Problem Statement

Ray tracing is a method of generating images of 3D models. It involves tracing light rays in reverse, from a camera to light sources in the scene. Sparse Voxel Octrees (SVO) is a tree-based data structure used to store volumetric pixels. This structure can be used for 3D models and can be traversed efficiently, which makes it suitable for ray tracing. Recently there has been efforts to perform ray tracing of SVOs for the purpose of rendering real-time graphics. There exists pure software implementations and solutions based on general purpose graphics processing units.

Review the existing solution that implement this technique. Outline a design for a system that can accelerate this technique in dedicated hardware. Implement a unit that can traverse SVOs, a control unit which can schedule multiple traversal units and a cache and memory management that is optimized for the memory access pattern of the traversal units. Find and implement optimizations that can speed up rendering. Finally, explain how this system could be integrated with traditional graphics processing units in order to render images using both the proposed technique, and the traditional rendering technique. Discuss the possible challenges that will be encountered.

Abstract

Ray tracing of sparse voxel octrees is a method of rendering images of 3D models, which could soon become practical for use in real time applications. This is desirable as ray tracing can produce very realistic visualizations, while voxel models can represent models with very fine geometric detail. For these reason the method has attracted significant attention in recent years, but no hardware solution has been published yet. This thesis presents a design of ray tracing of sparse voxel octrees in hardware. The objective is to show if it is sensible to implement the method in hardware, and if it could be integrated on modern GPUs alongside rasterization. To this end, the techniques used in existing software implementations of this method is reviewed, and an algorithm suitable for hardware implementation is presented. The problems of integrating the method with rasterization is explored, and the algorithm is analyzed and optimized to improve efficiency in hardware. A software implementation is presented, which supports the development of a hardware design. This design is implemented using the Verilog hardware description language, and it has been simulated and synthesized for an FPGA prototype. Multiple versions of the design has been synthesized and tested, and to evaluate the impact of design parameters the test results from these designs is presented. The thesis provides a comprehensive evaluation of the proposed design, and the results indicate that the algorithm is well suited for hardware implementation. Although real-time performance was not achieved, there are indications that further optimizations should allow real-time performance on the same platform, and that a full scale implementation on a modern GPU could probably allow ray tracing with a quality which is competitive with rasterization.

Sammendrag

Ray tracing av sparse voxel octrees er en metode for å tegne bilder av 3D modeller, som snart kan bli praktisk for bruk i sanntidsapplikasjoner. Dette er ønskelig fordi ray tracing kan produsere veldig realistiske visualiseringer, mens voxel modeller kan representere modeller med veldig mye geometriske detaljer. Av disse årsakene har metoden fått betydelig oppmerksomhet de siste årene, men ingen hardwareløsning har vært publisert enda. Denne oppgaven presenterer et design for ray tracing av sparse voxel octrees. Hensikten er å vise om det er fornuftig å implementere metoden i hardware, og om den kan integreres ved siden av rasterisering på en GPU. Derfor blir teknikkene som har vært brukt i eksisterende softwareløsninger gjennomgått, og en algoritme som er passelig for å implementeres i hardware presenteres. Problemene med å integrere metoden med rasterisering har blitt undersøkt, og algoritmen analyseres og optimaliseres for å forbedre ytelsen i hardware. En software implementering presenteres, som underbygger utviklingen at et hardwaredesign. Dette designet implementeres i hardware språket Verilog, og har blitt simulert og syntetisert for en FPGA prototype. Flere versjoner av designet har blitt syntentisert og testet, og for å evaluere påvirkningen av designparametere blir disse resultatene presentert. Oppgaven gir en omfattende evaluering av det foreslåtte designet, og resultatene indikerer at algoritmen er egnet for implementering i hardware. Selv om sanntidsytelse ikke ble oppnådd, er det indikasjoner på at videre optimaliseringen vil kunne føre til sanntidsytelse på samme platform, og at en fullstending implementering på en moderne GPU sannsynligvis vil tillate ray tracing med kvalitet som er konkurransedyktig med rasterisering.

Preface

Computer graphics has always fascinated me, and I have found it highly motivating to work with. A couple of years ago I stumbled over an interview with the co-found and technical director of id Software, John Carmack. In it he commented, *"there is a very strong possibility, as we move towards next generation technologies, for a ray tracing architecture that [...] involves ray tracing into a sparse voxel octree"*. Ray tracing is a very enticing form of rendering, because it closely mimics the nature of light. It can be fun to work with because it is easy to get realistic results, but it is hard to make it fast. In April 2010 I e-mailed him asking whether implementing this on an FPGA would be a good idea. To my surprise he replied: *"I have actually thought specifically about this – much of the work would be very amenable to an FPGA implementation in a far more efficient manner than when implemented on general purpose hardware."*. This has served as a huge source of motivation while working on this thesis.

This project is a testament to the value and maturity of the open source hardware design community. I am confident that open source hardware will become increasingly influential in the coming years.

I am grateful to everyone who has contributed to the ORPSoC project, which has been an invaluable platform in this thesis. I am in particular very grateful for the work done by Stefan Kristiansson who ported ORPSoC to the Atlys prototype board I have been using in this thesis. Without it I would surely have spent a lot of time on work that is not directly relevant to the problems I wanted to explore. It is also amazing the extent of support I received through the members of the ORPSoC IRC chat channel.

Chris McClelland's "FPGALink" project was also an important tool in simplifying the development process. It enabled me to write software which could communicate directly with the FPGA over a USB link. He was also extremely helpful in ironing out issues related to the use of this software. Other tools which have contributed to this thesis are "binvox" by Patrick Min and the "vmath" vector library by Jan Bartipan

My supervisor has been Per Gunnar Kjeldsberg (Department of Electronics and Telecommunications, NTNU, Trondheim). He has given me some absolutely essential advice on being focused and making the right decisions with regards to the scope of the thesis. More importantly, he has given precious encouragement by showing genuine enthusiasm for the value of my work.

I would also like to thank my friends in Trondheim who made the time I've spent in this

city the best years of my life, and helped me maintain a healthy balance between work and play.

This thesis is dedicated to my grandparents, Arthur and Irene Saunes, who let me stay with them during the last stretch of the thesis. They gave me the perfect environment to concentrate on my work which gave me some very productive weeks while I was there.

Contents

Chapter 1

Introduction

Ray tracing of sparse voxel octrees (SVO) is a method for rendering images of three dimensional models. It has a wide range of potential applications; medical imaging[31], visualization of scientific data[23], video production[9, 6] and real-time graphics[29, 10, 56]. This thesis will outline a practical design for ray tracing of SVOs in hardware.

This method has had some attention in recent years, and there are currently several implementations of real-time ray tracing of SVO in software[29, 10, 42, 56]. There are also a few implementations of a somewhat similar technique, ray tracing of polygon models, in hardware[43, 64]. Although some algorithms for ray tracing of SVOs seem to be particularly suited for hardware implementation, there seems to have been no such attempt yet.

Making hardware optimized for a specific domain, such as medical imaging, can be prohibitively expensive. Technology is often developed and popularized within a consumer market before being adopted in another domain. Consider the massively parallel general purpose graphics processor units (GPGPUs). These are now being used to process e.g., scientific and medical data[22], and NVIDIA is producing versions of their GPGPUs specifically for this market[62, 34]. However, these processors grew out graphics cards for the consumer gaming industry.

Similarly, this thesis will present a hardware implementation which could conceivably be integrated in a graphics processing units (GPUs) for consumer games, with the understanding that this might be the most practical path for developing hardware that can be used within other domains as well. It should be noted that one can not expect this method to supplant the established method of rendering graphics for consumer games: rasterization. There is a large investment in rasterization in terms of toolchains, rendering engines and knowledge. For ray tracing of sparse voxel octrees to gain popularity within games, it must be implemented alongside rasterization, and it should be possible to use it within the same rendering pipeline. This way the method can initially be used for special effects, and over time gain more uses as the capabilities of the hardware grows and the mind share of the method increases. Consequently, a practical architecture for raytracing of SVOs should be as compatible with rasterization as possible.

Using sparse voxel octrees to represent 3D models is a logical evolution of virtual texturing techniques which have been developed recenty[57, 19, 18]. They can allow full freedom to define the shape and texture of the models, in addition to being a potentially more efficient way to store the color and geometry data[47]. More importantly, these structures can be ray traced efficiently[29, 10]. Although ray tracing is generally slower than rasterization, ray tracing is a much more accurate model of how light behaves, and can enable effects like accurate reflections, detailed shadows and ambient occlusion[9]. There is also a benefit to developers as expression visual effects with rays of light is more natural and easy to deal with than the tricks that rasterization employ[48]. As hardware becomes faster, as demand for more realistic graphics grows, and as algorithms for ray tracing become more developed, there may soon come a time when using ray tracing for real-time computer games is feasible. This is effectively demonstrated by Schmittler and Pohls work of converting existing games to use ray tracing[44].

Field-programmable gate arrays (FPGAs) allow designs for digital integrated circuit hardware to be prototyped and tested at a price which is several orders of magnitude cheaper than making application-specific integrated circuits (ASICs). This thesis uses an FPGA prototype platform with the intention of presenting a design that could later be integrated on an ASIC. Although there are additional challenges to putting a design on an ASIC, an FPGA prototype should solve many of the initial problems.

In this thesis an algorithm for ray tracing of SVOs has been chosen and analyzed. A software implementation has been made to produce reference renderings and behavioral simulations, which provide data relevant to a hardware implementation. The software has also been used to evaluate techniques to combine rasterization and ray tracing based rendering. A hardware module which implements the algorithm has been described in the hardware description language Verilog. This module has been simulated, integrated in a system-on-chip solution for an FPGA, and several variations of the design has been synthesized, tested and benchmarked. This has provided useful data on the impact of various design parameters.

Chapter 2 presents the background material for the thesis. Topics related to rasterization, ray tracing, sparse voxel octrees, caching techniques and FPGAs are covered. Chapter 3 presents previous work which is related to the thesis, and these have been categorized into work related to software-based ray tracing of SVOs, other forms of ray tracing in hardware and work on sparse voxel octree data structures. Chapter 4 presents and explains an existing algorithm for ray tracing of SVOs, which serves as a good basis for a hardware implementation. Chapter 5 discusses some challenges related to integrating ray tracing and rasterization and presents some solutions which should lead to an architecture that makes it easier to combine the two techniques. Chapter 6 presents analysis of aspects of the algorithm which is significant to a hardware implementation, and modifications to the algorithm that make it feasible to implement it in hardware. Chapter 7 describes the software implementation of the hardware optimized algorithm and discusses the results this produced. Chapter 8 describes a hardware implementation, the methodology for testing the resulting design and discusses the results from these tests. Chapter 9 discusses the conclusions drawn from this thesis and future work necesssary to reach a comprehensive and practical design for ray tracing of SVOs in hardware.

Chapter 2

Background

2.1 Models

In 3D computer graphics, we usually want to visualize real world objects. These objects must necessarily be represented in the computer by some kind of *model*. We use a data structure or a mathematical model to represent these objects. A common way to model objects is using *polygon models*. These are constructed from sheets of polygons. The polygons themselves are usually subdvided into triangles, which can be represented by three points. A normal vector is often also included, which is useful to indicate which side of the triangle is facing outwards, and how light should behave on it.

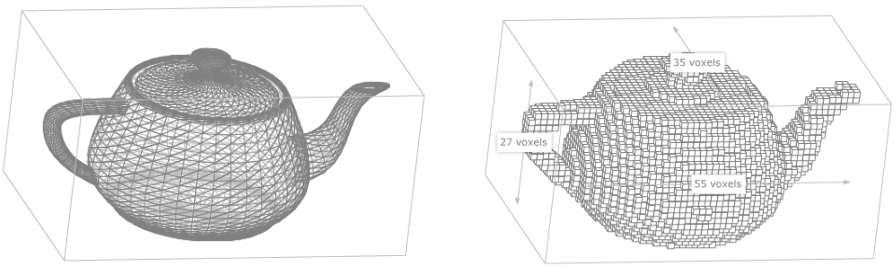


Figure 2.1: A triangle based (left) and a voxel based (right) model.

Another way to represent models is through voxels. Voxels are to 3D models what pixels are to 2D images. You represent the space as a uniform collection of points, and you store parameters for each of these points. The parameters can be a color value and normal vector of the voxel. Voxels can be drawn as points, circles or more commonly cubes. Although the cubes result in aliased hard edges, these can be softened through adaptive blurring techniques[29].

Given a type of model (polygons or voxels), there can be many different ways to render that model to a 2D image. Rendering algorithms are usually optimized for one type of

model. Rasterization is optimized for polygon models, and the algorithm presented in this thesis is optimized for voxel models.

2.2 Transforms

Transforms are essential tools in computer graphics[3]. It takes an entity and moves, rotates, scales or performs any other kind of geometrical conversion. They are useful for positioning objects in the scene, moving cameras and lights, etc. The most useful transforms are linear transforms. These can be represented as a matrix. In computer graphics points, vectors and colors are usually represented as a 4-component vector. For points and vectors these represent the x, y and z-axis, with the fourth component (w) usually being set to 1. The w-component is useful if a transform needs to add a constant to a component. Any combination of linear transforms can then be applied using a 4x4 matrix.

Model transform Models are defined within their own coordinate systems. But to model a scene with several different models in it, the models must be positioned in a common coordinate system, the *world space*. A model of a teapot could be centered in the origo of its coordinate system, and the nose could point along the z-direction. The model transform for the tea-pot could then for instance rotate it around the y-axis to point at an angle θ and position it at the coordinate $(10, 0, 20)$. The matrix for this example is illustrated below, with $\mathbf{p}$ and $\mathbf{q}$ being a vector before and after transformation.

$$M = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 10 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 20 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

$$\mathbf{q} = M\mathbf{p} \tag{2.2}$$

View transform The coordinate system of the world space is usually defined in an intuitively useful manner. The ground could for instance be defined to be parallel to the X-Z plane. A virtual camera could be positioned freely in this space. For rendering algorithms however, it is useful to have the camera positioned in the origo of the coordinate system with the camera looking along the z-axis. The *view transform* transforms the scene from the world space to the *camera space* (or *eye space*). The model and view transform can be combined in a single *model-view matrix*.

Projection transform A projection transform maps the visible parts of the scene to some *canonical view volume*, a simple axis aligned cube with corners at e.g., $(-1, -1, -1)$ and $(1, 1, 1)$. The nearest visible objects are mapped to the *near plane* ($z = -1$). The furthest visible objects are mapped to the *far plane* ($z = 1$). Similarly there is a top, bottom, left and right plane[3]. Anything outside these planes is outside the view of the camera, and should be ignored.

2.3 Perspective Projection

A perspective projection is a projection which closely match how we perceive the world: objects gets smaller as they get further away, and larger when they come closer. Consequently in scene space, the *far plane* is larger than the *near plane*, while in the canonical view volume they are equally big. This projection is represented by the following transform matrix[3] (for the OpenGL API), where position of the near, far, left, right, bottom and top planes are denoted respectively n, f, l, r, b and t:

$$\mathbf{P}_p = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \tag{2.3}$$

After this matrix is applied, the resulting vector must be divided by its w-component to complete the perspective transform. The w-component has in other words been used to temporarily store a value which is a linear function of the other components, and all components will be divided by this value. I.e., to get from a view space point $\mathbf{q}$ to a canonical view space point $\mathbf{v}$:

$$\begin{aligned} \mathbf{v}' &= \mathbf{P}_p \mathbf{q} \\ \mathbf{v} &= \mathbf{v}'/\mathbf{v}'_w \end{aligned} \tag{2.4}$$

2.4 Rasterization

The most popular form of 3D rendering today is rasterization of polygon models, or just *rasterization* for short[59]. It is the technique used by virtually all commercial GPUs. The process of rasterization can be divided into a pipeline. The three conceptual stages of the pipeline is the *application, geometry and rasterizer stages*[3]. The application stage is defined by the developer and runs on the CPU, it runs simulations and provides the models, textures, data and other parameters for the following stages in the pipeline. The task of the geometry stage is to apply transforms and effects that work on the geometry of the models, and eventually ending up with triangles mapped to a surface that represents the screen. The triangles are passed on to the rasterizer stage. This stage iterates through all the given triangles, and generates *fragments* for every pixel the triangle covers. These fragments can be further processed by a *pixel shader*, a small program that applies per-pixel visual effects. Finally the fragments for each pixel is merged, producing a single color-value for that pixel.

Figure 2.3 illustrates in further detail how the geometry stage works. The model transforms positions different models in a common coordinate system. The view transform positions the camera in the origo, and points it along the z-axis. The perspective transform maps the part of the world which is visible to the camera to a canonical view volume. Anything that is outside this volume is discarded. The geometry is then passed on to the rasterizer which is responsible for drawing each triangle.

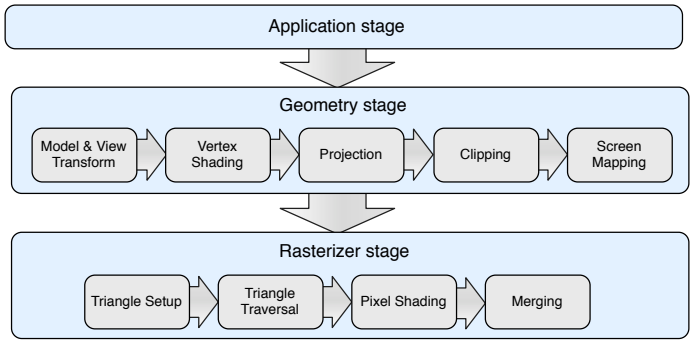


Figure 2.2: A typical GPU pipeline[3]

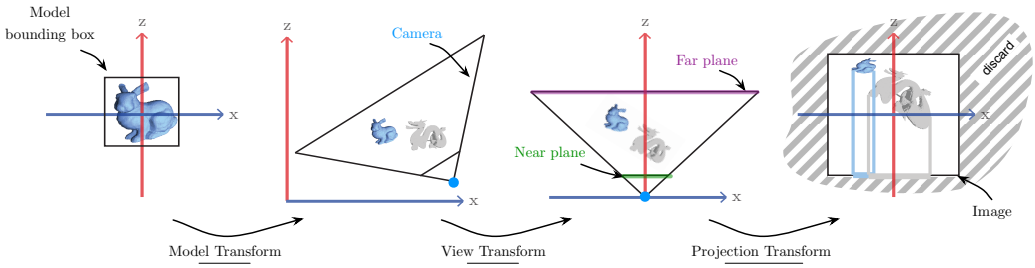


Figure 2.3: The geometry stage in action

2.5 Z-buffer

When drawing a fragment, it must be possible to determine whether the fragment should be drawn in front of or behind those that were drawn before. This is usually accomplished using a *Z-buffer* (or *depth buffer*)[3]. When drawing a fragment of a triangle to a pixel in the image, the z-coordinate (in the canonical view volume) of the fragment is stored in the Z-buffer. If the z-coordinate of the fragment is larger than the one in the buffer, we know that the fragment is behind those that were drawn before, and should not be drawn to the image.

One thing which is important to note about the Z-buffer, is that when using a perspective transform, the value stored in the buffer is not proportional to the distance of the camera to the fragment. The z-coordinate in the canonical view volume is derived from Equation 2.4. It can be shown that this implies that the z-coordinate in the canonical view volume is a function of $1/z_c$, where z_c is the z-coordinate in the camera space.

2.6 Ray tracing

Ray tracing is a method which generates 3D images by tracing the path of a ray of light in reverse. A virtual camera is defined, with a location and direction. From this camera, rays are traced in to a scene (a collection of objects we want to render). These rays

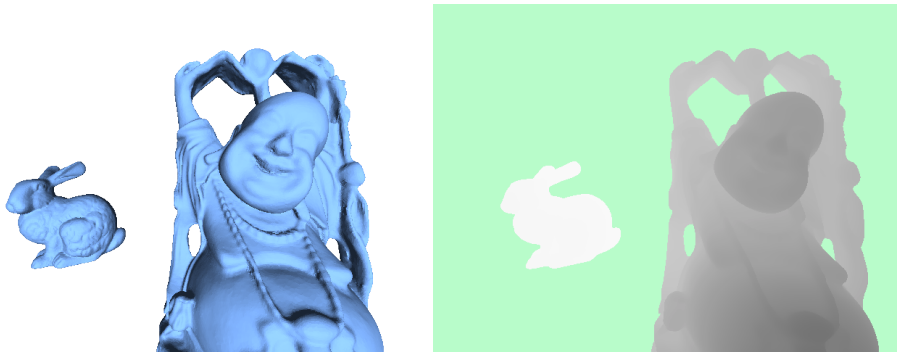


Figure 2.4: An image of the Stanford bunny and a buddha statue, and its Z-buffer. Whiter intensity implies further distance from camera. The background has been colored green to allow the bunny to stand out.

represent the pixels on the resulting image. Ideally, a pixel represents a volume in the space as seen in Figure 2.5. Accurately rendering this would be too expensive, but the pixel can be approximated by a ray instead. This will result in anti-aliasing issues, so it is common to average the colors of several rays to find the color of a pixel[14].

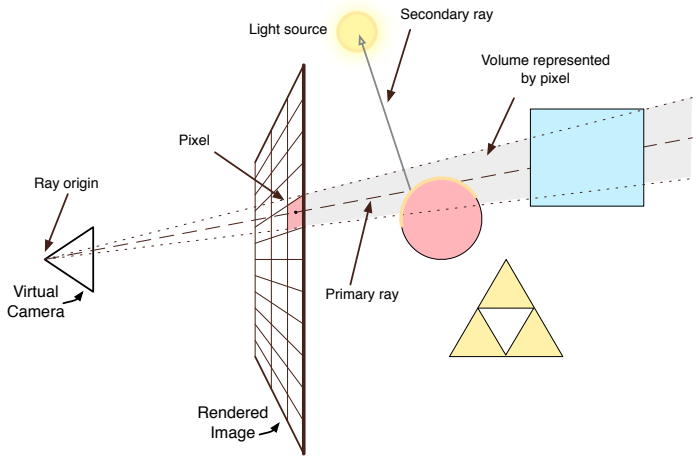


Figure 2.5: Ray tracing

The rays going from the camera to the objects in the scene are called *primary rays*. After a primary ray hits an object, additional rays can be sent out. These can be used to find whether the point is in a shadow or not, by tracing a ray to the light sources in the scene. They can also be used to calculate reflections and refractions on the object, allowing us to draw accurate transparent objects such as glass or water. These rays are often called secondary rays.

