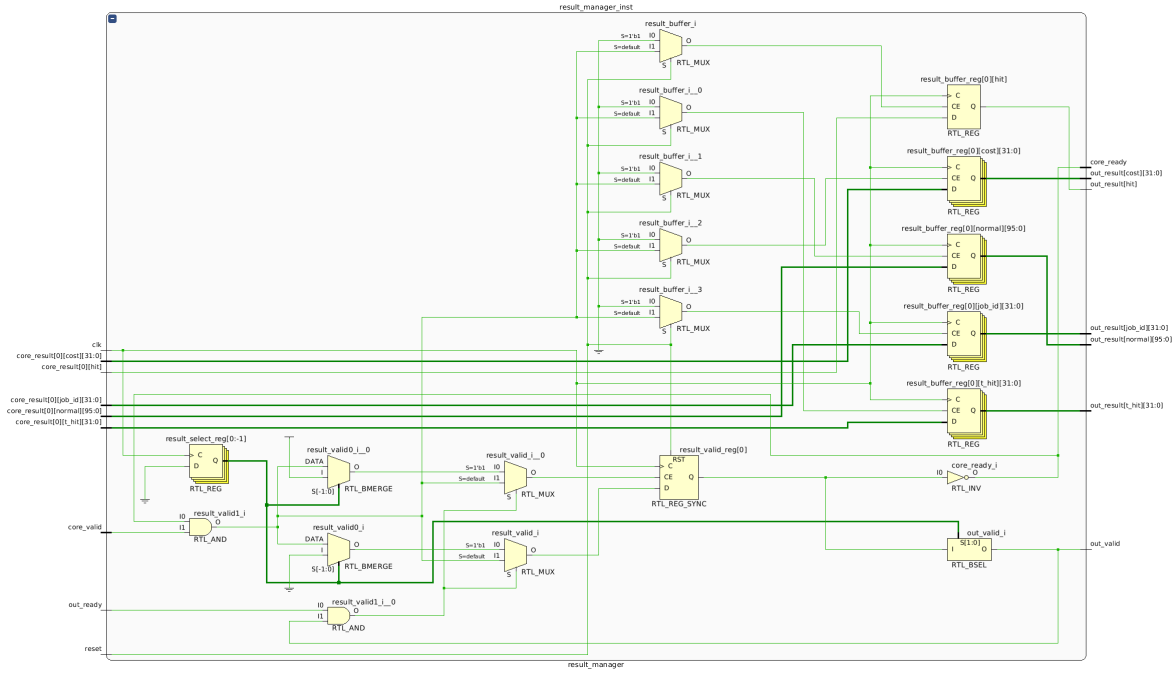



Figure 7.7: Schematic of the result manager module internals. The module has 210 cells and 408 nets.

### Comparison to the design specification

The implementation of the result manager resonates with its design specification to a great extent. It is connected to every core, and contains an array of registers so that each core result can be buffered. Whenever a core is finished, it may hand its result over to the result manager and signal that it is ready to process a new job. In order to expose these buffers on the external lines of the system, the result manager contains a counter that cycles through them. This selection mechanism functions in similar manner to how the scheduler cycles through the traversal cores.

### 7.3.5 Memory

As specified in the requirement analysis, the memory module is not the main focus of this thesis. A fully-functioning memory module has not been designed and implemented in an effort to reduce the overall scope of the project work. A simplified memory module with no latency was included to simulate interactions with memory so that a full module may be revisited and developed as part of future work.

## 7.4 Implementation context and wrapper modules

The Pynq development board requires some additional modules to be included in order to properly implement designs its FPGA. The top-level block diagram of the design implemented in hardware, along with these wrapper modules is shown in Figure 7.8.

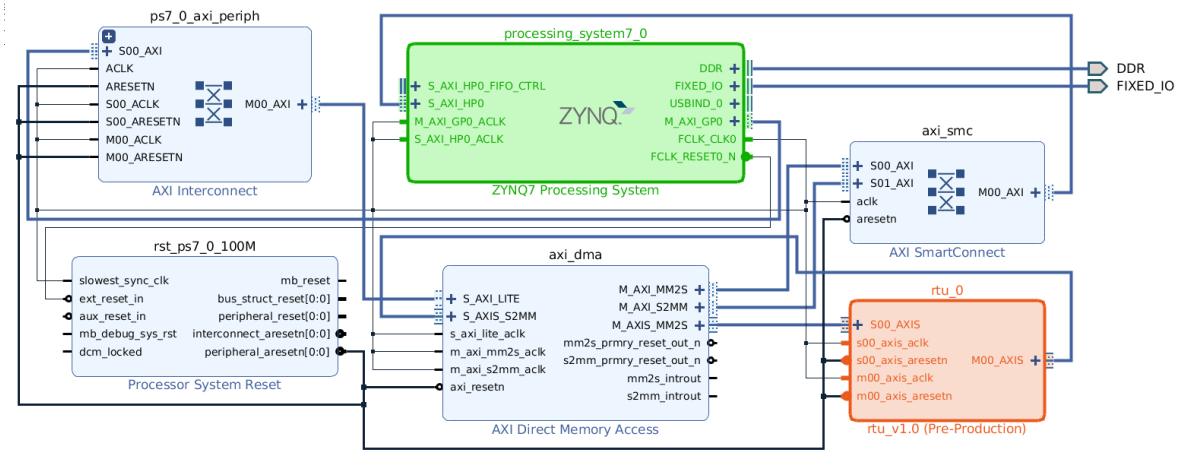



Figure 7.8: Block diagram of the system, as rendered in Vivado's *block design* function. The RTU is implemented as an IP and included in the design using Vivado's IP integrator.

In the block diagram, the RTU system is realised as a single module, highlighted in red. The Zynq7 processing system, which represents the Arm processor on the SoC, is also shown as its own module, coloured green. The other modules in this diagram are responsible for the communication between these two main system modules, as well as basic functionality such as system reset. Some of these helper modules were automatically generated by the Vivado wizard, while others, such as the DMA module, were placed and configured by the author.

At this stage, it is worth noting that the interface of the RTU module in the block diagram closely matches the interface presented as part of the requirement specification in Section 5.3.3. While not immediately apparent, perhaps, the point can be argued by explaining that the ports `S00_AXIS` and `M00_AXIS` on the RTU module in the block diagram are actually visually collapsed buses that contain all signals of the AMBA 4 AXI4-Stream protocol. This means that these buses contain—among a few additional signals—the ready-valid signals and the data lines required of the RTU interface. The four other signals, namely `s00_axis_aclk`, `s00_axis_aresetn`, `m00_axis_aclk`, and `m00_axis_aresetn`, are clock signals and reset signals.

The AMBA 4 AXI4-Stream interface to the RTU is included in Appendix D. It is configured to use a word width of 256 bits, so that an entire 8-byte ray input packet can be transferred each clock cycle. The results have a nominal size of 7 bytes, but are padded to 8 bytes so that they match the bit width of the AXI4-Stream interface.

### 7.4.1 DMA controller

As can be seen in Figure 7.8, the AXI4-Stream buses from the RTU module are connected directly to a DMA module. Direct memory access (DMA) was chosen as the main interface between the processor and the FPGA on the Zynq SoC mainly because of its suitability and simplicity.

The software driver inputs a sequence of rays to the RTU and obtains the results by employing the direct memory access controller present on the Zynq SoC. The DMA controller is initially set up by defining two regions of memory—the first of which holds the data that should be input to the design on the FPGA, the second of which will hold the results output from the design. On the CPU side, the whole interface consists only of these two blocks of memory which makes communication with the RTU very simple. The input block of memory is set up by the driver to contain all the ray jobs that are to be traced. This block of memory is then streamed to the RTU by the DMA controller through the AXI4-Stream input bus. Simultaneously, the second block of memory is ready to be filled with results. The DMA controller streams the results outputted from the RTU into this block of memory, one by one.

A limitation with this approach is that the dimensions of the frames to be rendered are constrained by the size of the DMA memory regions. According to the Pynq specification [64], a DMA memory region must be defined as a contiguous buffer in memory, which is only feasible up to a certain size. Moreover, to properly communicate both input and output, two of these contiguous buffers are needed. On the Pynq-Z1 the largest supported DMA buffers were emperically determined to be around 8 MB each, for a total of 16 MB. Since the input rays and output results each have a size of 256 bits or 8 bytes, this corresponds to a $16 : 9$ resolution of just above $1280 \times 720$. The derivation is shown in Equation (7.1).

$$1280 \times 720 \cdot 8\,\mathrm{B} = 7.37\,\mathrm{MB} \tag{7.1}$$

A consequence of this, unfortunately, is that the full-HD resolution $1920 \times 1080$ is not supported by the DMA approach, as it would require two buffers of 16.6 MB. This is fact is demonstrated by trying to allocate a DMA region for a frame larger than $1280 \times 720$—it simply results in the error "`Failed to allocate memory`". For this

reason, the results presented in Chapter 8 do not contain any performance data for resolutions above $1280 \times 720$.

### 7.4.2 Software driver

In order to simplify the interface between the software and the RTU system on the FPGA, a software driver for it was written. The driver is written in Python and runs in the Pynq environment on the Arm CPU. It is responsible for downloading the hardware design to the FPGA, setting up the DMA controller, and allocating memory regions to be used for communication.

Running as part of the *Jupyter* [65] interactive web interface, the driver enables the user to initiate the rendering sequence and save the resulting render to a file. It can also be used for benchmarking and was employed to record the performance metrics presented in Chapter 8. An example of this usage of the driver is shown in Figure 7.9, while the source code for it can be found in its entirety in Appendix F.

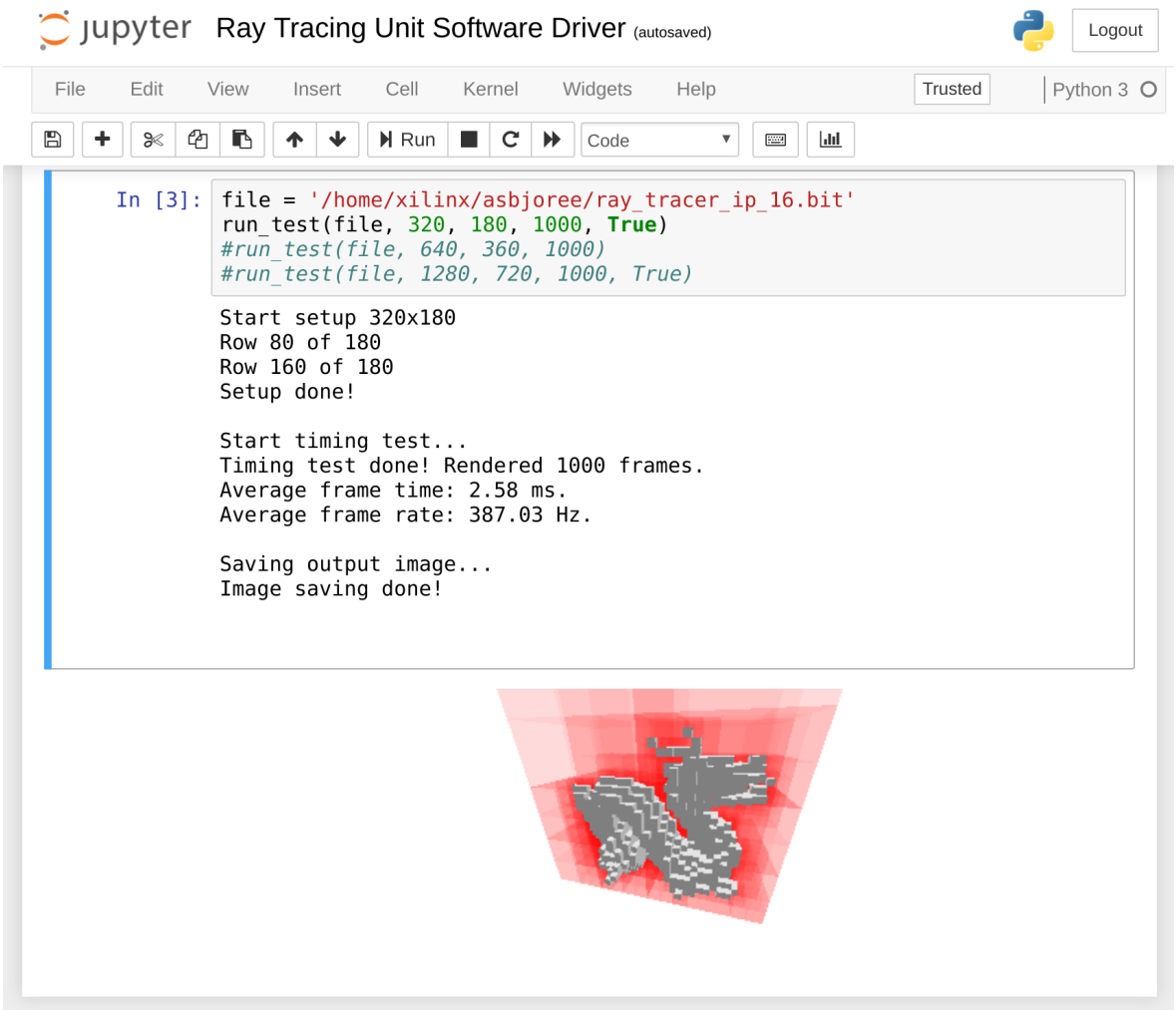



Figure 7.9: Example usage of the driver. The `run_test` function is included in Appendix F.2.

## 7.5 Verification methodology

A major part of the hardware implementation process is the verification of the system and its individual modules. The verification methodology used in the implementation work will be presented in this section, while the initial requirements of the system will be revisited and discussed in Chapter 8.

### 7.5.1 Software modelling

As discussed in Section 7.2, a software model of central modules of the design was prepared before the actual implementation work started. This model served not only as a tool for understanding and design exploration, but was also heavily employed in verification of the implementation in the early stages of the design process.

The software model was first written as a direct implementation of the underlying algorithms. Subsequently, it was modularised and the central modules of the software model were rewritten to reflect how their internal functionality might be implemented in hardware. This means, for instance, that certain routines which were initially implemented as blocking function calls were converted to software modules with internal state machines. In order to model the synchronous nature of hardware, these modules were called repeatedly—once each simulated clock pulse.

By converting an existing functional implementation of the algorithms to these synchronous software modules, the author was forced to think ahead and plan how the design would be best implemented. The process also aided in determining which states the internal state machines should contain, or whether an explicit state machine was needed at all.

Once the hardware design process began, the software model served a central role in functional verification. Used in conjunction with simulations of the hardware modules being designed, the software model was employed as a reference for the modules' desired functionality. For instance, the state machine in the SVO traversal core was verified by submitting the same job to both the hardware design and the software model. The behaviour of both designs could then be recorded and compared down to the smallest detail. Many bugs and design errors were quickly discovered using this method.

### 7.5.2 Simulation

The bulk of the verification effort was carried out utilising software simulations of the RTL code. Simulation runs are much faster than verification using the software model, which means that more exhaustive testing can be done using this method compared to the model. In addition, correctness against the specification can be checked on-the-fly using SystemVerilog's extensive assertion logic.

In a modular design, *unit tests* are often used to verify the functionality of each individual module [58, p. 65]. Since the design in this thesis is heavily modularised, it is natural that unit testing should be employed for the verification of its functional correctness. The concept of modularisation in conjunction with unit testing generally makes verification much easier than for large monolithic designs.

Simulations of a hardware design typically utilise unit testing so that each module of the design can be tested separately. To this end, *test benches* for the given RTL

modules must be written. A test bench is a piece of non-synthesisable HDL code that instantiates the DUT and performs tests on its functionality [58, p. 207]. In a majority of cases, the test bench is an example of a unit test.

Shown in Figure 7.10 is a demonstration of a test bench simulation, highlighting its usage in the verification of system functionality. In the figure, the entire RTU system is simulated by a test bench which submits ray tracing jobs and verifies the outputs. This exact test bench is therefore not an example of a unit test, since it essentially tests the entire system. As signified by the blue cursor, a ray tracing job with the unique ID `0x06F7` is submitted to the system at time 45 034 ns. The ray tracer then processes this job, and its result with the same unique ID is outputted at time 45 860 ns, as highlighted by the yellow cursor.

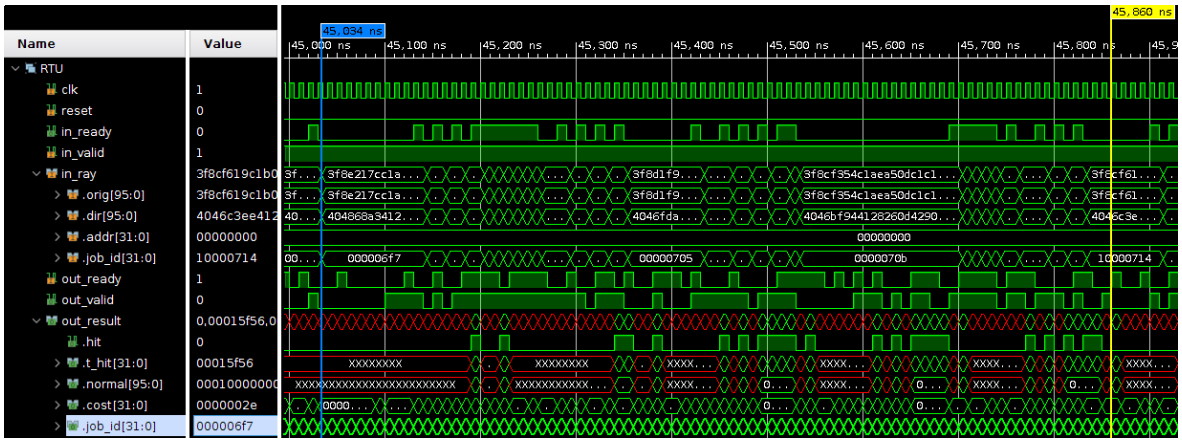



Figure 7.10: Simulation of the top-level RTU module. The `out_ready` signal is randomised.

