TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

MATIAS KOSKELA

SOFTWARE-BASED RAY TRACING FOR MOBILE DEVICES

Master of Science thesis

Examiners: D.Sc. Pekka Jääskeläinen &
Prof. Jarmo Takala
Examiners and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on 4th February 2015

ABSTRACT

MATIAS KOSKELA: Software-Based Ray Tracing for Mobile Devices
Tampere University of Technology
Master of Science thesis, 51 pages
August 2015
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiners: D.Sc. Pekka Jääskeläinen & Prof. Jarmo Takala
Keywords: Ray Tracing, Bounding Volume Hierarchy, Compression, Half-Precision Floating-Point

Ray tracing is a way to produce realistic images of three dimensional virtual scenes. It scales more to the number of pixels in the image than to the amount of details in the scene. This makes it an interesting application for mobile systems, which in general have smaller screens.

Modern high-performance ray tracing depends on special acceleration data structures such as bounding volume hierarchies. Compressing the size of the bounding volume hierarchy leads to smaller memory bandwidth usage. This should be especially beneficial for mobile systems, which in general have smaller memory bandwidth. Compression also reduces cache misses and memory usage. Unfortunately, compression reduces the quality of the data structure, leading the ray traversal into unnecessary computations. In addition, compression increases the amount of work which needs to be carried out in the performance critical inner loop.

The previous work on bounding volume hierarchy compression concentrates on inferring some of the coordinates from other coordinates or using different integer precisions. This thesis concentrates on using half-precision floating-point numbers, which have potential due to their greater dynamic range. If the halfs are too inaccurate for use as plain world coordinates, they can be used with hierarchical encoding. This restores the quality of the data structure back to original, but it requires even more work in the inner loop.

Halfs reduce the whole memory usage by 7% and cache misses by 16%. Furthermore, they reduce power usage by 1.7%. The halfs' effect on the performance is heavily dependent on the targeted hardware's support for them. If decompression of the halfs is too slow, they will have a negative impact. Compared to integers, halfs have better performance in the so-called teapot-in-a-stadium problem.

TIIVISTELMÄ

MATIAS KOSKELA: Ohjelmistopohjainen säteenjäljitys mobiililaitteille
Tampereen teknillinen yliopisto
Diplomityö, 51 sivua
Elokuu 2015
Tietotekniikan koulutusohjelma
Pääaine: Pervasive Systems
Tarkastajat: TkT Pekka Jääskeläinen & prof. Jarmo Takala
Avainsanat: Säteenjäljitys, Rajaustilavuushierarkia, Pakkaus, 16-bittinen Liukuluku

Säteenjäljityksellä tuotetaan realistisia kuvia kolmiulotteisista virtuaalimaailmoista. Säteenjäljityksen nopeus riippu enemmän kuvan koosta, kuin virtuaalimaailman yksityiskohtien määrästä. Tämän takia se on kiinnostava vaihtoehto mobiilialustoilla, joilla yleisesti ottaen on käytössä pienemmät näytöt.

Tavallisesti säteenjäljitystä kiihdytetään erityisillä kiihdytysrakenteilla, kuten rajaustilavuushierarkialla. Rajaustilavuushierarkian tiedon tiivistäminen johtaa pienempään muistikaistan käyttöön, jonka pitäisi olla erityisen hyödyllistä yleisesti vähemmän muistikaistaa omaavilla mobiililaitteilla. Lisäksi tiivistäminen vähentää välimuistihuteja ja muistin käyttöä. Valitettavasti tiivistys huonontaa kiihdytysrakenteen laatua, joka johtaa rakenteen turhaan läpikäyntiin. Se myöskin lisää työtä suorituskyvyn kannalta tärkeimmässä sisäsilmukassa.

Aikaisempi tutkimus rajaustilavuushierarkioiden tiivistämisessä rajoittuu arvojen päättelyyn toisista arvoista ja eri tarkkuisten kokonaislukujen käyttöön. Tässä työssä tarkastellaan 16-bittisten liukulukujen käyttöä, joka on kiinnostavaa suuremman dynaamisen alueen ansiosta. Jos 16-bittiset liukuluvut ovat liian epätarkkoja suoraan maailman koordinaatteina, voidaan niitä käyttää hierarkisesti. Tämä palauttaa tarkkuuden alkuperäiselle tasolle, mutta lisää entisestään laskentaa sisäsilmukkaan.

Säteenjäljittimessä kokonaismuistinkulutus vähenee 7% ja välimuistihudit vähenevät 16%, kun käytetään 16-bittisiä liukulukuja. Lisäksi tehon kulutus pienenee 1,7%. 16-bittisten liukulukujen vaikutus säteenjäljityksen nopeuteen on erittäin riippuvainen laitteiston tuesta niille. Suorituskykyä ei voida parantaa, jos 16-bittisten liukulukujen purkaminen on liian hidasta. Kokonaislukuihin nähden liukuluvut toimivat paremmin, jos virtuaalimaailmassa on enemmän yksityiskohtia yksittäisellä alueella.

PREFACE

This thesis was done as a part of the ray tracing research carried out in the *Customized Parallel Computing* (CPC) group at the *Tampere University of Technology* (TUT). Most of the research described in this thesis was carried out by the author. However, the use of half-precision floating-points in ray tracing was Timo Viitanen's idea and he also did the practical measurements that are described in Section 6.3. The hierarchical encoding explained in Subsection 5.2.1 was pointed out by Ken Cameron.

All images of 3D worlds in this thesis are made using the author's own ray tracer, unless mentioned otherwise. In addition, the 3D version of TUT-logo, used in example images, was made by the author, based on the original printed TUT-logo. Other models used in the example images were made by Martin Newell, Anat Grynberg, Greg Ward, Frank Meinl, Marko Dabrovic, Guedis Cardenas, Morgan McGuire and Stanford 3D Scanning Repository.

I would not have managed without the help and comments of my supervisors Pekka Jääskeläinen and Jarmo Takala. In addition, Timo Viitanen's ray tracing and Heikki Kultala's technical comments were very helpful. Furthermore, I would like to thank the entire CPC group for fruitful discussions and comments.