Any geometrical construct for which a ray-object intersection test can be formulated, can be rendered using ray tracing. The ray can fundamentally be described using Equa-

tion 2.5. The ray has an origin $\mathbf{o}$ and a direction vector $\mathbf{d}$. t can be considered a virtual time parameter, and $\mathbf{p}$ a position along the ray at different times. In these terms ray tracing is the problem of finding the values t where the ray intersect objects.

$$\mathbf{p}(t) = \mathbf{o} + t \cdot \mathbf{d} \tag{2.5}$$

This equation can for instance be solved for the equation of a perfect sphere, which is commonly used to demonstrate simple ray tracers. Equation 2.5 can also be solved for a triangle[14], making it possible to ray trace polygon models. A naive implementation of this would do an intersection test on every triangle in the model for every pixel in the screen, making the algorithm very slow. To optimize ray tracing, a space partitioning data structure is often used. Instead of testing every primitive, one performs a search of this data structure instead to find a set of primitives the ray could intersect. Therefore, ray tracing is cited as having the advantage over rasterization in that performance depends logarithmically, not linearly, on geometric complexity[25]. Although it should be mentioned that level-of-detail (LOD) based techniques to achieve similar performance scaling exists for rasterization[20].

2.7 Space Partitioning

Space partitioning is a method where a given space is divided into smaller non-overlapping spaces. These sub-spaces can be arranged in a hierarchical tree, where each sub-space can be divided into further partitions. This results in a nonuniform spatial subdivision[14]. Space partitioning data structures is often used in computer games to accelerate certain operations. In rasterization, it can used to exclude regions of polygons that will not be rendered (occlusing culling[7]), or to sort objects by depth, which is important for certain visual effects[3]. For these purposes, the BSP/kd-tree structures have been found to be better than octrees[55]. Although kd-trees have been used for ray tracing voxel data[52], sparse voxel octrees seem particularly suited for this purpose because they can be traced by a remarkably simple algorithm which will be presented in a later chapter.

2.8 Sparse Voxel Octree

An octree is a hierarchical space-partitioning data structure[41] where a space is divided in eight equal sub-regions, or *octants*, and where each octant that contains objects is further divided until the resulting tree reaches a desired height. A voxel is a volumetric pixel, in other words, an axis-aligned rectangular prism[14]. A sparse voxel octree is a data structure used to store a voxel model when most of the voxels are empty (the space is "sparse"). In visualization of real-world objects, this would be the case, as all the voxels exists on the surface of objects.

Often, when using a space-partitioning data structure, a list of objects is stored in the leaf nodes. In a sparse voxel octree the leaf node is a voxel, either an empty space or a solid cube. A solid voxel may have a color value. For an inner node, we can take the

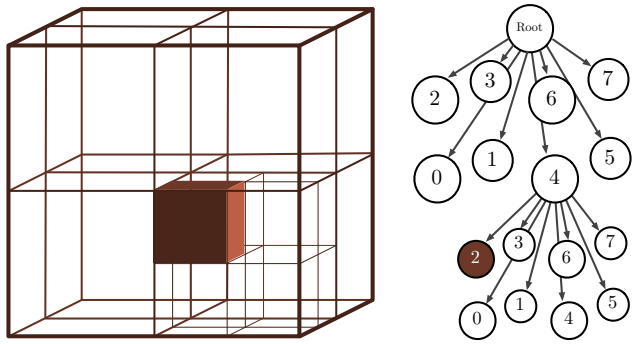


Figure 2.6: An octree with 2 levels and a single solid voxel

average color of all the child nodes, and attach it to this node. Doing so means the scene represented by the tree can be rendered at varying resolutions, depending on how deep we traverse. This is desirable because it is a waste to render objects that are far away with high detail. Furthermore, we can even avoid loading into memory those parts of the tree that represent high detail in far away parts of the scene, and stream them into memory as the camera moves towards those areas[10].

2.9 Traversal of Voxel Octrees

A variety of algorithms to traverse space partitions have been described in the literature. They have different complexities, and they can have very different performance on a given computer architecture. A significant attribute of the algorithms are whether they use a stack or not. A stack is useful for keeping track of the path taken as the tree is traversed, but can be detrimental on certain architectures. The following are descriptions of a few of the most common methods.

Restart The simplest methods are those based on the kd-restart algorithm. This is a stack-less algorithm: A starting point, the ray origin, is chosen. The tree is traversed until the voxel representing this point is found. If the voxel represented by the node is empty, the point is moved to where the ray exits the voxel. The algorithm then restarts from the new point. In other words, the algorithm traverses one octant at a time, and has to descend the tree from the root node down every iteration. This is repeated until a collision is found, or the ray exits the voxel represented by the root node.

Backtrack Backtracking algorithms (or *push-down optimization* [21]) are similar to kd-restart. But it exploits the fact that we can keep track of the last node which must be parent to all the nodes traversed by the ray. It is a simple optimization than can provide a performance boost. But the parent node might be close to the root, which can make the gain minimal in some iterations. It only eliminates traversal steps near the top of the tree, which are the ones that are most shared by neighboring primary rays [21].

Full stack A tree traversal algorithm can use a stack to keep track of the nodes visited as the algorithm descends the tree. The algorithms utilizing a stack is similar to the restarting algorithms when descending the tree, but will push the traversal state to a stack for each node. If an empty space is found, the stack is unwounded until it finds the next node the ray intersects. The descent can continue as usual from that point. In this thesis the algorithms using a stack as deep as the tree the algorithm was designed for is referred to as *full stack* algorithms, to distinguish them from *short stack* algorithms.

Short stack Sometimes a full stack can be too expensive to implement. In GPGPU implementations, the processing cores are often not optimized for the memory access pattern required by the stack. An alternative is to keep the state for the last N_s nodes traversed. If the stack is exhausted (a stack underflow), the algorithm can perform a restart.

The restarting algorithms can be considered a special case of a short stack with $N_s = 0$. Similarly, a full stack can be considered a special case with $N_s = N_t$ where N_t is the depth of the octree. Even if an algorithm is designed with a full stack, it could be wise to support restarts, as it will allow the algorithm to support octrees deeper than it was originally designed for.

Bottom up methods Revelles et al. [41] describes bottom-up methods as those starting at the first terminal node intersected by the ray. The algorithm must then find the neighboring nodes by some method. Some algorithms use neighbor pointers in the octree[15], which are pointers stored in the leaf nodes, pointing to the neighboring nodes at each face of its voxel. This enables the algorithm to jump between nodes without ascending the tree. It can improve traversal speed, but at a significant memory cost, as the additional pointers have to be stored in the octree.

2.10 Representation of Numbers

There are several ways to represent numbers. For performing calculations, the two most popular schemes are probably fixed point and floating point representations. They both have strength and weaknesses.

Fixed point numbers The *fixed point* format is named so because the decimal point is fixed in one place. Equation 2.6 shows the number 5.625 represented in binary fixed-point, with three bits for the integer and fractional part, or a total of six bits.

$$\begin{gathered} 101.101 \\ 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} \end{gathered} \tag{2.6}$$

The position of the decimal point must be chosen carefully. It will determine what range of numbers can be represented, and how accurately they can be represented. Given N

number of integer bits, and a two's complement representation, the range of integers is -2^{N-1} to $2^{N-1}-1$. Given M number of fractional bits, the smallest fraction that can be represented is 2^{-M}.

Fixed point numbers are relatively cheap to implement in hardware. Table 2.1 shows the resource use of a 32 bit adder/subtract unit and a 32 bit multiplier unit, as estimated by the Xilinx Core Generator tool[67, 65].

FPGA	Operation	LUTs	FF	Latency	DSPs	Max Clock Freq.
Virtex-5	Add/Subtract	70	91	3	0	410Mhz
Virtex-5	Add/Subtract	32	32	0	0	388Mhz
Spartan-6	Multiplier(35x35)	58	107	8	4	250Mhz

Table 2.1: Resource use of fixed point operations[67, 65]

Floating point numbers Floating point numbers are similar to scientific notation; it represents numbers of the form $m \cdot 2^{e-b}$. A standard for floating point representation called IEEE 754[1] is now used almost exclusively in digital computing platforms[24]. The numbers are stored as a *mantissa*, m, and an *exponent*, e. The exponent is offset by a constant bias, b. A sign bit, S, indicates whether the number is positive or negative. Floating point numbers provide a much larger dynamic range than fixed point numbers given the same amount of bits. That is, it is possible to represent both very large and very small numbers in the same representational system. This has several benefits over fixed point numbers, one being that the programmer or hardware designer does not have to worry as much about overflowing the boundaries of the representation. Floating point numbers are essential in modern GPUs, and a modern high-end GPU can perform over 1000 billion floating point operations per second[60]. Table 2.2 shows the estimated resource use for single precision (32 bit) floating point numbers[66]. Comparing to Table 2.1, we see that floating point numbers are significantly more expensive to implement.

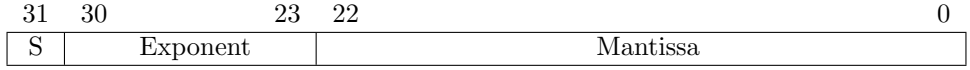


Figure 2.7: Bit organization of single precision (32 bit) IEEE 754 floating point representation[1].

FPGA	Operation	LUTs	FF	Latency	DSPs	Max Clock Freq.
Virtex-5	Add/Subtract	432	558	12	0	420Mhz
Virtex-5	Add/Subtract	388	76	2	0	110Mhz
Spartan-6	Multiplier	140	187	8	0	150Mhz

Table 2.2: Resource use of floating point operations[66]

2.11 Data Cache

Processors can perform operations on its internal registers significantly faster than it can access its main memory. For modern systems the cost of accessing main memory can be about 120 times as much as accessing a register [39]. It has long been known that using a cache can improve the practical cost of accessing main memory. A cache is a fast memory which can hold some of the content of the main memory, and which is inserted between the processor and main memory, sometimes with multiple levels of increasingly faster and smaller caches.

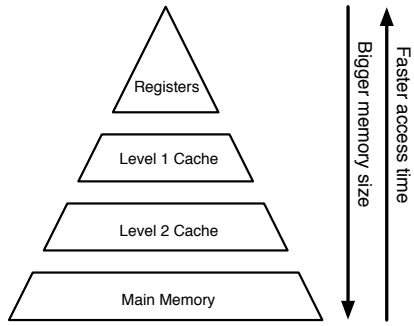


Figure 2.8: A cache hierarchy

A cache exploits two features inherent in the way many algorithms access memory: *temporal locality* and *spatial locality*. Temporal locality refers to cases where the algorithm accesses the the same content several times in a small amount of time. Spatial locality is where the algorithm accesses memory that is located nearby the last address it accessed. When the processor attempts to access main memory, the cache will be checked to see if it contains the relevant word. If it did (a *cache hit*) the cache will return the data. If it did not (a *cache miss*) the cache will fetch the data from main memory, usually in a block (or "line") of several words at a time. The ratio of cache misses over the number of accesses is called the *miss rate*, and can be used as a simple indicator of the performance of the cache [17].

The cache can be considered to have a set of slots in which the blocks can be inserted. Each slot will have an associated *tag* which indicates the address of the block which is stored there. There are different ways of determining the position of a block in the cache [39]:

Direct mapped : Each block is mapped to a single slot. This implies that when a new block is fetched, the old block occupying its slot *must* be evicted.

N-way set associative : Each block is mapped to one of N slots. When fetching a new block a replacement policy determines which of the old slots are chosen. A common policy is *least recently used* (LRU). The policy tracks the usage of blocks and the block that was used least recently will be evicted in favor of the new one.

Fully associative : A block can be inserted into any slot. This implies that we must compare the tags of all the slots in the cache to determine if a word is present or not.

These approaches have various pros and cons. A direct mapped cache is cheap to implement, but will have a lower hit rate. An N-way set associative cache requires more logic, and higher N implies a higher cost. The cost is also dependent on the replacement policy. LRU is cheap to implement in a 2-way set associative cache, requiring only a single bit to track which block was used last. For 4 and 8-way set associative cache, using a LRU policy can become prohibitively expensive and some implementations opt for a *pseudo-LRU* policy [53, 13].

2.12 FPGA

A field-programmable gate array (FPGA) is a type of integrated circuit which can be programmed (or configured) to perform the function of almost any kind of digital circuit or system[28]. The FPGAs is an array of logic blocks, memory and a routing fabric which connects these together. The logic blocks, or look-up tables (LUTs), can be configured to perform arbitrary combinatorial functions on a number of inputs. FPGAs can also contain some specialized blocks. The FPGA used in this thesis contains for example digital signal processing and memory controller blocks[68]. These specialized blocks implement commonly used functions and using these is more efficient than implementing the same functions in the general logic blocks.

Although FPGAs are use more area, consume more power and are slower than application-specific integrated circuits (ASICs), they are still very useful due to significantly smaller investment costs in terms of both time and money[28]. It is therefore beneficial for projects with low volume productions or short development time, or for prototyping and testing.

2.13 ORPSoC

OpenRISC Reference Platform SoC ("ORPSoC") is a project aimed at creating an open-source platform for OpenRISC development. The current version of ORPSoC uses the OpenRISC processor *OR1200* and integrates it in a system with several other modules and interfaces. The system includes modules for JTAG, Ethernet MAC, on-chip RAM/ROM, SPI, UART, I2C and USB. Every module in the system is connected through the Wishbone bus. There are derivate configurations available for a few different FPGA boards, both based on Actel and on Xilinx.

Wishbone[35] is an open-source hardware bus standard, and specifies the interface between components in an ASIC or FPGA design. It is an effort by the OpenCores community to develop a common interface between open-source hardware components, and it's used by many of the modules available on the OpenCores website. The standard only specifies the digital signals and their protocol. Analog effects or bus topology is intentionally left ambiguous.

Data is transferred during an interval called a *bus cycle*. During a bus cycle the master will

request a read or write operation on a slave and the relevant data will be transferred. Of particular interest in this thesis is *burst cycles*, which is an optional feature of Wishbone. In a 32-bit Wishbone bus, a single 32-bit word is the most that can be transferred in a single clock cycle. By using burst cycles the master can negotiate the transfer of multiple consecutive words during a single bus cycle. This provides two benefits: it avoids having to set up a new bus cycle for each word and it allows the slave to know up front the next address that should be transferred. This can potentially allow it to service the following requests with less delay. When negotiating a burst cycle, the master provides the first address that should be transferred and indicates how the next addresses should be calculated. There are four ways the addresses can be incremented: *linear*, *wrap-4*, *wrap-8* and *wrap-16*. With the linear mode the address is incremented linearly for each transfer, while with wrap-4, wrap-8 and wrap-16, the address is looped around a 4, 8 or 16 word block boundary. This feature is useful for reading cache lines for the cache of a processor. The word that the processor requested can be fetched first and sent directly to the processor, allowing it to continue executing, while the rest of the words in the cache line can be transferred subsequently.

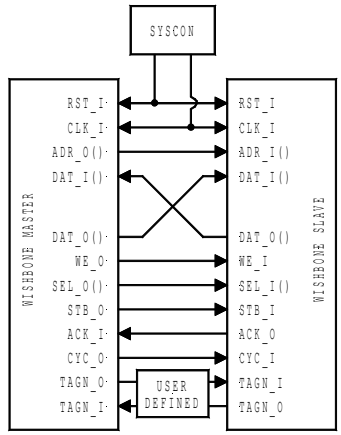


Figure 2.9: Wishbone signal lines[35]

OpenRISC is an effort to develop a series of general purpose, open-source RISC CPU designs. There is currently one OpenRISC architecture, called OpenRISC 1000. The first implementation is OpenRISC 1200 (*"OR1200"*). The *OR1200* is a 32-bit scalar RISC with Harvard micro-architecture (separate data and instruction interface), 5 stage integer pipeline, IEEE 754 compliant single precision FPU, virtual memory support (MMU) and basic DSP capabilities[36]. It includes an 8KB data cache and an 8KB instruction cache by default. The data and instruction interfaces is implemented as a Wishbone interface.

2.14 Digilent Atlys

Atlys is an FPGA prototype board made by Digilent. It uses a Spartan-6 XC6SLX45 FPGA chip. The FPGA chip has 43,661 logic cells (27,288 6-input LUTs), 54,576 flip-

flops and 116 18Kbit Block RAM blocks (total 2,088Kbit)[68]. The Atlys prototype board includes 128Mbyte DDR2 RAM, an Ethernet port, two HDMI inputs, two HDMI outputs, an AC-97 codec, a 16Mbyte SPI Flash, USB Keyboard/Mouse input, 8 LEDs, 8 switches, 5 push buttons, a 100MHz clock and two expansion ports[11]. The FPGA is programmed through a Cypress CY7C68013A-56 USB microcontroller. This microcontroller is has communication lines connected to the FPGA which can be used to communicate with the configured FPGA from a personal computer.

Chapter 3

Previous Work

No hardware implementations of ray tracing of SVOs was found in the literature. The work this project builds on can be divided in four categories: implementations of ray tracing of SVOs in software, implementations in GPUs, implementations of other forms of ray tracing in hardware and work related to data structures for SVOs.

3.1 Software Implementations

These works are all in software on CPUs, which is typical for papers from the 90s, as GPGPUs were not available yet. The first descriptions found of octree traversal was as a volume rendering technique by Levoy [31]. It illustrates that introducing the hierarchical octree structure improves voxel traversal speed. The next works are the *SMART* algorithm by Spackman and Willis [49] and an algorithm by Stolte and Caubet [51]. They are both based on a 3D DDA (Digital Differential Analyzer) as described by Sung [54], which works by stepping through the space by adding a delta value to a vector. Revelles et al. [41] presents an algorithm based on keeping track ray parameters (t from Equation 2.5) for the voxels during traversal. It appears to be the first work that constructs an algorithm which is specifically optimized for traversing octrees. The algorithm is compared to three previous bottom-up algorithms, and one top-down algorithm, and it was found to have improved performance over existing algorithms. A paper by Whang et al. [58] was also reviewed. It presents a variation of octrees where the middle planes could be shifted. This made it possible to balance the tree depending on the content of the scene. However, it was not intended for rendering voxels, as the resulting voxels would not be equilateral.

3.2 GPU Implementations

The arrival of GPGPUs allowed new rendering techniques to be implemented. Gobbetti et al. [15] implemented one of the first techniques for ray casting octrees on the GPU.

Their algorithm is based on an extension of an efficient ray traversal algorithm for kd-trees. Crassin et al. [10] presents a full ray tracing architecture for GPUs, which is capable of streaming parts of the octree to the GPU on demand. Their work is similar to that of Gobbetti et al. [15], but provides better quality and performance. Their algorithm is based on *kd-restart*, and avoids the use of a stack which is potentially inefficient on a GPU. The solution is geared towards rendering large volumetric data sets, e.g., medical 3D images.

Laine and Karras [29] created an implementation that utilizes a stack, and the algorithm appears nearly identical to that of Revelles et al. [41]. It is not clear why they chose to use a stack, but considering it was research by NVIDIA, who makes GPGPUs, it is probably an informed choice. Another difference from Crassin et al. [10] is that the implementation is made more suitable for rendering surfaces than volumetric data, which they claim is more relevant for most real-world content.

A master thesis by Römisch [42] presented three implementations: one using a bottom-up method utilizing neighbor pointers, the second using a kd-restart based method, and the third using a short stack. Neighbor pointers was found to be fastest, but at the cost of some flexibility. Between kd-restart and short stack, kd-restart was fastest. It's hypothesized that it's because of the limited performance of GPU processors when complex control-flow statements are used.

Römisch [42] and Laine and Karras [29] both explored short stacks on GPGPUs but came to differing conclusions. Römisch and Møller-Nielsen found that using a stack, even a short one, resulted in worse performance. While Laine found that a full stack was better than a short stack. This could possibly be due to different GPGPU architectures, or that the specifics of the implementations are different. In a hardware implementation, this issue is also particularly interesting, as the stack will require registers that consume area. This area could be used for other logic that could improve performance. Another important consideration is memory bandwidth. A short stack or restart algorithm will require more node lookups, causing more traffic on the data bus.

3.3 Ray Tracing in Hardware

The only hardware designed for ray tracing found in the literature are two prototypes by Woop and Schmittler from the computer graphics laboratory at Saarland University[43, 64]. They render polygon data, and use a kd-tree as an space partition structure to accelerate ray tracing. The first prototype is a fixed function ray tracer called SaarCOR. The second is a programmable ray processing unit (RPU), which allows for higher quality in the rendered image at the cost of area or performance. Presumably the kd-tree traversal unit described in these papers bears some similarity to the octree traversal unit in this thesis. Unfortunately the papers do not go into detail regarding the traversal unit, except to mention that it uses a stack and requires 4 adders, 4 dividers, 13 comparators and 44.5KB of internal memory.

3.4 Data Structures

There are many ways to encode an octree. The main considerations is to compress the size of the encoded octree as much as possible without making decoding of the structure too expensive. A naive implementation would have 8 child pointers for each inner node, which would either point to another innor node, or leaf data. One optimization to this scheme is to store data about child nodes within the inner nodes as a set of flags[6, 29]. Each node can be either an empty octant, an inner node, or a solid voxel. Using 2 bits to represent these three states for each child results in 16 bits of data. Instead of having a pointer to each child node, a single pointer can point to an array of node data. The size of this pointer depends on the potential size of the octree file. Laine and Karras [29] showed that if we use a relative pointer and organize the data in a depth-first order, 15 bits is enough for most child pointers. For the few cases where the pointer falls outside the 15-bit range, an additional bit indicates that the pointer is a *far pointer*, which points to a 32-bit pointer instead. The contour pointer and page header/info section shown in Figure 3.1 is used to refine the shape of the voxels and attach additional data to them.

Octree-based geometry compression is introduced by Botsch et al. [8] and Peng et al. [38]. Schnabel and Klein [45] builds on this work and presents a method for compression of point-sampled models. The general idea is that it is possible to predict the configuration of a child node as you descend the tree, and that this can be exploited by storing this configuration using variable-length encoding. These techniques obviously add complexity to the encoding and decoding of the octree, but the size of the encoded octree is smaller.

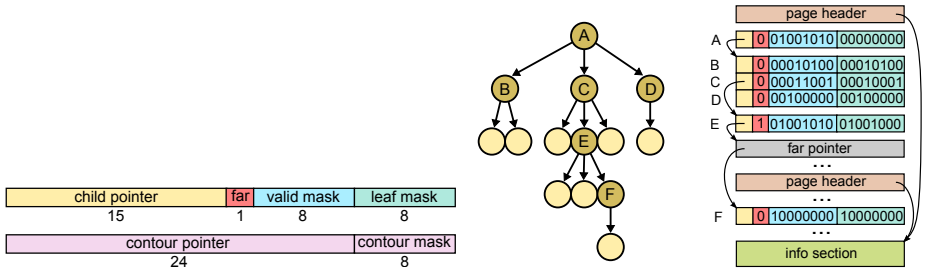


Figure 3.1: Data structure organization from Laine and Karras [29].