While software simulations are extremely useful in verification, they do not show the whole picture. One central aspect that is not explored is the viability of a design in terms of timing. Furthermore, the usage of resources and area is not calculated, and whether or not the physical routing of a design is feasible is not determined. This means that quite a number of bugs may pass undiscovered through the verification of a simulated design. In fact, since simulations often contain test benches with non-RTL code, they will not reveal situations where parts of the design itself is not synthesisable. While a simple software simulation will not uncover such design issues, prototyping on an FPGA can be used to this end.

### 7.5.3 FPGA prototyping

In this thesis, FPGA prototyping was employed with huge success in the verification effort. Since the design had been thoroughly modelled in software before the design work, and exhaustively simulated to root out bugs before implementation, the validation load at the FPGA stage was very light. FPGA prototyping is among the fastest verification methods in terms of the raw speed at which the design can be run. Depending on the implemented design, in the case of the chosen FPGA, the clock frequency can be well over 500 MHz [64]—a switching rate that is not feasible using software modelling or

simulation. This means that tests with considerable coverage can be run on the design very quickly.

The downside, however, when the design has been implemented on an FPGA is that the verification engineer forfeits a great deal of the debugging capacity that they had access to with software modelling or design simulation. Observing what happens inside the FPGA when a test fails—and determining the circumstances that lead to the failure—is not straightforward. Nonetheless, there exist techniques that may be used to increase the observability of a design implemented on an FPGA. The most common, perhaps, is the implementation of performance counters.

As stated in the previous section, there are several bugs and issues that might not be rooted out by simulation. One such design error that may not be discovered by simulation, but will manifest itself when prototyping on an FPGA is rooted in the unintended consequences related to the semantics of the `X`-value in Verilog and SystemVerilog. In RTL code, the `X` represent an unknown value [66, p. 83]. As stated by Turpin [101] in his paper *The Dangers of Living with an X*, unqualified use of the `X`-value in designs can be extremely dangerous as RTL bugs can be masked, allowing RTL simulations to incorrectly pass where netlist simulations can fail. Such `X`-bugs are often missed because simulations and formal equivalence checkers are configured to ignore them.

# Chapter 8

# Results and discussion

In the following sections, numerical metrics and results will be presented and discussed. Many of these are reported from the chosen logic synthesis tool, Vivado, while some are sourced from other experiments. The full reports from Vivado are included in Appendix E. Further discussion about the overall design and implementation can be found in Section 8.5.

## 8.1 Timing

The design timing results after implementation of the RTU system with different numbers of duplicated SVO traversal cores are shown in Table 8.1. The table is a summary of the implementation timing, and shows the worst and total slacks for three kinds of timing metrics: setup timing, hold timing, and pulse width timing.

Table 8.1: Worst and total negative slacks after implementation, taken from the Vivado design timing summary included in Appendix E.2. All timing numbers are given in ns.

| SVO cores | Setup | | Hold | | Pulse width | |
|---|---|---|---|---|---|---|
| | Worst | Total | Worst | Total | Worst | Total |
| 1 | 0.562 | 0.000 | 0.019 | 0.000 | 3.750 | 0.000 |
| 2 | 0.342 | 0.000 | 0.025 | 0.000 | 3.750 | 0.000 |
| 4 | 0.348 | 0.000 | 0.020 | 0.000 | 3.750 | 0.000 |
| 8 | 0.249 | 0.000 | 0.017 | 0.000 | 3.750 | 0.000 |
| 16 | 0.124 | 0.000 | 0.022 | 0.000 | 3.750 | 0.000 |
| 24 | −0.104 | −0.826 | 0.015 | 0.000 | 3.750 | 0.000 |

The *setup time* refers to the time before each rising clock edge that data signals reach their correct value, while the *hold time* denotes how long after the clock edge that the data remains stable [59, p. 199][58, p. 497]. The *pulse width* is a measure of

the how the clock signal itself is distributed between high and low signal level during a clock period [59, p. 474]. Slack describes in this context the margin that the design has in relation to these timing values [58, p. 499]. A negative slack is desired if the system is to function correctly—otherwise, the system is said to fail the timing requirements.

The system in this thesis is initially configured to run at 100 MHz, which translates to a clock period of 10 ns. As demonstrated by the timing results in Table 8.1, the design passes all timing requirements up to and including 16 SVO cores. Beyond this core count—as highlighted by implementing the design with 24 cores—the timing constraints are violated. This is most likely due to the signal routing becoming too complex, or simply impossible, for more than 16 cores on the given area. Typically, the signal paths of the system become too long, leading to a situation where the routing delay exceeds the timing requirements.

### 8.1.1 Critical path and maximum frequency

The critical path of the 16-core implementation is shown in Figure 8.1. The schematic is generated by Vivado and is unfortunately not legible, but it shows when zoomed in that the critical path is the in the SVO traversal core. Starting in the memory, the path subsequently goes through the carry signal of an adder, and ends up in the stack. This path is used in the `PUSH` state when fetching the next node from memory and calculating the new parameters. Redesigning the module in an attempt to improve this critical path is the key to speeding up the implementation, and is something that can be looked into as part of future work.

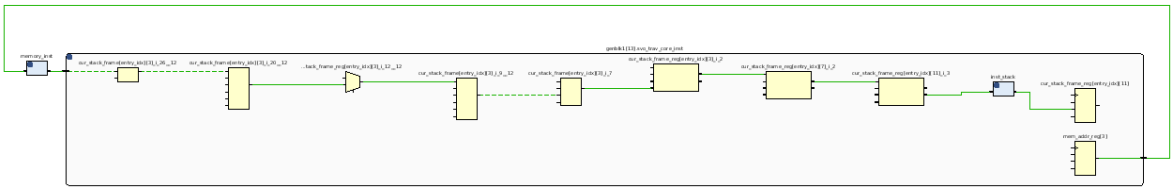


Figure 8.1: The critical path of the design as rendered by Vivado.

For a 16-core implementation, the worst negative slack (WNS) when running at 100 MHz is reported in Table 8.1 to be 0.124 ns. This means that the clock period could be reduced by this value while still meeting the timing requirements [58, p. 499]. The theoretical max frequency of the design in its current configuration may be calculated as shown in Equation (8.1).

$$f_{\max} = \frac{1}{10\,\text{ns} - 0.124\,\text{ns}} = 101.26\,\text{MHz} \tag{8.1}$$

It should be noted, however, that by changing the timing constraints, these numbers may change as well. This effect can for instance be seen by looking at the difference in WNS for the implementation of 2 and 4 SVO cores. While it may seem logical for the slack for 2 cores to be much larger than for 4 cores, it is not the case in Table 8.1. The most likely cause for this strange effect is that the synthesis tool stops its implementation effort once it reaches a configuration that meets all the constraints that have been posed. In other words, there might exist a more efficient layout and routing that could be achieved by further processing, which may be discovered by imposing stricter timing constraints.

## 8.2 Resource utilisation

Shown in Table 8.2 are the utilisation metrics for the entire system as reported by Vivado. The table sums up how adjusting the number of SVO traversal cores affects the usage of available resources. By visual examination, there appears to be a trend that slice register usage rises significantly slower than slice LUT usage when increasing the number of SVO cores in the system. This tendency is in all likelihood caused by the traversal cores being very logic-heavy. Furthermore, since the cores mainly perform calculations in-place on existing data, they do not require much register storage.

Table 8.2: Utilisation for the whole system by different SVO core counts. Taken from the Vivado resource utilisation report included in Appendix E.1.

| SVO cores | Slice LUTs | | | Slice registers | | |
|---|---|---|---|---|---|---|
| | Used | Available | Utilisation [%] | Used | Available | Utilisation [%] |
| 1 | 10 796 | 53 200 | 20.29 | 12 718 | 106 400 | 11.95 |
| 2 | 12 578 | ″ | 23.64 | 13 488 | ″ | 12.68 |
| 4 | 15 798 | ″ | 29.70 | 14 976 | ″ | 14.08 |
| 8 | 22 795 | ″ | 42.85 | 17 940 | ″ | 16.86 |
| 16 | 36 714 | ″ | 69.01 | 23 874 | ″ | 22.44 |
| 24 | 50 656 | ″ | 95.22 | 29 815 | ″ | 28.02 |

As witnessed by its 95 % slice LUT utilisation, the 24-core implementation is right on the limit of what can be synthesised on the given FPGA in terms of logic slice resources. Attempting to synthesise the system with more than 24 cores fails due to the implementation exceeding the available number of slice LUTs. However, as was discussed in the timing results section, a 24-core implementation would not be feasible in any event, since it does not satisfy the timing requirements of the system.

### 8.2.1 Per-module utilisation

Table 8.3 breaks down the resource utilisation further. The table shows how the FPGA resources—such as the number of slice LUTs and slice registers—are distributed among the different modules in the case of a 16-core system.

The SVO traversal cores represent the largest consumption of resources by a significant margin, especially in terms of LUT usage. The metrics lend credence to the claim that the traversal cores are the most complex parts of the design. The scheduler, on the other hand, uses almost no resources at all. As detailed in its implementation description, the scheduler is essentially implemented as a counter and a demultiplexer circuit which selects which of the cores' ready-valid signals to pass through. It is therefore only logical that this module should use very little resources when synthesised.

Table 8.3: Utilisation per module when implemented with 16 SVO cores. Wrapper modules has been chosen as a catch-all term for modules that are required for synthesis, but not directly part of the design. Examples of such logic are the AXI and DMA modules, as well as reset logic.

| Module | Slice LUTs | | Slice registers | |
|---|---|---|---|---|
| | Used | % of total | Used | % of total |
| Job manager | 3334 | 9.08 | 1208 | 5.06 |
| Scheduler | 17 | 0.05 | 4 | 0.02 |
| SVO cores | 23 806 | 64.84 | 10 232 | 42.86 |
| Result manager | 449 | 1.22 | 1626 | 6.81 |
| Memory | 2110 | 5.75 | 0 | 0.00 |
| Wrapper modules | 6998 | 19.06 | 10 804 | 45.25 |

The result manager uses quite a number of slice registers. This is supported by the fact that it contains an array of buffers for storing results before delivering them to the environment. The job manager, however, employs a lot of slice LUTs, as well as a fair number of slice registers. The module consists of a pipeline of calculations, so it is logical that both LUTs and registers are utilised. The registers store the intermediate results between the pipeline steps, while the LUTs implement the logic operations that form each stage.

As expected, the number of slice registers employed by the memory module is listed as 0. This metric is caused by the manner in which the memory module is currently implemented. In an effort to reduce the scope of this thesis, a fully functional memory module has not been developed. According to its requirements and design specification in Chapters 5 and 6, the memory module has been implemented in the system as a read-only data store, with the SVO model to be traced hard-coded. The memory is in essence a module of combinational logic which the synthesis tool has implemented as a set of look-up tables instead of employing conventional memory units such as block RAM or registers. This is substantiated by the module's LUT count in the utilisation table. To improve the design, memory should be prioritised for development as part of future work.

## 8.3 Performance

In Figure 8.2 is an SVO model is shown as rendered by the FPGA implementation. The SVO model was generated by the process described in Section 2.4.2, and has a spatial resolution of $32^3$ data points. The object modelled is known as the *Stanford Dragon*, and is provided by *Stanford 3D Scanning Repository* [51]. Due to the previously discussed limitations in memory, a model of the size $32^3$ was the largest synthesisable SVO data structure that would fit in memory.

In this output image, the computational cost of each ray is highlighted in red, employing the `cost` field in the output packet. A more intense red colour signifies that more clock cycles were spent in the SVO traversal core to calculate the result. It should be noted that only ray misses are part of this ray-cost visualisation, as ray hits are utilised to render the model itself.

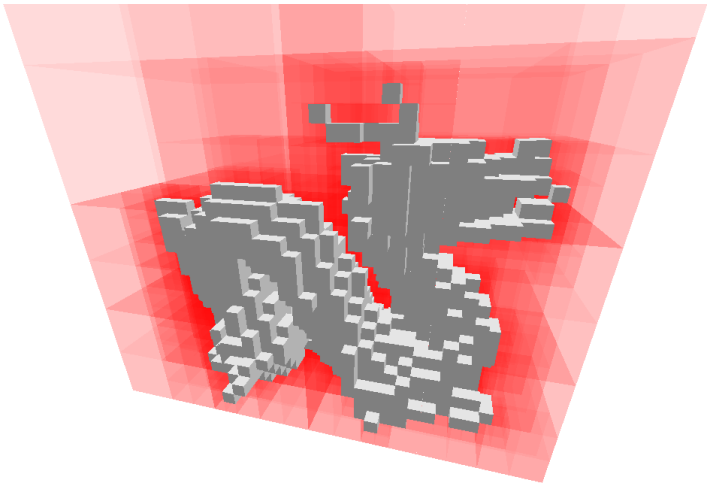


Figure 8.2: The model rendered when timing the implementation. Red colour signifies the ray cost.

The output image testifies to the claim that the implementation fulfils the first functional requirement posed in Chapter 5. This requirement stated that the system must be able to render SVO models. Specifically, the system had to be able to traverse an SVO model on the format described in Section 4.2, employing the algorithm detailed in Section 4.1. The validity of the solution in terms of these primary functional and non-functional requirements will be further discussed in Section 8.5.

### 8.3.1 Render times

In Table 8.4, the recorded render times of the implementation are listed. The table includes several different resolutions, as well as different numbers of cores, in order to illustrate what effect these configuration parameters have on performance. The content that is rendered is the Stanford Dragon shown in Figure 8.2. The data was gathered by rendering the same frame 1000 times and averaged in order to filter out any overhead associated with the setup of the FPGA and DMA controller.

The main reason why an attempt was made to filter out the overhead delays is because these latencies are not strictly part of the design and will only obscure the actual performance. During the verification, it was revealed that downloading the design to the FPGA and setting up the DMA controller and its memory regions were very slow operations. This process could take many seconds, or even minutes, to complete. Since this time delay could not be attributed to the design itself, but rather was an issue with the speed of the Pynq system, it was decided that it should be filtered out in order to paint a picture of the true performance of the system, unlimited by external factors.

Table 8.4: Actual render times of the implementation as timed by the software driver for different resolutions and SVO traversal core count. The frame rate for a given configuration is calculated as the inverse of the recorded render time. The relevant functions of the software driver are included in Appendix F.

| Resolution | SVO cores | Render time [ms] | Frame rate [Hz] |
|---|---|---|---|
| $320 \times 180$ | 1 | 9.04 | 110.57 |
| 〃 | 2 | 5.11 | 195.72 |
| 〃 | 4 | 3.14 | 318.61 |
| 〃 | 8 | 2.58 | 387.87 |
| 〃 | 16 | 2.58 | 387.86 |
| $640 \times 360$ | 1 | 35.36 | 28.28 |
| 〃 | 2 | 19.63 | 50.93 |
| 〃 | 4 | 12.29 | 81.38 |
| 〃 | 8 | 9.56 | 104.65 |
| 〃 | 16 | 9.50 | 105.30 |
| $1280 \times 720$ | 1 | 140.58 | 7.11 |
| 〃 | 2 | 78.75 | 12.70 |
| 〃 | 4 | 50.32 | 19.87 |
| 〃 | 8 | 38.44 | 26.02 |
| 〃 | 16 | 37.15 | 26.91 |

The first thing one may notice is that the performance of the implementation meets the requirement for real-time performance in a majority of the configurations. The requirement specified in Chapter 5 stated that any performance above 16 Hz would satisfy the performance requirement. As demonstrated by the render times, the requirement is satisfied for any number of cores for the resolutions $320 \times 180$ and $640 \times 360$. For $1280 \times 720$ resolution, however, the system must be implemented with at least 4 cores in order to satisfy the requirement.

### 8.3.2 Identifying bottlenecks

It can be observed in Table 8.4 that for configurations up to and including 4 SVO traversal cores, it appears that a further increase in the number of cores improves overall performance. This happens regardless of rendering resolution, and indicates that for these core counts, the bottleneck is internal to the system. In fact, it can be established that for an RTU implemented with 1, 2, or 4 traversal cores, the bottleneck

resides in the section of the system responsible for the traversal of the SVO models.

Beyond this point, however, there is not much difference in performance when adding traversal cores. The render times and frame rate of an 8-core implementation and a 16-core implementation are very similar across the board. For the given content they perform almost identically—an additional 8 cores have practically no impact on the throughput. In other words, the SVO traversal cores do not represent the bottleneck of the system in these cases. One possible explanation is that the content that is rendered—i. e. the dragon in Figure 8.2—is not very complex. Its SVO data structure has a maximum depth of 5, which means that the cores in the system do not spend a sufficient amount of clock cycles traversing it to warrant the need for a large amount of cores. Additional cores would presumably have a greater impact on throughput for more complex content, such as deeper SVO structures.

An interesting metric that may yield insight into potential bottlenecks of the system in these cases is the average amount of clock cycles the system uses to process each ray. This metric can be calculated from the performance data using the general formula listed in Equation (8.2). Shown in Table 8.5 are the results of employing this equation to calculate the average number of clock cycles per ray for the 16-core system at different resolutions.

$$100\,\mathrm{MHz} \times \frac{\text{Frame time}}{\text{Rays per frame}} \tag{8.2}$$

Table 8.5: Average number of clock cycles per ray for the three resolutions.

| Resolution | Average rendering time [ms] | Rays per frame | Average clock cycle count per ray |
|---|---|---|---|
| $320 \times 180$ | 2.58 | $6.84 \times 10^4$ | 3.77 |
| $640 \times 360$ | 9.50 | $2.30 \times 10^5$ | 4.13 |
| $1280 \times 720$ | 37.15 | $9.22 \times 10^5$ | 4.03 |

The arithmetic mean of the averages listed in the right-most column is determined to be 3.97, which means that the system on average outputs a result about every fourth clock cycle. Based on this, one may conclude that there must be a bottleneck somewhere, since it from a design standpoint should be able to process one ray per clock cycle when not limited by the SVO traversal cores. Conclusively determining where this bottleneck is located is not straightforward. However, it is theorised that since the DMA controller is set up to transfer 8-byte words, and no 256-bit buses exist natively on the Pynq, the performance might be limited by external factors. In fact, according to *AXI DMA v7.1 - LogiCORE IP Product Guide* [102], the maximum throughput of the DMA controller is around $300\,\mathrm{MB\,s^{-1}}$, which corresponds to 3 bytes per clock cycle at 100 MHz. This means that the DMA controller must use at least 3 clock cycles to transfer 8 bytes. On the basis of this, the system throughput is presumably limited—at least partially—by IO transfers.