I would also like to thank Toimi Teelahti, Adrian Benfield and Ken Cameron for helping with grammatical corrections.

Finally, I would like to thank my wife for all her support. Without it, graduating from the university would have been impossible.

Tampere, 20.7.2015

Matias Koskela

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AABB	Axis-Aligned Bounding Box
ALU	Arithmetic Logic Unit
BIH	Bounding Interval Hierarchy
BVH	Bounding Volume Hierarchy
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DLP	Data-Level Parallelism
GPU	Graphics Processing Unit
HLBVH	Hierarchical Linear Bounding Volume Hierarchy
ILP	Instruction-Level Parallelism
k-d tree	k-dimensional tree
LBVH	Linear Bounding Volume Hierarchy
LoD	Level of Detail
MBVH	Multi Bounding Volume Hierarchy
nD	n Dimensional
OpenCL	Open Computing Language
OpenMP	Open Multi-Processing
SAH	Surface Area Heuristic
SBVH	Split Bounding Volume Hierarchy
SIMD	Single Instruction, Multiple Data
SIMT	Single Instruction, Multiple Thread
SoC	System on a Chip
TLP	Thread-Level Parallelism
TUT	Tampere University of Technology

1. INTRODUCTION

Images and videos produced by computer graphics are a crucial part of our lives today. They are used in several engineering applications, such as computer-aided design, simulation and promotion, as well as in entertainment including computer gaming, movie special effects and photo manipulation.

There are two main principles for drawing *three-dimensional* (3D) computer graphics. The usual way is to rasterize 3D objects defined as a cluster of triangles. *Rasterization* of a triangle means setting the colors of those pixels that the triangle is contributing to. This method is sometimes called the conventional rendering pipeline. With the pipeline, rendering can be done quickly and in parallel. That is why it has been the most common way to draw 3D scenes. Additionally the majority of today's graphics hardware is optimized for rasterization computation [PH11, Appendix A]. However there are effects, such as depth-of-field, reflection and refraction, which are hard to achieve using the pipeline [AMHH08, Kos13]. Such effects are easily achieved with the other main 3D rendering method of tracing rays. In fact *ray tracing* is much closer to the way our eyes observe our surroundings. In a virtual 3D world, ray tracing simulates how photons would travel in the physical world. The main difference between rasterization and ray tracing is the data used in the main loop. Rasterization is done for each triangle and ray tracing is done for each pixel. In other words, rasterization calculates in front of which pixels each triangle is and ray tracing calculates which triangles are in front of each pixel.

The problem with ray tracing is that it is not cost-effective with regards to computation power. This is the reason for the ongoing search for algorithms, which would make ray tracing achieve real-time frame rates. There are already some frameworks, which claim to be fast enough [WWB*14, PBD*10, LSL*13]. Unfortunately, they are not able to achieve the same frame rates and resolutions as the conventional rendering pipeline. In particular, the performance is lower with more realistic rendering results which are the main reason for using ray tracing.

The frame rate of ray tracing is more dependent on the numbers of pixels than the number of details in the scene. This makes it ideal for mobile devices, which in

general have lower screen resolutions than laptop and desktop computers. Other characteristics of mobile devices are low power usage requirement and the smaller memory bandwidth. The low power usage requirement comes firstly from the fact that mobile devices run on battery power. The more power the system uses, the less time the battery lasts. Unfortunately, in the era of big displays and high-speed connections, the backlight of the display and the radio use most of the power and the other parts of the systems need to cope with what is left [CH10]. The other source for the low power requirement is that the mobile devices cannot overheat as much as desktop devices can, since it is hard for the user to use too hot handheld devices. Furthermore, active cooling of the device would require more power, which would make the battery die even faster.

One way to save power is to use low-precision data types. Both the storage of fewer bits and the computations using them can be done with lower power usage. This requires that the hardware has support for the data type. Unfortunately, compressed data is usually less accurate, which needs to be taken into account somehow. Either the user needs to bear with the lower quality or the data needs to be recovered to a level where it is indistinguishable from the original data.

This thesis focuses on optimizing ray tracing and in particular it describes what influences compressing the data structures have. This is expected to be especially beneficial on mobile systems, due to the smaller memory bandwidth.

The previous work in this field focuses on inferring numbers from other numbers or using fixed-point representation. This thesis compresses a commonly used data structure, the bounding volume hierarchy, using half-precision floating-point numbers. Floating-point numbers are interesting compared to integers due to their greater dynamic range. Additionally, some mobile devices have native support for half-precision floating-point numbers and they can be better suited for floating-point calculations than integer calculations.

Sophisticated techniques to achieve complex light effects are left out of the thesis to keep it focused on the optimization and the compression. Nevertheless, the principles of different ray tracing algorithms are explained briefly in Chapter 2, which also describes implementation of a simple ray tracer. Common parallel hardware architectures used for accelerating ray tracing are described in Chapter 3. General optimization strategies, that can be applied to the most of the ray tracing applications, are explained in Chapter 4. Chapter 5 describes how the bounding volume hierarchy can be compressed with half-precision floating-point numbers. Chapter 6 evaluates the proposed method and finally Chapter 7 concludes the thesis.

2. RAY TRACING

Ray tracing can be divided into four main categories: *ray casting*, *Whitted-style ray tracing*, *distributed ray tracing* and *physically based rendering*. These categories have different capabilities of producing complex effects. In general, more complex effects can be produced by using categories that shoot more rays. Fortunately, ray traversal can be optimized in a similar manner regardless of from which category the ray tracing algorithm is used.