Chapter 4

An Algorithm for SVO Traversal

Out of the various algorithms reviewed, the one described by Revelles et al. [41] appeared to be most suited for this thesis. It was written with the intention of CPU implementation, and it is a recursive top-down algorithm using a stack. However, it can easily be modified to support restarting and short stacks. The algorithm is thoroughly described, which makes it easy to implement. Furthermore, the operations performed in the kernel of the algorithm is well suited for implementation using fixed point numbers; additions, comparisons and simple bit manipulation. And as a very similar algorithm was used by Laine and Karras [29], it appears the algorithm is still competitive. However, the addition of contour data that they presented was not used in this thesis, as it would add too much complexity to the hardware implementation. The algorithm described by Gobbetti et al. [15], uses several multiplications which would increase the cost of a hardware implementation, and it is uncertain if the algorithm is faster. Bottom-up methods were also discarded. Although they were found to be a little faster than restart based and short stack algorithms in one GPGPU implementation[42], the stack in this implementation made the algorithm slower due to the architecture of the GPGPU. It is expected that a stack will have a large performance benefit in a hardware implementation, without the drawbacks of larger memory consumption and less flexibility that the bottom-up method had.

This chapter describes the algorithm presented by Revelles et al. [41], with some minor modifications and differences in notation. Finally, an illustration has been created that demonstrates in detail how the algorithm works.

4.1 Overview

The algorithm takes each face of the octants in the tree, and extends them to planes. It then finds the t-value at which the ray intersects these planes, as per Equation 2.5. First

it calculates these values for the root voxel. The ray could in some applications start outside the root, meaning the actual octree we want to traverse might just be part of the scene being rendered. We must therefore check that the ray hits the root voxel at all. If it does, the algorithm enters the recursive part. It takes the t-values of the voxel, and uses these to determine which child node the ray enters. If the child node is not a leaf, it calculates the t-values for the child, and recurses on it. It moves through each child the ray intersects, until it exits the current voxel, at which point the function returns.

For now we assume that the ray traverses in a positive direction along each axis. I.e., the components of $\mathbf{d}$ are all positive. According to Revelles et al. [41] this assumption makes it more efficient to describe and implement the algorithm, while it is easy to modify it to support negative directions.

4.2 Parameters

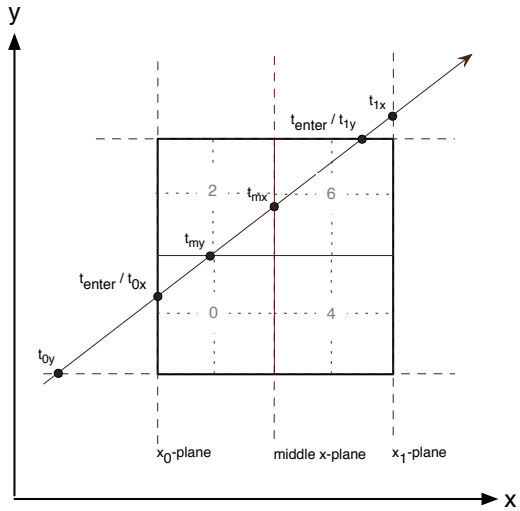


Figure 4.1: Parameters in the octree traversal algorithm[41].

The octree consists of axis-aligned cubes. Each face of the cube is extended to a plane. Unless the ray is running parallell to this plane, there will be a time, t, at which the ray intersects the plane. The *entry planes* are the ones where the ray potentially enters the octant. The *exit planes* are the ones where the ray exits the octant. The fundamental parameters of the algorithm are the t-values for the three entry planes, $\mathbf{t_0}$ or t_{0x}, t_{0y}, t_{0z}, and for the three exit planes $\mathbf{t_1}$ or t_{1x}, t_{1y}, t_{1z}. At every step in the algorithm, it must keep track of $\mathbf{t_0}$ and $\mathbf{t_1}$ for the current octant.

Assuming that the components of the direction vector are positive, the t-values can be found using Equation 4.1, where x_0 and x_1 are the x-coordinate of the two faces of the

octant that are normal to the x-axis.

$$\begin{aligned} t_{0x} &= (min(x_0, x_1) - o_x)/d_x \\ t_{1x} &= (max(x_0, x_1) - o_x)/d_x \end{aligned} \tag{4.1}$$

The ray will not enter an octant unless it has crossed all the entry planes. Consequently, the t at which the ray enters the octant is the t of the last entry plane it crosses. Since t is strictly increasing as we travel along the ray, the t at which we enter the octant, t_{enter}, is the largest of t-values of the three entry planes. By the same reasoning, the time at which we exit the octant is the smallest of the t-values of the three exit planes.

$$\begin{aligned} t_{enter} &= max(t_{0x}, t_{0y}, t_{0z}) \\ t_{exit} &= min(t_{1x}, t_{1y}, t_{1z}) \end{aligned} \tag{4.2}$$

4.3 Child Nodes

Assuming that the ray intersects an octant, and given the t-values for that octant, the t-values for the child nodes can be easily derived. As each child nodes shares either the entry plane or exit plane for each side, one set of t-values can be copied from the parent node. The other set can be found by finding the t-values for the middle planes of the node ($\mathbf{t_m}$ or t_{mx}, t_{my}, t_{mz}):

$$\mathbf{t_m} = (\mathbf{t_0} + \mathbf{t_1})/2 \tag{4.3}$$

The t-values for the child node can now be found. Equation 4.4 shows an example for the x-values, and can be extended to the other components.

$$\left.\begin{aligned} t_{0x\text{-child}} &= t_{0x} \\ t_{1x\text{-child}} &= t_{mx} \end{aligned}\right\} \quad \text{If child-node is on the low side of the middle plane} \tag{4.4a}$$

$$\left.\begin{aligned} t_{0x\text{-child}} &= t_{mx} \\ t_{1x\text{-child}} &= t_{1x} \end{aligned}\right\} \quad \text{If child-node is on the high side of the middle plane} \tag{4.4b}$$

For instance, child 0 in Figure 4.1 is on the left (lower) side of the middle x-plane, and thus its t_{0x} and t_{1x} is the parents t_{0x} and t_{mx}. Determining which side of the middle plane a given child node is on is made easy by carefully selecting the indices of the child nodes. There are 8 child nodes, which can be represented by a 3-bit integer. The child nodes are numbered so that each of the 3 bits determine which side of the middle plane the nodes are on. Bit 2, 1 and 0 represent respectively X, Y and Z. I.e., if bit 2 is set, the child node is on the lower end of the X-axis. This results in the numbering shown in Figure 4.2. The hidden node is node #1.

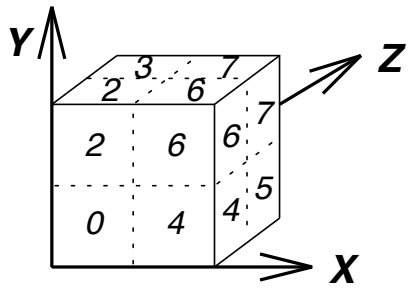


Figure 4.2: Indexing of child nodes[41].

Finding child indices When processing a node, the child-nodes have to be traversed in the right order. The first child node the ray enters have to be found, and if a collision is not found there, there are one or two more child nodes the ray can intersect. The first child node is determined by looking at which side of the various middle planes the ray enters. This is best illustrated by Algorithm 1.

Algorithm 1 FIRSTINDEX($t_{enter}, \mathbf{t_m}$)

```
1: idx ⇐ 0b000        (3-bit integer)
2: if t_enter > t_mx then
3:     idx[2] ⇐ 1
4: end if
5: if t_enter > t_my then
6:     idx[1] ⇐ 1
7: end if
8: if t_enter > t_mz then
9:     idx[0] ⇐ 1
10: end if
11: return idx
```

After traversing a child node, finding the index of the next child node is a function of the previous index and the exit plane of the child. This function is illustrated in Algorithm 2. It states that if the ray exits the child through the YZ-plane, which is equivalent to t_{1x} being the smallest of $\mathbf{t_1}$, then the ray should step to the next child along the x-axis. If the child is already on the high side of the x-axis, it means the ray exited the parent voxel.

Algorithm 1 and Algorithm 2 has here been transformed from the tabular form found in Revelles et al. [41] to a functional form closer to the form which we will use in the hardware implementation in later chapters.

Algorithm 2 NEXTINDEX($idx_{prev}, t_{\text{exit-child}}$)

```
1:  idx ⇐ idx_prev        (3-bit integer)
2:  exit-node ⇐ false
3:  if t_1x-child = t_exit-child then
4:     if idx[2] = 1 then
5:        exit-node ⇐ true
6:     end if
7:     idx[2] ⇐ 1
8:  else if t_1y-child = t_exit-child then
9:     if idx[1] = 1 then
10:       exit-node ⇐ true
11:    end if
12:    idx[1] ⇐ 1
13: else
14:    if idx[0] = 1 then
15:       exit-node ⇐ true
16:    end if
17:    idx[0] ⇐ 1
18: end if
19: return idx, exit-node
```

4.4 Negative Directions and Parallel Rays

Negative Directions It was mentioned that we have assumed that the components of the direction vector are all positive. This is obviously not the case for real rays (i.e., a ray can point in any direction in the space), and so the algorithm must be modified slightly to handle this case. If one of the components of the direction vector is negative, the ray is mirrored about the corresponding middle plane of the octree. Furthermore, a bit mask is set to recall this fact, labeled *dir-mask* in Algorithm 3. The bit mask is only used when we retrieve a child node. The corresponding bits in the child node index is then flipped (implemented using **xor** in Algorithm 3), and the the algorithm will consequently descend into the correct child node. Why this works is explained more thoroughly by Revelles et al. [41].

Parallel Rays If a ray runs parallel to an axis, t_0 and t_1 can not be found (or will be infinite). There are two ways to deal with this. Revelles et al. [41] suggests setting t_0/t_1 to $+\infty$ or $-\infty$ depending on conditions, and taking this into account in all calculations that involve the parameters. A different solution is to check if the components of the ray direction vector $\mathbf{d}$ is zero, and if so, set them to a tiny non-zero value[30]. This is slightly less accurate, but makes the algorithm easier to implement.

4.5 The Tracing Kernel

Algorithm 3 TRACEKERNEL(*node*, $\mathbf{t_0}$, $\mathbf{t_1}$)

```
1: if t_1x < 0 or t_1y < 0 or t_1z < 0 then
2:    return false
3: end if
4: t_enter ⇐ max(t_0x, t_0y, t_0z)
5: if CHILDISLEAF(node) then
6:    PROCESSLEAF(node, t_enter)
7:    return true
8: end if
9: t_m ⇐ (t_0 + t_1)/2
10: child-idx ⇐ FIRSTINDEX(t_enter, t_0)
11: repeat
12:    child-node ⇐ GETCHILDNODE(node, child-idx ⊕ dir-mask)
13:    t_0-child, t_1-child ⇐ SELECTCHILDT(child-idx, t_0, t_m, t_1)
14:    if ISVALID(child-node) then
15:       found-leaf ⇐ TRACEKERNEL(child-node, t_0-child, t_1-child)
16:       if found-leaf then
17:          return true
18:       end if
19:       child-node, exit-node ⇐ NEXTINDEX(child-idx, t_1-child)
20:    end if
21: until exit-node
22: return false
```

The kernel of the tracing algorithm can now be defined, and is listed in Algorithm 3. It is formulated as a recursive function for clarity. The kernel can be called with the parameters calculated for the root node. It will traverse the octree until a leaf node is found, or the ray exits the octree. The first lines will skip all nodes it encounters before the start of the ray ($t = 0$). It then checks if the current node is a leaf node. If not, it finds the first child node index, and starts to iterate over each child node the ray intersects. If a valid node is found it recurses on that node, if not it calculates the next child index and iterates untill the ray exits the current node.

To illustrate how the algorithm works in practice, Figure 4.3 shows the algorithm running on a small three-level octree with two solid voxels. The ray will pass close to the first voxel and hit the second one.

Initialization The t-values are calculated for the root, and we check if the ray hits the octree. Since it does, the trace kernel is called for the root node.

Root node The first child index is found (node 0). The child is empty. The next child index is found (node 2). The child is an inner node. The algorithm recurses on node 2. ①

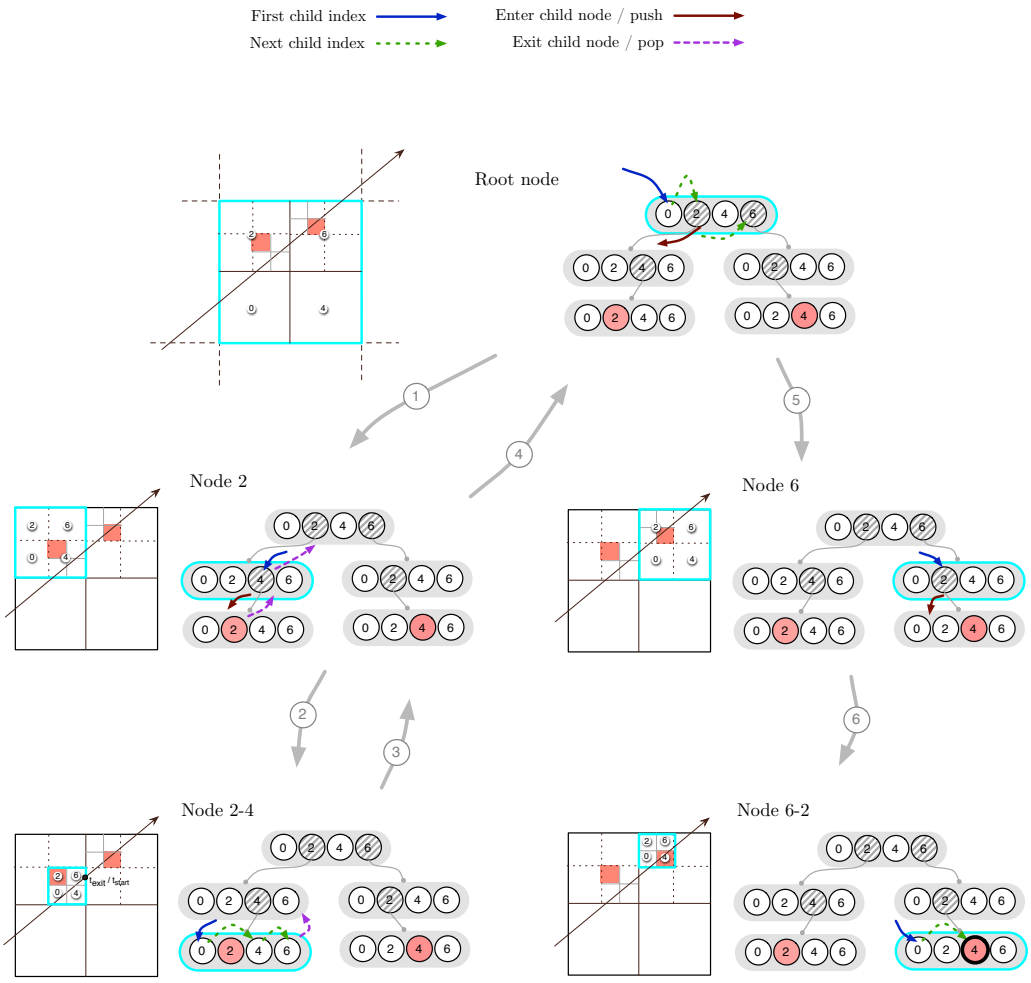


Figure 4.3: Illustration of the octree tracing algorithm.

Node 2 The first child index is found (node 4). The child is an inner node. The algorithm recurses on node 2-4. ②

Node 2-4 The first child index is found (node 0). The child is empty. The next child index is found (node 4). The child is empty. The next child index is found (node 6). The child is empty. The ray exits the current node and the algorithm goes up a level. If we used a restarting algorithm, we would use $t_{\text{exit-child}}$ to set a parameter t_{start}, and restart from the root. Then, when evaluating a node, we would pass it if $t_{enter} < t_{start}$. ③

Node 2 We re-enter node 2 from child index 4. The ray exits the current node and the algorithm goes up a level. ④

Root node We re-enter the root node from child index 2. The next child index is found (node 6). The child is an inner node. The algorithm recurses on node 6. ⑤

Node 6 The first child index is found (node 2). The child is an inner node. We recurse on node 6-2. ⑥

Node 6-2 The first child index is found (node 0). The child is empty. The next child index is found (4). The child is a solid voxel. The algorithm terminates.

Chapter 5

A Ray Tracer Geometry Stage

With traditional rasterization, the model-view matrix and the perspective projection matrix is the main data which dictates the size and position of the models relative to the camera, and how they should be projected to a 2D image. If ray tracing of SVOs is to be implemented alongside rasterization, it would be very benificial to use the same data in the ray tracer. This chapter outlines a geometry stage for a ray tracer which mirrors the one used in rasterization. Although it is probable that these techniques have been developed previously, for instance in commercial ray tracing systems, it was not described in any of the literature covered in this thesis. Therefore, these techniques were developed independently.

5.1 Normalizing the Octree

In rasterization, the world is transformed so the camera occupies a fixed position and orientation, and the volume that should be rendered occupies a fixed space. With ray tracing of sparse voxel octree, it is the octree that must occupy a fixed position. As mentioned, the algorithm requires an axis-aligned octree. The algorithm can in principle work with an octree of any size. However, to simplify analysis and implementation, we choose to normalize the octree so that it occupies the volume defined that by the points (-1,-1,-1) to (1,1,1). I.e., the length of the sides of the octree is always 2, and the center of the octree is in the origo of the coordinate system. Instead of defining the size and the position of the model by the corners of the octree, we can use a model transform matrix. This has two benefits: the core of the algorithm can make simplifying assumptions, and the same techniques used to transform polygon model can be used to transform the octree, which should simplify and ease programming.

5.2 Generating Primary Rays

In a ray tracer, it is the position and direction of the primary rays which dictates how the model end up looking on the screen. We can easily define these rays in the canonical view volume. Each ray starts at $z = -1$ and ends at $z = 1$, and they are evenly spaced along the X/Y axis. The distance between each ray along the X and Y axis is defined by the width and height of the image. Now consider the transforms discussed in Chapter 2.2. If we follow these in reverse, we get the position and direction of the rays in the coordinate system where the octree is axis aligned and normalized. In essence, the idea of the ray tracer geometry stage is to perform the same operations as in the rasterization geometry stage (Figure 2.3), and perform them in reverse on the ray data rather than the model data. This is illustrated in Figure 5.1. Fortunately, the model and view transform can be reversed simply by taking the inverse of the model/view matrix, but the perspective projection transform requires some elaboration.

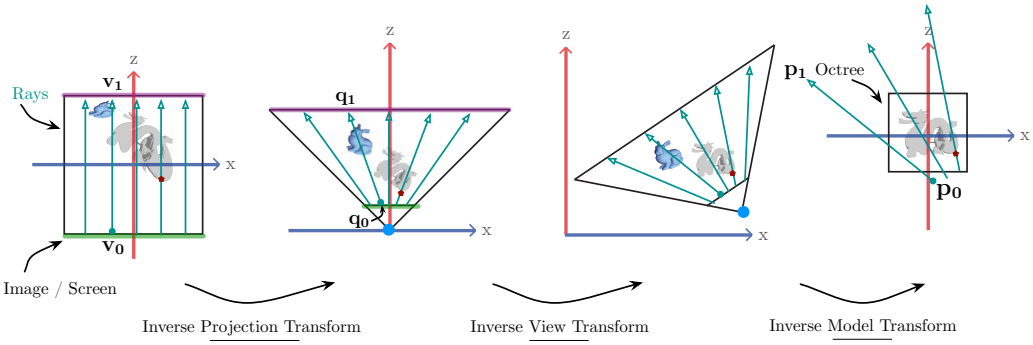


Figure 5.1: A ray tracer geometry stage in action

Equation 5.1 shows the equations for the ray tracer geometry stage, where x and y are the x and y coordinates of the pixel in the image, w and h is the width and height of the image, v_0 and v_1 are the starting and end point of the ray in the canonical view space, q_0, q_1, p_0 and p_1 are the same points in the camera space and the normalized octree space. These points are also illustrated in Figure 5.1. $\mathbf{P}_i$ is the inverse perspective projection and $\mathbf{M_{mv}}$ is the model-view tranform matrix.

$$
\begin{aligned}
\mathbf{v_0} &= (x/w, y/h, -1, 1) \\
\mathbf{v_1} &= (x/w, y/h, 1, 1) \\
\mathbf{q_0'} &= \mathbf{P}_i \mathbf{v_0} \\
\mathbf{q_1'} &= \mathbf{P}_i \mathbf{v_1} \\
\mathbf{q_0} &= \mathbf{q_0'} / \mathbf{q_{0w}'} \\
\mathbf{q_1} &= \mathbf{q_1'} / \mathbf{q_{1w}'} \\
\mathbf{p_0} &= \mathbf{M_{mv}}^{-1} \mathbf{q_0} \\
\mathbf{p_1} &= \mathbf{M_{mv}}^{-1} \mathbf{q_1}
\end{aligned}
\tag{5.1}
$$

The ray origin $\mathbf{o}$ and direction vector $\mathbf{d}$ for the ray traversal algorithm (see Equation 4.1) may be found as such:

$$\begin{aligned} \mathbf{o} &= \mathbf{p_0} \\ \mathbf{d} &= \mathbf{p_1} - \mathbf{p_0} \end{aligned} \tag{5.2}$$

5.3 Inverse Perspective Projection

The model/view matrix can be inverted using standard matrix inversion algorithms such as Gauss-Jordan elimination. The perspective projection however, can not be inverted by simply inverting the perspective projection matrix, because the result is divided by the w-component(Equation 2.4). Using simple linear algebra, we can find a similar equation to go from a canonical view space point $\mathbf{v}$ to a camera space point $\mathbf{q}$.