### 8.3.3 Extrapolating the data

A data set that is lacking from Tables 8.4 and 8.5 is the performance of the system when rendering a $1920 \times 1080$ frame. Unfortunately, this data could not be gathered as a consequence of a different limitation in the Pynq system. It was briefly explained in the section describing the DMA controller in Chapter 7, that the Pynq does not allow the allocation of large enough contiguous memory buffers to support the $1920 \times 1080$ resolution. In fact, the largest standard resolution that is supported is the $1280 \times 720$ resolution.

It would be very interesting to see how the system would perform rendering a full-HD frame, but the data could therefore not be gathered directly. Nonetheless, the performance of the other data sets could be extrapolated in an attempt to predict the behaviour. Assuming that the system still uses on average 4 clock cycles to process each ray, the performance can be calculated as shown in Equation (8.3).

$$\frac{4 \cdot 2.07 \times 10^6}{100\,\text{MHz}} = 82.80\,\text{ms} \tag{8.3}$$

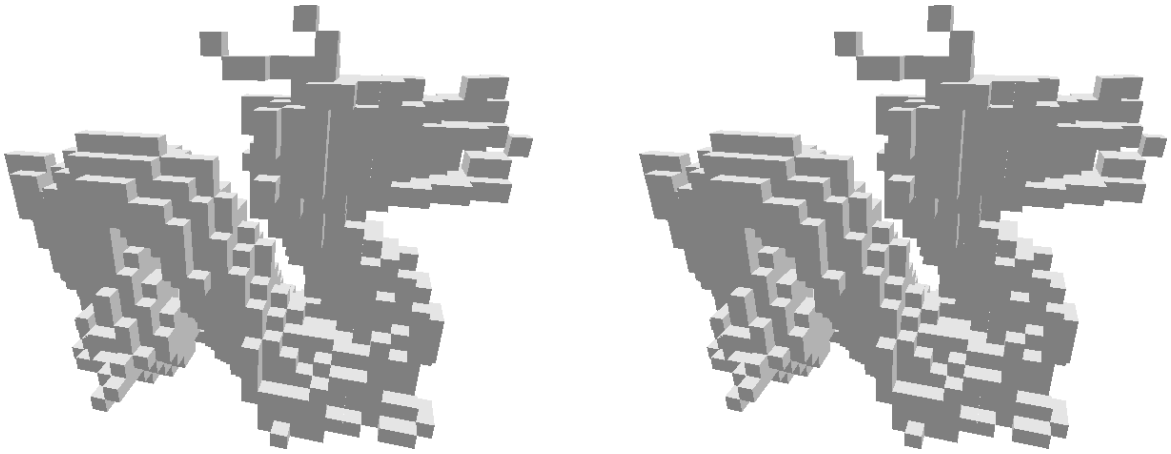
This corresponds to a frame rate of just above 12 Hz. Such a low frame rate may not be classified as real-time, so the system might struggle to meet its requirements at a resolution this high. However, it should be noted that the system currently only runs at 100 MHz. Optimising the digital design to attain higher frequencies could be part of future work. If the frequency were to double, the frame rate for $1920 \times 1080$ could very well be classified as real-time. It is not too far-fetched to consider that higher clock frequencies may be attainable, as the current generation of graphics cards from Nvidia have a base clock speed of 1350 MHz [5].

## 8.4 Visual results

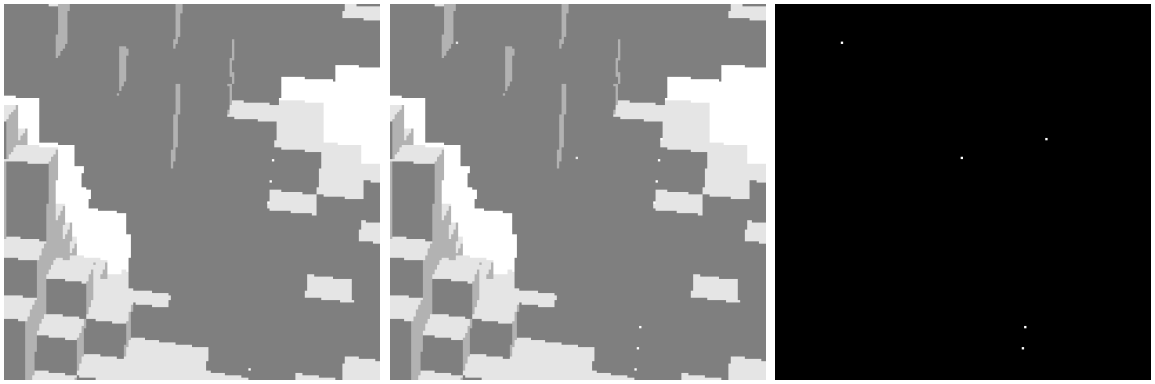
Shown in Figure 8.3 is a visual comparison of the outputs from the software model and the hardware implementation. Figures 8.3a and 8.3b show the original outputs rendered with a resolution of $1280 \times 720$. The latter of these outputs was rendered by an FPGA implementation with 16 SVO traversal cores. To highlight differences in the individual pixels, Figures 8.3c and 8.3d show zoomed-in sections of these outputs. The per-pixel difference between the two cropped outputs is shown in Figure 8.3e as a binary image—a black pixel implies that the corresponding pixels in the two cropped outputs are identical, while a white pixel signals that there was a difference.

The comparison shows that there is an almost negligible difference in the visual outputs between the software model and the FPGA implementation, and that the latter is very nearly a *bit-exact* implementation of the former. Since the zoomed-in images each have a resolution of $160 \times 90$, it follows that for a total of 14 400 pixels, only 5 differ. Furthermore, for the complete renders, only 12 out of the total 921 600 pixels are different between the software model and the hardware implementation. This last fraction corresponds to a percentage of 0.0013 %.

From visual inspection, it appears that the few pixels which differ are somehow related to the edges of the voxels. On the basis of this observation, it would stand to reason that the anomalies may be attributed to some section of the SVO traversal



(a) Model (b) FPGA



(c) Model (zoomed) (d) FPGA (zoomed) (e) Difference (zoomed)

Figure 8.3: The outputs from the software model and the implementation on FPGA are shown in (a) and (b), each with a resolution of $1280 \times 720$. Zoomed-in sections of these outputs, with sizes $160 \times 90$, are shown in (c) and (d). The differences between the zoomed outputs are highlighted in (e).

algorithm. One hypothesis is that they might stem from slight differences in the mathematical comparison operators that are employed in the design. These comparison operators are used when determining which node children that are to be recursed into, as well as to decide which face of a node that is hit if the ray pierces a solid node. In the model, these comparison operators are implemented using simple software comparison operators, while in the FPGA, they are implemented as hardware comparison circuits. While no such discrepancy has been explicitly confirmed, there may be slight differences in how these behave.

Another potential source for these minor differences is that the internal fixed-point numbers may be rounded differently in the two implementations. In the software model, the fixed-point numbers are implemented as a custom data type employing standard integers. This means that after mathematical operations, the numbers will be rounded according to the rules of integer rounding. As for the FPGA implementation, no such logic for conventional rounding has been designed. In other words, in the hardware design, the numbers are truncated and not properly rounded after any mathematical

operation. The small differences this rounding discrepancy introduces may add up over the course of a traversal routine and lead to slightly different results.

## 8.5 Further discussion

In the following sections, further discussion about the overall design and implementation will be presented. Limitations of the current design and future work will be discussed, and the finalised system will be compared to its requirement specification.

### 8.5.1 Validation with respect to primary requirements

In conclusion, the system design presented in Chapter 6 fulfils all three of the primary functional and non-functional requirements presented in Chapter 5. The implementation of this design, however, only meets two of the three requirements. In the following, each of the primary requirements will be systematically revisited and discussed.

#### First requirement: ray tracing of SVO models

This first functional requirement was satisfied by both the system design and the system implementation. The design specification presented a system which implemented the correct algorithms and data structures, and its correctness was verified by the software model. For the hardware implementation, this was proved by the output render shown in Figure 8.2, as well as the comparison in Figure 8.3.

#### Second requirement: SVO model animation

The second functional requirement was satisfied by the design specification, which discussed and formulated all the necessary modules and their functionality for the proper animation of SVO models. The implementation, however, does not meet this requirement primarily as a consequence of the required functionality not being completed in time. A system implementation meeting this requirement should be the principal concern for future work.

#### Third requirement: real-time performance

The last requirement is the non-functional requirement that the solution should exhibit real-time performance, which was defined to be a frame rate of more than 16 Hz. The initial calculations in Chapter 5 showed that such a system was indeed feasible, and the implementation satisfied the performance requirement by providing frame rates well above the required minimum.

### 8.5.2 Animation

The primary limitation of the implementation is the fact that animation is not supported. The design specification includes all necessary modules to facilitate animation, but—as has been repeatedly discussed in previous relevant sections—part of these modules were not implemented as they proved too time-consuming to develop.

There are two main features that are missing from the current implementation, which therefore need to be revisited as part of future work. The first is the matrix multiplication circuit that provides the actual animation from a mathematical standpoint. This circuit should be quite straightforward to design as an array of *multiply-accumulate* (MAC) operations, and may even be parallelised by pipelining or module duplication. In Chapter 6, many relevant examples from the literature were highlighted, including solutions by Zhuo and Prasanna [96]; Sajish et al. [97]; and Tertei, Piat, and Devy [98].

The second operation is the floating-point division circuit needed to calculate the traversal parameters after the animation transformations have been applied. The division operation is usually a bit more involved to implement in hardware, but there are multiple viable approaches. A couple of relevant designs were discussed, such as the solutions by Savas et al. [99], and Peng et al. [100]. The design by Savas et al. [99] is especially relevant since it is comprised of two main steps: reciprocal calculation and multiplication. It was devised in Chapter 6 that these two stages may potentially be incorporated in an effort to optimise the traversal parameter calculation, on account of the reciprocal being a central part of this operation.

### 8.5.3 Memory

The design implemented in this thesis does not employ a true memory module. This limitation in the system design was introduced in the requirement analysis in Chapter 5 as part of an effort to reduce the scope of the thesis, which had grown to become very large. Unfortunately, this means that any latency associated with memory operations is not properly manifested in the timing metrics presented in the results, and the validity of these in a real world setting is therefore debatable. All the results in Table 8.4 were gathered by rendering a model hard-coded in memory as a set of LUTs. A consequence of this is that all memory operations were completed within the same clock cycle as they were requested, and none of the SVO traversal cores were ever stalled due to memory latency.

The idea of a fully-functioning memory module has been entertained in peripheral discussion throughout this thesis—as evidenced by the discussion of memory caches and the potential inclusion of a `BLOCKED` state in the SVO traversal cores. As a topic for future work, it would be interesting to see what effect a fully-functioning memory module would have on performance. This memory module could for instance employ the block RAM present on the FPGA. The Pynq development board has a total of 630 kB of such memory available, so by storing the model to be rendered in this RAM, a much larger and more complex scene can be rendered.

The introduction of memory caches should help alleviate the latency introduced by such a memory module. Exploring how a memory cache would mitigate latency issues is certainly a potential subject for future research. It has been suggested in relevant earlier discussion that multiple levels of cache could be employed. For instance, placing a L1 cache in each SVO traversal core, and an L2 cache in the memory module may further improve performance.

### 8.5.4 Tracing complex scenes

The performance when tracing complex scenes consisting of multiple models may be quite limited in the current design specification. The reason for this is that tracing multiple models currently requires submitting multiple ray tracing jobs to the RTU. The system is configured to interpret one ray tracing job to mean tracing a single ray through a single model, and is pipelined, parallelised, and optimised for this exact purpose. Complex scenes may be traced by traversing every model in the scene by every ray, and subsequently using the depth information—the `t_hit` field—as part of a depth buffering mechanism to sort the results. There would presumably be much to gain in terms of performance by extending the design specification so that multiple objects may be traced by a single ray, and an effort to this end can be part of future work.

One approach to enable such functionality would be to include a module responsible for traversing the scene. This module will in the following referred to as the *scene manager*, and the RTU structure after introducing this scene manager module is shown in Figure 8.4. A description of the scene and which models it contains would be supplied by some data structure in memory, or be submitted to the RTU as an initial setup before tracing a frame. Using this scene description, the scene manager would for each model in the scene submit an internal ray tracing job to the job manager. In addition, it would keep track of the results from the result manager and select the correct result to return for a given ray.

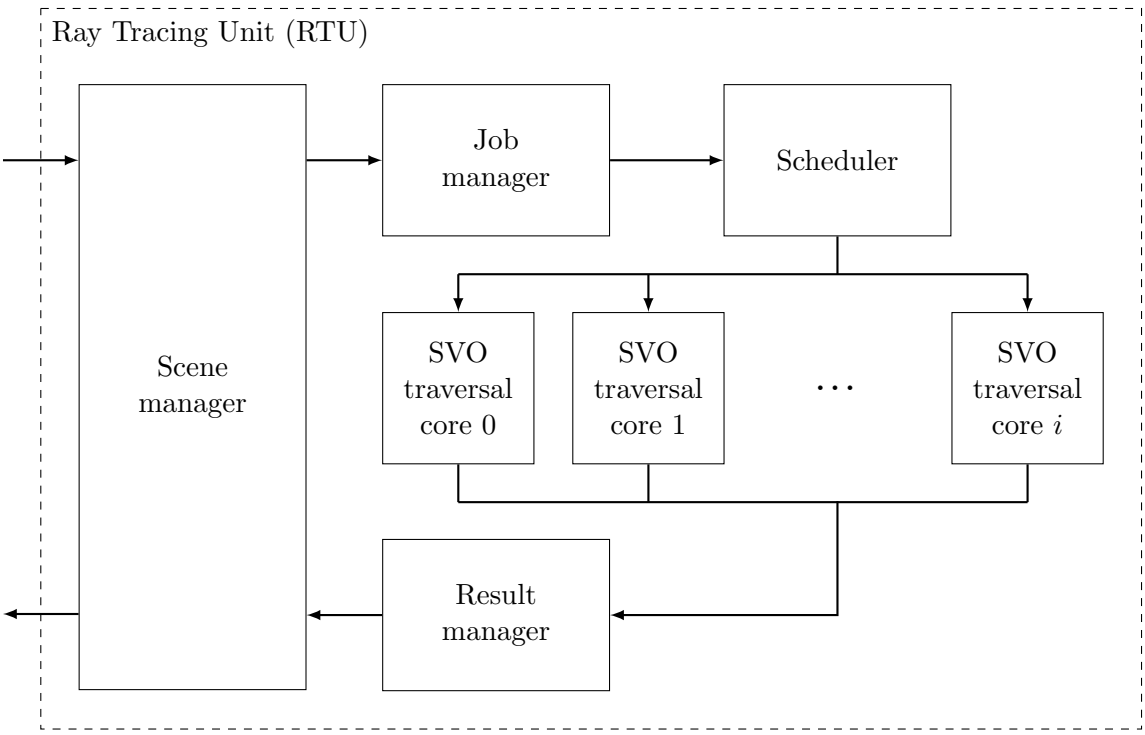



Figure 8.4: How a scene manager module may be realised in the system in order to allow efficient traversal of complex scenes.

### 8.5.5 Algorithmic optimisations

Introduced in the project thesis were several optimisations that may further improve general performance of animated SVO models. These optimisations were briefly described in Section 4.3, and should be evaluated as part of future work. Especially relevant are the hit buffer object (HBO) and bounding-sphere sorting optimisations which were employed with huge success in the project thesis.

These optimisations work on the scene level, and therefore might be coupled with the introduction of a scene manager module as described in the previous.

# Chapter 9

# Conclusions

In this master's thesis a solution for the real-time ray tracing of animated sparse voxel octrees is presented. The solution—titled *ray tracing unit* (RTU)—is designed to run in hardware on the FPGA present on the Pynq-Z1 development board. For its design, the system employs the works of Revelles, Ureña, and Lastra [53], and Laine and Karras [35] as a foundation, where the former paper provides the octree traversal algorithm, and the latter contains the formulation upon which the SVO format is based. The method for animation of SVO models was introduced by Espe [1], in the specialisation project thesis that was antecedent to this master's thesis.

An analysis of the system requirements was conducted in Chapter 5, and later employed in the formulation of a requirement specification. Three primary requirements were established—two functional requirements and one non-functional requirement. The first functional requirement stated that the solution must be able to render SVO models as specified by the chosen foundational algorithms detailed in Chapter 4. The second functional requirement posed that the solution must support animation of the SVO models in the manner detailed by the project thesis. The third non-functional requirement asserted that the solution must be capable of rendering these models in real-time. The requirements were revisited in later chapters as part of the validation of the solution. Following these primary requirements, further derivative requirements and considerations were discussed, such as system scalability, pipelining, communication interface, and overall feasibility.

In Chapter 6, the overall design of the system was derived. The internal modularisation of the solution was determined, and the function of each module discussed and examined in contrast to the requirements posed in the previous chapter. The main modules of the system introduced as part of the design were: the job manager, the scheduler, the SVO traversal core, the result manager, and the memory module. All but one of these modules were specified in detail; in an effort to limit the scope of the implementation workload, the memory module was only partially designed, leaving the design and implementation of a proper memory module for future work.