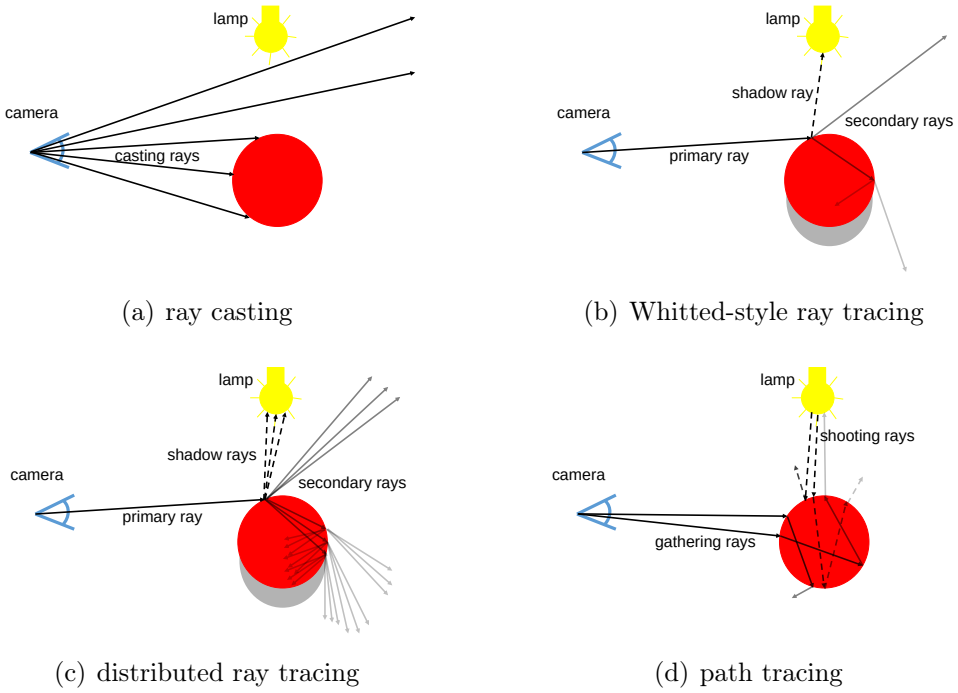
2.1 Ray Casting

The idea of ray casting is to send one ray for each pixel from the camera. Each ray finds the closest intersection with the scene, which can be seen in Fig. 2.1(a). With this information, similar shading to conventional rasterization pipeline can be achieved. Ray casting is non-recursive ray tracing. It is a special sub-category of ray tracing in which only the first rays are traced from the camera, without any reflections or refractions [Bou05, p. 6].

Some volume rendering algorithms are also called ray casting. Those algorithms trace only the first rays from the camera, but they trace them through all of the geometry in the whole scene. This way they are able to find out all the intersections and calculate how much each volume affects the color of the pixel.

2.2 Whitted-Style Ray Tracing

In Whitted-style ray tracing, once the closest intersection is found, new *secondary rays* are sent to the directions of reflection and refraction [Whi79]. These directions need to be examined only if the properties of the material state so. Spawning new rays means that algorithm works recursively. The final color of the pixel is the weighted sum of all the rays. The weights are decided based on the material properties. For example, a perfect mirror material would only use the result of the ray sent from the mirror surface to the reflection direction.



(a) ray casting

(b) Whitted-style ray tracing

(c) distributed ray tracing

(d) path tracing

Figure 2.1 *The concepts of different ray tracing styles. For visualization reasons only few rays are shown.*

In addition to the reflection and the refraction rays, algorithms are usually using shadow rays. These are rays which are sent from the found intersect position to the light source to check if there are any objects between them. If there is anything between the position and the light source that means that the position is not affected by the light. These concepts are visualized in Fig. 2.1(b).

If the scene contains just a single point light, a typical Whitted-style ray tracer sends one to three new rays from the closest intersection point of each ray. This is continued recursively until the material of the intersected primitive states that it does not need any more extra rays or some kind of hard threshold for recursion depth is met. This threshold could be, for example, a limit in the recursion depth. Another example is a lower limit in the recursion ray's weight.

2.3 Distributed Ray Tracing

Distributed ray tracing is an advanced version of Whitted-style ray tracing. The difference is that instead of just one reflection and one refraction secondary ray distributed ray tracing might sent more of them based on the material properties [CPC84]. This is visualized in Fig. 2.1(c) . The result of this is a more realistic image with effects, such as, soft shadows, fuzzy reflections, depth-of-field and motion blur.

Soft shadows can be achieved by sampling an area light using multiple shadow rays. The brightness of a point in the 3D scene is decided by how many of shadow rays are able to see the light. Soft transition on the shadow edges is achieved because in those areas some of the shadow rays are blocked by shadow casting objects and some are not.

Similarly to soft shadows, fuzzy reflections can be done by sending multiple secondary rays to varied directions and calculating an average of them. Depth-of-field is a result of averaging multiple samples of primary rays from different origins. In contrast, motion blur is a result of averaging multiple samples in time.