Solving for $\mathbf{P}_i$ results in the following matrix:

$$\mathbf{P}_i = \begin{pmatrix} -\frac{f(r-l)}{f-n} & 0 & 0 & -\frac{f(r+l)}{f-n} \\ 0 & -\frac{f(t-b)}{f-n} & 0 & -\frac{f(t+b)}{f-n} \\ 0 & 0 & 0 & \frac{2fn}{f-n} \\ 0 & 0 & 1 & -\frac{f+n}{f-n} \end{pmatrix} \tag{5.3}$$

5.4 Normalizing Ray Length

A ray has a starting point and a direction. The direction is often defined as a normalized vector with a length of 1. However, the output of the proposed geometry stage is a starting point and an end point. The starting point is positioned at the near plane and the end point is positioned at the far plane. Although a far plane is strictly speaking not necessary in a stand-alone ray tracer, when combining with rasterization it can be be useful to have the option of terminating the ray tracing at the far plane. Following equation 2.5, the near plane is placed at $t = 0$ and the far plane at $t = 1$. If we standardize on this, our algorithm can be designed to optionally terminate at $t = 1$. This functionality should be useful when calculating light/shadow rays as well. In this case we also have a starting point (the point at which the primary ray hits an object) and an end point (the light source), and we are only interested in finding if the line between these points intersects some object. I.e., we should terminate the ray tracing at $t = 1$. Another useful feature of having the start and end point be at the near and far plane, is that the resulting t-value can be used to calculate a value for the Z-buffer.

5.5 Z-Buffering With Ray Tracing

In rasterization, the Z-buffer ensures that the geometry appears appears in the correct order from front to back. If we wish to merge the result of the ray traced image with a

rasterized image, and have everything appear in the correct order, we need to be able to generate the correct values for the Z-buffer.

When a primary ray hits a voxel, the algorithm returns a t-value (t_{out}) which represent how far along the ray the voxel is. Because we placed the start and end of the ray at the near and far plane, this t-value also represents the position of the voxel between the near and the far plane. This is illustrated by the red dot in Figure 5.1, which can be followed in reverse from its position in the octree on the right, to its position in the canonical view volume on the left.

However, this can not be directly used as a Z-value. The t_{out} is linear with regards to distance in the camera space. In rasterization the Z-value is linear with regards to distance in the canonical view volume. If we use perspective projection, this Z-value is not linear with regards to camera space. If we look at Equation 2.4, we can see that the Z-value in the canonical view volume, v_z, is divided by v'_w. This implies that v_z is a function of $1/q_z$ where q_z is the Z-value in the camera space.

To solve this problem, we can first find the collision point in the camera space, given the t-value at the collision point t_{out}:

$$\mathbf{q_c} = \mathbf{q_0} + (\mathbf{q_1} - \mathbf{q_0}) \cdot t_{out} \tag{5.4}$$

Using the perspective projection matrix $\mathbf{P_p}$ defined in Equation 2.3, we can transform this coordinate in to the canonical view volume:

$$\mathbf{v_c} = \mathbf{P_p q_c} \tag{5.5}$$

Taking the z-coordinate from this vector will give us a Z-value which is compatible with the Z-buffer generated by the rasterizer.

Chapter 6

Hardware Optimizations

In order to optimize the ray traversal algorithm for hardware implementation, various considerations were made, and the algorithm was transformed to a form which is feasible to implement in a hardware description language. This chapter will present the rationale for using fixed point numbers, we will analyze the use of a stack in hardware, and the final hardware optimized version of the algorithm will be presented.

6.1 Floating Point vs Fixed Point

A fundamental decision regarding how to implement the algorithm, was whether to use floating point or fixed point representation for the parameters in the algorithm. Since the algorithm could be integrated with a modern GPU – which extensively support use of floating point numbers – using a floating point representation might be pertinant. However, looking at the algorithm, the operations performed on the parameters in the core of the algorithm are only additions and divisions by two. Additions require much more resources to implement and is slower to execute with floating point numbers. The difference is probably even more dramatic with the division by two. This can be done with a simple sign-extended shift operation in fixed point arithmetic[26]. Because of these differences in resource use, a fixed point representation was chosen in this thesis.

To integrate the hardware module presented in this thesis with a system using floating point numbers, the module could integrate a floating point to fixed point conversion unit. Even given the cost of this unit, it is probably still better than using floating point units within the core of the module. The floating point parameters passed to the module could fall out of range of the chosen fixed point representation, but this could be solved by scaling the parameters by a common factor such that the largest parameter falls within the range of the fixed point representation. When the results are passed back out of the module, they could be scaled back using the same scaling factor.

The number of bits used to represent the number was chosen to be 32 bits. This was due to the fact that the bus width is 32 bits, and that the OpenRISC processor that was used

is a 32-bit processsor.

6.2 The Decimal Point

The position of the decimal point in a fixed point number dictates the range and precision of the numbers that can be represented. In this thesis, it was decided to use 16-bit each to represent the integer and fractional part of the number. Since the octree is normalized to exist within the range $-1..1$, the octree is contained within the fractional part of the number. Consequently we have 16 bits to represent a position within the octree.

Numbers equal to or larger than 2^{15} can not be represented by the chosen representation. When calculating the initial t-values for the algorithm, it is important to make sure the calculations to not overflow, which would make the results nonsensical. Taking Equation 4.1 and applying the constraints of the normalized octree, we end up with Equation 6.1, where o_x and d_x is x-component of the ray origin and direction vectors.

$$\begin{aligned} t_{0x} &= (-1 - o_x)/d_x \\ t_{1x} &= (1 - o_x)/d_x \end{aligned} \tag{6.1}$$

Assuming the ray origin is somewhere outside the octree, i.e. $o_x \gg 1$, we have $t_{0x} \approx o_x/d_x$. The value of t_{0x} can easily overflow if o_x is large and d_x is small. We should therefore put a constraint on o_x. Constraining o_x arbitrarily to be less than 2^5 gives us a lower bound for d_x:

$$d_x = o_x/t_{0x} = 2^5/2^{15} = 2^{-10} \tag{6.2}$$

This implies that if $2^{-10} < d_x < -2^{-10}$, then d_x should be clipped to $\pm 2^{-10}$ before calculating $\mathbf{t_0}$ / $\mathbf{t_1}$. If we do this we also avoid the problem discussed in Chapter 4.4, where $d_x = 0$. This clipping implies there is a limit to the resolution of the direction vector, and we have shown that there is a relationship between this limit, the maximum values of the origin vector and the upper bound of our number representation.

6.3 Stack

Algorithm 3 is formulated as a recursive function. Since functions are not a native primitive in hardware design, an explicit stack must be used to achieve the same functionality. Instead of recursing, the algorithm will push the necessary variables in the stack, and instead of returning the variables will be popped from the stack.

The stack requires memory, and since the purpose of the stack is to avoid slow memory requests, the stack memory should be fast, preferably accessible in one clock cycle. This implies that this memory will be expensive in terms of resources, and we should seek to minimize the size.

The state that must be stored on the stack are the node data and address, the current child index, and the t-values $\mathbf{t_0}$ and $\mathbf{t_1}$. The node address is a 32-bit pointer pointing to the node data. The node data is a 32-bit value consisting of the 16-bit child table pointer and the 16-bit child data. The node data is not strictly necessary since we could request the node data from memory using the node address. Howevery, this would partly defeat the performance enhancing benefits of the stack.

It is possible to eliminate the need to store the 16-bit child pointer on the stack. Whenever the algorithm enters a child node, we must calculate the child address. Taking the current node address and adding the 16-bit child table pointer gives us the address of the table of child nodes. Adding the child offset to this value gives us the address of the desired child node. By storing the resolved child node table address instead of the parent node address on the stack, we avoid the need to store the child table pointer, as the only use of these two values is to calculate the child table address. This optimization has a secondary benefit aswell: if the child table pointer is a far pointer, we need to perform an extra memory request to fetch the far pointer in order to calculate the child table address. By storing the child table address on the stack we avoid having to resolve the far pointer again later.

The child index is a 3-bit value. It is not strictly necessary to store this value on the stack. If we implement restarting, as will be described later, the t_{start} parameter is sufficient to insure that we do not visit nodes we have visited before. However, this would come at a cost of a few extra wasted clock cycles each time the stack is popped. Considering that the child index is so small relative to the node data/address, it is probably worth keeping on the stack.

The t-values $\mathbf{t_0}$ and $\mathbf{t_1}$ are each a vector of three 32-bit numbers. However, it is not necessary to store the full value of these numbers on the stack. When the algorithm enters a child node, the t-values for the child node is picked from $\mathbf{t_0}$, $\mathbf{t_m}$ and $\mathbf{t_1}$. If we can reconstruct the parents t-values from the child t-values we can avoid storing them on the stack. Consider how the child t-values are calculated:

$$\mathbf{t_m} = (\mathbf{t_0} + \mathbf{t_1})/2 \tag{6.3a}$$

$$\left.\begin{aligned} t_{0x\text{-child}} &= t_{0x} \\ t_{1x\text{-child}} &= t_{mx} \end{aligned}\right\} \quad \text{If } \textit{child-idx[2]} \text{ is 1} \tag{6.3b}$$

$$\left.\begin{aligned} t_{0x\text{-child}} &= t_{mx} \\ t_{1x\text{-child}} &= t_{1x} \end{aligned}\right\} \quad \text{If } \textit{child-idx[2]} \text{ is 0} \tag{6.3c}$$

If we store *child-idx* on the stack, we can use this value to find either t_{0x} or t_{1x}. We call this value t_{pass}:

$$\left.\begin{aligned} t_{pass} &= t_{0x} = t_{0x\text{-child}} \\ t_{mx} &= t_{1x\text{-child}} \end{aligned}\right\} \quad \text{If } \textit{child-idx[2]} \text{ is 1} \tag{6.4a}$$

$$\left.\begin{array}{l} t_{pass} = t_{1x} = t_{1x\text{-child}} \\ t_{mx} = t_{0x\text{-child}} \end{array}\right\} \quad \text{If } \textit{child-idx[2]} \text{ is } 0 \tag{6.4b}$$

The other value, which we will name $t_{recover}$, must be recovered from t_{mx}. However, the division by two in Equation 6.3a, which is implemented as a shift-operation in hardware, discards one bit of information. To recover $t_{recover}$, we need to save this bit, which we will call t_{bit}.

$$t_{mx}, t_{bit} = (t_{pass} + t_{recover})/2 \tag{6.5a}$$

$$t_{recover} = (t_{mx}, t_{bit}) \cdot 2 - t_{pass} \tag{6.5b}$$

Finally we set t_{0x} or t_{1x} to $t_{recover}$ depending on the value of *child-idx[2]*.

Using this method, we have reduced the stack memory requirement of the t-values to one bit per component, i.e., three bits in total. Adding up the total size of the required variables we get $32 + 16 + 3 + 3 = 54$ bits. For a stack of depth D the total size of the stack in bits S is:

$$S = 54D \tag{6.6}$$

The optimal depth of the stack, and thus the total size of the stack, has been optimized experimentally. This is presented in Chapter 8.

6.4 Restarting

Implementing restarting is useful because it makes it possible to trace octrees which are deeper than the depth of our stack. We can adjust the size of the stack to a size that gives the best performance for the resources we use, and we do not have to worry about the size of the octree. Even if the stack is large enough to trace the octrees we are currently using, we might want to use the hardware for larger octrees in the future.

First, we need to find the t-value at which the ray exits a node:

$$t_{exit} = min(t_{1x}, t_{1y}, t_{1z}) \tag{6.7}$$

Restarting requires one extra variable t_{start}. When we exit a node, instead of popping the stack to return to the parent node, we store t_{exit} in t_{start}. We then restart the algorithm from the root node. Each time the algorithm processes a child node it will pass the node if $t_{start} >= t_{exit-child}$. I.e., the next time we visit the parent node of the last node we visited before we restarted, the algorithm will pass that node.

This operation is quite expensive compared to popping the stack, since we have to process every node from the root node down to the parent node of the node we exited, while when we pop the stack we go directly back to the parent node.

6.5 Hardware Optimized Algorithm

To implement the raytracing algorithm in hardware, it first had to be converted to a form which is suitable for hardware implementation. The goal was to create a state machine where the operations performed in each state was limited and reuse of functional units was possible between states. This process did not follow a strict methodology, but it was informed by techniques from high-level synthesis[12] and compiler techniques[2].

First, the kernel listed in Algorithm 3 was changed from a recursive form to a non-recursive form. The recursive function call was replaced by a loop and a stack was introduced which could save and restore the required state, after which the algorithm consisted of a set of nested loops. The algorithm was converted to a state machine by introducing a state for each basic block in the function, and replacing loop branching with state transitions.

In synchronous digital circuits, it is important to keep the delay of the critical path (i.e., the longest path through a combinatorial circuit) below the period of the clock. A shorter critical path increases the maximum clock frequency of the design. This can be achieved through pipelining, which splits the critical path with registers, and the same operation is divided over additional clock cycles[4]. In this thesis the algorithm was implemented in a system alongside the 32-bit OpenRISC processor and the ray tracing module was designed to run at the same clock frequency as this processor. Running the module and the processor at different clock frequencies would have resulted in a design with multiple clock domains which are more difficult to implement[4]. The OpenRISC processor performs 32-bit addition, subtractions and comparisons in one clock cycle[36]. To keep the maximum clock frequency of the ray tracing module similar to that of the processor, the algorithm was designed so that the critical path would go through no more than one 32-bit adder/subtractor/comparator, a couple of multiplexer and some simple combinatorial logic. To achieve this, any state that had two or more chained arithmetic operations was split into two or more states.

To minimize the amount of resources a digital design uses, it is important to consider resource sharing. By allowing the operations in different states to share functional units such as arithmetic units, we can minimize the area (or LUTs) consumed by the design[40]. Several operations in the presented algoritm requires three comparators. These are the *max*, *min* and FirstIndex(Algorithm 1) functions in Algorithm 4. These are all performed in different states, and could potentially share comparators. As adders and comparators are similar functional units, the calculation of $\mathbf{t_m}$ could potentially also share these units, by splitting up the CalcT state. Even further reduction of resource use could be achieved by further pipelining the design and perform a single addition/comparison in each cycle.

Algorithm 4 shows the kernel of the algorithm converted to a form which should be suitable for hardware implementation. Each label represents a state in a state machine, and each state has been assigned operations that can be executed in a single cycle. This algorithm was used in the ray traversal core of the hardware implementation. Further

descriptions along with a state diagram can be found in Chapter 8.5.

Algorithm 4 TRACECORE(*root-adr*, *root*-$\mathbf{t_0}$, *root*-$\mathbf{t_1}$, *dir-mask*)

```
 1: globals node, level, idx
 2: globals is-first-node, exits-node, pass-node
 3: globals t-start, t-enter, t-exit, t-exit-child
 4: globals t_0, t_1, t_m, t_bit
 5: globals t_0-child, t_1-child
 6: globals is-leaf, t-out
 7: INIT:
 8:    t_0 ⇐ root-t_0
 9:    t_1 ⇐ root-t_1
10:    is-first-node ⇐ true
11:    level ⇐ 0
12:    node ⇐ GETROOTDATA(root-adr)
13:    goto CALCT

14: CALCT:
15:    t_m, t_bit ⇐ (t_0 + t_0)/2
16:    t-enter ⇐ max(t_0x, t_0y, t_0z)
17:    if is-first-node then
18:       goto FIRSTIDX
19:    else
20:       goto NEXTIDX
21:    end if

22: FIRSTIDX:
23:    if got-node-data then
24:       idx ⇐ FIRSTINDEX(t-enter, t_m)
25:       exits-node ⇐ false
26:       is-first-node ⇐ false
27:       goto CALCCHILDT
28:    else
29:       goto FIRSTIDX
30:    end if
```

TraceCore *continued*

```
31: NextEval:
32:   idx, exits-node ⇐ NextIndex(idx, t-exit-child)
33:   if exits-node then
34:     if level = 0 then
35:       is-leaf ⇐ false
36:       t-out ⇐ t-exit-child
37:       goto Finished
38:     else if stack-empty then
39:       t-start ⇐ t-exit-child
40:       goto Init
41:     else
42:       node, t0, t1, idx ⇐ Pop
43:       level ⇐ level − 1
44:       goto CalcT
45:     end if
46:   else
47:     goto CalcChildT
48:   end if

49: CalcChildT:
50:   t0-child, t1-child ⇐ SelectChildT(idx, t0, tm, t1)
51:   t-exit-child ⇐ min(t1x−child, t1y−child, t1z−child)
52:   goto Eval

53: Eval:
54:   pass-node ⇐ t-exit-child <= t-start
55:   if pass-node then
56:     goto NextIdx
57:   else if IsChildLeaf(node, idx⊕dir-mask) then
58:     t-out ⇐ t-enter
59:     goto Finished
60:   else if IsChildValid(node, idx⊕dir-mask) then
61:     Push(node, tbit, idx)
62:     t0 ⇐ t0-child
63:     t1 ⇐ t1-child
64:     level ⇐ level + 1
65:     is-first-node ⇐ true
66:     node ⇐ GetData(node, idx⊕dir-mask)
67:     goto CalcT
68:   else
69:     goto NextIdx
70:   end if

71: Finished:
72:   return  is-leaf, t-out
```

Chapter 7

Software Implementation

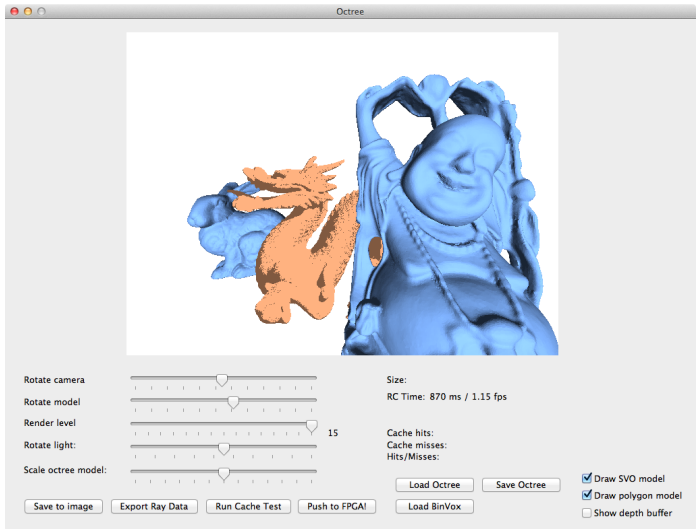


Figure 7.1: A screenshot of the software application used to support this thesis.

Implementing an algorithm in a hardware description language like Verilog can be a cumbersome task, and if there is an error in the design it can be hard to track down where and how the error occured. Implementing the system in a software application can help ease the process of implementing it in hardware later. It can provide a better intuitive understand of the algorithm and it can make debugging easier by comparing the output from the two implementations.

This chapter will present the software application which was created to support this thesis. The software was used to generate sparse voxel octrees with the correct format, to implement the ray tracing algorithm, to test the proposed ray tracer geometry stage, to generate an image using hybrid rendering and to create a model of a cache in order to evaluate the attributes of a cache in hardware.

The application was created using Xcode on Mac OS X. The GUI-related code was written in Objective-C, while the code to generate and trace the octree was written in C++. The software was executed on a laptop with a 2 GHz Intel Core i7 processor, 4 GB of 1333 MHz DDR3 memory and an AMD Radeon HD 6490M GPU with 256 MB graphics memory.

7.1 SVO Data Structure

Both the software and hardware implementation needs data in a sparse voxel octree format. The data structure in this thesis is organized in a way similar to that of Laine and Karras [29], but the contour data, page header and info section they describe was dropped as none of those features was required for this thesis. The resulting data structure is illustrated in Figure 7.3. The data structure is an array of 32-bit entries. Two 8-bit masks indicate the type of the child nodes. The 8-bit `leaf_mask` indicates whether a child node is a leaf node or not. The 8-bit `valid_mask` indicates whether a child node has solid voxels or not. A partially filled inner node will have its `valid_mask` set but not its `leaf_mask`. A solid leaf node (a solid voxel) will have both set. If the node has any inner node children, the 15-bit `child_table_ptr` is a relative pointer to a table of these child nodes. If the 15-bits are not enough to hold the pointer, the `is_far` bit is set, and `child_table_ptr` points to an entry with a 32-bit pointer. The final address of the child table is found by adding the 32-bit far pointer to the parent node address.

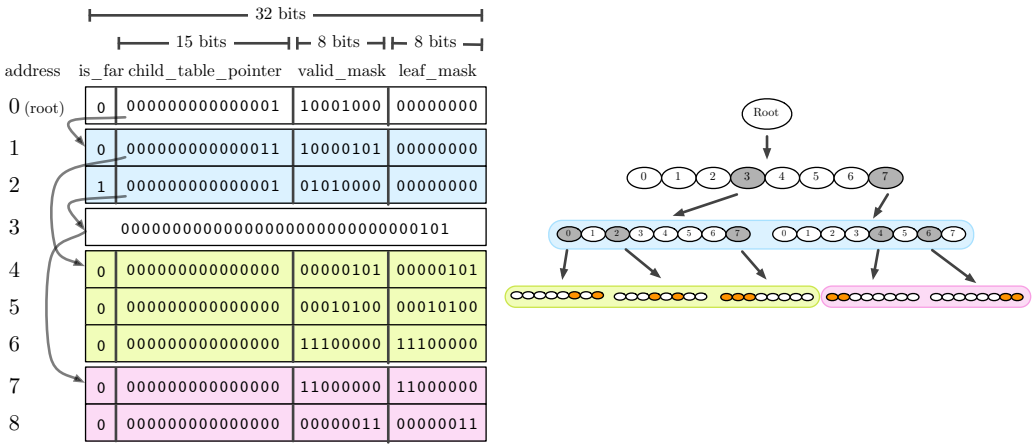


Figure 7.2: The organization of the sparse voxel octree data structure.

7.2 Generating Sparse Voxel Octrees

The models used in this thesis was the “Stanford Bunny”, “Happy Buddha” and “Stanford Dragon” models from Stanford Computer Graphics Laboratory [50]. These models were processed by the third-party program “binvox”[16], which converted the polygon models to voxel models. The program outputs a run-length encoded voxel format. A routine

was created to convert this format to an octree. The routine loads the entire voxel model to a three-dimensional array in memory. It then recursively divides the array into eight sub-regions untill a region contains a single voxel and constructs an inefficient octree in the process. This octree is then compacted into the binary format described previously. This data can then be used by the software application and be written to a binary file which can be uploaded to the FPGA memory. The voxel models are 1024 voxels wide along each axis, and the resulting octree is 10 levels deep.