An implementation of the system design was developed and presented in Chapter 7. The majority of the system was implemented in accordance with the design specification, but due to time constraints, some modules were only partially completed. The scheduler, the SVO traversal core, and the result manager closely reflect their design specification, while the job manager lacks a certain subset of the required features that

facilitate animation. As such, the implementation is a fully-functioning SVO ray tracer, but support for animation of the models could not be completed in time. Many relevant approaches to these unfinished features were presented in an attempt to streamline this part of future work. The memory module was implemented as required by the design specification, though its functionality is limited due to the reduction of scope detailed in the system design.

The results from the implementation show that the solution is capable of delivering a frame rate of 26.91 Hz when rendering static SVO models at a resolution of $1280 \times 720$. This performance was recorded with 16 SVO traversal cores. In order to achieve real-time performance at this resolution, the system must be implemented with at least 4 SVO traversal cores, which gives the borderline frame rate of 19.87 Hz. On the basis of this, it is argued that the non-functional requirement of real-time performance is satisfied. Even though the implementation itself does not satisfy the second functional requirement of supporting animation, this requirement is satisfied by the system design. It is theorised that real-time performance rendering animated models is achievable if implemented efficiently—especially if certain optimisations are explored, such as the general optimisation techniques presented in the project thesis.

It is demonstrated that if the system is implemented with sufficient resources, the bottleneck is not attributed to limitations in its internal throughput, and that the RTU performance is consequently limited by external factors. Specifically, when not constrained by the throughput of the SVO traversal cores, the system performance was limited by a bottleneck identified as the transfer of data from the DMA controller to the RTU system. A side-by-side comparison of the outputs from a software implementation of the chosen algorithms and the hardware implementation was presented. The visual difference between the outputs is negligible, which substantiates the claim that the first functional requirement is fulfilled.

## 9.1 Limitations

The limitations of the design and implementation were detailed in Chapter 8, but the main points will be briefly reiterated here. A major limitation of the implementation is the fact that animation is not supported. The scope of this thesis was apparently too large, and lead to certain required features not being completely implemented. A limitation in the design of the system itself is that scenes consisting of several models can only be properly rendered by tracing multiple rays. Moreover, the memory module is only partially designed. For this reason the system is currently constrained by the available memory, with only small SVO models currently supported.

A limitation attributed to the chosen target technology is that the performance of the implementation is currently restricted by the speed of the CPU and DMA controller on the Pynq. This was indeed identified as the bottleneck of the system. Additionally, the setup and configuration of the Pynq—including the downloading of the implementation to the FPGA and the allocation of DMA memory—is very slow. Lastly, due to memory constraints associated with the selected DMA approach, the system is limited to a maximum resolution of $1280 \times 720$.

## 9.2 Future work

When nearing the end of implementation work, it was evaluated that a hardware implementation of the entire design specification had been too ambitious, and resulted in a scope too large for a master's thesis. To put it somewhat extremely, the implementation of a matrix multiplication circuit or floating-point division circuit could conceivably constitute entire theses on their own.

A fair share of material has been identified as suitable for future work and continued research. The chief topic, perhaps, is completing the implementation of the job manager module according to its design specification. The job manager was only partially implemented due to time constraints, which lead to animation not being supported by the implementation. Finishing the implementation of the required operations, such as matrix multiplication and floating-point division in hardware, should be the primary focus of future work. Many solutions providing these specific functions were found in the literature, so their implementation should be unproblematic given sufficient time.

Secondly, a proper memory module should be designed and implemented. The memory module could make use of the block RAM present on the FPGA to enable the storage of larger SVO models, as well as the transformation matrices associated with animation. This memory module would be coupled with proper support for stalls in the SVO traversal cores. In addition, to mitigate the adverse effects linked to memory latency, memory caches could be introduced wherever suitable. For instance, a large L2 cache could be implemented as part of the memory module, while placing L1 caches in the job manager and SVO traversal cores.

Complex scenes consisting of several SVO models currently require that multiple ray jobs are submitted to the system in order to trace properly. Extending the system design to allow a single ray to be used for the traversal of multiple models would presumably speed up the implementation significantly, since post-processing of the results and the maintaining of a depth buffer in software would no longer be required. This could be achieved, for instance, by designing a scene manager module.

As a consequence of the parallel nature of the system, ray tracing jobs submitted to the RTU may be received out of order. Hence, the results must be post-processed and reordered in software so that each result ends up in the appropriate memory location. The result manager could in all likelihood be extended to perform this sorting in hardware, and subsequently return the result in raster order—the order in which they were submitted. This extension would also alleviate the need for the `job_id` field, which is required solely for the purpose of identifying results.

Optimisations of the current solution may yield significant improvements in performance, and as such is an area that could be explored as part of future work. The critical path identified in Chapter 8 is the primary limitation constraining the maximum frequency of the implementation. If improved, the design could accommodate higher frequencies and in turn better performance. Another approach would be to introduce multiple clock domains in order to speed up certain parts of the design. The optimisations introduced in the project thesis—such as the HBO and the set of bounding-sphere optimisations—should be evaluated, as they may be suited for hardware implementation and grant further improvements in performance once animation is supported.

# Bibliography


[1] A. E. Espe. *A Method for Rigid-Body Animation of Sparse Voxel Octrees*. Specialisation project thesis. 2018.

[2] J. Carmack. *John Carmack Keynote Q&A*. Aug. 2011. URL: https://www.youtube.com/watch?v=00Q9-ftiPVQ (visited on 02/06/2019).

[3] S. Sakar. *Nvidia bringing new ray tracing tech to GTX graphics cards*. Mar. 18, 2019. URL: https://www.polygon.com/2019/3/18/18271633/nvidia-rtx-gtx-ray-tracing-gdc-2019 (visited on 04/28/2019).

[4] C. Schodt. *Ray tracing explained: The future of hyper-realistic graphics*. Apr. 16, 2018. URL: https://www.engadget.com/2018/04/16/the-future-of-ray-tracing-explainer/ (visited on 05/21/2019).

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

[6] J. Evangelho. *Nvidia RTX 20 Series: Why You Should Jump Off The Hype Train*. Aug. 21, 2018. URL: https://www.forbes.com/sites/jasonevangelho/2018/08/21/nvidia-rtx-20-graphics-cards-why-you-should-jump-off-the-hype-train/#8fa428b3f8e3 (visited on 05/21/2019).

[7] B. Stelmaszek. *Ray Tracing Is Just A Giant Gimmick*. Sept. 22, 2018. URL: https://www.youtube.com/watch?v=68QqaksxuEg (visited on 05/21/2019).

[8] N. Subtil. *Introduction to Real-Time Ray Tracing with Vulkan*. Nvidia Developer Blog. Oct. 10, 2018. URL: https://devblogs.nvidia.com/vulkan-raytracing/ (visited on 05/20/2019).

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

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

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

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

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

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

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

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

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

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

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

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

[21] 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.

[22] 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.

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

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

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

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

[27] 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.

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

[29] 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.

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

[31] J. Packer. *Exploiting concurrency: a ray tracing example*. Tech. rep. 72-TCH-007-01. INMOS Technical Note 7, 1987. URL: http://www.transputer.net/tn/07/tn07.html.

[32] D. 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.

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

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

[35] S. Laine and T. Karras. "Efficient Sparse Voxel Octrees". In: *IEEE Transactions on Visualization and Computer Graphics* 17.8 (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).

[36] 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.

[37] 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.

[38] *IEEE Standard for Floating-Point Arithmetic*. IEEE, 2008. ISBN: 978-0-7381-5753-5. DOI: 10.1109/ieeestd.2008.4610935.

[39] M. L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. Society for Industrial and Applied Mathematics, 2001. ISBN: 978-0-89871-482-1. DOI: 10.1137/1.9780898718072.

[40] A. S. Tanenbaum and T. Austin. *Structured Computer Organization (6th Edition)*. Pearson, 2012. ISBN: 978-0-13-291652-3.

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

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

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

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

[45] 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).

[46] H. Brönnimann and M. 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.

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

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

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

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

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

[52] 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.

[53] J. Revelles, C. Ureña, and M. Lastra. "An Efficient Parametric Algorithm for Octree Traversal". In: (2000), pp. 212–219.

[54] 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.

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

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

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

[58] W. J. Dally, R. Curtis Harting, and T. M. Aamodt. *Digital Design Using VHDL: A Systems Approach*. Cambridge University Press, 2015. ISBN: 978-1-107-09886-2.

[59] M. M. R. Mano and M. D. Ciletti. *Digital Design: With an Introduction to the Verilog HDL*. Pearson, 2012. ISBN: 978-0-13-277420-8.

[60] K. S. Mohamed. *IP Cores Design from Specifications to Production: Modeling, Verification, Optimization, and Protection (Analog Circuits and Signal Processing)*. Springer, 2015. ISBN: 978-3-319-22034-5.

[61] I. Kuon and J. Rose. "Measuring the Gap Between FPGAs and ASICs". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 26.2 (2007), pp. 203–215. DOI: 10.1109/tcad.2006.884574.

[62] *PYNQ-Z1: Python Productivity for Zynq-7000 ARM/FPGA SoC*. Digilent Inc. URL: https://store.digilentinc.com/pynq-z1-python-productivity-for-zynq-7000-arm-fpga-soc/ (visited on 04/19/2019).

[63] *Python 3.6.0 documentation*. Python Software Foundation. URL: https://docs.python.org/release/3.6.0/ (visited on 04/22/2019).

[64] *Python productivity for Zynq (Pynq)*. Xilinx, Inc. URL: https://pynq.readthedocs.io/en/latest/getting_started.html (visited on 04/22/2019).

[65] *Jupyter Notebook Documentation*. Jupyter Team. URL: https://jupyter-notebook.readthedocs.io/en/latest/index.html (visited on 04/22/2019).

[66] *IEEE Standard for SystemVerilog*. IEEE, 2018. ISBN: 978-1-5044-4509-2. DOI: 10.1109/ieeestd.2018.8299595.

[67] E. Cerny et al. *The Power of Assertions in SystemVerilog*. Springer US, 2010. DOI: 10.1007/978-1-4419-6600-1.

[68] L. Lavagno et al. *Electronic Design Automation for IC Implementation, Circuit Design, and Process Technology*. CRC Press, 2017. ISBN: 978-1-4822-5461-7.

[69] *Vivado Design Suite*. Version 2018.3. Xilinx, Inc. Dec. 10, 2018. URL: https://www.xilinx.com/products/design-tools/vivado.html (visited on 04/22/2019).

[70] C. Fletcher. *EECS150: Interfaces: "FIFO" (a.k.a. Ready/Valid)*. UC Berkeley College of Engineering. Feb. 24, 2009. URL: https://inst.eecs.berkeley.edu/~cs150/Documents/Interfaces.pdf (visited on 05/15/2019).

[71] *AMBA Specifications*. Arm Ltd. URL: www.arm.com/products/silicon-ip-system/embedded-system-design/amba-specifications (visited on 04/09/2019).

[72] *AMBA 4 AXI4-Stream Protocol Specification*. Arm Ltd. 2010. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf (visited on 05/26/2019).

[73] 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.

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

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

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

[77] 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.

[78] 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.

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

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

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

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

[83] 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).

[84] 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).

[85] 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.

[86] C. Cassagnabère, F. Rousselle, and C. Renaud. "Path tracing using the AR350 processor". In: *Proceedings of the 2nd international conference on Computer graphics and interactive techniques in Austalasia and Southe East Asia - GRAPHITE '04*. ACM Press, 2004. DOI: 10.1145/988834.988838.

[87] J. Fender and J. Rose. "A high-speed ray tracing engine built on a field-programmable system". In: *Proceedings. 2003 IEEE International Conference on Field-Programmable Technology (FPT) (IEEE Cat. No.03EX798)*. IEEE. DOI: 10.1109/fpt.2003.1275747.

[88] G. Knittel and W. Straßer. *VIZARD - Visualization Accelerator for Realtime Display*. 1997. DOI: 10.2312/eggh/eggh97/139-146.

[89] J. Schmittler et al. "Realtime ray tracing of dynamic scenes on an FPGA chip". In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware - HWWS '04*. ACM Press, 2004. DOI: 10.1145/1058129.1058143.

[90] S. Woop, J. Schmittler, and P. Slusallek. "RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing". In: *ACM Transactions on Graphics* 24.3 (2005), p. 434. DOI: 10.1145/1073204.1073211.

[91] S. Collinson. "Efficient Ray Tracing on FPGAs". PhD thesis. Dec. 2014.

[92] 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.

[93] 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 (2007), pp. 143–152. DOI: 10.1016/j.nimb.2007.04.262.

[94] G. M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)*. ACM Press, 1967. DOI: 10.1145/1465482.1465560.

[95] J. W. Nilsson and S. Riedel. *Electric Circuits (9th Edition)*. Pearson, 2010. ISBN: 978-0-13-611499-4.

[96] L. Zhuo and V. K. Prasanna. "Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on Reconfigurable Computing Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 18.4 (Apr. 2007), pp. 433–448. DOI: 10.1109/tpds.2007.1001.

[97] C. Sajish et al. "Floating Point Matrix Multiplication on a Reconfigurable Computing System". In: *Current Trends in High Performance Computing and Its Applications*. Springer-Verlag, pp. 113–122. DOI: 10.1007/3-540-27912-1_11.

[98] D. T. Tertei, J. Piat, and M. Devy. "FPGA design and implementation of a matrix multiplier based accelerator for 3D EKF SLAM". In: *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*. IEEE, Dec. 2014. DOI: 10.1109/reconfig.2014.7032523.

[99] S. Savas et al. "Efficient Single-Precision Floating-Point Division Using Harmonized Parabolic Synthesis". In: *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, July 2017. DOI: 10.1109/isvlsi.2017.28.

[100] Y. Peng et al. "Single/Double Precision Floating-Point Division and Square Root Unit Based on SRT-8 Algorithm". In: *Communications in Computer and Information Science*. Springer Singapore, 2016, pp. 3–14. DOI: 10.1007/978-981-10-3159-5_1.

[101] M. Turpin. *The Dangers of Living with an X (bugs hidden in your Verilog)*. Arm Ltd. Aug. 14, 2003. URL: http://infocenter.arm.com/help/topic/com.arm.doc.arp0009a/Verilog_X_Bugs.pdf (visited on 05/11/2019).

[102] *AXI DMA v7.1 - LogiCORE IP Product Guide*. Xilinx, Inc. URL: https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf (visited on 05/20/2019).

# Appendix A

# Attached ZIP file contents

A ZIP archive is attached to this master's thesis. Its contents are as listed in the following.

### Contents

```
masters_thesis.pdf      # This master's thesis
project_thesis.pdf      # The project thesis

rtu/                    # The SystemVerilog source code for the RTU
    axi/                    # AXI wrapper modules
    modules/                # Main modules
        rtu.sv                  # The RTU top module
        job_man.sv              # The job manager module
        float_to_fixed.sv       # The floating-point to fixed-point converter
        scheduler.sv            # The scheduler module
        svo_trav_core.sv        # The SVO traversal module
        result_man.sv           # The result manager
        memory.sv               # The memory module
        data_types.sv           # Data types
model/                  # The Python source code for the software model
svo_model_generation/   # SVO model generation routines (from the project thesis)
figures/                # Figures used in this thesis
```

# Appendix B

# Acronyms and abbreviations

**ABV** Assertion-Based Verification. 29

**AMBA** Advanced Microcontroller Bus Architecture. 31, 89

**API** Application Programming Interface. 1

**ASIC** Application-Specific Integrated Circuit. 26, 79

**AXI** Advanced eXtensible Interface. xiii, 31, 89, 98, 143–147

**BSP** Binary Space Partitioning. 20

**CAD** Computer-Aided Design. 3

**CLB** Configurable Logic Block. 26, 80

**CPU** Central Processing Unit. 27, 35, 89, 90, 110

**CSG** Constructive Solid Geometry. 5

**CUDA** Compute Unified Device Architecture. 35

**DMA** Direct Memory Access. xii, 88–90, 98, 99, 101, 102, 110

**DSP** Digital Signal Processing. 80

**DUT** Device Under Test. 29, 92

**EDA** Electronic Design Automation. 30

**FPGA** Field-Programmable Gate Array. v, vii, xi, 25–27, 29, 30, 35, 67, 79–81, 88–90, 92, 93, 97–99, 102, 103, 105, 109–111

**GPGPU** General-Purpose computing on Graphics Processing Unit. 35, 37

**GPU** Graphics Processing Unit. 4

**GUI** Graphical User Interface. 3

**HBO** Hit Buffer Object. 51, 52, 67, 107, 111

**HCI** Human-Computer Interaction. 3

**HDL** Hardware Description Language. xi, 25, 28–30, 92

**HDMI** High-Definition Multimedia Interface. 27

**HDVL** Hardware Description and Verification Language. 28

**IO** Input/Output. v, vii, 101

**IP** Intellectual Property. 88

**LoD** Level of Detail. 21–24

**LUT** Look-up Table. 26, 80, 97, 98, 105

**MAC** Multiply-accumulate. 67, 69, 105

**MIMD** Multiple Instruction, Multiple Data. 35

**NaN** Not-a-number. 12

**NTNU** Norges Teknisk-Naturvitenskapelige Universitet. 2

**PC** Personal Computer. 4

**RAM** Random-Access Memory. 78, 80, 98, 105, 111

**RLE** Run-Length Encoding. 23, 24

**RTL** Register-Transfer Level. 29, 30, 81, 91–93, 125

**RTU** Ray Tracing Unit. xiii, 58, 60, 61, 63, 64, 66, 76, 79, 83, 85, 87–90, 92, 95, 100, 106, 109–111, 125, 149

**SFX** Special Effects. 3

**SIMD** Single Instruction, Multiple Data. 35, 68

**SoC** System-on-Chip. 30, 80, 88, 89

**SVO** Sparse Voxel Octree. iii, v, vii, xii, xiii, 21–24, 34, 35, 46–52, 54–56, 58, 59, 61–69, 71, 73–78, 80, 81, 83–87, 91, 95–102, 104–107, 109–111, 131, 133, 135, 137, 149

**WNS** Worst Negative Slack. 96

# Appendix C

# Hardware modules

The RTL source code for the RTU implementation is listed in this appendix.