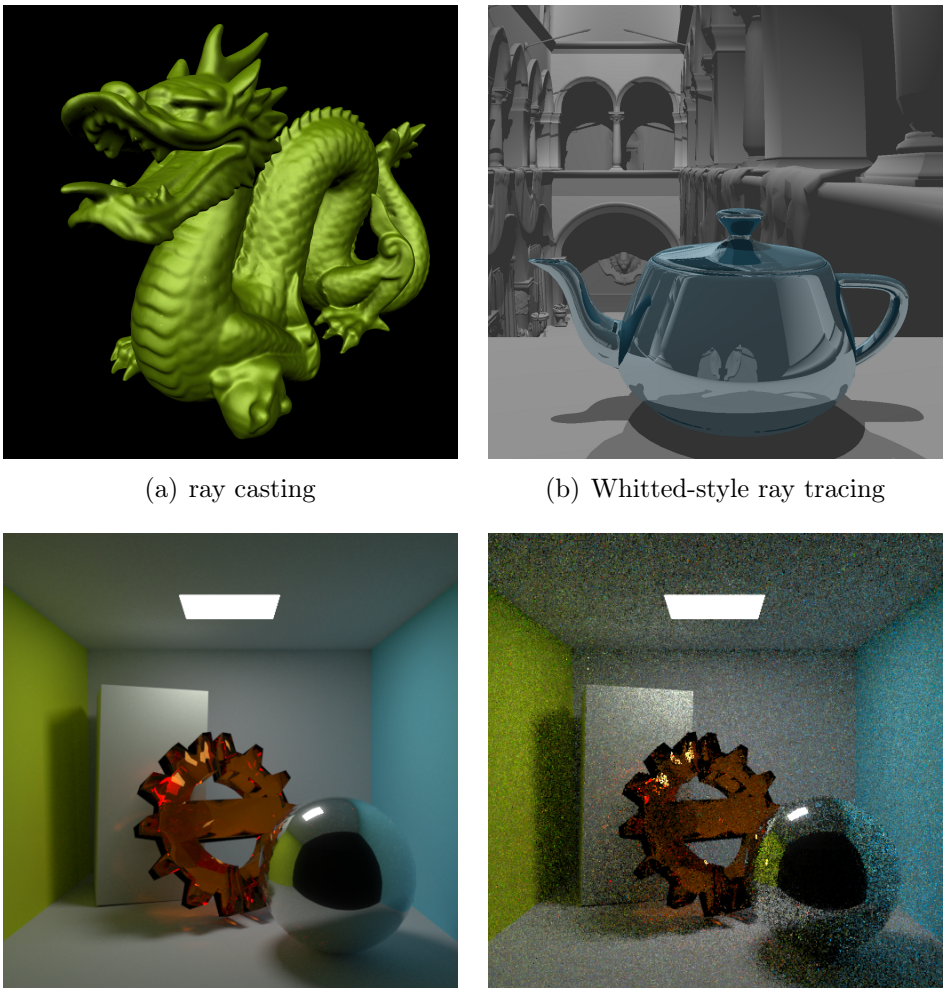
2.4 Physically Based Rendering

The human eye measures the amount of photons coming from different directions. The paths of a single photon or a group of them can be easily simulated with rays. In general, physically based rendering tries to solve the so-called *rendering equation*. The result of the equation is the total amount of light emitted in to the viewer's direction from a point in 3D space [Kaj86]. This is calculated based on the incoming light and material properties.

There are many algorithms that try to solve the rendering equation. One of them is called *path tracing*. Path tracing software can use rays starting from each light source or rays starting from the camera. The third option is to use a combination of both of these approaches.

Shooting rays are sent from the light sources of the 3D world. Every time a shooting ray intersects something, it is randomly reflected or refracted. Instead of completely random directions, the distribution is weighted so that more rays are sent in the actual reflection direction. The distribution weights are decided based on the material properties. For example, a mirror material sends rays primarily towards the reflection direction. This is repeated until the ray intersects the imaginary camera or exits the scene area. Every time a ray finds the camera it adds photons to the pixels. Based on the photons, the image starts slowly emerge. The more rays are traced, the better image quality is. Of course, there is one great problem with this technique; it takes a lot of computation time to produce even small images. As stated above the rays are going in random directions and, therefore, they are unlikely to find the camera.

The simplest way to optimize shooting rays is to reverse the ray direction. Instead of tracing rays from the lights, they are traced from the camera. It is more likely to find



(a) ray casting (b) Whitted-style ray tracing

(c) path tracing (rendered in hours) (d) path tracing (rendered in seconds)

Figure 2.2 *Samples of different ray tracing styles. Path tracing images were rendered with LuxRenderer [Lux15].*

one of the many lights than to find the only camera. Additionally, the lights can even be area lights that cover areas instead of just one point. Intersection with an area is more likely than with a point. Rays starting from the camera are called *gathering rays*, because they gather the information of what they intersect and set the color of pixel based on that. Just like the direction of the shooting ray, also gathering ray's direction is decided based on material dependent weighted distribution. Examples of shooting rays and gathering rays can be seen in Fig. 2.1(d).

Path tracing can be made faster by combining the two basic path tracing approaches. For example in *photon mapping* the shooting rays distribute light to the scene. Then gathering rays gather some of this distributed light. Finally, the color is decided based the gathered light. On the other hand, in *bidirectional path tracing* one gathering ray and one shooting ray are traced at the same time. After every found

intersection, tests are performed to see if any of the gathering ray's intersections can see any of the shooting ray's intersections. If the rays can see each other, the amount of light which is going along the clear path can be calculated.

Examples of image quality of three ray tracing categories can be seen in Fig. 2.2. Ray caster's (Fig. 2.2(a)) colors are only darker on the parts that are pointing away from the light and there are no shadows cast. For example, the inside of the dragon's mouth should be in the shadow of the upper jaw. In Whitted-style ray tracing (Fig. 2.2(b)) there are shadows and reflections (it also supports transparent objects). These are possible because of the shadow rays and recursive secondary rays. There are two point lights in the scene. This can be observed from the two hard shadows on the floor under the teapot. The maximum recursion count in this example was one, which can be seen in the spout's reflection on the teapot which is not reflecting like the actual spout is. In addition to basic recursive ray effects, path tracing (Fig. 2.2(c)) supports caustics and soft shadows which make the image much more realistic. A caustic effect can be seen under the orange glass TUT-logo where there are some brighter areas. An example of the soft shadows can be seen on the wall next to the white box. In the other path tracing image (Fig. 2.2(d)) the result of tracing only the first rays can be seen. In the image there are bright spots which are results of the ray randomly finding the light very quickly. After many more rays the actual color emerges.

2.5 A Basic Ray Caster

In this section a basic ray casting algorithm is demonstrated. The basic ray casting software is straightforward to implement. First, the software needs some kind of ray-object intersection test. The objects of the 3D scene are either loaded from a file or procedurally generated. Then, one ray is sent from every pixel. Each ray is tested against every object with the intersection test, and the color is chosen based on the material at the closest intersection point. Naturally, it is important that the intersection tests are as optimized as possible because the software is going to have to compute an enormous number of them.

A ray-object test can be any kind of 3D ray intersection test, but whatever kind of test is selected it will limit the kind of objects that can exist in the 3D world. For example, if the test is a ray-ball test, there can be only balls in the scene. Of course there can be several different tests in the software to enable support for multiple object types. The most common test is the ray-triangle test, because 3D object design tools usually save their objects as triangles.

3D objects are designed using design software which saves the resulting triangles to a file. At the start of the execution, the ray casting software reads the file and stores the objects in its memory. Another way to do this is to procedurally generate objects, but it is hard work to achieve interesting and detailed objects this way. While procedural generation often requires complicated math, most design tools can be used intuitively by artistic people. After the objects have been loaded, the software initializes the rays.