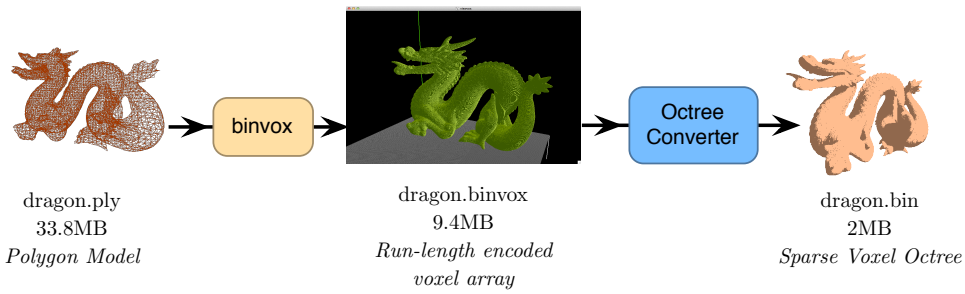


Figure 7.3: The process of generating sparse voxel octree models.

7.3 Software Ray Tracer

The ray tracing algorithm was implemented in software to provide reference images and facilitate in debugging the hardware solution. Initially the algorithm was implemented according to Algorithm 3, but it was later rewritten to closely match the hardware version. This proved immensely useful in debugging. When rendering the same image using the same input, the software and hardware simulation could be executed and the value of the variables in the algorithm could be compared at different points during execution. This made it easier to locate and fix bugs in the hardware implementation. The software ray tracer was also used to generate the parameters required to initialize the ray tracing core in the hardware implementation. The same function which generates these parameters for the ray tracing kernel in software can be redirected to write these parameters to a file. This file can then be uploaded to the FPGA memory and used by the hardware ray tracing core.

There are some important differences between the software and the hardware ray tracer. The software version does not run on multiple cores in parallell. It can use either floating point numbers or fixed point numbers in the calculation. Furthermore, although both a stack and cache is implemented in the software version, they are used to simulate the behavior of the hardware version, not to boost the performance in software. Finally, the shading, i.e., the coloring of the rendered image is different. In the software implementation, simple hard shadows are rendered by tracing secondary rays from the collision point of the ray to a point light source. These shadows are used to better visualize the octree model, and to provide preliminary indications that such techniques are possible using the algorithm. The software can also visualize the cost of rendering a ray by coloring the

pixel of the ray according to the number of memory requests required to trace that ray as seen in Figure 7.5

The core functions of the code for the software ray tracer is included in the appendix.

7.4 Merging Ray Tracing and Rasterization

To test the proposed technique for merging ray tracing and rasterization, the software ray tracer was designed with the proposed geometry stage, and the corrected Z-values was extracted and stored in a Z-buffer. The GPU on the computer executing the software was utilized through the OpenGL API to perform rasterization. The dragon voxel model was rendered with the ray tracer and the resulting image and Z-buffer was uploaded to the image buffer and Z-buffer of the GPU. The bunny and buddha polygon models were then drawn using the GPU, positioning them such that the bunny would appear behind the dragon, and the buddha in front of the dragon, if the Z-buffer generated by the ray tracer was correct. The results from this experiment is illustrated in Figure 7.4. It shows the image and depth buffer from the octree ray tracing and polygon rasterization, and that the three models appear in correct order in the combined image.

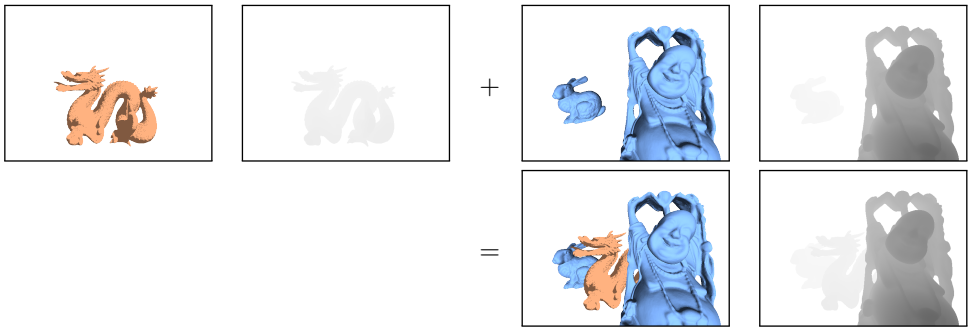


Figure 7.4: Ray traced image and Z-buffer + rasterized image and Z-buffer = combined image and Z-buffer

7.5 Cache Profiling

It was suspected that the memory bus of the hardware module would be a bottleneck for the algorithm. To investigate the benefits of a data cache in the hardware implementation a behavioral model of a cache was created in software. Each access to a node or far pointer in the octree was logged by a software cache model. This cache has three configurations: the cache size (in number of 32-bit words), the block size (in words) and block placement policy (direct mapped and 2/4-way set associative). A small test routine rendered a given SVO model from four different angles, logged the cache hits/misses and calculated the average miss rate.

As the data cache is designed for a specific algorithm, it should in theory be possible

to implement novel and specific optimizations to the cache which exploits the memory access patterns of the algorithm. For instance, nodes that are closer to the root are more likely to be accessed often than nodes deeper in the tree. It is possible that it is inefficient to cache nodes deeper in the tree. If they are accessed only once or twice, the benefit of caching them is small compared to the penalty of evicting a node closer to the root. To investigate this, a cache model was created where the cache was given the level of the node accessed in addition to its address. The cache could then be configured to disable caching for deeper nodes.

7.6 Results

Direct mapped

	Block size			
Cache size	1	2	4	8
32	0.46	0.54	0.60	0.61
128	0.25	0.28	0.31	0.33
512	0.17	0.17	0.16	0.16

Two way set associative

	Block size			
Cache size	1	2	4	8
32	0.37	0.49	0.60	0.60
128	0.18	0.19	0.21	0.26
512	0.14	0.12	0.10	0.09

Four way set associative

	Block size			
Cache size	1	2	4	8
32	0.31	0.45	0.60	0.59
128	0.15	0.14	0.16	0.20
512	0.14	0.11	0.08	0.06

Two way set associative
1 word block size

	Cache size			
Max level*	8	32	128	512
2	0.66	0.53	0.48	0.47
4	0.67	0.43	0.31	0.28
6	0.70	0.40	0.24	0.17
8	0.72	0.42	0.24	0.18

*Max level of N: only the nodes N levels or less from the root are cached

Table 7.1: Cache miss-rate results from software profiling. Smaller numbers are better.

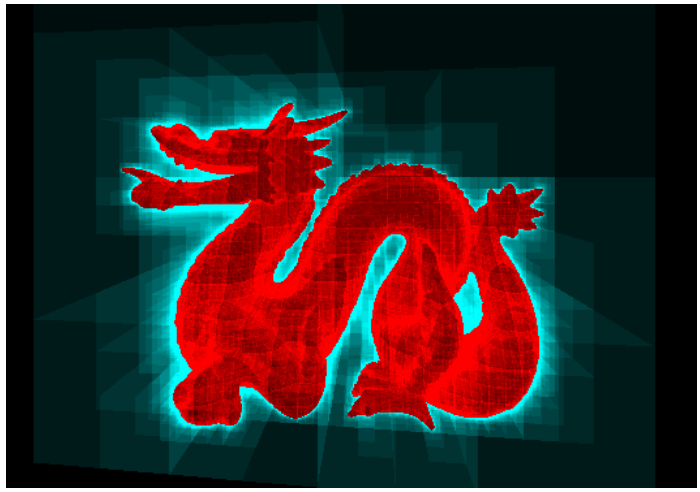
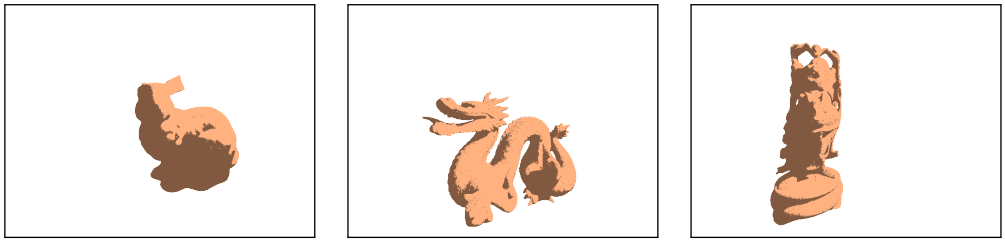


Figure 7.5: An illustration of the cost of tracing the primary rays. The lighter the color the more memory requests was used to trace that ray.



Stanford Bunny	Stanford Dragon	Happy Buddha
652ms	824ms	677ms

Figure 7.6: The three voxel models and their rendering speed in software.

7.7 Discussion

In the few experiments that were done, merging ray tracing and rasterization using the Z-buffer seemed to work correctly. In addition, since the model-view transform matrices used in ray tracing was extracted from the OpenGL API, the same OpenGL calls that were used to set up the rasterized models could be used for ray tracing as well. The same function calles used to set up the camera for the OpenGL rasterizer was used for the ray tracer as well. In other words, programming was made easy and tidy. However, this is a very basic demonstration. Commercial games rely on many different effects, e.g., translucent materials, reflections and shadows. Implementing these various effects using a hybrid rendering technique will probably be challenging.

The cache profiling resulted in some interesting data. Larger block size has a negative impact on the performance of the cache for small cache sizes. However, for large cache sizes the cache miss-rate seems to improve with larger block sizes, especially with two and four way set associative caches. It is possible that there is a rather small chance that the next memory access is one of the other words in the block. For small cache sizes that could imply that it is more likely that the algorithm will need one of the words that had to be evicted, than one of the other words in the block. However, as the cache size grows, the probability that a cache block read will evict a word that the algorithm will soon access diminishes.

A two way set associative cache is better than a direct mapped cache, and two way is better than four way. However, the improvement from two way to four way set associative is minor, and might not be worth the increased resources required to implement it. Finally, restricting which words are cache based on the depth of the node which is accessed seems to be detrimental to performance, for all but the very smallest caches. To understand why, consider the case where the ray for one pixel has been traced. The cache is now filled with nodes from this trace. In this implementation the next pixel to be traced is the next pixel to the right in the image. When tracing the ray for this pixel, it will follow a path very similar to the one for the last pixel, and consequently it will access mostly the same nodes, even for nodes deep in the tree. However, secondary rays such as those used to calculate shadows and reflections, are not as coherent as the primary rays. There is a possibility that a variation of this optimization could be useful in a more complex rendering system.

Figure 7.6 shows the time it took to render the three models in software. These are given to provide some frame of reference to the results from the hardware implementation. Because neither implementation is aggressively optimized, and given the vast differences in hardware capabilities, a direct comparison is of little interest. However, it would be surprising if the relative speed from one model to the others is different in the two implementations. Note that the dragon model takes longer to render. If a ray passes close to the edge of a model, it will be costly to trace because it will have to traverse deep in the octree to determine if it collides with a voxel or not. This effect is illustrated effectively in Figure 7.5, where brighter colors imply higher cost. Although the dragon appears to have a similar surface area as the bunny, the edge has more folds, and thus is more costly to render. In this figure one can also observe the structure of the octree. As the ray crosses into new octants the cost increases, and thus faint cubes can be observed in the picture.

Chapter 8

Hardware Implementation

The goal of this thesis was to create a practical design for ray tracing of sparse voxel octrees in hardware. To demonstrate that the proposed design could work, a hardware prototype had to be created. An FPGA prototype board was used, and a system-on-chip platform for that board was modified to include a ray tracing module. This module was written in the Verilog hardware description language.

This chapter presents the design of the hardware system and the details of the design of the ray tracing modules and its sub-components. The methodology used to test the complete system is presented, along with the results. Finally, the design and results are discussed.

8.1 Hardware Platform

As one of the goals of the thesis was to synthesize the module for an FPGA, a prototype board had to be chosen. The criteria for selecting an FPGA prototype board was:

- Academic Discount
- On-board RAM for storing ray data, octrees and the resulting image
- Video output (VGA or DVI/HDMI)

The Digilent Atlys prototype board was selected since it fitted well with the chosen criteria. One drawback of the board is that it contains relatively few switches and LEDs, and no display. However, there is a USB-port for a keyboard and/or mouse, and there is UART over USB. The UART module has been used extensively to debug the system, by communicating with a terminal on a personal computer. Additionally, the Cypress microcontroller that is used to configure/program the FPGA over USB, also connects to the FPGA through a set of data signals. Digilent provides a software API and a piece of software to transfer data to the FPGA over these signal lines.

Two different architectures was considered for the thesis. One was a standalone system, where the system would consist of only the ray tracing module, and the necessary support to connect it to RAM, to initiate ray traing, and to get image out to a computer or a display. The second was to use a soft core CPU, and implement the ray caster as an accelerator. A soft core CPU is a CPU which can be implemented on an FPGA[46], while an accelerator is a component attached to the same bus as the CPU which can provide performance increases for specific functions[63]. The benefits of having a CPU is that some parts of the system can be significantly easier to implement in software than in hardware. If those are not performance critical, then having a CPU to perform those tasks can cut down design time with little or no performance penalty. However, a soft core CPU can take up a significant amount of the FPGAs resources, leaving less for the ray tracing module, and increasing the time it takes to synthesize the system.

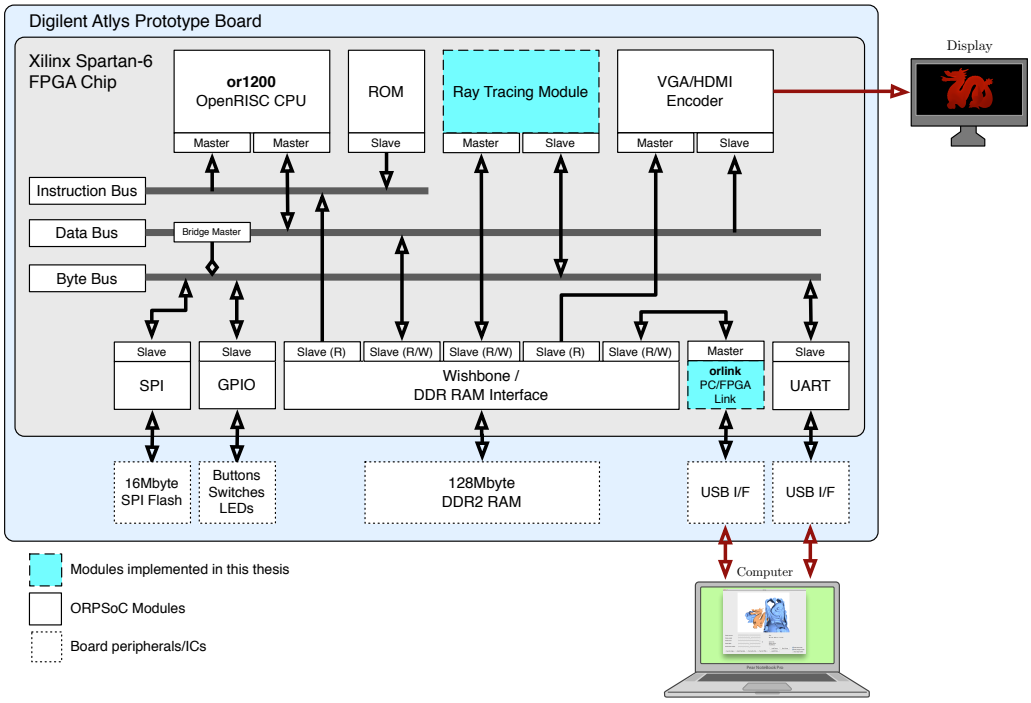


Figure 8.1: Hardware platform overview

The second architecture, with a CPU and the ray caster implemented as an accelerator, was chosen because the complexity of creating a stand-alone system was deemed to be too much for this thesis, and that debugging the system could be hard without an internal CPU which could quickly be programmed to change certain behavior in the system. To create the proposed system an existing system-on-chip solution had to be found. There was some investigation into using Xilinx' embedded development kit (EDK) and their MicroBlaze CPU. However, the tools were complicated to use and did not encourage automating the synthesis of the design. The open source system-on-chip solution *ORPSoC* has recently been ported to Atlys by Kristiansson [27]. Using this platform as a basis for the project meant that all the support modules were already in place. The ray tracing

module could easily be attached to the system bus, using the open interface standard *Wishbone*. In addition, the complete system could be built using *Makefiles*, which implies that the process of synthesizing the design could easily be automated, allowing many variations of the design to be synthesized.

A module named *orlink* was also created for this thesis. This module supports transfers between a computer and the FPGA system. This allowed OpenRISC software, ray data and octree data to be uploaded to the on-board RAM, and the resulting image to be downloaded to a computer. This work builds on McClelland's *FPGALink*[32] software, which uses a custom firmware for the Cypress microcontroller on the board. The software presents a software API similar to the one Digilent provides. However, FPGALink is cross-platform and open-source. *Orlink* consists of a verilog module, and a piece of software. The Verilog module translates request from the microcontroller, to Wishbone bus requests. The *orlink* software is a command-line interface that allows a user to halt and reset the CPU on the FPGA and transfer data between a personal computer and the on-board memory. As the module is not directly relevant to the results of the thesis, the details of this module will not be presented.

The final system is presented in Figure 8.1. The OR1200 OpenRISC CPU uses a Harvard architecture, which is why there is a separate instruction and data bus. The ROM module is a small read-only memory on the FPGA, which bootstraps the CPU. The bootstrapper can load software from an SPI Flash chip, or it can simply cause the CPU to jump to a reset vector in RAM. Both of these modes have been used in the project for various purposes. The instruction bus and the data bus are 32-bits wide, while the byte bus is 8-bits wide. All of them are implemented as cross-bar buses, which have good performance, but at a large routing cost[37]. The byte bus alleviates some of this cost, by having a narrower connection where bandwidth is not essential.

8.2 Ray Casting Module

The ray casting module is responsible for taking a collection of rays and tracing them through a given octree. It is divided into several sub-modules. The *slave module* implements a set of registers which are accessible from the CPU, these are used to configure and control the ray casting module. The slave module also generates an interrupt request to the CPU when a frame has finished rendering.

The module instantiates up to four cores, each of which can perform the tracing algorithm. The algorithm is inherently parallel, and the performance of the module should scale with the number of cores we use until we saturate the memory bandwidth. The *scheduler* iterates through each pixel in the image, loads ray parameters from memory, finds an available core to trace the ray, and once a core has finished tracing, it writes the correct pixel data to the frame buffer.

The module has a single master interface to read and write to the main memory. Both the scheduler and the cores require access to memory, so these requests are handled by a single *memory controller* module. It arbitrates the request from the scheduler and the cores, and implements a common cache for all of the cores.

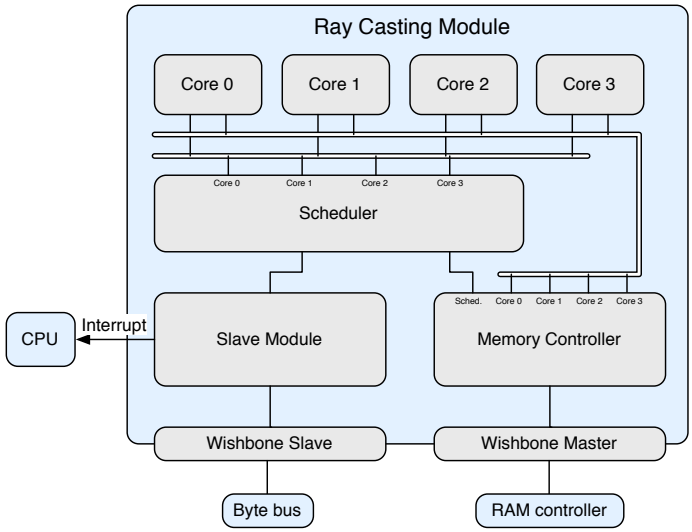


Figure 8.2: Ray casting module overview

8.3 Scheduler

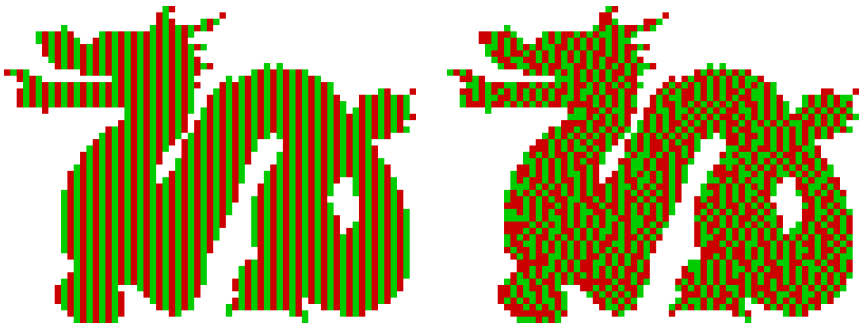
The scheduler has a variety of tasks. In essence it is responsible for initializing the cores and getting the results out to an image, in order to test the functionality of the ray tracing cores. It is designed to perform that function with the least amount of effort, and as such the design of this module should not be considered optimal for a practical ray tracing system.

The scheduler reads parameters from registers in the slave module. These parameters have been set up by software running on the OpenRISC CPU, and include the address of the octree data, the base address of the frame buffer, the address of the ray parameters data and the numbers of rays to trace (i.e., the numbers of pixel in the image). When a start signal is received from the slave module, the scheduler will start iterating through every pixel/ray in the image.

The first task of the scheduler is to load ray parameters from memory and assign a core to perform the tracing of that ray. The ray parameters has been pre-calculated by the software application and transferred to the memory on the FPGA prototype board beforehand. These parameters should ideally have been calculated in hardware, but this was not done in order to save effort for the design of the ray tracing cores. The parameters loaded from memory for each pixel are the *dir-mask* and the root $\mathbf{t_0}$ and $\mathbf{t_1}$ values for the ray. In total seven 32-bit words are loaded from memory for every pixel/ray.

The second task of the scheduler is to find an available core, load it with the ray parameters and send a start signal. Once the core has finished, a color value for the corresponding pixel is calculated. The scheduler will simply color the pixel according to the depth of ray. I.e., the resulting image is a modified Z-buffer. Finally, the color must be written to the frame buffer in memory.

Two ways of writing to the frame buffer was explored. Earlier designs would write each pixel directly to memory after each core had finished. This allowed some flexibility in which order the pixels were written, but was somewhat slow as each individual memory write request takes a significant amount of time. The final designs used a buffer of 24 bytes. This size was chosen as it is divisible by 3 bytes, which is the number of bytes in a pixel, and 4 bytes, which is the width of the memory bus. After eight pixels had been written to the buffer, the whole buffer could be written at once with a six word burst using the linear address increment burst cycle of the Wishbone bus. This solution resulted in improved rendering speed, but as the pixels had to be written in order of increasing memory address, there is no flexibility with regard to which order the pixels are written to memory.