## C.1 RTU top module

```
import types::*;

module rtu #(
    parameter CORE_CNT        = 16,
    parameter FX_PT_FRAC_BITS = 16
) (
    input  logic    clk,
    input  logic    reset,

    output logic    in_ready,
    input  logic    in_valid,
    input  ray_t    in_ray,

    input  logic    out_ready,
    output logic    out_valid,
    output result_t out_result
);

logic sched_ready;
logic sched_valid;
job_t pending_job;

job_manager #(FX_PT_FRAC_BITS) job_manager_inst (
    .clk(clk),
    .reset(reset),

    .in_ready(in_ready),
    .in_valid(in_valid),
    .in_ray(in_ray),

    .out_ready(sched_ready),
    .out_valid(sched_valid),
    .out_job(pending_job)
);
```

```
logic [CORE_CNT-1:0] core_in_ready;
logic [CORE_CNT-1:0] core_in_valid;

scheduler #(CORE_CNT) scheduler_inst (
    .clk(clk),
    .reset(reset),

    .in_valid(sched_valid),
    .in_ready(sched_ready),

    .core_ready(core_in_ready),
    .core_valid(core_in_valid)
);

logic [31:0] mem_addr [0:CORE_CNT-1];
logic [31:0] mem_data [0:CORE_CNT-1];
memory #(CORE_CNT) memory_inst(
    .addr(mem_addr),
    .data(mem_data)
);


logic    [CORE_CNT-1:0] core_out_ready;
logic    [CORE_CNT-1:0] core_out_valid;
result_t [CORE_CNT-1:0] core_out_result;

genvar i;
generate
    for (i = 0; i < CORE_CNT; i += 1) begin
        svo_trav_core svo_trav_core_inst(
            .clk(clk),
            .reset(reset),
            .mem_addr(mem_addr[i]),
            .mem_data(mem_data[i]),

            .in_ready(core_in_ready[i]),
            .in_valid(core_in_valid[i]),
            .in_job(pending_job),

            .out_ready(core_out_ready[i]),
            .out_valid(core_out_valid[i]),
            .out_result(core_out_result[i])
        );

    end
endgenerate

result_manager #(CORE_CNT) result_manager_inst (
    .clk(clk),
    .reset(reset),

    .core_ready(core_out_ready),
    .core_valid(core_out_valid),
    .core_result(core_out_result),
```

```
    .out_ready(out_ready),
    .out_valid(out_valid),
    .out_result(out_result)
);

endmodule
```

## C.2 Job manager

```
import types::*;

module job_manager #(
    parameter FX_PT_FRAC_BITS = 16
) (
    input  logic clk,
    input  logic reset,

    output logic in_ready,
    input  logic in_valid,
    input  ray_t in_ray,

    input  logic out_ready,
    output logic out_valid,
    output job_t out_job
);

logic pipeline_enable;
assign pipeline_enable = out_ready || ~out_valid;
assign in_ready = pipeline_enable;

vec3_t t0_float;
vec3_t t1_float;

genvar i;
generate
    for (i = 0; i < 3; i += 1) begin
        // Placeholder for future work
        ray_to_params ray_to_params(
            .clk(clk),
            .enable(pipeline_enable),
            .in_ray_orig(in_ray.orig[i*32+:32]),
            .in_ray_dir(in_ray.dir[i*32+:32]),
            .out_t0(t0_float[i*32+:32]),
            .out_t1(t1_float[i*32+:32])
        );

        float_to_fixed #(FX_PT_FRAC_BITS) float_to_fixed_t0(
            .clk(clk),
            .enable(pipeline_enable),
            .data_in(t0_float[i*32+:32]),
            .data_out(out_job.t0[i*32+:32])
        );
        float_to_fixed #(FX_PT_FRAC_BITS) float_to_fixed_t1(
            .clk(clk),
            .enable(pipeline_enable),
```

```
            .data_in(t1_float[i*32+:32]),
            .data_out(out_job.t1[i*32+:32])
        );
    end
endgenerate

`define P_STAGES 6

logic [`P_STAGES - 1:0]  valid_delay;
job_t pipeline [`P_STAGES];

assign out_valid      = valid_delay[`P_STAGES - 1];
assign out_job.a      = pipeline[`P_STAGES - 1].a;
assign out_job.addr   = pipeline[`P_STAGES - 1].addr;
assign out_job.job_id = pipeline[`P_STAGES - 1].job_id;

always @(posedge clk) begin
    if (reset == 'b1) begin
        valid_delay <= 'b0;
    end else if (pipeline_enable) begin
        valid_delay        <= {valid_delay[`P_STAGES - 2:0], in_valid & in_ready};
        pipeline[0].a      <= in_ray.job_id[30:28];
        pipeline[0].addr   <= in_ray.addr;
        pipeline[0].job_id <= in_ray.job_id;

        for (int i = 0; i < `P_STAGES - 1; i += 1) begin
            pipeline[i + 1] <= pipeline[i];
        end
    end
end
endmodule
```

### C.2.1 Floating-point to fixed-point conversion

```
import types::*;

module float_to_fixed #(
    parameter FX_PT_FRAC_BITS = 16
) (
    input  logic   clk,
    input  logic   enable,
    input  float32 data_in,
    output fixed32 data_out
);

typedef struct packed {
    logic        sign;
    logic        exp_sign;
    logic  [7:0] exponent;
    logic [63:0] fraction;
} pipeline_stage_t;

pipeline_stage_t s1;
pipeline_stage_t s2;
pipeline_stage_t s3;
pipeline_stage_t s4;
```

```
logic [31:0] output_select;
assign output_select = s4.fraction[(63-FX_PT_FRAC_BITS)-:32];

always @(posedge clk) begin
    if (enable) begin
        s1.exp_sign         <= 'b0;
        s1.sign             <= data_in[31];
        s1.fraction         <= 'b0;
        s1.fraction[32-:24] <= {1'b1, data_in[22:0]};

        if ($signed(data_in[30:23] - 127) >= 0) begin
            s1.exponent <= data_in[30:23] - 127;
        end else begin
            s1.exponent <= ~(data_in[30:23] - 127) + 1;
            s1.exp_sign <= 'b1;
        end

        if ($signed(data_in[30:23] - 127) >= (31 - FX_PT_FRAC_BITS)) begin
            s1.exponent <= 'b0;
            s1.sign     <= 'b0;
            if (data_in[31]) begin
                s1.fraction     <= 'b0;
                s1.fraction[63] <= 'b1;
            end else begin
                s1.fraction     <= 'b1;
                s1.fraction[63] <= 'b0;
            end
        end else if (
            $signed(data_in[30:23] - 127) <= $signed(1 - FX_PT_FRAC_BITS)
        ) begin
            s1.sign     <= 'b0;
            s1.exponent <= 'b0;
            s1.fraction <= 'b0;
        end
    end
end

always @(posedge clk) begin
    if (enable) begin
        s2 <= s1;

        if (s1.exponent[3]) begin
            if (s1.exp_sign) begin
                s2.fraction <= s1.fraction >> 8;
            end else begin
                s2.fraction <= s1.fraction << 8;
            end
        end
    end
end

always @(posedge clk) begin
    if (enable) begin
        s3 <= s2;
```

```
        if (s2.exponent[2]) begin
            if (s2.exp_sign) begin
                s3.fraction <= s2.fraction >> 4;
            end else begin
                s3.fraction <= s2.fraction << 4;
            end
        end
    end
end

always @(posedge clk) begin
    if (enable) begin
        s4 <= s3;

        if (s3.exponent[1]) begin
            if (s3.exp_sign) begin
                s4.fraction <= s3.fraction >> 2;
            end else begin
                s4.fraction <= s3.fraction << 2;
            end
        end
    end
end

always @(posedge clk) begin
    if (enable) begin
        data_out <= s4.sign ? (~output_select + 1) : output_select;

        if (s4.exponent[0]) begin
            if (s4.exp_sign) begin
                data_out <= (
                    s4.sign ? (~(output_select >> 1) + 1) : (output_select >> 1)
                );
            end else begin
                data_out <= (
                    s4.sign ? (~(output_select << 1) + 1) : (output_select << 1)
                );
            end
        end
    end
end
endmodule
```

## C.3 Scheduler

```
module scheduler #(
    parameter CORE_CNT = 16
) (
    input  logic clk,
    input  logic reset,

    output logic in_ready,
    input  logic in_valid,

    input  logic [CORE_CNT-1:0] core_ready,
```

```
    output logic [CORE_CNT-1:0] core_valid
);

logic [$clog2(CORE_CNT)-1:0] core_select;

assign in_ready = core_ready[core_select];
assign core_valid = in_valid << core_select;

always @(posedge clk) begin
    if (reset == 'b1) begin
        core_select <= 'b0;
    end else begin
        core_select <= 'b0;

        if (core_select < CORE_CNT - 1) begin
            core_select <= core_select + 1;
        end
    end
end
endmodule
```

## C.4 SVO traversal core

```
import types::*;

typedef enum logic[2:0] {
    IDLE,
    INIT,
    EVAL,
    NEXT,
    PUSH,
    POP,
    OUT
} rt_core_state_t;

typedef struct packed {
    logic  [31:0] entry_idx;
    vec3_t        t0;
    vec3_t        tm;
    vec3_t        t1;
    logic   [2:0] cur_child_idx;
} stack_frame_t;

module svo_trav_core (
    input  logic        clk,
    input  logic        reset,
    output logic [31:0] mem_addr,
    input  logic [31:0] mem_data,

    output logic        in_ready,
    input  logic        in_valid,
    input  job_t        in_job,

    input  logic        out_ready,
    output logic        out_valid,
```

```
    output result_t     out_result
);

rt_core_state_t core_state;
assign in_ready = (core_state == IDLE);

stack_frame_t cur_stack_frame;
stack_frame_t stack_top;
logic         stack_push;
logic         stack_pop;
logic         stack_empty;
logic         push_indirect;
logic         stack_clear;

stack inst_stack(
    clk,
    reset,
    stack_clear,
    stack_push,
    stack_pop,
    cur_stack_frame,
    stack_top,
    stack_empty
);

logic [2:0] transform;
logic [2:0] cur_child_idx_t;
always_comb begin
    cur_child_idx_t = cur_stack_frame.cur_child_idx ^ transform;
end

logic [95:0] top_t0;
logic [95:0] top_t1;
logic        ray_valid;
get_ray_valid inst_get_ray_valid(clk, top_t0, top_t1, ray_valid);

vec3_t t0_child;
vec3_t t1_child;
get_child_t_values inst_t0_get_child_t_values(
    cur_stack_frame.cur_child_idx,
    cur_stack_frame.t0,
    cur_stack_frame.tm,
    t0_child
);

get_child_t_values inst_t1_get_child_t_values(
    cur_stack_frame.cur_child_idx,
    cur_stack_frame.tm,
    cur_stack_frame.t1,
    t1_child
);

logic [2:0] initial_child_idx;
get_initial_child_idx inst_get_initial_child_idx(
    cur_stack_frame.t0,
    cur_stack_frame.tm,
```

```
    initial_child_idx
);

logic [3:0] next_child_idx;
get_next_child_idx inst_get_next_child_idx(
    clk,
    cur_stack_frame.cur_child_idx,
    t1_child,
    next_child_idx
);


logic [2:0] ones_count [8];
get_ones_count inst_get_ones_count(mem_data[14:8] & ~mem_data[6:0], ones_count);

logic[31:0] next_entry_idx;

assign out_valid = (core_state == OUT);

always @(posedge clk) begin
    if (reset == 1'b1) begin
        core_state      <= IDLE;
        cur_stack_frame <= 'b0;
        stack_push      <= 'b0;
        stack_pop       <= 'b0;
        stack_clear     <= 'b0;
        push_indirect   <= 'b0;
        next_entry_idx  <= 'b0;
        mem_addr        <= 'bX;
        transform       <= 'bX;
        out_result      <= 'bX;
    end else begin
        if (core_state == IDLE) begin
            out_result.cost <= 'b0;
        end else if (core_state != OUT) begin
            out_result.cost <= out_result.cost + 1;
        end

        case (core_state)
            IDLE: begin
                if (in_ready & in_valid) begin
                    next_entry_idx    <= 'b0;
                    mem_addr          <= 'b0;
                    stack_clear       <= 'b1;

                    out_result.job_id <= in_job.job_id;

                    cur_stack_frame.entry_idx <= in_job.addr;
                    cur_stack_frame.t0.x <= in_job.t0[0 +:32];
                    cur_stack_frame.t0.y <= in_job.t0[32+:32];
                    cur_stack_frame.t0.z <= in_job.t0[64+:32];
                    cur_stack_frame.t1.x <= in_job.t1[0 +:32];
                    cur_stack_frame.t1.y <= in_job.t1[32+:32];
                    cur_stack_frame.t1.z <= in_job.t1[64+:32];
                    cur_stack_frame.tm.x <= (
                        $signed(in_job.t0[0 +:32] + in_job.t1[0 +:32]) >>> 1
```

```
            );
            cur_stack_frame.tm.y <= (
                $signed(in_job.t0[32+:32] + in_job.t1[32+:32]) >>> 1
            );
            cur_stack_frame.tm.z <= (
                $signed(in_job.t0[64+:32] + in_job.t1[64+:32]) >>> 1
            );

            top_t0     <= in_job.t0;
            top_t1     <= in_job.t1;
            transform  <= in_job.a;
            core_state <= INIT;
        end
    end

    INIT: begin
        cur_stack_frame.cur_child_idx <= initial_child_idx;

        mem_addr    <= cur_stack_frame.entry_idx;
        stack_clear <= 1'b0;
        core_state  <= EVAL;
    end

    EVAL: begin
        if (!ray_valid) begin
            out_result.hit <= 1'b0;
            core_state     <= OUT;
        end else if (mem_data[{2'b01, cur_child_idx_t}]) begin
            if (mem_data[{2'b00, cur_child_idx_t}]) begin
                out_result.hit    <= 1'b1;
                out_result.normal <= 'b0;

                if (
                    $signed(t0_child.x) >= $signed(t0_child.y) &&
                    $signed(t0_child.x) >= $signed(t0_child.z)
                ) begin
                    out_result.t_hit          <= t0_child.x;
                    out_result.normal[0 +:32] <= 32'h00010000;
                end else if (
                    $signed(t0_child.y) >= $signed(t0_child.x) &&
                    $signed(t0_child.y) >= $signed(t0_child.z)
                ) begin
                    out_result.t_hit          <= t0_child.y;
                    out_result.normal[32+:32] <= 32'h00010000;
                end else begin
                    out_result.t_hit          <= t0_child.z;
                    out_result.normal[64+:32] <= 32'h00010000;
                end

                core_state <= OUT;
            end else begin
                stack_push <= 1'b1;
                if (mem_data[31] == 1'b1) begin
                    mem_addr      <= (
                        cur_stack_frame.entry_idx +
                        {17'b0, mem_data[30:16]}
```

```
                );
                push_indirect <= 1'b1;
            end else begin
                next_entry_idx <= (
                    cur_stack_frame.entry_idx +
                    {17'b0, mem_data[30:16]}
                );
            end
            core_state <= PUSH;
        end
    end else begin
        core_state <= NEXT;
    end
end

NEXT: begin
    if (next_child_idx < 8) begin
        cur_stack_frame.cur_child_idx <= next_child_idx[2:0];
        core_state <= EVAL;
    end else begin
        if (stack_empty == 1'b1) begin
            out_result.hit <= 1'b0;
            core_state     <= OUT;
        end else begin
            cur_stack_frame <= stack_top;
            stack_pop       <= 1'b1;
            core_state      <= POP;
        end
    end
end

PUSH: begin
    stack_push      <= 1'b0;
    cur_stack_frame <= '0;
    if (push_indirect == 1'b1) begin
        cur_stack_frame.entry_idx <= (
            cur_stack_frame.entry_idx +
            mem_data + {29'b0, ones_count[cur_child_idx_t]}
        );
    end else begin
        cur_stack_frame.entry_idx <= (
            next_entry_idx +
            {29'b0, ones_count[cur_child_idx_t]}
        );
    end

    push_indirect        <= 1'b0;
    cur_stack_frame.t0   <= t0_child;
    cur_stack_frame.t1   <= t1_child;
    cur_stack_frame.tm.x <= $signed(t0_child.x + t1_child.x) >>> 1;
    cur_stack_frame.tm.y <= $signed(t0_child.y + t1_child.y) >>> 1;
    cur_stack_frame.tm.z <= $signed(t0_child.z + t1_child.z) >>> 1;
    core_state           <= INIT;
end

POP: begin
```

```
                stack_pop  <= 1'b0;
                mem_addr   <= cur_stack_frame.entry_idx;
                core_state <= NEXT;
            end

            OUT: begin
                if (out_ready) begin
                    core_state <= IDLE;
                end
            end
        endcase
    end
end
endmodule


module get_initial_child_idx(
    input  vec3_t       t0,
    input  vec3_t       tm,
    output logic  [2:0] initial_child_idx
);

always_comb begin
    initial_child_idx = 3'h0;
    if ($signed(t0.x) >= $signed(t0.y) &&
        $signed(t0.x) >= $signed(t0.z)) begin
        if ($signed(t0.x) >= $signed(tm.y)) begin
            initial_child_idx[1] = 1'b1;
        end
        if ($signed(t0.x) >= $signed(tm.z)) begin
            initial_child_idx[2] = 1'b1;
        end
    end else if ($signed(t0.y) >= $signed(t0.x) &&
        $signed(t0.y) >= $signed(t0.z)) begin
        if ($signed(t0.y) >= $signed(tm.x)) begin
            initial_child_idx[0] = 1'b1;
        end
        if ($signed(t0.y) >= $signed(tm.z)) begin
            initial_child_idx[2] = 1'b1;
        end
    end else begin
        if ($signed(t0.z) >= $signed(tm.x)) begin
            initial_child_idx[0] = 1'b1;
        end
        if ($signed(t0.z) >= $signed(tm.y)) begin
            initial_child_idx[1] = 1'b1;
        end
    end
end
endmodule


module get_next_child_idx(
    input  logic        clk,
    input  logic  [2:0] cur_child_idx,
    input  vec3_t       t1,
```

```
    output logic  [3:0] next_child_idx
);

logic [3:0] lookup [0:31] = '{
    4'b0001, 4'b0010, 4'b0100, 4'b1111,
    4'b1000, 4'b0011, 4'b0101, 4'b1111,
    4'b0011, 4'b1000, 4'b0110, 4'b1111,
    4'b1000, 4'b1000, 4'b0111, 4'b1111,
    4'b0101, 4'b0110, 4'b1000, 4'b1111,
    4'b1000, 4'b0111, 4'b1000, 4'b1111,
    4'b0111, 4'b1000, 4'b1000, 4'b1111,
    4'b1000, 4'b1000, 4'b1000, 4'b1111
};

always @(posedge clk) begin
    if (
        $signed(t1.x) <= $signed(t1.y) &&
        $signed(t1.x) <= $signed(t1.z)
    ) begin
        next_child_idx <= lookup[{cur_child_idx, 2'b00}];
    end else if (
        $signed(t1.y) <= $signed(t1.x) &&
        $signed(t1.y) <= $signed(t1.z)
    ) begin
        next_child_idx <= lookup[{cur_child_idx, 2'b01}];
    end else begin
        next_child_idx <= lookup[{cur_child_idx, 2'b10}];
    end
end
endmodule


module get_ray_valid(
    input  logic        clk,
    input  logic [95:0] t0,
    input  logic [95:0] t1,
    output logic        ray_valid
);

logic [31:0] t0_max;
logic [31:0] t1_min;

always_comb begin
    if (
        $signed(t0[0 +:32]) >= $signed(t0[32+:32]) &&
        $signed(t0[0 +:32]) >= $signed(t0[64+:32])
    ) begin
        t0_max = t0[0 +:32];
    end else if (
        $signed(t0[32+:32]) >= $signed(t0[0 +:32]) &&
        $signed(t0[32+:32]) >= $signed(t0[64+:32])
    ) begin
        t0_max = t0[32+:32];
    end else begin
        t0_max = t0[64+:32];
    end
```

```
end

always_comb begin
    if (
        $signed(t1[0 +:32]) <= $signed(t1[32+:32]) &&
        $signed(t1[0 +:32]) <= $signed(t1[64+:32])
    ) begin
        t1_min = t1[0 +:32];
    end else if (
        $signed(t1[32+:32]) <= $signed(t1[0 +:32]) &&
        $signed(t1[32+:32]) <= $signed(t1[64+:32])
    ) begin
        t1_min = t1[32+:32];
    end else begin
        t1_min = t1[64+:32];
    end
end

always @(posedge clk) begin
    if ($signed(t0_max) >= $signed(t1_min)) begin
        ray_valid = 0;
    end else begin
        ray_valid = 1;
    end
end

endmodule


module get_ones_count(
    input  logic [6:0] data,
    output logic [2:0] ones_count [8]
);

always_comb begin
    for (int idx = 0; idx < 8; idx += 1) begin
        ones_count[idx] = '0;
        for (int sub_idx = 0; sub_idx < idx; sub_idx += 1) begin
            ones_count[idx] += {2'b00, data[sub_idx]};
        end
    end
end
endmodule


module stack(
    input   logic clk,
    input   logic reset,
    input   logic clear,
    input   logic push,
    input   logic pop,
    input   stack_frame_t data,
    output  stack_frame_t top,
    output  logic empty
);
```

```
stack_frame_t [10:0] stack;
logic          [7:0] stack_idx;
always_comb begin
    empty = (stack_idx == 8'h0);
    if (empty) begin
        top = 'b0;
    end else begin
        top = stack[stack_idx-1];
    end
end

always @(posedge clk) begin
    if (reset == 1'b1 || clear == 1'b1) begin
        stack_idx <= 0;
    end else if (push == 1'b1) begin
        stack[stack_idx] <= data;
        stack_idx        <= stack_idx + 1;
    end else if (pop == 1'b1) begin
        stack_idx <= stack_idx - 1;
    end
end

endmodule

module get_child_t_values(
    input  logic  [2:0] cur_child_idx,
    input  vec3_t       t0,
    input  vec3_t       t1,
    output vec3_t       t_child
);

always_comb begin
    if (cur_child_idx[0] == 1'b1) begin
        t_child.x = t1.x;
    end else begin
        t_child.x = t0.x;
    end
    if (cur_child_idx[1] == 1'b1) begin
        t_child.y = t1.y;
    end else begin
        t_child.y = t0.y;
    end
    if (cur_child_idx[2] == 1'b1) begin
        t_child.z = t1.z;
    end else begin
        t_child.z = t0.z;
    end
end
endmodule
```