The ray casting starts by deciding the directions in which the primary rays are to be sent. Usually they start from one point which is designated as the imaginary camera. The direction of each is set based on the pixel that the ray is contributing to. The directions of the rays contributing to the upper half of the image are turned upwards and those contributing to the lower half are turned downwards. The further the pixel is from the center of the image the more the ray is turned. The same idea is used for the right and left halves of the image. The maximum turn angle is in accordance with the simulated focal length of the camera. The greater the maximum angle, the more of a fish-eye effect there is in the resulting image. A smaller maximum angle replicates a greater zoom on the camera lens. Once the direction has been decided the rays need to be traced.

Tracing means finding the intersection points of the ray with the objects in the 3D world. It might be the case that all of the intersections are required, or that only the closest intersection is needed, or even only the knowledge of whether the ray intersects anything at all. For example, in basic ray casting only the closest intersection is needed, and the color is decided based on the material of the closest object. The most obvious way to find the closest intersection is to test the ray against each object in the 3D world. However, with bigger worlds this is impractical to compute.

If only the closest intersection is wanted, it might seem a good idea to sort all the objects based on their distance from the camera and then to test them in order, from the closest to the farthest. Unfortunately, this sorting is still too computationally heavy, because the camera might move and turn between the frames, so the sorting needs to be done all over again. In addition, if the ray does not intersect anything, the software is still testing for intersections with every object. That is why special data structures are used for ray tracing. These structures divide the 3D space into smaller sections. In this way, the rays can avoid doing the intersection test with an object that is far from the ray. The most common of these structures are introduced in Section 4.1. Nevertheless, before optimizing the ray tracing code it is important to know common hardware architectures which execute it.

3. COMMON COMPUTER ARCHITECTURES FOR PARALLEL COMPUTING

Before reaching the so-called *power wall*, faster processing was done by increasing the clock rate of a processor [PH11, p. 39]. The power wall means that increasing the clock rate increases the power usage and temperature too much. Notably a higher clock rate requires more voltage and power is proportional to the voltage squared.

In order to work around the power wall, different ways of parallel execution have become more common ways of speeding up processing. In general, parallelism can be divided into three main categories: *Thread-Level Parallelism* (TLP), *Data-Level Parallelism* (DLP) and *Instruction-Level Parallelism* (ILP). This chapter describes these different parallel computation architectures. They can be used singly or combined to accelerate ray tracing. Chapter 4 presents how some of these accelerations are done.

3.1 Thread-Level Parallelism

A thread is a part of software executing on its own and having its own program counter. The difference between a process and a thread is that the process has its own memory address space, while many threads share the same. Furthermore, a thread can be thought to be a path of execution within a process and there can be multiple threads in a process. Multiple threads can be used for dividing the work of a single process to multiple processors on a *multiprocessor* or for so-called hardware *multithreading* on a single processor.

The multiprocessor is a chip which has multiple processing cores, which can run multiple threads in parallel. The only connection between the threads is the memory that is shared between the processors. There can be so-called *critical areas* of execution in a software, which require that the data in the common memory is not read or written by any other thread while the critical area is executed. For handling these areas processors have special techniques such as locks. A Lock can be checked

in one cycle and, if it is free, it is given to the checker. If the lock was not free, the checker will wait and check the lock again after some time. When the thread is leaving the critical area it needs to free the lock so that others can access the area. If locks are not handled correctly, it is possible to achieve a so-called deadlock. In deadlock a thread is waiting forever for a lock that is never going to be freed. Usually this causes the program to eventually freeze.

Hardware multithreading means that many threads share the functional units of a single processor [PH11, p. 645–648]. The main advantage of using multiple threads on a single processor is that they can hide costly stalls. Stalls occur when something is blocking the process from continuing the execution. For example, the process is waiting for data to be fetched from memory or it is waiting for a user input. The hardware can in that case change to a different thread automatically, which is called *coarse-grained multithreading*. The other option is that the hardware is changing threads all the time and skipping threads which are currently stalling. This can be called *fine-grained multithreading* if the hardware is changing the thread on every cycle or *simultaneous multithreading* if resources are dynamically scheduled to the threads which need them. Modern desktop processors utilize simultaneous multithreading.

Multithreading processors have the same programming challenges as multiprocessors. Both multiprocessors and hardware multithreading are commonly used in modern *Central Processing Units* (CPU) and *Graphics Processing Units* (GPU), including mobile CPUs and GPUs.

3.2 Data-Level Parallelism

Single Instruction, Multiple Data (SIMD) is an example of data level parallelism. Additionally, *Single Instruction, Multiple Thread* (SIMT) is another example of data level parallelism, but it also resembles thread level parallelism.

3.2.1 Single Instruction, Multiple Data

SIMD means that a single instruction performs the same operation for multiple pieces of data. This can be thought to be the same as piecewise vector computations. For that reason SIMD instructions are also commonly referenced as vector instructions. Contemporary desktop processor hardware supports 256-bit SIMD operations. This can be divided into, for example, eight 32-bit floating point calculations or four 64-bit calculations. In addition, not all of the bits need to be used,

which means that narrower vectors can be computed. The current trend is that new generations of SIMD extensions double the amount of bits each instruction can compute simultaneously.

Basically, a programmer can order the processor to fetch, for example, four numbers from memory and perform the same computation for all of them with a single instruction. This will lead to almost four times the speed, although loading four times as much data could slow it down. Even if the hardware has a capability for fetching all of the vector data at once, all of the data needs to be there in time. If a piece of the data is missing, all of the pieces have to wait for it to be loaded from slower levels of caches. Usually, the miss of some data can be avoided with proper data layout for the cache lines. If the computations are done one by one, other pieces of the data can be loaded while the first pieces are computed. Nevertheless, the computation itself is around four times faster than with a scalar processing unit performing same computations on four pieces of data.

With SIMD instructions the data streams are synchronized, which means that there is no need for complicated concurrence handling such as locks. The downside is that it can be impossible for some algorithms to have the type of data parallelism which allows them to be computed with vector instructions.