Round-robin	*First-available*
890229 cycles	800803 cycles

Figure 8.3: Scheduling of ray tracing cores and the rendering time in simulation

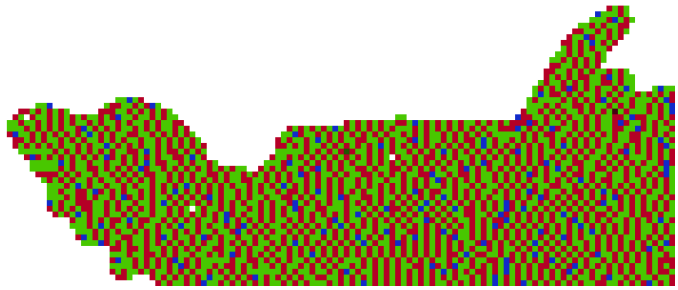


Figure 8.4: First-available scheduling with 3 cores on the FPGA

Two ways of scheduling the cores was explored. A *round-robin* approach was tried, where core 1 would be scheduled first, then core 2, core 3, etc., untill it runs out of cores and loops around to core 1 again. The second approach, *first-available*, would simply look for any core which is not busy and use that. These are illustrated in Figure 8.3 where a pixel is colored by the core that traced it. As different rays require different amount of clock cycles to trace, the first-available approach utilizies cores more efficiently, and so this approach can use fewer clock cycles to render the same image. However, the first-available scheduling technique could not be used in combination with the buffering and

burst writing of pixel values discussed previously. The reason is that if one ray takes a particular long time to trace, once it is finished the pixel buffer could have been flushed to memory. Experiments indicated that the round-robin scheduling in combination with pixel buffering was faster than the first-available scheduling without buffering. A more advanced design could conceivably combine a variation of first-available scheduling with buffering though.

8.4 Memory Controller

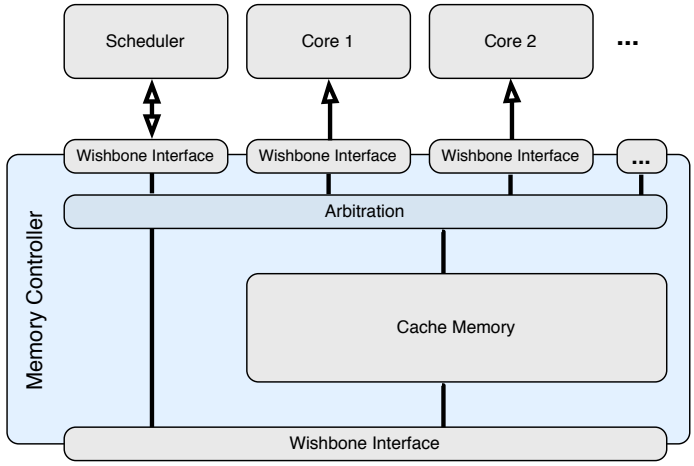


Figure 8.5: Memory controller overview

The memory controller arbitrates requests from the scheduler and the cores. The requests are very different in nature. The scheduler issues both read and write requests, and uses burst transfers to fetch ray data. The cores issues only read requests, and in addition, these requests pass through a cache. The cache is unnecessary for the scheduler, as the data it reads is only read once per frame. The arbitration scheme is a simple fixed-priority scheme, where the scheduler has top priority, followed by core 1, core 2, etc. This implies that if the bus bandwidth is saturated, the upper cores will be starved first. This is assumed to be unproblematic, as we are only interested in the total time it takes to render an image. If a single trace finishes faster at the expense of another it should not impact the overall performance. This starvation effect can be observed in Figure 8.4. Core 3 is colored blue in this picture, and we can observe that this core traces fewer pixels as it uses more time to trace each ray.

The cache unit uses two blocks of RAM on the FPGA. These blocks are instantiated as ordinary Verilog arrays. Although the Spartan-6 FPGA has specialized 18kbit block RAM cells[68], they are slightly more cumbersome to use, but it would be reasonable to use them in more refined versions of the design. One block of RAM is used to store the cached data, and the second is used to store tags indicating the address of the data in the cache. The size of these can theoretically be configured to any power-of-two size. The cache unit can be configured to replace cache entries in blocks of 1, 4, 8 or 16 words, which

is the sizes which are directly supported by the Wishbone bus cycle functionality. With block transfers the size of the tag required to identify a cache block is $32\text{bits} - log_2(b)\text{bits}$, where b is the block size. Both a direct mapped cache and a two-way set associative cache was implemented. The two-way set associative cache uses two sets of data and tag memory.

When a core requests a word from memory, the tag memory is checked to see if the block this words belongs to is resident in the cache. For the two-way set associative cache both sets of tags must be checked. If the data is not present, the block is transferred from memory using the burst cycle transfer functionality of the Wishbone bus. The first address requested by the burst cycle is the word that the core requested, and this word is returned to the core as soon as it is received. After this, the rest of the words in the cache block is transferred and stored in the cache.

8.5 Ray Traversal Core

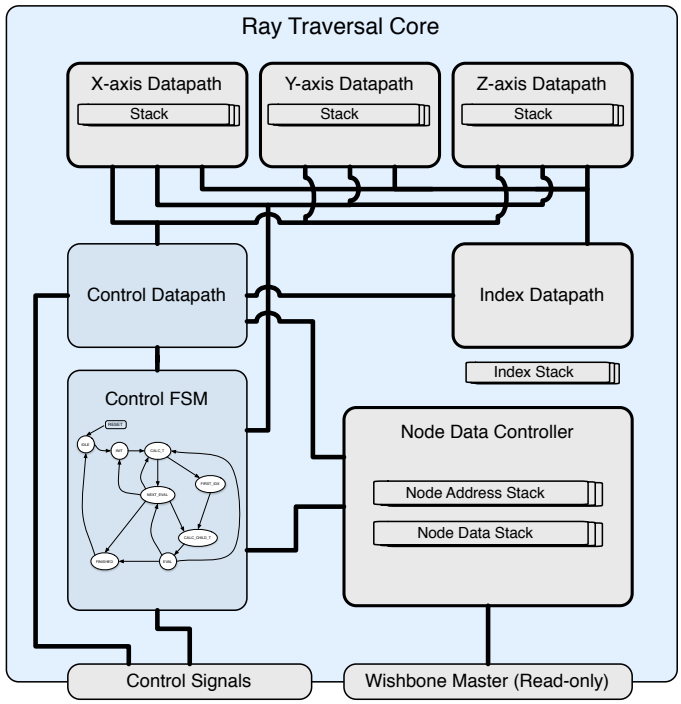


Figure 8.6: Core schematic

The kernel of the ray traversal algorithm was implemented as illustrated in Figure 8.6. The module has two interfaces. A set of control signals lets the scheduler feed the module with the root address, *dir-mask* and the parameters $\mathbf{t_0}$ and $\mathbf{t_1}$. A Wishbone port, modified for read-only operation, gives the module access to the memory. A finite state machine (FSM) controls a collection of datapaths. The data processing related to calculating t_m

and finding $\mathbf{t_0}$ and $\mathbf{t_1}$ for the child nodes, have been seperated into three datapaths – one for each of the X, Y and Z components. These are implemented as a single Verilog module which can be instantiated three times within the core. These also contain the stack of $\mathbf{t_0}/\mathbf{t_1}$ values. The index datapath implements the functions FirstIndex and NextIndex. It outputs the next index, and a flag indicating if the ray has exited the octant or not. The node data controller is tasked with generating memory request to fetch data for a node, and contains the node address and data stacks. It also contains a small state machine to generate the memory requests, since far pointers requires multiple memory requests. The control datapath performs calculations that integrate data between all the other datapaths.

These design choices were made so that it would be easy to know which calculations were performed in which clock cycles, and to attempt to limit the amount of calculations performed within a single clock cycle. The critical path through the combinatorial circuits for a calculation, should pass through no more more than one 32-bit adder/subtracter/comparator, a couple of multiplexers and some basic logic. This should keep the maximum clock frequency on the same level as that of the OpenRISC 32-bit processor, meaning it is both possible and sensible to run the processor and the cores from the same clock.

The size of the stacks can be configured, and even be completely removed. This made it possible to evaluate the impact of stack size on the performance of the ray traversal core. The node address, node data and index stack is instantiated from the same Verilog module. For this module, both a shift-register and a memory-based approach was tried, and it was found that the latter approach used less resources. For the X/Y/Z-axis datapath stacks, a single shift-register was used, as they only required a single bit per stack level, and it was assumed that using an addressed memory would not be beneficial here.

The Verilog code for the core module is included in the appendix.

8.6 Core State Machine

The core state machine implements the logic of the algorithm. It evaluates output from the various calculations and generates control signals to latch the result of calculations to registers or to initialize a pop or push of the stack and request node data from memory. It is illustrated in Figure 8.7, where the conditions for advancing to a state is labeled at the base of the arrows. The task of each state is given below:

- Idle: When the start signal is received, it latches all the parameters given by the scheduler to internal registers.
- Init: Resets all working parameters to the root parameters. Initializes a request for the root data.
- Calc-T: Calculate $\mathbf{t_m}$ and t_{enter}.
- First-Index: Calculate the first child index by latching output from index datapath. Wait for node data request to finish.

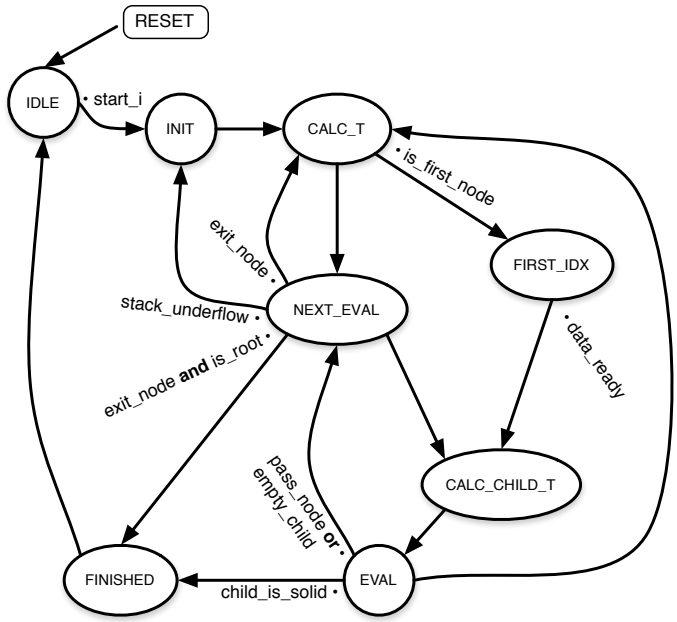


Figure 8.7: Core state machine

- NEXT-EVAL: Calculate the next child index and evaluate which decision take, based on whether the ray exited the current node:
  - Exited the root node: Go to FINISHED, and indicate that we did not hit a leaf.
  - Exited the current node: If there's a stack underflow, restart from the root by going to INIT. Otherwise, go up a level, pop the stack, and go to CALC-T.
  - Otherwise: the ray did not exit the node, latch the next child index and go to CALC-CHILD-T.
- CALC-CHILD-T: Calculate $t_{exit-child}$.
- EVAL: Calculate *pass-node*, evaluate the current child node and
  - If *pass-node*: Go to the next child index.
  - If the node data indicates the child is a solid voxel: Go to FINISHED, and indicate that we hit a leaf.
  - If the node data indicates the child is an internal node: Go down a level, push the stack, initiate a node data request and go to CALC-T.
  - Otherwise: The child must be empty: Go to the next child index.

8.7 Testing

In order to test whether the design worked and to benchmark the performance impact of various design parameters, the following test procedure was used. First, the design was simulated using the free open source verilog simulator "Icarus Verilog". A small test image was rendered, and if there was a problem with the simulation, the waveform viewer Scansion was used to inspect the output of the simulation. If the design passed simulation, it was synthesized using Xilinx ISE.

A small test program was written for the OpenRISC processor. After the FPGA board is reset it will initialize the HDMI output module and display a blank image on the screen. It will wait for a start signal from a physical button or input from the USART port. It will then set up the ray tracer module with its parameters, the ray data address and the octree model address. It starts a clock cycle counter, and send a start signal to the ray tracer module. It will then wait for an interrupt signal from the ray tracer module. Once the interrupt is received, the clock cycle counter is stopped, and its value is printed over the USART port. This value is then used to calculate the time it took to render the image.

In addition to the SVO data, the raytracer module needs data to initialize the ray cores. This data is the framebuffer address of the pixel represented by the ray, the direction mask, and the root t-values root-t_0 and root-t_1. This data is generated by the software implementation and saved to a file. The ray data file and SVO model file is uploaded to the FPGA using a small software tool.

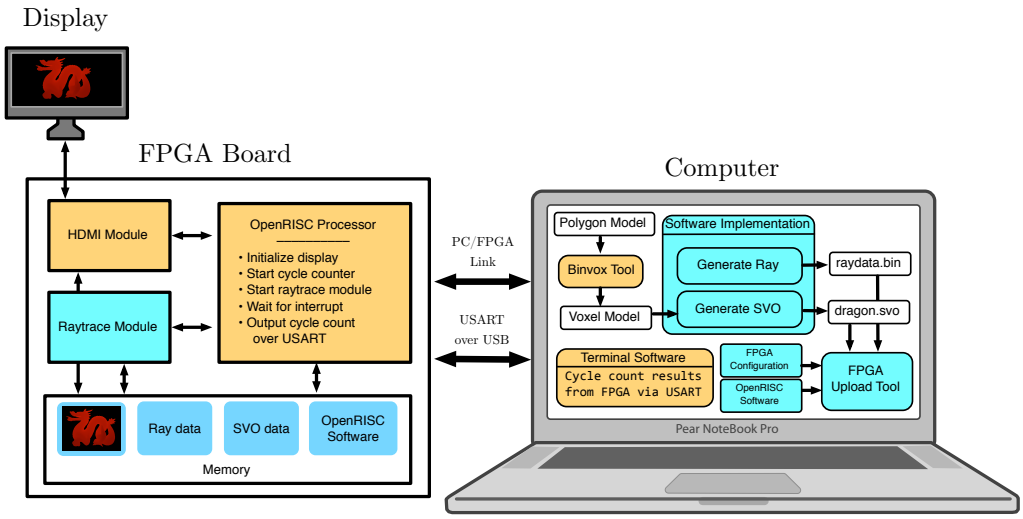


Figure 8.8: Setup for testing the raytracing module on the FPGA. Software/hardware written for this thesis indicated in teal. Third party software/hardware in orange.

There are four design parameters which could be adjusted to generate different permutations of the design. The *core count* is the number of ray tracing cores. That is, the

designed can be synthesized without some of the cores illustrated in Figure 8.2, to generate designs with 1, 2, 3 or 4 cores. The *stack depth* is the depth of the stack as discussed in Chapter 6.3, and can be set to any power of two up to 16. The *cache block size* is the number of words which is transferred to the stack after a cache miss. The *cache size* is the size of the cache in 32-bit words. The block size and cache size are the same parameters as were profiled by the software model in Chapter 7.5. In addition, the version of the cache with a two-way set associative replacement policy was synthesized for some of the permutations.

These four parameters were used to generate a total of 67 permutations of the design with the intent of investigating the impact of the design parameters and their interactions on the performance of the design. For every permutation the design was tested by rendering the same image of the dragon model from a specific angle. The time it took to render the image was logged and inserted into a table of results.

To evaluate the impact on resource use of the various design parameters, five different design with similar rendering speed, but different design parameters was synthesized and tested. The number of resources measured in registers and LUTs was extracted from the reports generated by the Xilinx synthesis tools.

To provide a frame of reference for the generated results, the ray tracing module was executed once without starting the cores. The only task performed by the module was to load the ray parameters from memory, and to write a blank pixel value to memory. This provides the maximum theoretical speed of the design which can be achieved without improving the scheduler, or increasing the clock speed. Furthermore, the total resources used by a typical design has been reported. The maximum memory bandwidth of the memory channel used by the ray tracing module was calculated by testing the time it took to load a large dataset using block cycle transfers.

8.8 Results

Table 8.1: Asorted results and constraints

FPGA:	Spartan-6 XC6SLX45
Available registers:	54 576
Available LUTs:	27 288
System clock speed:	50 MHz
Memory bandwidth:	95 MB/s
Output image resolution:	640 x 480 pixels
Number of pixels/rays:	307 200
Minimum "real-time" speed:	24fps / 42ms
Maximum theoretical speed:	150ms
Time to load ray data:	130ms
Best achieved speed:	303ms / 3.3fps
Best achieved rays / second:	1 013 760
Avg. clock cycles per ray @ 24fps :	7
Avg. clock cycles per ray @ 3.3fps:	50
Typical total resource utilization	
Total number of registers used:	6 919 (12%)
Total number of LUTs used:	12 849 (47%)
OpenRISC CPU registers used:	2 383
OpenRISC CPU LUTs used:	6 695
Ray tracer registers used:	2 294
Ray tracer LUTs used:	3 412
State of the art GPU:	GeForce GTX 680[61]
Release date:	March 22, 2012
Core clock speed:	1 006 MHz
Memory bandwidth:	192 GB/s

Table 8.2: Results from FPGA testing. Smaller numbers are better.

A: Cache block size vs cache size

Cache type:	Direct mapped
Core count:	1
Stack size:	8

Cache size	Block size			
	1	4	8	16
128	741ms	921ms	1085ms	1070ms
512	605ms	696ms	796ms	736ms
1024	582ms	641ms	641ms	651ms

B: Stack size vs cache size

Cache type:	Direct mapped
Block size:	1
Core count:	2

Stack size	Cache size			
	None	128	512	1024
None	1910ms	822ms	646ms	617ms
2	1230ms	566ms	455ms	434ms
4	1090ms	505ms	413ms	394ms
8	1053ms	491ms	403ms	385ms
16	1053ms	490ms	403ms	385ms

C: Stack size vs core count

Cache type:	Direct mapped
Block size:	1
Cache size:	128

Stack size	Core count			
	1	2	3	4
None	1257ms	822ms	728ms	712ms
2	831ms	566ms	526ms	526ms
4	756ms	505ms	473ms	478ms
8	741ms	491ms	458ms	463ms
16	741ms	490ms	457ms	463ms

D: Cache size vs core count

	Cache type:	Direct mapped		
	Block size:	1		
	Stack size:	8		
Cache size	Core count			
	1	2	3	4
None	1044ms	1053ms	1026ms	1052ms
128	741ms	491ms	458ms	463ms
512	605ms	403ms	365ms	358ms
1024	582ms	385ms	346ms	336ms

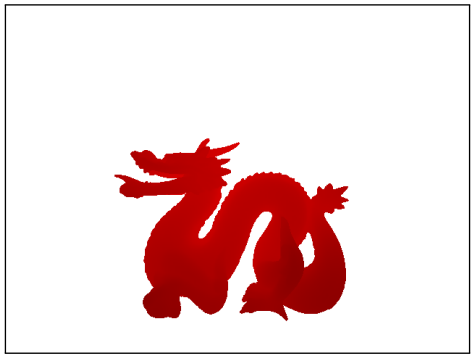
E: Cache size vs core count

	Cache type:	Two-way		
	Block size:	1		
	Stack size:	8		
Cache size	Core count			
	1	2	3	4
128	669ms	446ms	402ms	394ms
512	549ms	365ms	324ms	312ms
1024	539ms	357ms	315ms	303ms

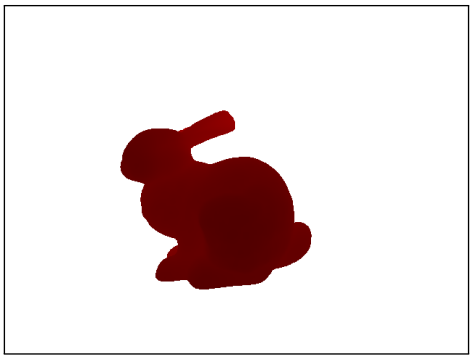
Table 8.3: Resource use of a selection of designs with approximately 440ms render speed.

Cores	Stack size	Cache size	Cache type*	Render speed	Registers	LUTs	LUT-RAM
2	8	128	Two-way	446ms	2358	3357	210
2	8	256	DM	430ms	2294	3257	340
2	2	512	DM	455ms	2258	3541	592
3	8	128	DM	458ms	3078	4221	238
3	2	256	DM	438ms	4353	4353	368

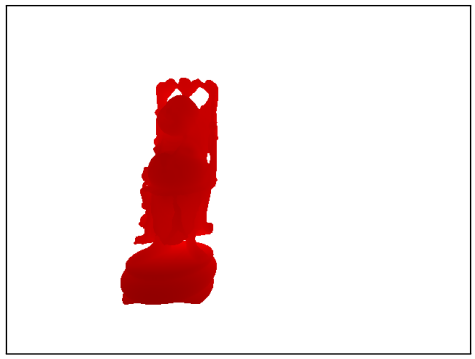
*DM = Direct Mapped, Two-way = Two way set associative



(a) The Stanford Dragon



(b) The Stanford Bunny



(c) The Happy Buddha

Figure 8.9: Rendered images extracted from the FPGA implementation

8.9 Discussion

The test results show that with a direct mapped cache, greater block size does not in fact improve rendering speed, even with large cache sizes (Table 8.2A). The small improvements seen with larger cache sizes in the software model is not apparent here. This can probably be attributed to inaccuracies in the software cache model. It does appear that a 16 word block size is slightly better than an 8 word block size, and one can speculate if even larger block sizes would improve performance further. It is possible that the two way set associative cache would have benefited from larger cache block sizes, but as the the way set associative cache was implemented late in the thesis work, this has not been explored.

The relationships between the stack size and the cache size (Table 8.2B) show that larger stack sizes improve the performance of the design even when the cache size is large. With both no cache and a 512 word cache, the design is almost twice as fast with a 8 level cache compared to no cache. There is no benefit to have a 16 level cache compared to 8 levels, which is not surprising as the octree is only 10 levels deep. There is a low probability that a restart will be necessary when the stack is almost as deep as the octree, and if a restart does occur it is relatively cheap.

Table 8.2C shows that increasing stack size and core counts seems to improve performance mostly independently when in the presence of a small cache. However, the jump from 3 to 4 cores seems to provide no performance benefit. In some instances it even decreases performance, but this difference is so small that no conclusions can be drawn from it. It is probably due to random variations in the rendering speed. The reason why more cores do not necessarily lead to better performance is probably due to congestion on the memory bus. When the memory bus is saturated the cores spend most their time waiting for access to the memory bus.