## C.5 Result manager

```
import types::*;

module result_manager #(
```

```
    parameter CORE_CNT = 16
) (
    input  logic                   clk,
    input  logic                   reset,

    output logic    [CORE_CNT-1:0] core_ready,
    input  logic    [CORE_CNT-1:0] core_valid,
    input  result_t [CORE_CNT-1:0] core_result,

    input  logic                   out_ready,
    output logic                   out_valid,
    output result_t                out_result
);

logic    [$clog2(CORE_CNT)-1:0] result_select;
logic            [CORE_CNT-1:0] result_valid;
result_t         [CORE_CNT-1:0] result_buffer;

assign out_valid  = result_valid[result_select];
assign core_ready = ~result_valid;
assign out_result = result_buffer[result_select];

always @(posedge clk) begin
    if (reset == 1'b1) begin
        result_valid  <= 'b0;
        result_select <= 'b0;
    end else begin
        for (int i = 0; i < CORE_CNT; i += 1) begin
            if (core_ready[i] & core_valid[i]) begin
                result_valid[i]  <= 'b1;
                result_buffer[i] <= core_result[i];
            end
        end

        if (out_ready & out_valid) begin
            result_valid[result_select] <= 'b0;
        end

        result_select <= 'b0;

        if (result_select < CORE_CNT - 1) begin
            result_select <= result_select + 1;
        end
    end
end
endmodule
```

## C.6 Data types

```
package types;
    typedef logic [31:0] float32;
    typedef logic [31:0] fixed32;
    typedef logic [95:0] vec3_pack_t;

    typedef struct packed {
        fixed32 x;
        fixed32 y;
        fixed32 z;
    } vec3_t;

    typedef struct packed {
        vec3_pack_t        orig;
        vec3_pack_t        dir;
        logic       [31:0] addr;
        logic       [31:0] job_id;
    } ray_t;

    typedef struct packed {
        logic        [2:0] a;
        vec3_pack_t        t0;
        vec3_pack_t        t1;
        logic       [31:0] addr;
        logic       [31:0] job_id;
    } job_t;

    typedef struct packed {
        logic              hit;
        fixed32            t_hit;
        vec3_pack_t        normal;
        logic       [31:0] cost;
        logic       [31:0] job_id;
    } result_t;
endpackage
```

# Appendix D

# AXI hardware modules

These modules were originally generated by the Vivado AXI wizard, but have heavily modified to fit the use case of this thesis.

## D.1 AXI top module

```
import types::*;

module rtu_v1_0 #(
    parameter integer C_S00_AXIS_TDATA_WIDTH = 256,
    parameter integer C_M00_AXIS_TDATA_WIDTH = 256,
    parameter integer CORE_CNT               = 16,
    parameter integer FX_PT_FRAC_BITS        = 16
)(
    // Ports of Axi Slave Bus Interface S00_AXIS
    input  wire                                    s00_axis_aclk,
    input  wire                                    s00_axis_aresetn,
    output wire                                    s00_axis_tready,
    input  wire     [C_S00_AXIS_TDATA_WIDTH-1 : 0] s00_axis_tdata,
    input  wire [(C_S00_AXIS_TDATA_WIDTH/8)-1 : 0] s00_axis_tstrb,
    input  wire                                    s00_axis_tlast,
    input  wire                                    s00_axis_tvalid,

    // Ports of Axi Master Bus Interface M00_AXIS
    input  wire                                    m00_axis_aclk,
    input  wire                                    m00_axis_aresetn,
    output wire                                    m00_axis_tvalid,
    output wire     [C_M00_AXIS_TDATA_WIDTH-1 : 0] m00_axis_tdata,
    output wire [(C_M00_AXIS_TDATA_WIDTH/8)-1 : 0] m00_axis_tstrb,
    output wire                                    m00_axis_tlast,
    input  wire                                    m00_axis_tready
);

    wire         job_ready;
    wire         job_valid;
    wire [255:0] job_data;

    wire  [31:0] job_count;
    wire         in_done;
    wire         out_done;
```

```
wire         result_ready;
wire         result_valid;
wire [255:0] result_data;

// Instantiation of Axi Bus Interface S00_AXIS
rtu_v1_0_S00_AXIS #(
    .C_S_AXIS_TDATA_WIDTH(C_S00_AXIS_TDATA_WIDTH)
) rtu_v1_0_S00_AXIS_inst (
    .job_ready(job_ready),
    .job_valid(job_valid),
    .job_data(job_data),
    .job_count(job_count),
    .in_done(in_done),
    .out_done(out_done),

    .S_AXIS_ACLK(s00_axis_aclk),
    .S_AXIS_ARESETN(s00_axis_aresetn),
    .S_AXIS_TREADY(s00_axis_tready),
    .S_AXIS_TDATA(s00_axis_tdata),
    .S_AXIS_TSTRB(s00_axis_tstrb),
    .S_AXIS_TLAST(s00_axis_tlast),
    .S_AXIS_TVALID(s00_axis_tvalid)
);

// Instantiation of Axi Bus Interface M00_AXIS
rtu_v1_0_M00_AXIS #(
    .C_M_AXIS_TDATA_WIDTH(C_M00_AXIS_TDATA_WIDTH)
) rtu_v1_0_M00_AXIS_inst (
    .result_ready(result_ready),
    .result_valid(result_valid),
    .result_data(result_data),
    .job_count(job_count),
    .in_done(in_done),
    .out_done(out_done),

    .M_AXIS_ACLK(m00_axis_aclk),
    .M_AXIS_ARESETN(m00_axis_aresetn),
    .M_AXIS_TVALID(m00_axis_tvalid),
    .M_AXIS_TDATA(m00_axis_tdata),
    .M_AXIS_TSTRB(m00_axis_tstrb),
    .M_AXIS_TLAST(m00_axis_tlast),
    .M_AXIS_TREADY(m00_axis_tready)
);

wire     clk;
wire     reset;
wire     in_ready;
wire     in_valid;
ray_t    in_ray;
wire     out_ready;
wire     out_valid;
result_t out_result;

rtu #(
    .CORE_CNT(CORE_CNT),
```

```
        .FX_PT_FRAC_BITS(FX_PT_FRAC_BITS)
    ) rtu_inst (
        .clk(clk),
        .reset(reset),

        .in_ready(in_ready),
        .in_valid(in_valid),
        .in_ray(in_ray),

        .out_ready(out_ready),
        .out_valid(out_valid),
        .out_result(out_result)
    );

    assign clk          =  s00_axis_aclk;
    assign reset        = ~s00_axis_aresetn;

    assign job_ready     = in_ready;
    assign in_valid      = job_valid;

    assign in_ray.job_id = job_data[0  +:32];
    assign in_ray.orig   = job_data[32 +:96];
    assign in_ray.dir    = job_data[128+:96];
    assign in_ray.addr   = 'b0;

    assign out_ready    = result_ready;
    assign result_valid = out_valid;
    assign result_data  = {
        32'b0,
        out_result.cost,
        out_result.normal,
        out_result.t_hit,
        31'b0, out_result.hit,
        out_result.job_id
    };
endmodule
```

## D.2 AXI4-Stream slave module

```
module rtu_v1_0_S00_AXIS #(
    parameter integer C_S_AXIS_TDATA_WIDTH = 256
)(
    input  wire        job_ready,
    input  wire        out_done,
    output reg         job_valid,
    output reg [255:0] job_data,

    output reg  [31:0] job_count,
    output reg         in_done,

    // Do not modify the ports beyond this line
    input  wire                                 S_AXIS_ACLK,
    input  wire                                 S_AXIS_ARESETN,
    output wire                                 S_AXIS_TREADY,
    input  wire     [C_S_AXIS_TDATA_WIDTH-1 : 0] S_AXIS_TDATA,
```

```
    input  wire [(C_S_AXIS_TDATA_WIDTH/8)-1 : 0] S_AXIS_TSTRB,
    input  wire                                  S_AXIS_TLAST,
    input  wire                                  S_AXIS_TVALID
);
    reg          axi_ready;
    wire         axi_valid;
    wire         axi_last;
    wire [255:0] axi_data;

    assign S_AXIS_TREADY = axi_ready;
    assign axi_valid     = S_AXIS_TVALID;
    assign axi_data      = S_AXIS_TDATA;
    assign axi_last      = S_AXIS_TLAST;

    always @(posedge S_AXIS_ACLK) begin
        if (!S_AXIS_ARESETN) begin
            axi_ready <= 'b1;
            job_valid <= 'b0;
            job_data  <= 'b0;
            job_count <= 'b0;
            in_done   <= 'b0;
        end else begin
            if (axi_ready && axi_valid) begin
                if (axi_last) begin
                    in_done <= 'b1;
                end

                job_data  <= axi_data;
                axi_ready <= 'b0;
                job_valid <= 'b1;
                job_count <= job_count + 1;
            end

            if (job_ready && job_valid) begin
                axi_ready <= 'b1;
                job_valid <= 'b0;
                job_data  <= 'b0;
            end

            if (in_done && out_done) begin
                in_done <= 'b0;
            end
        end
    end
endmodule
```

## D.3 AXI4-Stream master module

```
module rtu_v1_0_M00_AXIS #(
    parameter integer C_M_AXIS_TDATA_WIDTH = 256
)(
    output reg          result_ready,
    input  wire         result_valid,
    input  wire [255:0] result_data,
    input  wire  [31:0] job_count,
```

```
    input  wire          in_done,
    output reg           out_done,

    // Do not modify the ports beyond this line
    input  wire                                     M_AXIS_ACLK,
    input  wire                                     M_AXIS_ARESETN,
    output wire                                     M_AXIS_TVALID,
    output wire     [C_M_AXIS_TDATA_WIDTH-1 : 0]    M_AXIS_TDATA,
    output wire [(C_M_AXIS_TDATA_WIDTH/8)-1 : 0]    M_AXIS_TSTRB,
    output wire                                     M_AXIS_TLAST,
    input  wire                                     M_AXIS_TREADY
);
    wire        axi_ready;
    wire        axi_valid;
    reg         axi_last;
    reg [255:0] axi_data;
    reg  [31:0] result_count;

    assign M_AXIS_TVALID = axi_valid;
    assign M_AXIS_TDATA  = axi_data;
    assign M_AXIS_TLAST  = axi_last;
    assign M_AXIS_TSTRB  = {(C_M_AXIS_TDATA_WIDTH/8){1'b1}};
    assign axi_ready     = M_AXIS_TREADY;
    assign axi_valid     = ~result_ready;

    always @(posedge M_AXIS_ACLK) begin
        if (!M_AXIS_ARESETN) begin
            axi_last     <= 'b0;
            axi_data     <= 'b0;
            result_count <= 'b0;
            result_ready <= 'b1;
            out_done     <= 'b0;
        end else begin
            if (result_ready && result_valid) begin
                axi_data     <= result_data;
                result_count <= result_count + 1;
                axi_last     <= in_done & (result_count + 1 == job_count);
                out_done     <= in_done & (result_count + 1 == job_count);
                result_ready <= 'b0;
            end

            if (axi_ready & axi_valid) begin
                result_ready <= 'b1;
                axi_last     <= 'b0;
                out_done     <= 'b0;
            end
        end
    end
endmodule
```

# Appendix E

# Vivado synthesis reports

The synthesis reports from Vivado are attached in this appendix. The reports are generated during the synthesis of an RTU system with 16 SVO traversal cores.