In addition to speed improvements, SIMD advantages include reduced program memory size, reduced need for control logic and reduced need for checking so-called hazards (both data and control) [PH11, p. 648–653]. With SIMD there is no need to store and fetch the same program multiple times for each piece of data, which saves power significantly. Additionally, the same control logic can control multiple *Arithmetic Logic Units* (ALU), because they are all doing the same work just for different data.

SIMD is commonly used in all kinds of CPUs and in GPUs including mobile devices. In addition, some GPUs use the SIMT execution model, which can include SIMD instructions.

3.2.2 Single Instruction, Multiple Thread

The difference between SIMT and SIMD is that SIMT applies the same instruction to multiple threads as well as to multiple pieces of data [LNOM08]. In SIMT, a group of threads, a so-called *warp*, is executed all at once in a SIMD like execution unit. Each single thread can even execute SIMD instructions, but depending on the hardware, this might reduce the warp size.

The downside of SIMT is that if there is any branch divergence in the program the execution can handle only one branch at the time. This is done by running all of the commands in all of the branches to all of the threads, but masking out those that should not actually execute the branch. If there is a single if – else statement with equally long execution paths on both branches (and at least one thread takes the other branch than the other threads), only 50% of the benefits are achieved [PH11, p. A-29]. This means that heavily branching code likely runs faster on a scalar processor than a SIMT processor.

Some of the divergence could be avoided by reordering the warps so that as many warps as possible contain only threads that are executing the same branch or similar code. The easiest way to achieve this is if the programmer can reorder the data so that the ones that take the same branch are sent to the same warp. Otherwise, especially for small branches this might be too complicated for the hardware to manage.

This kind of execution model is especially beneficial for applications where the same small program, without much branch divergence, is executed for many different pieces of data. Examples of these kind of programs are graphics programs in other words the *shaders*, which do the same computations for every primitive vertex or pixel. The difference between each primitive vertex and each pixel is that their data is different and, therefore, the same small program needs to be run for every one of them individually. This is the reason why many desktop GPUs and some mobile GPUs utilize the SIMT execution model.

A common SIMT architecture has long latencies for each instruction [PH11, p. A-29–A-30]. The latency can be hidden by using hardware multithreading (described in Section 3.1). This means that there can be even more threads running in parallel. Today's SIMT-based GPUs can theoretically have thousands of active threads running in parallel. The amount of actual threads compared to theoretical threads is called *thread occupancy*. In contrast, modern CPUs can have tens of active threads, but these threads are much more independent from each other than the threads on a GPU.

SIMT has same hardware level advantages as SIMD. On the software level, the difference to SIMD is that the programmer does not need to manually define the data parallel parts of the code. The code can be written in the scalar way for a single piece of data and the SIMT driver and hardware can decide the data parallelism dynamically. This way even if the hardware changes, thread occupancy can be kept high with the same code. The difference to conventional multithreading on a CPU

is that all of the threads in a warp are running the same code. In contrast, with regular multithreading each thread might as well be running a totally different piece of the code.

3.3 Instruction-Level Parallelism

In *Instruction-Level Parallelism* (ILP) the hardware is actually running multiple instructions at the same time, even if it is abstracted in a way so that the programmer and the compiler can think that the hardware is running sequential code [PH11, p. 41]. This can be done, for example by sending instructions to a *pipeline* so that the computation of the next instruction starts before the computation of the previous one ends. Another idea is to have multiple hardware units for different tasks and have a control logic, which realizes that some of the instructions can be executed in parallel, which is called *Superscalar* execution.

There are ways to make ILP utilization easier for the hardware. One is to unroll loops so that there are more sequential instructions modifying different data, but this usually requires more registers. In the author's experience, ray tracing is somewhat register bound. Therefore, unrolling is probably not a good idea. Furthermore, it is the compiler's job to do the unrolling, if it realizes that the loop is a good candidate for it.

3.4 Mobile Device Hardware Architecture

Mobile devices use all of the parallel computing architectures described in this chapter, but usually their resources are more limited compared to desktops. One notable characteristic of mobile device architectures is the reduced memory bandwidth, which reduces the power consumption. For this reason, algorithms for mobile devices are designed so that they use less memory bandwidth [AMS08].

A modern trend is that mobile devices have a so-called *System on a Chip* (SoC). A SoC is a piece of hardware that contains multiple different kind of hardware units, which all are dedicated to their own tasks. For example, there could be a processor just for signal processing and a separate processor just for manipulating images from the device's camera. SoCs use less power by shutting down some parts of the chip [EBA*11]. In a way, SoC is a high level parallel computing architecture, because the hardware units can work simultaneously on their own tasks.

In the future, it is possible that the SoC contains a hardware unit dedicated to ray tracing. Either the ray tracing unit could be a part of the GPU, capable of drawing

triangles with the conventional pipeline, or it could be a completely separate graphics processor. The advantage of the first approach is that the same resources could be reused, but it also sets many requirements for the unit. There is already a large body of work on this area [LSL*13, SWW*04, WSS05].

4. GENERAL RAY TRACING OPTIMIZATION ALGORITHMS

In this chapter, the most common generic optimization algorithms for ray tracing are introduced. These are algorithms which can be used with many kinds of ray tracers on different kind of platforms.

Algorithms can be divided into two categories: online and offline optimizations. Offline optimizations are done before the actual tracing of the rays start and online optimizations try to optimize the code that is doing the tracing. Space partitions are ways of doing offline optimization. They are sometimes done again for every frame, but they are still done before the tracing. Online optimizations include parallel computing and optimizing the inner loop.

4.1 Space Partition

The most important optimization for ray tracing is data structure optimization. In this process, the 3D space is usually divided into smaller volumes. These volumes are usually stored in a tree-like structure. The space can also be divided into equally sized portions and stored in 3D array as in grid space partition. By using these principles, the ray traversal can be terminated as soon as possible. All of the most common space partition methods for ray tracing *grids*, *k-dimensional trees* (k-d trees) and *Bounding Volume Hierarchies* (BVH) are described in this section [WMG*09].