The same congestion issues can be observed in Table 8.2D, where cache size and core count is contrasted. These results illustrate that the larger the cache – the more cores the memory bus can support. With no cache, adding extra cores does not improve performance. With a cache, adding a second core improves rendering speed significantly. Although adding cores does seem to add some improvement for larger cache sizes, the most dramatic improvement is from 1 to 2 cores.

The same test of cache size vs. core count was done using a two way set associative cache (Table 8.2E). The results indicate a significant improvement across all configurations. This cache seems to improve utilization of the cores without increasing the cache size. This table also shows the design that achieved the best performance in the tests that were performed: a design with 4 cores, 1024 word cache, 1 word cache block size and a two way set associative cache. Larger cache size were not tested due to issues with synthesizing.

Table 8.3 provides some interesting data on the resource use of the different design parameters. For instance, the first two rows illustrate that a two way set associative cache requires more LUTs but fewer LUT-RAMs. The reason is probably that the set associative cache requires an extra comparison unit and some extra logic. On the Spartan-6 FPGA the LUT-RAMs seem to be interchangeable with LUTs, so the version with a two way set associative cache wins on that measure. But it also requires more registers. In

essence, the resource use is very similar between these designs. However, it is possible that the benefit of a two way set associative cache increases as the cache size increases, since the overhead of the two way set associative cache should be somewhat constant.

Looking at the design with a small stack size, it seems clear that it is better to have a large stack size and a smaller cache, than a small stack and a larger cache. It is also clear that among those designs the jump from 2 to 3 cores is not economical.

If we assume the design can achieve real time speed with a 640 x 480 image, by improving the scheduler and the cache system further, then we can try to estimate whether the design should allow real time speeds at higher resolutions when implemented with state of the art integrated circuit technology. A commercial system should be able to render a full HD image at a resolution of 1920 x 1080 pixels, which would imply that we need to trace at least 2 million pixels per frame. As we have seen, memory bandwidth is the critical bottleneck in the design, and it is not unreasonable to assume that the performance scales somewhat linearly with the memory bandwidth. A state of the art GPU has more than 2 000 times more memory bandwidth than our system[61], which could potentially allow us to trace 2 billion rays per second, or 84 million rays per frame at 24 frames per second. This gives us 42 rays per pixel with full HD resolution, which should be enough for a wide range of effects, e.g., multiple light sources, reflections, ambient lighting and anti-aliasing.

To achieve this throughput the number of cores would probably also have to be increased, but it is difficult to evaluate how many cores could be included on a modern GPU. The NVIDIA GeForce GTX 680 GPU contains 1536 CUDA cores[33], which are more capable than the ray tracing cores presented in this thesis. A similar anecdote, is that Laine and Karras [29] achieved 24-170 million rays per second on a modern GPU by using software running on the CUDA cores. If the ray tracing cores are significantly smaller than the CUDA cores, and can perform the ray tracing algorithm faster, then that is also an indication that a GPU could conceivably integrate enough cores to achieve real time speed for high resolutions within a reasonable area.

There are many other factors that could affect this estimate. The GPU will probably require much of the memory bandwidth to load color/texture data. In addition, some games run at 60 frames per second. However, smarter caching and other optimizations could allow the design to be even faster without increasing bandwidth requirements. Furthermore, the SVO model that was rendered in this thesis was only 10 levels deep, but if a game is to use the technique to render a vast landscape, it could require an SVO which is much deeper, and the performance impact of rendering a larger SVO has not been analyzed.

Chapter 9

Conclusions

In this thesis the outline of a practical design for ray tracing of sparse voxel octree in hardware has been presented. The final system approaches real-time speeds at a low clock frequency, and with further work it is possible that this milestone could be passed without increasing the clock frequency. There is much work left before the design is a complete and optimal system for ray tracing of SVOs in hardware. There are also many unknown factors that affect the performance should the design be integrated in a modern GPU. However, the results shows that the technique is promising.

We have seen the value of maintaining a software version of the algorithm to aid in the design of a hardware implementation. It generated input data, implemented parts of the algorithm which had not been done in hardware yet, was used to debug simulations of the hardware and provided data which could inform the choice of design parameters. Most importantly the software version demonstrated that rasterization and ray tracing could potentially be integrated in a hybrid rendering system, which would be essential in a commercially viable design.

The use of the ORPSoC system has shown the utility of open source in hardware design. It provided a complete and functional system for the chosen FPGA prototype board, which greatly simplified the process of creating a ray tracing system. The OpenRISC CPU proved very valuable in debugging the design, and adjusting the functionality of the system on the fly. In addition, the open source FPGALink software/firmware allowed for easy and efficient communication with the board. The verilog simulator was open source as well. In conclusion, almost all the intellectual property used in this thesis was community created open source software and hardware. An added benefit of this was that most of the tools used was cross-platform, allowing development on any of the three big operating systems. Only the commercial Xilinx synthesis tools were limited to just Windows and Linux.

The work on hardware optimizations have shown that the algorithm explored in this thesis can be optimized in ways that are quite specific to hardware. By doing a careful evaluation of the information required to implement the stack for the algorithm, we can drastically reduce the number of bits required for each level of the stack. The results

from the hardware design testing shows the importance of evaluating the impact of design parameters. Using a stack which is at least half the size of the octree is one of the most economical ways of boosting performance. Using a data cache is essential to improving performance, both by improving the speed of a single core, and to support more cores using the same memory bus. Finally, using a two way set associative cache improves the utilization of the cores without increasing cache size.

The future of ray tracing in hardware is still uncertain. First, there is the question of whether one should implement ray tracing of polygon models, as demonstrated by Woop et al. [64], rather than using sparse voxel octrees. It is also conceivable that one could construct a system which was flexible enough to do both. Second, it is possible that the software implementations on GPGPUs, such as that of Laine and Karras [29], will be fast enough on future GPGPUs, and that a dedicated hardware solution will be deemed too inflexible. There are many open questions, and more work is needed before we can conclude whether ray tracing in hardware is truly practical or not.

9.1 Future Work

We have not looked thoroughly at the possible data structures for sparse voxel octrees. As indicated by the experiments, memory bandwidth is crucial to performance, and a more efficient data structure could reduce the impact on bandwidth. The challenge is to create a data structure which can efficiently compress the SVO, while being fast enough to decode in hardware without excessive resources. The data structure would have to incorporate and compress color data as well.

A large scale system for ray tracing of SVOs should also support streaming of voxel data from primary storage, as was demonstrated in software by Crassin et al. [10]. To use SVOs to model a large terrain could require tens of gigabytes if not more, and it is unreasonable to expect to hold all the data in memory at once. Enabling streaming of voxel data will present additional design challenges in both the design of the hardware and the data structure.

There is currently very little research on the animation and deformation of SVOs[5]. Animation of SVOs could increase the number of applications of the technique. To enable this, one should perhaps look at accelerating the *construction* of SVOs in hardware as well. If one could dynamically modify the SVOs in real time one could for instance create tire tracks in the dirt as a car drives across it. This is also something worth considering when researching data structures for SVOs.

If these techniques are to be implemented in dedicated hardware on GPUs, one should try to enable as many uses as possible to justify the cost. Octrees are a space partitioning structure, and as mentioned in the chapter on space partitioning, these structures have many uses in computer games. It could be wise to attempt to design the ray traversal module in a way that enables other uses.

It is clear that the design of the caches for the ray tracing cores has a very large impact on performance. Introducing cache hierarchies could further improve performance. One could introduce a small private cache to each core, have a set of cores share a larger

cache, and have several sets of cores share an even larger one. The interplay between the scheduler and the cache is also interesting. One could investigate a scheduler which starts a set of cores at the exact same time. If the cores are tracing coherent rays they should access some of the same node data at the same time, which could enable additional optimizations. Finally, there could yet be smart optimizations to the cache which could exploit the specific memory access patterns of the ray tracing cores. There is still much research that could be done on this subject.

Bibliography

[1] IEEE Standard for Floating-Point Arithmetic (754-2008), 2008.

[2] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, us ed edition, January 1986.

[3] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Third Edition*. AK Peters, 3 edition, July 2008.

[4] Mohit Arora. *The Art of Hardware Architecture: Design Methods and Techniques for Digital Circuits*. Springer, 2012 edition, October 2011.

[5] Dennis Bautembach. Animated Sparse Voxel Octrees, 2011. URL `http://bautembach.de/wordpress/wp-content/uploads/asvo.pdf`. Retrieved on 2012-06-17.

[6] David Benson, Joel Davis, David Benson, and Joel Davis. Octree textures. In *SIGGRAPH '02 Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 785–790, New York, USA, July 2002. ACM.

[7] J Bittner, V Havran, and P Slavik. Hierarchical Visibility Culling with Occlusion Trees. In *Computer Graphics International*, pages 207–219. IEEE Comput. Soc, 1998.

[8] Mario Botsch, Andreas Wiratanaya, and Leif Kobbelt. *Efficient high quality rendering of point sampled geometry*. Eurographics Association, July 2002.

[9] Per Christensen, Julian Fong, David Laur, and Dana Batali. Ray Tracing for the Movie 'Cars'. In *2006 IEEE Symposium on Interactive Ray Tracing*, pages 1–6. IEEE, 2006.

[10] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 15–22, New York, NY, USA, January 2009.

[11] Digilent Inc. *Atlys Booard Reference Manual*, 2011. URL `http://digilent.us/Data/Products/ATLYS/Atlys_rm.pdf`. Retrieved on 2011-12-16.

[12] Daniel D Gajski, Nikil D Dutt, Allen C-H Wu, and Steve Y-L Lin. *High-Level Synthesis: Introduction to Chip and System Design*. Springer, 1st edition, February 1992.

[13] H Ghasemzadeh, S Mazrouee, and M R Kakoee. Modified pseudo LRU replacement algorithm. In *13th Annual IEEE International Symposium and Workshop on Engineering of Computer-Based Systems (ECBS'06)*, pages 6 pp.–376. IEEE, 2006.

[14] Andrew S Glassner. *An introduction to ray tracing*. Academic Press, 1989.

[15] Enrico Gobbetti, Fabio Marton, and José Antonio Iglesias Guitián. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7-9):797–806, 2008.

[16] Eric Haines and Qingnan Zhou. binvox 3D mesh voxelizer, 2012. URL `http://www.cs.princeton.edu/~min/binvox/`. Retrieved on 2012-05-22.

[17] John L Hennessy and David A Patterson. Memory Hierarchy Design. In *Computer Architecture: A Quantitative Approach, 4th Edition*, pages 288–342. Morgan Kaufmann, September 2006.

[18] Charles Hollemeersch, Bart Pieters, Peter Lambert, and Rik Van de Walle. Accelerating virtual texturing using CUDA. In *GPU Technology Conference*, 2009.

[19] Charles-Frederik Hollemeersch, Bart Pieters, Aljosha Demeulemeester, Frederik Cornillie, Bert Van Semmertier, Erik Mannens, Peter Lambert, Piet Desmet, and Rik Van de Walle. Infinitex: An interactive editing system for the production of large texture data sets. *Computers & Graphics*, 34(6):643–654, December 2010.

[20] H Hoppe. Smooth view-dependent level-of-detail control and its application to terrain rendering. In *Visualization '98*, pages 35–42,. IEEE, 1998.

[21] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive kd tree GPU raytracing. In *Symposium on Interactive 3D Graphics and Games*, Seattle, Washington, 2007.

[22] Fumihiko Ino, Jun Gomita, Yasuhiro Kawasaki, and Kenichi Hagihara. A GPGPU Approach for Accelerating 2-D/3-D Rigid Registration of Medical Images. In Minyi Guo, Laurence Yang, Beniamino Di Martino, Hans Zima, Jack Dongarra, and Feilong Tang, editors, *Parallel and Distributed Processing and Applications*, pages 939–950. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2006.

[23] Arie Kaufman. Volume visualization. *The Visual Computer*, 6(1):1–1, January 1990.

[24] Shoab Ahmed Khan. *Digital Design of Signal Processing Systems: A Practical Approach*. Wiley, 1 edition, February 2011.

[25] Aaron M Knoll, Ingo Wald, and Charles D Hansen. Coherent multiresolution isosurface ray tracing. *The Visual Computer*, 25(3):209–225, March 2008.

[26] Isreal Koren. *Computer arithmetic algorithms*. Ak Peters Series. A K Peters, 2002.

[27] Stefan Kristiansson. ORPSoC on Digilent Atlys, 2012. URL `http://www.chokladfabriken.org/projects/orpsoc-atlys`. Retrieved on 2012-06-16.

[28] Ian Kuon, Russell Tessier, and Jonathan Rose. FPGA Architecture: Survey and Challenges. *Foundations and Trends® in Electronic Design Automation*, 2(2): 135–253, 2007.

[29] Samuli Laine and Tero Karras. Efficient Sparse Voxel Octrees. *Symposium on Interactive 3D Graphics and Games*, February 2010.

[30] Samuli Laine and Tero Karras. Efficient Sparse Voxel Octrees – Analysis, Extensions, and Implementation, February 2010. URL `http://code.google.com/p/efficient-sparse-voxel-octrees/`.

[31] Marc Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[32] Chris McClelland. FPGALink, 2011. URL `http://www.makestuff.eu/wordpress/?page_id=1400`. Retrieved on 2011-12-15.

[33] NVIDIA Corporation. Nvidia geforce gtx 680, 2012. URL `http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf`. Retrieved on 20212-06-16.

[34] NVIDIA Corporation. High Performance Computing, 2012. URL `http://www.nvidia.com/object/tesla-supercomputing-solutions.html`. Retrieved on 2012-05-18.

[35] OpenCores. *Wishbone B4 - Wishbone System-on-chip (SoC) Interconnection Architecture for Portable IP Cores*. OpenCores, 2011. URL `http://opencores.org/opencores,wishbone`. Retrieved on 2011-12-15.

[36] OpenCores. *OpenRISC 1200 IP Core Specification (Preliminary Draft)*. OpenCores, 2012. URL `http://opencores.org/openrisc,or1200`. Retrieved on 2012-06-15.

[37] Sudeep Pasricha and Nikil Dutt. *On-Chip Communication Architectures: System on Chip Interconnect (Systems on Silicon)*. Morgan Kaufmann, 1 edition, May 2008.

[38] Jingliang Peng, C.-C. Jay Kuo, Jingliang Peng, and C.-C. Jay Kuo. Geometry-guided progressive lossless 3D mesh coding with octree (OT) decomposition. *ACM Transactions on Graphics (TOG)*, 24(3):609–616, July 2005.

[39] Naila Rahman. Algorithms for Hardware Caches and TLB. In Ulrich Meyer, Peter Sanders, and Jop Sibeyn, editors, *Lecture Notes in Computer Science*, pages 171–192. Springer Berlin Heidelberg, Berlin, Heidelberg, February 2003.

[40] S. Raje and R. A. Bergamaschi. *Generalized resource sharing*. IEEE Computer Society, November 1997.

[41] J. Revelles, C. Ureña, and M. Lastra. An Efficient Parametric Algorithm for Octree Traversal. *Journal of WSCG*, pages 212–219, January 2000.

[42] Kristof Römisch. Sparse voxel octree ray tracing on the GPU. Master's thesis, Aarhus University, 2009.

[43] Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J. Paul, and Philipp Slusallek. Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, page 95, New York, New York, USA, 2004. ACM Press.

[44] Jörg Schmittler, Daniel Pohl, Tim Dahmen, Christian Vogelgesang, and Philipp Slusallek. Realtime ray tracing for current and future games. In *ACM SIGGRAPH 2005 Courses*, page 23, New York, New York, USA, 2005. ACM Press.

[45] Ruwen Schnabel and Reinhard Klein. Octree-based Point-Cloud Compression. In *Eurographics Symposium on Point-Based Graphics (2006)*. Institut für Informatik II, Universität Bonn, Germany, 2006.

[46] D Sheldon, R Kumar, F Vahid, D Tullsen, and R Lysecky. Conjoining Soft-Core FPGA Processors. In *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, pages 694–701, 2006.

[47] Ryan Shrout. John Carmack on id Tech 6, Ray Tracing, Consoles, Physics and more, 2008. URL `http://www.pcper.com/reviews/Graphics-Cards/John-Carmack-id-Tech-6-Ray-Tracing-Consoles-Physics-and-more`. Retrieved on 2012-06-15.

[48] Ryan Shrout. John Carmack Interview: GPU Race, Intel Graphics, Ray Tracing, Voxels and more!, 2011. URL `http://www.pcper.com/reviews/Editorial/John-Carmack-Interview-GPU-Race-Intel-Graphics-Ray-Tracing-Voxels-and-more`. Retrieved on 2012-06-15.

[49] John Spackman and Philip Willis. The SMART navigation of a ray through an oct-tree. *Computers & graphics*, 15(2):185–194, 1991.

[50] Stanford Computer Graphics Laboratory. Scanning and surface reconstruction, 2012. URL `http://graphics.stanford.edu/data/3Dscanrep/`. Retrieved on 2012-05-22.

[51] Nilo Stolte and René Caubet. Discrete Ray-Tracing of Huge Voxel Spaces. *Computer Graphics Forum*, 14(3):383–394, August 1995.

[52] K R Subramanian and D S Fussell. Applying Space Subdivision Techniques to Volume Rendering. In *First IEEE Conference on Visualization: Visualization '90*, pages 150–159,. IEEE Comput. Soc. Press, 1990.

[53] TSB Sudarshan and RA Mir. Highly efficient LRU implementations for high associativity cache memory. In *Proceedings of 12th ...*, 2004.

[54] Kelvin Sung. A DDA Octree Traversal Algorithm for Ray Tracing. In *Eurographics Conference Proceedings 1991*, 1991.

[55] L Szirmay-Kalos, V Havran, and B Balázs. On the efficiency of ray-shooting acceleration schemes. In *Proceedings of the 18th ...*, 2002.

[56] Branislav Síleš. Atomontage Engine - Home, 2012. URL `http://www.atomontage.com/`. Retrieved on 2012-05-18.

[57] JMP Van Waveren. Van Waveren: id Tech 5 Challenges-From Texture Virtualiza... - Google Scholar. In *SIGGRAPH 2009: ...*, 2009.

[58] Kyu-Young Whang, Ju-Won Song, Ji-Woong Chang, Ji-Yun Kim, Wan-Sup Cho, Chong-Mok Park, and Il-Yeol Song. Octree-R: an adaptive octree for efficient ray tracing. *IEEE Transactions on Visualization and Computer Graphics*, 1(4):343–349, 1995.

[59] Wikipedia. Rasterization. URL `http://en.wikipedia.org/wiki/Rasterisation`. Retrieved on 2012-06-17.

[60] Wikipedia. Geforce 500 series, 2012. URL `http://en.wikipedia.org/wiki/GeForce_500_Series`. Retrieved on 2012-03-22.

[61] Wikipedia. Geforce 600 series, 2012. URL `http://en.wikipedia.org/wiki/GeForce_600_Series`. Retrieved on 2012-06-16.

[62] Wikipedia. Nvidia tesla, 2012. URL `http://en.wikipedia.org/wiki/Nvidia_Tesla`. Retrieved on 2012-05-18.

[63] Wayne Wolf. *Computers as Components, Second Edition: Principles of Embedded Computing System Design*. Morgan Kaufmann, 2 edition, June 2008.

[64] Sven Woop, Jörg Schmittler, and Philipp Slusallek. RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. *ACM SIGGRAPH 2005 Papers*, pages 434–444, April 2005.

[65] Xilinx Inc. *LogiCORE IP Adder/Subtracter v11.*, 2011. URL `http://www.xilinx.com/support/documentation/ip_documentation/addsub_ds214.pdf`. Retrieved on 2012-03-22.

[66] Xilinx Inc. *LogiCORE IP Floating-Point Operator v5.0*, 2011. URL `http://www.xilinx.com/support/documentation/ip_documentation/floating_point_ds335.pdf`. Retrieved on 2012-03-22.

[67] Xilinx Inc. *LogiCORE IP Multiplier v11.2*, 2011. URL `http://www.xilinx.com/support/documentation/ip_documentation/mult_gen_ds255.pdf`. Retrieved on 2012-03-22.

[68] Xilinx Inc. *Spartan-6 Family Overview*, 2011. URL `http://www.xilinx.com/support/documentation/data_sheets/ds160.pdf`. Retrieved on 2011-12-16.