## E.1 Utilisation report

```
Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
------------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.3 (lin64) Build 2405991 Thu Dec  6 23:36:41 MST 2018
| Date         : Mon May 20 11:43:54 2019
| Host         : asbjorn-pc running 64-bit Manjaro Linux
| Command      : report_utilization -file
                     ray_tracer_bd_wrapper_utilization_placed.rpt -pb
                     ray_tracer_bd_wrapper_utilization_placed.pb
| Design       : ray_tracer_bd_wrapper
| Device       : 7z020clg400-1
| Design State : Fully Placed
------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Slice Logic Distribution
3. Memory
4. DSP
5. IO and GT Specific
6. Clocking
7. Specific Feature
8. Primitives
9. Black Boxes
10. Instantiated Netlists

1. Slice Logic
--------------

+----------------------------+-------+-------+-----------+-------+
|          Site Type         |  Used | Fixed | Available | Util% |
```

| | | | | |
|---|---|---|---|---|
| Slice LUTs | 36714 | 0 | 53200 | 69.01 |
| LUT as Logic | 31795 | 0 | 53200 | 59.77 |
| LUT as Memory | 4919 | 0 | 17400 | 28.27 |
| LUT as Distributed RAM | 4670 | 0 | | |
| LUT as Shift Register | 249 | 0 | | |
| Slice Registers | 23874 | 0 | 106400 | 22.44 |
| Register as Flip Flop | 23874 | 0 | 106400 | 22.44 |
| Register as Latch | 0 | 0 | 106400 | 0.00 |
| F7 Muxes | 901 | 0 | 26600 | 3.39 |
| F8 Muxes | 101 | 0 | 13300 | 0.76 |

1.1 Summary of Registers by Type

| Total | Clock Enable | Synchronous | Asynchronous |
|---|---|---|---|
| 0 | _ | - | - |
| 0 | _ | - | Set |
| 0 | _ | - | Reset |
| 0 | _ | Set | - |
| 0 | _ | Reset | - |
| 0 | Yes | - | - |
| 0 | Yes | - | Set |
| 0 | Yes | - | Reset |
| 246 | Yes | Set | - |
| 23628 | Yes | Reset | - |

2. Slice Logic Distribution

| Site Type | Used | Fixed | Available | Util% |
|---|---|---|---|---|
| Slice | 11177 | 0 | 13300 | 84.04 |
| SLICEL | 7517 | 0 | | |
| SLICEM | 3660 | 0 | | |
| LUT as Logic | 31795 | 0 | 53200 | 59.77 |
| using O5 output only | 7 | | | |
| using O6 output only | 21774 | | | |
| using O5 and O6 | 10014 | | | |
| LUT as Memory | 4919 | 0 | 17400 | 28.27 |
| LUT as Distributed RAM | 4670 | 0 | | |
| using O5 output only | 0 | | | |
| using O6 output only | 2 | | | |
| using O5 and O6 | 4668 | | | |
| LUT as Shift Register | 249 | 0 | | |
| using O5 output only | 15 | | | |
| using O6 output only | 111 | | | |
| using O5 and O6 | 123 | | | |
| Slice Registers | 23874 | 0 | 106400 | 22.44 |

```
|   Register driven from within the Slice    | 14157 |       |           |       |
|   Register driven from outside the Slice   |  9717 |       |           |       |
|     LUT in front of the register is unused |  5264 |       |           |       |
|     LUT in front of the register is used   |  4453 |       |           |       |
| Unique Control Sets                        |   472 |       |     13300 |  3.55 |
+--------------------------------------------+-------+-------+-----------+-------+
* Note: Available Control Sets calculated as Slice Registers / 8, Review the
Control Sets Report for more information regarding control sets.


3. Memory
---------

+-------------------+------+-------+-----------+-------+
|     Site Type     | Used | Fixed | Available | Util% |
+-------------------+------+-------+-----------+-------+
| Block RAM Tile    |    9 |     0 |       140 |  6.43 |
|   RAMB36/FIFO*    |    8 |     0 |       140 |  5.71 |
|     RAMB36E1 only |    8 |       |           |       |
|   RAMB18          |    2 |     0 |       280 |  0.71 |
|     RAMB18E1 only |    2 |       |           |       |
+-------------------+------+-------+-----------+-------+
* Note: Each Block RAM Tile only has one FIFO logic available and therefore can
accommodate only one FIFO36E1 or one FIFO18E1. However, if a FIFO18E1 occupies a
Block RAM Tile, that tile can still accommodate a RAMB18E1


4. DSP
------

+-----------+------+-------+-----------+-------+
| Site Type | Used | Fixed | Available | Util% |
+-----------+------+-------+-----------+-------+
| DSPs      |    0 |     0 |       220 |  0.00 |
+-----------+------+-------+-----------+-------+


5. IO and GT Specific
---------------------

+-----------------------------+------+-------+-----------+--------+
|          Site Type          | Used | Fixed | Available |  Util% |
+-----------------------------+------+-------+-----------+--------+
| Bonded IOB                  |    0 |     0 |       125 |   0.00 |
| Bonded IPADs                |    0 |     0 |         2 |   0.00 |
| Bonded IOPADs               |  130 |   130 |       130 | 100.00 |
| PHY_CONTROL                 |    0 |     0 |         4 |   0.00 |
| PHASER_REF                  |    0 |     0 |         4 |   0.00 |
| OUT_FIFO                    |    0 |     0 |        16 |   0.00 |
| IN_FIFO                     |    0 |     0 |        16 |   0.00 |
| IDELAYCTRL                  |    0 |     0 |         4 |   0.00 |
| IBUFDS                      |    0 |     0 |       121 |   0.00 |
| PHASER_OUT/PHASER_OUT_PHY   |    0 |     0 |        16 |   0.00 |
| PHASER_IN/PHASER_IN_PHY     |    0 |     0 |        16 |   0.00 |
| IDELAYE2/IDELAYE2_FINEDELAY |    0 |     0 |       200 |   0.00 |
| ILOGIC                      |    0 |     0 |       125 |   0.00 |
```

| OLOGIC | 0 | 0 | 125 | 0.00 |

6. Clocking

| Site Type | Used | Fixed | Available | Util% |
|---|---|---|---|---|
| BUFGCTRL | 1 | 0 | 32 | 3.13 |
| BUFIO | 0 | 0 | 16 | 0.00 |
| MMCME2_ADV | 0 | 0 | 4 | 0.00 |
| PLLE2_ADV | 0 | 0 | 4 | 0.00 |
| BUFMRCE | 0 | 0 | 8 | 0.00 |
| BUFHCE | 0 | 0 | 72 | 0.00 |
| BUFR | 0 | 0 | 16 | 0.00 |

7. Specific Feature

| Site Type | Used | Fixed | Available | Util% |
|---|---|---|---|---|
| BSCANE2 | 0 | 0 | 4 | 0.00 |
| CAPTUREE2 | 0 | 0 | 1 | 0.00 |
| DNA_PORT | 0 | 0 | 1 | 0.00 |
| EFUSE_USR | 0 | 0 | 1 | 0.00 |
| FRAME_ECCE2 | 0 | 0 | 1 | 0.00 |
| ICAPE2 | 0 | 0 | 2 | 0.00 |
| STARTUPE2 | 0 | 0 | 1 | 0.00 |
| XADC | 0 | 0 | 1 | 0.00 |

8. Primitives

| Ref Name | Used | Functional Category |
|---|---|---|
| FDRE | 23628 | Flop & Latch |
| LUT6 | 14369 | LUT |
| LUT4 | 12877 | LUT |
| LUT3 | 8422 | LUT |
| RAMD32 | 7004 | Distributed Memory |
| CARRY4 | 3137 | CarryLogic |
| LUT2 | 2929 | LUT |
| LUT5 | 2449 | LUT |
| RAMS32 | 2334 | Distributed Memory |
| MUXF7 | 901 | MuxFx |
| LUT1 | 763 | LUT |
| SRL16E | 327 | Distributed Memory |
| FDSE | 246 | Flop & Latch |

```
| BIBUF    |   130 |                   IO |
| MUXF8    |   101 |                MuxFx |
| SRLC32E  |    45 |   Distributed Memory |
| RAMB36E1 |     8 |         Block Memory |
| RAMB18E1 |     2 |         Block Memory |
| PS7      |     1 | Specialized Resource |
| BUFG     |     1 |                Clock |
+----------+-------+----------------------+


9. Black Boxes
--------------

+----------+------+
| Ref Name | Used |
+----------+------+


10. Instantiated Netlists
-------------------------

+----------+------+
| Ref Name | Used |
+----------+------+
```

## E.2 Timing summary report

```
Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.
------------------------------------------------------------------------------------
| Tool Version : Vivado v.2018.3 (lin64) Build 2405991 Thu Dec  6 23:36:41 MST 2018
| Date         : Mon May 20 11:46:43 2019
| Host         : asbjorn-pc running 64-bit Manjaro Linux
| Command      : report_timing_summary -max_paths 10 -file
                     ray_tracer_bd_wrapper_timing_summary_routed.rpt -pb
                     ray_tracer_bd_wrapper_timing_summary_routed.pb -rpx
                     ray_tracer_bd_wrapper_timing_summary_routed.rpx
                     -warn_on_violation
| Design       : ray_tracer_bd_wrapper
| Device       : 7z020-clg400
| Speed File   : -1  PRODUCTION 1.11 2014-09-11
------------------------------------------------------------------------------------

Timing Summary Report

------------------------------------------------------------------------------------
| Timer Settings
| --------------
------------------------------------------------------------------------------------

  Enable Multi Corner Analysis               :  Yes
  Enable Pessimism Removal                   :  Yes
  Pessimism Removal Resolution               :  Nearest Common Node
  Enable Input Delay Default Clock           :  No
  Enable Preset / Clear Arcs                 :  No
  Disable Flight Delays                      :  No
```

```
  Ignore I/O Paths                              :  No
  Timing Early Launch at Borrowing Latches      :  false

  Corner  Analyze    Analyze
  Name    Max Paths  Min Paths
  ------  ---------  ---------
  Slow    Yes        Yes
  Fast    Yes        Yes



check_timing report

Table of Contents
-----------------
1. checking no_clock
2. checking constant_clock
3. checking pulse_width_clock
4. checking unconstrained_internal_endpoints
5. checking no_input_delay
6. checking no_output_delay
7. checking multiple_clock
8. checking generated_clocks
9. checking loops
10. checking partial_input_delay
11. checking partial_output_delay
12. checking latch_loops

1. checking no_clock
--------------------
 There are 0 register/latch pins with no clock.


2. checking constant_clock
--------------------------
 There are 0 register/latch pins with constant_clock.


3. checking pulse_width_clock
-----------------------------
 There are 0 register/latch pins which need pulse_width check


4. checking unconstrained_internal_endpoints
--------------------------------------------
 There are 0 pins that are not constrained for maximum delay.

 There are 0 pins that are not constrained for maximum delay due to constant clock.


5. checking no_input_delay
--------------------------
 There are 0 input ports with no input delay specified.

 There are 0 input ports with no input delay but user has a false path constraint.
```

```
6. checking no_output_delay
---------------------------
 There are 0 ports with no output delay specified.

 There are 0 ports with no output delay but user has a false path constraint

 There are 0 ports with no output delay but with a timing clock defined on it or
 propagating through it


7. checking multiple_clock
--------------------------
 There are 0 register/latch pins with multiple clocks.


8. checking generated_clocks
----------------------------
 There are 0 generated clocks that are not connected to a clock source.


9. checking loops
-----------------
 There are 0 combinational loops in the design.


10. checking partial_input_delay
--------------------------------
 There are 0 input ports with partial input delay specified.


11. checking partial_output_delay
---------------------------------
 There are 0 ports with partial output delay specified.


12. checking latch_loops
------------------------
 There are 0 combinational latch loops in the design through latch input



------------------------------------------------------------------------------------
| Design Timing Summary
| ---------------------
------------------------------------------------------------------------------------

    WNS(ns)      TNS(ns)  TNS Failing Endpoints  TNS Total Endpoints
    -------      -------  ---------------------  -------------------
      0.124        0.000                      0               114194

    WHS(ns)      THS(ns)  THS Failing Endpoints  THS Total Endpoints
    -------      -------  ---------------------  -------------------
      0.022        0.000                      0               114194

    WPWS(ns)     TPWS(ns)  TPWS Failing Endpoints  TPWS Total Endpoints
```

```
--------     --------  ----------------------  --------------------
   3.750        0.000                       0                33605

All user specified timing constraints are met.
```

# Appendix F

# Software driver

The software driver for the hardware implementation is listed in this appendix. The driver runs in the Jupyter environment on the Pynq development board.

## F.1 Driver core functions

```
import numpy as np
from pynq import Xlnk
from pynq import Overlay
import time
import math
import struct
from PIL import Image

class RTU:
    def __init__(self, path):
        self.overlay = Overlay(path)
        self.dma = self.overlay.axi_dma

    def setup(self, width, height, pos, rot):
        global inv_proj
        global inv_view

        inv_proj = self.get_inv_proj(width, height)
        inv_view = self.get_inv_view(pos, rot)

        self.width = width
        self.height = height

        xlnk = Xlnk()
        self.input_buffer = xlnk.cma_array(
            shape=(width * height, 8),
            dtype=np.uint32
        )
        self.output_buffer = xlnk.cma_array(
            shape=(width * height, 8),
            dtype=np.uint32
        )
        np.set_printoptions(formatter={'int': '{: x}'.format})
```

```
        for y in range(0, height):
            for x in range(0, width):
                self.input_buffer[x + y*width,:] = self.get_ray_job(
                    x,
                    y,
                    width,
                    height
                )
            if (y + 1) % 80 == 0:
                print("Row {} of {}".format(y + 1, height))

    def start_input(self, block, debug=False):
        if debug:
            print("Starting transfer of {} jobs...".format(
                len(self.input_buffer)
            ))

        self.dma.sendchannel.transfer(self.input_buffer)
        if block:
            self.dma.sendchannel.wait()

    def start_output(self, block, debug=False):
        if debug:
            print("Trying to receive {} results...".format(
                len(self.output_buffer)
            ))

        self.dma.recvchannel.transfer(self.output_buffer)
        if block:
            self.dma.recvchannel.wait()

    def render_frame(self):
        self.start_input(block=False)
        self.start_output(block=True)

    def save_output(self, file, highlight_cost=True):
        data = np.zeros((self.height, self.width, 3))

        for result in self.output_buffer:
            i = result[0] & 0x0FFFFFFF
            x = i % self.width
            y = int(i / self.width)
            if not result[1]:
                if highlight_cost:
                    data[y, x] = [
                        1.0,
                        0.95**((result[6] - 2) / 2),
                        0.95**((result[6] - 2) / 2)
                    ]
                else:
                    data[y, x] = [1.0, 1.0, 1.0]
                continue

            if result[3]:
                data[y, x] = [0.7, 0.7, 0.7]
            elif result[4]:
```

```
            data[y, x] = [0.9, 0.9, 0.9]
        else:
            data[y, x] = [0.5, 0.5, 0.5]

    img = Image.fromarray((data * 255).astype('uint8'))
    img.save(file)


def get_inv_proj(self, width, height):
    far = 100.0
    near = 0.001
    fov = 75.0
    aspect = float(width)/height

    f = far
    n = near
    t = math.tan(fov / 180 * math.pi / 2) * n
    b = -t
    r = t * aspect
    l = -r


    return np.array([
        [-f * (r - l) / (f - n), 0, 0, -f * (r + l) / (f - n)],
        [0, -f * (t - b) / (f - n), 0, -f * (t + b) / (f - n)],
        [0, 0, 0, 2 * f * n / (f - n)],
        [0, 0, 1, -(f + n) / (f - n)]
    ])


def get_inv_view(self, pos, rot):
    inv_view = np.array([
        [1, 0, 0, 0],
        [0, 1, 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 1]
    ])

    rotation_y = np.array([
        [ math.cos(-rot[1]), 0, math.sin(-rot[1]), 0],
        [                 0, 1,                 0, 0],
        [-math.sin(-rot[1]), 0, math.cos(-rot[1]), 0],
        [                 0, 0,                 0, 1]
    ])

    s_rot = math.sin(rot[0])

    rotation_x = np.array([
        [1,                     0,                     0, 0],
        [0, math.cos(0.5 * s_rot), -math.sin(0.5 * s_rot), 0],
        [0, math.sin(0.5 * s_rot),  math.cos(0.5 * s_rot), 0],
        [0,                     0,                     0, 1]
    ])

    rotation = rotation_x.dot(rotation_y)
```

```
        translation = np.array([
            [1, 0, 0, pos[0]],
            [0, 1, 0, pos[1]],
            [0, 0, 1, pos[2]],
            [0, 0, 0,      1]
        ])

        inv_view = np.transpose(rotation).dot(inv_view)
        inv_view = translation.dot(inv_view)
        return inv_view


    def get_ray(self, x, y):
        v0 = np.array([x, y, -1, 1])
        v1 = np.array([x, y, 1, 1])

        q0 = inv_proj.dot(v0)
        q1 = inv_proj.dot(v1)

        q0 = q0 / q0[3]
        q1 = q1 / q1[3]

        q0 = inv_view.dot(q0)
        q1 = inv_view.dot(q1)

        ray_orig = [q0[0], q0[1], q0[2]]
        ray_dir = [q1[0] - q0[0], q1[1] - q0[1], q1[2] - q0[2]]

        return {'orig': ray_orig, 'dir': ray_dir}


    def get_initial_params(self, ray_orig, ray_dir):
        ray_len = (ray_dir[0]**2 + ray_dir[1]**2 + ray_dir[2]**2)**0.5
        ray_dir[0] /= ray_len
        ray_dir[1] /= ray_len
        ray_dir[2] /= ray_len

        a = 0
        dim = 1

        for comp in range(0, 3):
            if ray_dir[comp] == 0:
                ray_dir[comp] = 0.0001
            if ray_dir[comp] < 0:
                ray_orig[comp] *= -1
                ray_dir[comp] *= -1
                a |= 2**comp

        t0_float = (
            (-dim - ray_orig[0]) / ray_dir[0],
            (-dim - ray_orig[1]) / ray_dir[1],
            (-dim - ray_orig[2]) / ray_dir[2]
        )

        t1_float = (
            (dim - ray_orig[0]) / ray_dir[0],
```

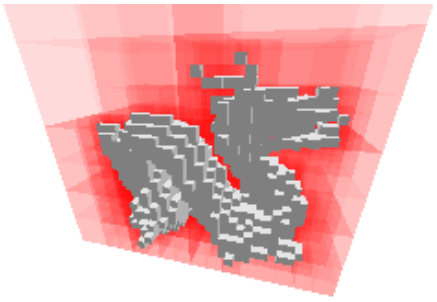
```
        (dim - ray_orig[1]) / ray_dir[1],
        (dim - ray_orig[2]) / ray_dir[2]
    )

    return {'a': a, 't0_float': t0_float, 't1_float': t1_float}


def get_float(self, val):
    return struct.unpack('I', struct.pack('f', val))[0]


def get_ray_job(self, x, y, width, height):
    ray = self.get_ray(
        (x + 0.5) / width * 2 - 1.0,
        ((height - y - 1) + 0.5) / height * 2 - 1.0
    )
    params = self.get_initial_params(ray['orig'].copy(), ray['dir'].copy())

    return [
        (x + y * width) | params['a'] << 28,
        self.get_float(params['t0_float'][0]),
        self.get_float(params['t0_float'][1]),
        self.get_float(params['t0_float'][2]),
        self.get_float(params['t1_float'][0]),
        self.get_float(params['t1_float'][1]),
        self.get_float(params['t1_float'][2]),
        0x00000000
    ]
```

## F.2 Example driver usage

```
def run_test(file, width, height, count, save=False):
    rtu = RTU(file)

    print('Start setup {}x{}'.format(width, height))
    rtu.setup(
        width,
        height,
        [-0.5, 0.35, -2.1],
        [math.pi * 4/12, math.pi * 13/12, 0]
    )
    print('Setup done!\n')

    print('Start timing test...')
    start = time.time()

    for i in range(0, count):
        rtu.render_frame()

    end = time.time()
    print("Timing test done! Rendered {} frames.".format(count))
    print("Average frame time: {:.2f} ms.".format((end - start) / count * 1000))
    print("Average frame rate: {:.2f} Hz.\n".format(1 / ((end - start) / count)))

    if save:
        print('Saving output image...')
        rtu.save_output('img_test.png')
        print('Image saving done!\n\n')

    del rtu


file = '/home/xilinx/asbjoree/ray_tracer_ip_16.bit'
run_test(file, 320, 180, 1000, True)
```

### F.2.1 Example output

```
Start setup 320x180
Row 80 of 180
Row 160 of 180
Setup done!

Start timing test...
Timing test done! Rendered 1000 frames.
Average frame time: 2.58 ms.
Average frame rate: 387.03 Hz.

Saving output image...
Image saving done!
```



# Appendix G

# Software model

The software model created during the design and implementation of the system is included in this appendix.