Space partition algorithms are demonstrated in the scene illustrated in Fig. 4.1. In the figure, the primitives are represented by different shapes, but all of them could as well be triangles. If one would like to make a ray tracer that would work with, for instance, star shaped primitives, they would need to either split the star into easier shapes like triangles or just implement a test which tells if a ray intersects the star or not. The figures presented in this section might not be the optimal solutions for each space partition algorithm in the example scene. They try to visualize how the algorithms work. All the algorithms work in 3D space, but due to the author's

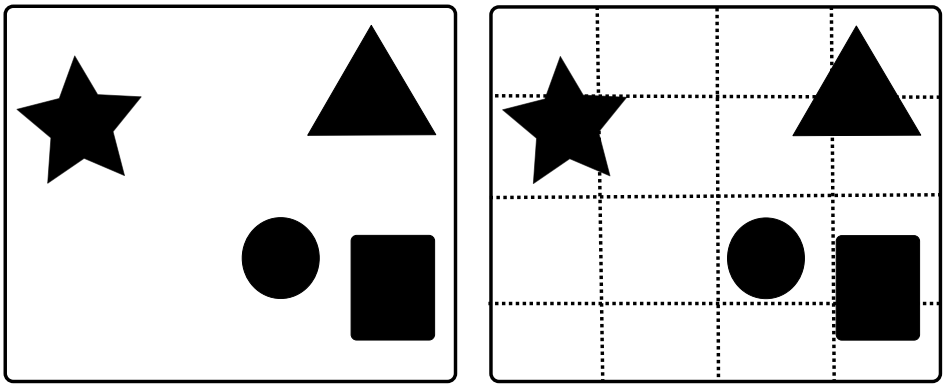


Figure 4.1 *Left: Scene used for demonstrating the space partition algorithms. Right: A grid partitions the space into equally sized boxes. Notice that memory is used for the empty lower left corner and many primitives occupy more than one grid box.*

rather limited drawing skills, every algorithm's 2D equivalent is used in the figures.

4.1.1 Grids

Grids divide the space into equally sized boxes, which can be easily stored in a 3D array. The advantage of this is that if a ray's starting position is in the middle of the scene, examination can start from the grid's box corresponding to that position. This only requires a conversion function between the world coordinates and the 3D array coordinates. In addition, it is fast to find out which grid boxes a ray is going to intersect and traverse them in closest first order. The 2D space partitioned into a 2D array is illustrated in Fig. 4.1.

Traversing is done by first loading the grid box in which the ray starts and checking if the ray intersects primitives in it. Then traversing continues by taking the closest grid box which the ray is intersecting and is not yet examined. This is done over and over again as long as an intersection is found or the ray goes out of the scene. If there are multiple primitives in a box, and the ray is intersecting more than one of them, the result is the primitive with the closest intersection.

Disadvantages of the grids are handling big primitives and handling empty space. Big primitives are spread out to multiple grid boxes. This means that they need to be referenced in all of them. During traversal, the same primitive must be tested multiple times, even if it was already tested (and realised that the ray does not intersect with it). This can be avoided by adding a buffer of last tested primitives, but even with a quite small buffer, checking it becomes slower than just testing the primitive again. If there is an empty area in the scene, the grid is using memory

to store empty nodes. In the worst case this means that there are only a couple of nodes storing all the primitives (consider a case of two detailed models far apart from each other). In this kind of worst case scenario, using the grid might even be almost as bad as testing the ray intersection against every single primitive.

4.1.2 Octrees

Instead of storing data in equally sized grid boxes, data can be stored in a tree structure. This way some of the branches can terminate earlier and shallower branches can be applied to empty areas of the scene.

While traversing the tree structure, at the root level, the ray is first tested to see if it can intersect with anything at all. If not, the traversal can terminate straight away. If the traversal indicates that the ray intersects a node in the tree, a similar test is performed for all the child nodes of the node. This is continued until all the branches have terminated, or reached the leaf level. At the leaf level, primitive intersection tests are performed to find out if the ray actually intersects any primitives in the leaf. This can be optimized by going through nodes in a depth first order and starting from the branch that contains the closest node to the ray origin and continuing to branches further away in the ray's direction. In this way, the whole traversal can terminate as soon as an intersection is found.

An octree is one way to store the space into a tree structure [Mea82]. The idea of the octree is to split the space recursively into eight equally sized partitions. The recursion is terminated if there are no primitives at the branch at all. This way, memory is used for areas that actually need them. The octrees do not need to store the split position into memory, because they are always going to be in the middle

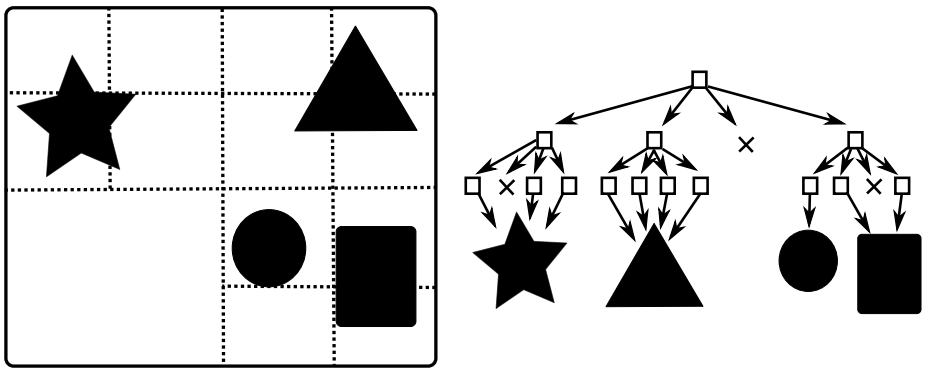


Figure 4.2 *Octree's 2D equivalent quadtree partitions the space recursively into four equally sized boxes.*

of the inner node. The inner node just needs to store eight pointers, which can be pointing to child branches, child leaves or null (empty node).

The example scene is partitioned using octree's 2D equivalent quadtree in Fig. 4.2. The optimal quadtree would have been one where recursion had been stopped after the first split but it was continued in this example for demonstration purposes. Nevertheless, if the primitives are not equally sized and they are not laid out to fixed coordinate values, the octree nodes are not tightly bounded around their geometry. This means that there is empty space in the leaf nodes and that the same primitives are referenced in multiple leafs. Given these points the disadvantage of octrees is their rigidity. As mentioned in Subsection 4.1.1, referencing a primitive in multiple leaves forces traversal to test the same primitive multiple times. In contrast, empty space forces the traversal examine a leaf node, even the ray is actually far away from the node's geometry. For these reasons octrees are not that commonly used in ray tracing.