Appendix A

Attached Files

```
raycaster/    -- The Verilog source of the ray caster module
   raycast_core_axis_dp.v    -- The datapath of the per-axis calculations
   raycast_core_idx.v        -- The datapath of the index calculations
   raycast_core_master.v     -- The memory request unit of the core
   raycast_core.v            -- The ray casting core
   raycast_ctrl.v            -- The scheduler / control unit
   raycast_master.v          -- The ray caster memory interface / cache
   raycast_slave.v           -- The slave interface
   raycaster.v               -- The ray caster top level module
oppgave.pdf   -- This report
orpsoc/     -- HDL, Software and Makefiles for the full ORPSoC-based system
sim/     -- The bench for simulation of the hardware module
out/     -- The source and synthesis output of the designs
data/    -- The octree and ray data used to test the designs
Octree2/  -- The software application
orlink/
    hw/  -- HDL code and Sim. bench for orlink
    sw/  -- CLI software to communicate with orlink
```

Appendix B

The Software Ray Tracing Core Functions

```
    RayCastOutput rayCast(Ray r)
    {
        Ray r2 = r;
        int dir_mask=0;

//        if (fabs(r.o.x)>o_max ||
//            fabs(r.o.y)>o_max ||
//            fabs(r.o.z)>o_max)
//            throw;

        if (r.d.x==0)
            r.d.x+=0.001;
        if (r.d.x<0.0) {
            r.o.x = -r.o.x;
            r.d.x = -r.d.x;
            dir_mask |= 4;
        }
        if (r.d.y==0)
            r.d.y+=0.001;
        if (r.d.y<0.0) {
            r.o.y = -r.o.y;
            r.d.y = -r.d.y;
            dir_mask |= 2;
        }
        if (r.d.z==0)
            r.d.z+=0.001;
        if (r.d.z<0.0) {
            r.o.z = -r.o.z;
            r.d.z = -r.d.z;
            dir_mask |= 1;
        }

        float tx0, tx1, ty0, ty1, tz0, tz1;

        tx0 = (-1-r.o.x)/r.d.x;
        tx1 = (1-r.o.x)/r.d.x;
```

```
        ty0 = (-1-r.o.y)/r.d.y;
        ty1 = (1-r.o.y)/r.d.y;
        tz0 = (-1-r.o.z)/r.d.z;
        tz1 = (1-r.o.z)/r.d.z;

        if (debug_ray) {
//             printf("Init:\n\t%f %f %f\n\t%f %f %f\n",
//                    tx0, ty0, tz0,
//                    tx1, ty1, tz1);
            printf("\tdir_mask: %d\n", dir_mask);
        }

//        float t_min = min(tx0,min(ty0,tz0));
//        float t_max = max(tx1,max(ty1,tz1));
//        float t_absmax = max(abs(t_min),abs(t_max));

        float t_enter = max(max(tx0,ty0),tz0);
        float t_exit = min(min(tx1, ty1), tz1);


        tx0 *= tnum_scale; ty0 *= tnum_scale; tz0 *= tnum_scale;
        tx1 *= tnum_scale; ty1 *= tnum_scale; tz1 *= tnum_scale;


        tnum_clip(&tx0);
        tnum_clip(&ty0);
        tnum_clip(&tz0);
        tnum_clip(&tx1); tnum_clip(&ty1); tnum_clip(&tz1);

        assert(tx0<tnum_max && tx0>tnum_min);
        assert(ty0<tnum_max && ty0>tnum_min);
        assert(tz0<tnum_max && tz0>tnum_min);

        assert(tx1<tnum_max && tx1>tnum_min);
        assert(ty1<tnum_max && ty1>tnum_min);
        assert(tz1<tnum_max && tz1>tnum_min);

        if (debug_ray)
            printf("%.8x %.8x %.8x\n%.8x %.8x %.8x\n\n",
                   (tnum)tx0, (tnum)ty0, (tnum)tz0,
                   (tnum)tx1, (tnum)ty1, (tnum)tz1);

        if (t_enter < t_exit) {
            OctreeCursor cursor = ot->newRootCursor();
            cursor.cache = cache;

            if (exportMode){
                exportRay(tx0,ty0,tz0,
                          tx1,ty1,tz1,
                          dir_mask);
                return RayCastOutput(0,0);
            }
            else
                return rayCastCore(r2,
                             tx0,ty0,tz0,
                             tx1,ty1,tz1,
                             dir_mask,
                             cursor);
        }
        else {
```

```
        if (exportMode)
            exportRay(0,0,0,0,0,0, (1<<31));
    }
    if (debug_ray) printf("End.\n");
    return RayCastOutput(0,false);
}



RayCastOutput rayCastCore(Ray ray,
                tnum r_tx0, tnum r_ty0, tnum r_tz0,
                tnum r_tx1, tnum r_ty1, tnum r_tz1,
                int dir_mask,
                OctreeCursor r_cur)
{

    int level;
    int idx, idx_flip;
    bool is_first_node;
    bool exits_node;
    bool pass_node;
    tnum t_start = 0;
    tnum t_enter;
    tnum t_exit_child;

    tnum tx0, ty0, tz0;
    tnum tx1, ty1, tz1;
    tnum txm, tym, tzm;
    tnum tx0_child, ty0_child, tz0_child;
    tnum tx1_child, ty1_child, tz1_child;

    bool is_leaf;
    tnum t_out;

    OctreeCursor cur;

S_INIT:
    cur = r_cur;

    tx0 = r_tx0; ty0 = r_ty0; tz0 = r_tz0;
    tx1 = r_tx1; ty1 = r_ty1; tz1 = r_tz1;

    is_first_node = true;
    level = 0;
    exits_node = false;

    stack->reset();

    if (debug_ray)
        printf("Init - t_start: %.8x\n",  t_start);
    goto S_CALC_T;


S_CALC_T:
    txm = (tx0+tx1)/2;
    tym = (ty0+ty1)/2;
    tzm = (tz0+tz1)/2;

    t_enter = max(tx0,max(ty0,tz0));
```

```
    if (debug_ray)
        printf("t_enter: %.8x\n",  t_enter);

    if (is_first_node)
        goto S_FIRST_IDX;
    else
        goto S_NEXT_IDX;

S_FIRST_IDX:
    idx = (txm < t_enter)<<2 | (tym < t_enter)<<1 | (tzm < t_enter);
    exits_node = false;
    is_first_node = false;
    if (debug_ray)
        printf("FrstIdx: %d / %d\n", idx, idx^dir_mask);
    goto S_DATA_WAIT;

S_DATA_WAIT:
    // if (got_data)
    if (debug_ray)
        printf("Data: %.8x : %.8x\n", cur.adr*4, *((int*)cur.node));
    goto S_CALC_CHILD_T;



S_NEXT_IDX:
    exits_node = false;
    if (t_exit_child == tx1_child) {
        if (idx & 4)
            exits_node = true;
        else
            idx = idx ^ 4;
    }
    else if (t_exit_child == ty1_child) {
        if (idx & 2)
            exits_node = true;
        else
            idx = idx ^ 2;
    }
    else {
        if (idx & 1)
            exits_node = true;
        else
            idx = idx ^ 1;
    }
    if (debug_ray)
        printf("NextIdx: %d / %d   exit: %d\n", idx, idx^dir_mask, exits_node);
    goto S_NEXT_EVAL;

S_NEXT_EVAL:
    if (exits_node) {
        if (level==0) {
            is_leaf = false;
            goto S_FINISHED;
        }
        else if (stack->empty()) {
            if (debug_ray)
                printf("Underflow\n\n");
            restart_count++;
            t_start = t_exit_child;
```

```
                goto S_INIT;
            }
            else {
                if (debug_ray) printf("Pop\n");
                stack->pop(&cur,
                           &tx0, &ty0, &tz0,
                           &tx1, &ty1, &tz1,
                           &idx);
                level--;
                goto S_CALC_T;
            }
        }
        else {
            goto S_CALC_CHILD_T;
        }

    S_CALC_CHILD_T:
//        if (debug_ray) printf("Idx %d / %d\n", idx, idx ^ dir_mask);
        tx0_child = (idx&4) ? txm : tx0;
        tx1_child = (idx&4) ? tx1 : txm;
        ty0_child = (idx&2) ? tym : ty0;
        ty1_child = (idx&2) ? ty1 : tym;
        tz0_child = (idx&1) ? tzm : tz0;
        tz1_child = (idx&1) ? tz1 : tzm;

        t_exit_child = min(tx1_child,min(ty1_child,tz1_child));

        goto S_EVAL;

    S_EVAL:
        idx_flip = idx ^ dir_mask;
        pass_node = t_exit_child <= t_start;
        if (pass_node) {
            goto S_NEXT_IDX;
        }
        else if (cur.childIsFilled(idx_flip) || level==terminLevel) {
            tx0 = tx0_child; ty0 = ty0_child; tz0 = tz0_child;
            tx1 = tx1_child; ty1 = ty1_child; tz1 = tz1_child;
            is_leaf = true;
            goto S_FINISHED;
        }
        else if (cur.childIsValid(idx_flip)) {
            if (debug_ray) {
//                printf("%.8x %.8x %.8x\n%.8x %.8x %.8x\n",
//                       tx0, ty0, tz0, tx1, ty1, tz1);
                if (cur.childPtrIsFar())
                    printf("Far %x\n", cur.farPtr());
                printf("Push\n\n");
            }
            if (cur.childPtrIsFar()) {
                far_count++;
            }
            push_count++;
            stack->push(cur,
                        tx0, ty0, tz0,
                        tx1, ty1, tz1,
                        idx);

            tx0 = tx0_child; ty0 = ty0_child; tz0 = tz0_child;
            tx1 = tx1_child; ty1 = ty1_child; tz1 = tz1_child;
```

```
        level++;
        is_first_node = true;

        cur = cur.getChild(idx_flip);

        goto S_CALC_T;
    }
    else {
        goto S_NEXT_IDX;
    }

S_FINISHED:
    t_enter = max(tx0,max(ty0,tz0));
    t_out = is_leaf ? t_enter : t_exit_child;
    if (debug_ray) {
     printf("Leaf: %d %.8x\n\n\n", is_leaf, t_out);
    }
    return RayCastOutput((float)t_out/(float)tnum_scale, is_leaf);
}
```

Appendix C

The Ray Tracing Core Module

```
// TODO: Implement error when level overflows

`include "raycast_defines.v"

module raycast_core
    (
        clk, rst,
        start_i,

        root_adr_i,
        dir_mask_i,
        tx0_i, ty0_i, tz0_i,
        tx1_i, ty1_i, tz1_i,

       m_wb_adr_o,  m_wb_dat_i,
        m_wb_cyc_o, m_wb_stb_o,
       m_wb_ack_i,

        finished_o, leaf_o, t_o, level_o
    );

    // = Parameters =
    parameter dw = 32; // Data width
    parameter stack_size = 8;
    parameter stack_size_log2 = 3;
    parameter max_lvl = 32;
    parameter max_lvl_log2 = 5;

    // --

    // = Ports =
    input clk;
    input rst;
    input start_i;
    input [31:0] root_adr_i;
    input [2:0] dir_mask_i;
    input [dw-1:0] tx0_i, ty0_i, tz0_i;
```

```
 input [dw-1:0] tx1_i, ty1_i, tz1_i;

 // WISHBONE master
output [31:0]    m_wb_adr_o;
input [31:0]        m_wb_dat_i;
output           m_wb_cyc_o;
output           m_wb_stb_o;
input               m_wb_ack_i;

 output reg finished_o;    reg finished_o_n;
 output reg leaf_o;      reg leaf_o_n;
 output [dw-1:0] t_o;
 output [max_lvl_log2-1:0] level_o;
// --

 // = States =
 parameter S_IDLE = 0, S_INIT = 1,
     S_CALC_T = 2, S_FIRST_IDX = 3,
     S_NEXT_EVAL = 4, S_CALC_CHILD_T = 5, S_EVAL = 6, S_FINISHED = 7;
 // --

 parameter EXIT_X = 2'b01, EXIT_Y = 2'b10, EXIT_Z = 2'b00;

 // = Registers/Memories =

 reg [31:0] root_adr, root_adr_n;
 reg [2:0]  dir_mask, dir_mask_n;

 reg signed [dw-1:0] tx0_root, tx0_root_n;
 reg signed [dw-1:0] ty0_root, ty0_root_n;
 reg signed [dw-1:0] tz0_root, tz0_root_n;

 reg signed [dw-1:0] tx1_root, tx1_root_n;
 reg signed [dw-1:0] ty1_root, ty1_root_n;
 reg signed [dw-1:0] tz1_root, tz1_root_n;

 reg [2:0]                 state,     state_n;
 reg [max_lvl_log2-1:0]     level;

 reg [1:0]                expl_child, expl_child_n; // Child exit plane
 reg signed [dw-1:0]        t_exit_child, t_exit_child_n;
 reg signed [dw-1:0]        t_enter, t_enter_n;
 reg signed [dw-1:0]     t_start, t_start_n;

 reg [2:0]     idx,                idx_n;
 reg [2:0]    idx_prev,        idx_prev_n;
 reg         is_first_node, is_first_node_n;

 // --

 // = Wires/Aliases =
 wire signed [dw-1:0] tx0;
 wire signed [dw-1:0] ty0;
 wire signed [dw-1:0] tz0;

 wire signed [dw-1:0] txm;
 wire signed [dw-1:0] tym;
 wire signed [dw-1:0] tzm;

 wire signed [dw-1:0] tx1;
```

```
wire signed [dw-1:0] ty1;
wire signed [dw-1:0] tz1;

wire signed [dw-1:0] tx1_child;
wire signed [dw-1:0] ty1_child;
wire signed [dw-1:0] tz1_child;

reg tem_latch;

wire [2:0] idx_next;
wire exit_node;

reg t_enter_calc;
reg t_exit_child_calc;
reg pass_node_calc;

wire signed [dw-1:0] t_enter_next;
wire signed [dw-1:0] t_exit_child_next;
wire [1:0] expl_child_next; // Child exit plane

// = Calculations =

assign t_enter_next =
    (tx0>=ty0 && tx0>=tz0)    ? tx0 :
    (ty0>=tz0)                 ? ty0 : tz0;

assign expl_child_next[0] =
    (tx1_child <= ty1_child) && (tx1_child <= tz1_child);

assign expl_child_next[1] =
    !expl_child_next[0]  && (ty1_child <= tz1_child);

assign t_exit_child_next =
    expl_child_next[0] ? tx1_child :
    expl_child_next[1] ? ty1_child : tz1_child;

wire pass_node = (t_exit_child <= t_start);

wire node_is_root = (level==0);
wire [2:0] idx_flip    = idx ^ dir_mask;

assign t_o = leaf_o ? t_enter : t_exit_child;

// Stack signals
reg init;
reg push;
reg pop;
wire stack_empty;
wire [2:0] idx_stack_o;

// Master signals
reg node_data_req;
wire [15:0] node_data;
wire node_data_ready;
// Decode node data
wire [7:0] valid_mask         = node_data[7:0];
wire [7:0] leaf_mask         = node_data[15:8];

wire child_is_solid = leaf_mask[idx_flip];
wire child_is_valid = valid_mask[idx_flip];
```

```
// --

// = Datapath Instances =
raycast_core_axis_dp x_axis_dp
    (
        .clk       (clk),
        .rst       (rst),
        .init_i    (init),
        .push_i    (push),
        .pop_i     (pop),
        .tem_latch_i (tem_latch),
        .idx_i       (idx[2]),
        .idx_prev_i (idx_prev[2]),
        .te0_i     (tx0_root),
        .te1_i     (tx1_root),
        .te0_o     (tx0),
        .tem_o     (txm),
        .te1_o     (tx1),
        .te1_child_o (tx1_child)
    );
defparam x_axis_dp.dw = dw;
defparam x_axis_dp.stack_size = stack_size+1;

raycast_core_axis_dp y_axis_dp
    (
        .clk       (clk),
        .rst       (rst),
        .init_i    (init),
        .push_i    (push),
        .pop_i     (pop),
        .tem_latch_i (tem_latch),
        .idx_i (idx[1]),
        .idx_prev_i (idx_prev[1]),

        .te0_i     (ty0_root),
        .te1_i     (ty1_root),
        .te0_o     (ty0),
        .tem_o     (tym),
        .te1_o     (ty1),
        .te1_child_o (ty1_child)
    );
defparam y_axis_dp.dw = dw;
defparam y_axis_dp.stack_size = stack_size+1;

raycast_core_axis_dp z_axis_dp
    (
        .clk       (clk),
        .rst       (rst),
        .init_i    (init),
        .push_i    (push),
        .pop_i     (pop),
        .tem_latch_i (tem_latch),

        .idx_i (idx[0]),
        .idx_prev_i (idx_prev[0]),

        .te0_i     (tz0_root),
        .te1_i     (tz1_root),
        .te0_o     (tz0),
```

```
        .tem_o     (tzm),
        .te1_o     (tz1),
        .te1_child_o (tz1_child)
    );
defparam z_axis_dp.dw = dw;
defparam z_axis_dp.stack_size = stack_size+1;


raycast_core_idx idx_next_dp
    (
        .is_first_i (is_first_node),
        .idx_i (idx),
        .txm_i (txm),
        .tym_i (tym),
        .tzm_i (tzm),
        .t_enter_i (t_enter),
        .exit_plane_child_i (expl_child),

        .idx_next_o(idx_next),
        .is_exit_o(exit_node)
    );
defparam idx_next_dp.dw = dw;

raycast_core_master core_master
    (
        .clk                (clk),
        .rst                (rst),
       .m_wb_adr_o          (m_wb_adr_o),
        .m_wb_dat_i         (m_wb_dat_i),
        .m_wb_cyc_o         (m_wb_cyc_o),
        .m_wb_stb_o         (m_wb_stb_o),
       .m_wb_ack_i          (m_wb_ack_i),
        .init_i             (init),
        .pop_i              (pop),
        .root_adr_i         (root_adr),
        .idx_flip_i         (idx_flip),
        .node_data_req_i    (node_data_req),
        .node_data_ready_o  (node_data_ready),
        .node_data_o        (node_data)
    );
defparam core_master.stack_size = stack_size;
defparam core_master.stack_size_log2 = stack_size_log2;

// --

// = Control FSM =
always @(*)
begin
    root_adr_n = root_adr;
    dir_mask_n = dir_mask;
    tx0_root_n = tx0_root;
    ty0_root_n = ty0_root;
    tz0_root_n = tz0_root;
    tx1_root_n = tx1_root;
    ty1_root_n = ty1_root;
    tz1_root_n = tz1_root;

    state_n = state;

    expl_child_n = expl_child;
```

```
t_exit_child_n = t_exit_child;
t_enter_n = t_enter;
t_start_n = t_start;

idx_prev_n = idx_prev;
idx_n = idx;
is_first_node_n = is_first_node;

leaf_o_n = leaf_o;
finished_o_n = finished_o;

t_enter_calc = 0;
t_exit_child_calc = 0;
pass_node_calc = 0;
tem_latch = 0;

init = 0;
push = 0;
pop = 0;
node_data_req = 0;

case (state)
    S_IDLE: begin // 0
        if (start_i) begin
            root_adr_n = root_adr_i;
            dir_mask_n = dir_mask_i;
            tx0_root_n = tx0_i;
            ty0_root_n = ty0_i;
            tz0_root_n = tz0_i;
            tx1_root_n = tx1_i;
            ty1_root_n = ty1_i;
            tz1_root_n = tz1_i;
            t_start_n  = 0;
            finished_o_n = 0;
            state_n = S_INIT;
        end
    end
    S_INIT: begin // 1

        // Fetch root data
        // Set level = 0
        // Reset stack
        init = 1;
        is_first_node_n = 1;
        state_n = S_CALC_T;
    end
    S_CALC_T: begin // 2
        // tm <= (t0+t1)/2 (in axis_dp)
        tem_latch = 1;

        // t_enter <= min(t0)
        t_enter_calc = 1;
        t_enter_n = t_enter_next;

        if (is_first_node)
            state_n = S_FIRST_IDX;
        else
            state_n = S_NEXT_EVAL;
    end
    S_FIRST_IDX: begin // 3
```

```
                if (node_data_ready) begin
                    idx_n = idx_next;
                    is_first_node_n = 0;
                    state_n = S_CALC_CHILD_T;
                end
            end
            S_NEXT_EVAL: begin
                if (exit_node) begin
                    // We've exited current octant.
                    if (node_is_root) begin
                        // Exited octree, we're finished
                        leaf_o_n = 0;
                        state_n = S_FINISHED;
                    end
                    else if (stack_empty) begin
                        // Stack underflow, restart from root
                        t_start_n = t_exit_child;
                        state_n = S_INIT;
                    end
`ifndef RAYC_DISABLE_STACK
                    else begin
                        // Go up a level. Pop the stack.
                        pop = 1;
                        idx_prev_n = idx_stack_o;
                        idx_n = idx_prev;
                        state_n = S_CALC_T;
                    end
`endif
                end
                else begin
                    idx_n = idx_next;
                    state_n = S_CALC_CHILD_T;
                end
            end
            S_CALC_CHILD_T: begin
                t_exit_child_calc = 1;
                t_exit_child_n = t_exit_child_next;
                expl_child_n = expl_child_next;
                state_n = S_EVAL;
            end
            S_EVAL: begin
                pass_node_calc = 1;
                if (pass_node) begin
                    // Proceed to next child voxel
                    state_n = S_NEXT_EVAL;
                end
                else if (child_is_solid)    begin
                    // We hit a voxel
                    // Push to get child parameters
                    push = 1;
                    leaf_o_n = 1;
                    state_n = S_FINISHED;
                end
                else if (child_is_valid) begin
                    // We hit an octant containing something
                    // Push and get next node
                    // level <= level+1
                    push = 1;
                    idx_prev_n = idx;
                    is_first_node_n = 1;
```

```
                node_data_req = 1;

                state_n = S_CALC_T;
            end
            else begin
                // Empty octant, evaluate next child node
                state_n = S_NEXT_EVAL;
            end
        end
        S_FINISHED: begin
            // Calculate t_enter for child
            t_enter_calc = 1;
            t_enter_n = t_enter_next;

            finished_o_n = 1;
            state_n = S_IDLE;
        end

    endcase
end
// --

// = Registers =

always @(posedge clk)
begin
    if (rst) begin
        root_adr <= 32'd0;
        dir_mask <= 3'd0;
        tx0_root <= 32'd0;
        ty0_root <= 32'd0;
        tz0_root <= 32'd0;
        tx1_root <= 32'd0;
        ty1_root <= 32'd0;
        tz1_root <= 32'd0;
        state    <= 3'd0;
        level    <= 0;
        expl_child <= 2'd0;
        t_exit_child <= 32'd0;
        t_enter <= 32'd0;
        t_start  <= 32'd0;
        idx_prev    <= 3'd0;
        idx      <= 3'd0;
        is_first_node <= 0;
        finished_o <= 1;
        leaf_o <= 0;
    end
    else
    begin
        root_adr    <=    root_adr_n;
        dir_mask    <=    dir_mask_n;

        tx0_root    <=    tx0_root_n;
        ty0_root    <=    ty0_root_n;
        tz0_root    <=    tz0_root_n;

        tx1_root    <=    tx1_root_n;
        ty1_root    <=    ty1_root_n;
        tz1_root    <=    tz1_root_n;
```

```
            state    <=    state_n;

            expl_child <= expl_child_n;
            t_exit_child <= t_exit_child_n;
            t_enter <= t_enter_n;
            t_start    <=    t_start_n;

            idx    <=    idx_n;
            idx_prev <= idx_prev_n;
            is_first_node    <=    is_first_node_n;

            finished_o <= finished_o_n;
            leaf_o <= leaf_o_n;

            if (init)
                level <= 0;
            else if (push)
                level <= level + 1;
            else if (pop)
                level <= level - 1;
            else
                level <= level;
        end
    end


    // --

    // = Stacks =
`ifndef RAYC_DISABLE_STACK

    reg [stack_size_log2:0] stack_depth;
    wire stack_full = (stack_depth==stack_size);
    assign stack_empty = (stack_depth==0);


    always @(posedge clk)
        if (init) begin
            stack_depth <= 0;
        end
        else if (push) begin
            if (!stack_full)
                stack_depth <= stack_depth + 1;
        end
        else if (pop) begin
            stack_depth <= stack_depth - 1;
        end

    raycast_stack idx_stack_inst
        (
            .clk (clk),
            .push (push),
            .pop (pop),
            .data_i (idx_prev),
            .data_o (idx_stack_o)
        );
        defparam idx_stack_inst.dw = 3;
        defparam idx_stack_inst.depth = stack_size; ///-1
        defparam idx_stack_inst.depth_log2 = stack_size_log2;
`else
```

```
    assign stack_empty = 1;
`endif
    // --


endmodule
```