## G.1 Ray tracer core

```
from datatypes import *
import sys

def get_initial_child_idx(t0, tm):
    child_idx = 0

    if fp.gte(t0.x, t0.y) and fp.gte(t0.x, t0.z):
        child_idx |= (fp.gte(t0.x, tm.y) << 1);
        child_idx |= (fp.gte(t0.x, tm.z) << 2);
    elif fp.gte(t0.y, t0.x) and fp.gte(t0.y, t0.z):
        child_idx |= (fp.gte(t0.y, tm.x) << 0);
        child_idx |= (fp.gte(t0.y, tm.z) << 2);
    else:
        child_idx |= (fp.gte(t0.z, tm.x) << 0);
        child_idx |= (fp.gte(t0.z, tm.y) << 1);

    return child_idx


def get_max_t_value(t):
    if fp.gte(t.x, t.y) and fp.gte(t.x, t.z):
        return t.x
    if fp.gte(t.y, t.x) and fp.gte(t.y, t.z):
        return t.y
    else:
        return t.z


def get_next_child_idx(cur_child_idx, t):
    lookup = [
        [0b0001, 0b0010, 0b0100],
        [0b1000, 0b0011, 0b0101],
        [0b0011, 0b1000, 0b0110],
        [0b1000, 0b1000, 0b0111],
```

```
        [0b0101, 0b0110, 0b1000],
        [0b1000, 0b0111, 0b1000],
        [0b0111, 0b1000, 0b1000],
        [0b1000, 0b1000, 0b1000]
    ]

    if fp.lte(t.x, t.y) and fp.lte(t.x, t.z):
        return lookup[cur_child_idx][0]
    if fp.lte(t.y, t.x) and fp.lte(t.y, t.z):
        return lookup[cur_child_idx][1]
    else:
        return lookup[cur_child_idx][2]


def get_child_t_values(child_idx, t0, t1):
    t = vec3()
    t.x = t1.x if (child_idx & 0x1) else t0.x
    t.y = t1.y if (child_idx & 0x2) else t0.y
    t.z = t1.z if (child_idx & 0x4) else t0.z
    return t


def sum_and_right_shift(t0, t1):
    return vec3.shr(vec3.add(t0, t1))

state = None
stack = []
stack_idx = 0
out = {
    'ready': False,
    't_hit': None
}

t0_child = vec3()
t1_child = vec3()
next_child = 0
a = 0

def rt_core(reset, start, params, mem, debug):
    global state
    global stack
    global stack_idx
    global out
    global t0_child
    global t1_child
    global next_child
    global a
    global push_entry
    global push_offset_lookup


    if debug:
        print('=== %s ===' % state)
        print('\tOut: %s\n\tStack: %s' % (out, stack))

    if reset:
```

```
        state = 'IDLE'
        out['ready'] = True
        return out

    if state == 'IDLE':
        stack = []
        t0_child = vec3()
        t1_child = vec3()
        next_child = 0
        stack_idx = 0
        if start:
            state = 'INIT'
            out['ready'] = False
            stack.append(stack_frame())
            stack[stack_idx].entry = params['entry']
            stack[stack_idx].t0 = params['t0']
            stack[stack_idx].t1 = params['t1']
            stack[stack_idx].tm = sum_and_right_shift(
                params['t0'],
                params['t1']
            )
            a = params['a']
        else:
            return out
    elif state == 'INIT':
        stack[stack_idx].cur_child_idx = get_initial_child_idx(
            stack[stack_idx].t0,
            stack[stack_idx].tm
        );
        state = 'EVAL'
    elif state == 'EVAL':
        t0_child = get_child_t_values(
            stack[stack_idx].cur_child_idx,
            stack[stack_idx].t0,
            stack[stack_idx].tm
        )
        t1_child = get_child_t_values(
            stack[stack_idx].cur_child_idx,
            stack[stack_idx].tm,
            stack[stack_idx].t1
        )
        if (
            ((mem[stack[stack_idx].entry] & 0x0000FF00) >> 8) &
            (1 << (a ^ stack[stack_idx].cur_child_idx))
        ):
            if (
                (mem[stack[stack_idx].entry] & 0x000000FF) &
                (1 << (a ^ stack[stack_idx].cur_child_idx))
            ):
                out['hit'] = True
                state = 'OUT'
            else:
                state = 'PUSH'
                push_entry = stack[stack_idx].entry + \
                             ((mem[stack[stack_idx].entry] & 0x7FFF0000) >> 16)
                mask = ((mem[stack[stack_idx].entry] & 0x0000FF00) >> 8) & \
```

```
                    ~(mem[stack[stack_idx].entry] & 0x000000FF)
                push_offset_lookup = [
                    0,
                    mask & 0x1,
                    (mask & 0x1) + ((mask >> 1) & 0x1),
                    (mask & 0x1) + ((mask >> 1) & 0x1) + ((mask >> 2) & 0x1),
                    (mask & 0x1) + ((mask >> 1) & 0x1) + ((mask >> 2) & 0x1) +
                        ((mask >> 3) & 0x1),
                    (mask & 0x1) + ((mask >> 1) & 0x1) + ((mask >> 2) & 0x1) +
                        ((mask >> 3) & 0x1) + ((mask >> 4) & 0x1),
                    (mask & 0x1) + ((mask >> 1) & 0x1) + ((mask >> 2) & 0x1) +
                        ((mask >> 3) & 0x1) + ((mask >> 4) & 0x1) +
                        ((mask >> 5) & 0x1),
                    (mask & 0x1) + ((mask >> 1) & 0x1) + ((mask >> 2) & 0x1) +
                        ((mask >> 3) & 0x1) + ((mask >> 4) & 0x1) +
                        ((mask >> 5) & 0x1) + ((mask >> 6) & 0x1),
                ]
        else:
            next_child = get_next_child_idx(
                stack[stack_idx].cur_child_idx,
                t1_child
            )
            state = 'NEXT'
    elif state == 'PUSH':
        entry = 0
        if mem[stack[stack_idx].entry] & 0x80000000:
            entry = stack[stack_idx].entry + mem[push_entry] + \
                    push_offset_lookup[a ^ stack[stack_idx].cur_child_idx]
        else:
            entry = push_entry + \
                    push_offset_lookup[a ^ stack[stack_idx].cur_child_idx]

        stack_idx += 1
        stack.append(stack_frame())
        stack[stack_idx].entry = entry
        stack[stack_idx].t0 = t0_child
        stack[stack_idx].t1 = t1_child
        stack[stack_idx].tm = sum_and_right_shift(t0_child, t1_child)
        state = 'INIT'
    elif state == 'NEXT':
        if next_child < 8:
            stack[stack_idx].cur_child_idx = next_child
            state = 'EVAL'
        else:
            if stack_idx == 0:
                out['hit'] = False
                state = 'OUT'
            else:
                state = 'POP'
    elif state == 'POP':
        del stack[stack_idx]
        stack_idx -= 1
        t0_child = get_child_t_values(
            stack[stack_idx].cur_child_idx,
            stack[stack_idx].t0,
            stack[stack_idx].tm
```

```
        )
        t1_child = get_child_t_values(
            stack[stack_idx].cur_child_idx,
            stack[stack_idx].tm,
            stack[stack_idx].t1
        )
        next_child = get_next_child_idx(
            stack[stack_idx].cur_child_idx,
            t1_child
        )
        state = 'NEXT'
    elif state == 'OUT':
        out['t_hit'] = get_max_t_value(t0_child)
        if out['t_hit'] == t0_child.x:
            out['normal'] = vec3(1, 0, 0)
        elif out['t_hit'] == t0_child.y:
            out['normal'] = vec3(0, 1, 0)
        else:
            out['normal'] = vec3(0, 0, 1)
        state = 'IDLE'
        out['ready'] = True

    return out
```

## G.2 Data types

```
class fp:
    binval = 0
    global_prec = 16
    size = 32

    def __init__(self, value=2**16):
        if value > 2**(fp.size - 1 - fp.global_prec) - 1:
            value = 2**(fp.size - 1 - fp.global_prec) - 2**(-fp.global_prec)
        elif value < -2**(fp.size - 1 - fp.global_prec) - 1:
            value = -2**(fp.size - 1 - fp.global_prec)

        self.binval = int(value * 2**fp.global_prec) & (2**fp.size - 1)

    def __repr__(self):
        if self.binval > 2**(fp.size - 1) - 1:
            dec = (self.binval - 2**fp.size) / 2**fp.global_prec
        else:
            dec = self.binval / 2**fp.global_prec
        return "%08X (%f)" % (self.binval, dec)

    def add(left, right):
        retval = fp(0)
        if left.binval == 2**(fp.size - 1) or right.binval == 2**(fp.size - 1):
            retval.binval = 2**(fp.size - 1)
        else:
            retval.binval = (left.binval + right.binval) & (2**fp.size - 1)
        return retval

    def shr(val):
```

```
        retval = fp(0)
        if val.binval == 2**(fp.size - 1):
            retval.binval = 2**(fp.size - 1)
        elif val.binval < 2**(fp.size - 1):
            retval.binval = val.binval >> 1
        else:
            retval.binval = (val.binval | 2**fp.size) >> 1
        return retval

    def gt(left, right):
        if (
            (left.binval > 2**(fp.size - 1) - 1) ^
            (right.binval > 2**(fp.size - 1) - 1)
        ):
            return (left.binval <= 2**(fp.size - 1) - 1)
        else:
            return left.binval > right.binval

    def lt(left, right):
        if (
            (left.binval > 2**(fp.size - 1) - 1) ^
            (right.binval > 2**(fp.size - 1) - 1)
        ):
            return (left.binval > 2**(fp.size - 1) - 1)
        else:
            return left.binval < right.binval

    def gte(left, right):
        if (
            (left.binval > 2**(fp.size - 1) - 1) ^
            (right.binval > 2**(fp.size - 1) - 1)
        ):
            return (left.binval <= 2**(fp.size - 1) - 1)
        else:
            return left.binval >= right.binval

    def lte(left, right):
        if (
            (left.binval > 2**(fp.size - 1) - 1) ^
            (right.binval > 2**(fp.size - 1) - 1)
        ):
            return (left.binval > 2**(fp.size - 1) - 1)
        else:
            return left.binval <= right.binval


class vec3:
    x = fp()
    y = fp()
    z = fp()

    def __init__(self, x=None, y=None, z=None):
        if x is not None:
            self.x = fp(x)
        if y is not None:
            self.y = fp(y)
```

```
        if z is not None:
            self.z = fp(z)


    def __repr__(self):
        return "[%s | %s | %s]" % (self.x, self.y, self.z)

    def add(left, right):
        retval = vec3()
        retval.x = fp.add(left.x, right.x)
        retval.y = fp.add(left.y, right.y)
        retval.z = fp.add(left.z, right.z)
        return retval

    def shr(val):
        retval = vec3()
        retval.x = fp.shr(val.x)
        retval.y = fp.shr(val.y)
        retval.z = fp.shr(val.z)
        return retval

    def max(self):
        if fp.gte(self.x, self.y) and fp.gte(self.x, self.z):
            return self.x
        elif fp.gte(self.y, self.x) and fp.gte(self.y, self.z):
            return self.y
        else:
            return self.z

    def min(self):
        if fp.lte(self.x, self.y) and fp.lte(self.x, self.z):
            return self.x
        elif fp.lte(self.y, self.x) and fp.lte(self.y, self.z):
            return self.y
        else:
            return self.z
```

## G.3 Test bench

```
reset = False
start = False

out = {}
max_it = 0
max_params = {}

def run_rt_core(ray_orig, ray_dir, debug=False):
    global out
    global max_it
    global max_params
    params = get_initial_params(ray_orig.copy(), ray_dir.copy())
    params['ray_orig'] = ray_orig
    params['ray_dir'] = ray_dir

    if fp.gte(params['t0'].max(), params['t1'].min()):
```

```
        out['cost'] = 0
        return False

    params['entry'] = 0

    out = rt_core(True, False, {}, mem, debug)
    out = rt_core(False, True, params, mem, debug)

    for i in range(0, 1000):
        out = rt_core(False, False, params, mem, debug)
        if out['ready']:
            hit = out['hit']
            out = rt_core(False, False, {}, mem, debug)
            out = rt_core(False, False, {}, mem, debug)
            if hit and i > max_it:
                max_it = i
                max_params = params
            out['cost'] = i
            return hit


def get_initial_params(ray_orig, ray_dir):
    ray_len = (ray_dir[0]**2 + ray_dir[1]**2 + ray_dir[2]**2)**0.5
    ray_dir[0] /= ray_len
    ray_dir[1] /= ray_len
    ray_dir[2] /= ray_len

    a = 0
    dim = 1

    for comp in range(0, 3):
        if ray_dir[comp] == 0:
            ray_dir[comp] = 0.0001
        if ray_dir[comp] < 0:
            ray_orig[comp] *= -1
            ray_dir[comp] *= -1
            a |= 2**comp

    t0_float = (
        (-dim - ray_orig[0]) / ray_dir[0],
        (-dim - ray_orig[1]) / ray_dir[1],
        (-dim - ray_orig[2]) / ray_dir[2]
    )

    t1_float = (
        (dim - ray_orig[0]) / ray_dir[0],
        (dim - ray_orig[1]) / ray_dir[1],
        (dim - ray_orig[2]) / ray_dir[2]
    )

    t0 = vec3(t0_float[0], t0_float[1], t0_float[2])
    t1 = vec3(t1_float[0], t1_float[1], t1_float[2])

    return {
        'a': a,
        't0': t0,
```

```
        't0_float': t0_float,
        't1': t1,
        't1_float': t1_float
    }


def get_inv_proj(width, height):
    far = 100.0
    near = 0.001
    fov = 60
    aspect = float(width) / height

    f = far
    n = near
    t = math.tan(fov / 180 * math.pi / 2) * n
    b = -t
    r = t * aspect
    l = -r

    return np.array([
        [-f * (r - l) / (f - n), 0, 0, -f * (r + l) / (f - n)],
        [0, -f * (t - b) / (f - n), 0, -f * (t + b) / (f - n)],
        [0, 0, 0, 2 * f * n / (f - n)],
        [0, 0, 1, -(f + n) / (f - n)]
    ])


def get_inv_view(pos, rot):
    inv_view = np.array([
        [1, 0, 0, 0],
        [0, 1, 0, 0],
        [0, 0, 1, 0],
        [0, 0, 0, 1]
    ])

    rotation_y = np.array([
        [math.cos(-rot[1]), 0, math.sin(-rot[1]), 0],
        [0, 1, 0, 0],
        [-math.sin(-rot[1]), 0, math.cos(-rot[1]), 0],
        [0, 0, 0, 1]
    ])

    rotation_x = np.array([
        [1, 0, 0, 0],
        [0, math.cos(0.5* math.sin(rot[0])), -math.sin(0.5* math.sin(rot[0])), 0],
        [0, math.sin(0.5* math.sin(rot[0])), math.cos(0.5* math.sin(rot[0])), 0],
        [0, 0, 0, 1]
    ])

    rotation = rotation_x.dot(rotation_y)

    translation = np.array([[1, 0, 0, pos[0]],
                    [0, 1, 0, pos[1]],
                    [0, 0, 1, pos[2]],
                    [0, 0, 0, 1]])
```

```
    inv_view = np.transpose(rotation).dot(inv_view)
    inv_view = translation.dot(inv_view)
    return inv_view


def get_ray(x, y, width, height, pos, rot):
    v0 = np.array([x, y, -1, 1])
    v1 = np.array([x, y, 1, 1])

    inv_proj = get_inv_proj(width, height)
    inv_view = get_inv_view(pos, rot)

    q0 = inv_proj.dot(v0)
    q1 = inv_proj.dot(v1)

    q0 = q0 / q0[3]
    q1 = q1 / q1[3]

    q0 = inv_view.dot(q0)
    q1 = inv_view.dot(q1)

    ray_orig = [q0[0], q0[1], q0[2]]
    ray_dir = [q1[0] - q0[0], q1[1] - q0[1], q1[2] - q0[2]]

    return {'orig': ray_orig, 'dir': ray_dir}
```

Asbjørn Engmark Espe

Real-time Ray Tracing of Animated Sparse Voxel Octrees on FPGA

NTNU
Norwegian University of
Science and Technology