4.1.3 K-Dimensional Trees

In k-d trees, like octrees, the division is always made along axes, but unlike octrees the space is split, not divided into 8 pieces [WH06]. Additionally, the position of the split can be freely chosen within the parent volume as long as it is along one of the three axes. In other words, in the k-d tree the space is split recursively into two parts which can be different sized. In k-d trees if a primitive is in the area of two leaves, it needs to be referenced by both of them, but because of the freely chosen split position most of these cases can be avoided. The freely chosen split position also allows tightly bound boxes around the primitives.

One way of partitioning our example scene using a k-d tree is shown in Fig. 4.3. Notice how in the example scene getting rid of empty space next to the triangle requires an empty node. Empty nodes might seem like a good idea because they are terminating immediately and do not require memory, but they might be bad for performance when using the parallel traversal techniques discussed in Section 4.2.

K-d trees have been popular data structures for ray tracing for some time. In k-d trees, the node only needs to store two bits for the 3D axis and one floating point number for the split position. In addition, k-d trees are also fast to traverse. Unfortunately, k-d trees are still quite rigid. For example, it is difficult to split the world into more than two parts, which is required by some parallel traversal techniques. Furthermore, it is difficult to adapt k-d trees for animated scenes. This usually requires building the whole k-d tree from scratch after every change in the

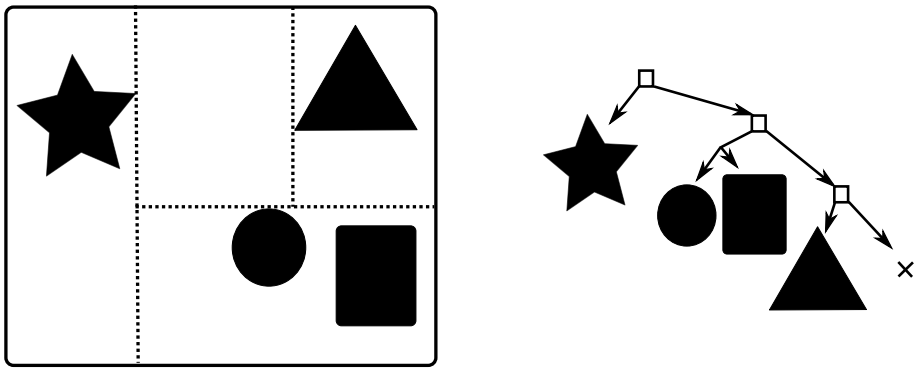


Figure 4.3 *The k-d tree splits the space recursively into two different sized halves. Notice how getting rid of the empty areas around primitives requires more splits.*

model [WIP08]. Therefore, more flexible data structures, like bounding volume hierarchies have been developed.

4.1.4 Bounding Volume Hierarchies

The idea of a BVH is to store the hierarchical structure of bounding volumes [KK86]. Usually these are *Axis-Aligned Bounding Boxes* (AABB), but spheres and other shapes can also be used. AABBs are boxes whose bounds are aligned to main axes of the scene. Compared to boxes with other orientations, intersection computations are faster using AABBs.

In BVHs, instead of just splitting the volume, smaller AABBs are taken from it. These smaller child AABBs can be positioned anywhere inside their parent. Using this approach, the child volumes are easily fitted so that no extra volume is left in them. When using k-d trees this process would require multiple splits. For this reason, some k-d tree algorithms utilize empty nodes [WH06]. Empty nodes are not as effective as the fitting achieved in BVHs. Because of the basic idea of the BVH, it is some times categorised as an object partition algorithm instead of the space partition algorithm.

An example of 2D BVH can be seen in Fig. 4.4 and visualization of 3D BVH for much more complex scene can be seen in Fig. 4.5. This kind of visualization can easily be made by setting a ray tracer to count how many intersection tests it does for each pixel to find the closest intersection with the primitives. Then the count of intersection tests is used to set the color of the pixel. In this example the pixel color was set to white and then darkened for each test the pixel had to make.

When using BVH, each node needs to store six values, which correspond to the

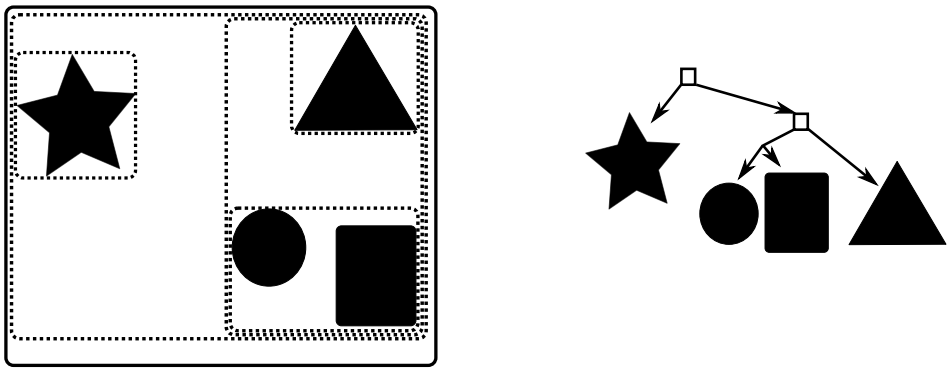


Figure 4.4 *BVH fits tight axis-aligned bounding boxes around groups of primitives and produces a hierarchy of them.*

minimum and maximum values of the AABB on every axis. Unfortunately, this requires more memory in the inner nodes than in the inner nodes of a k-d tree. Nevertheless, the k-d tree might require more memory in total, because it might be referencing to the same triangles so many times in the leaf nodes. There are ways to reduce the memory requirement of BVH inner node at the cost of extra computations. These ways are described in Section 4.4. To find out if a ray intersects the AABB of a BVH node, there needs to be a ray-AABB intersection test. The test requires some computation, even if it is as optimized as possible [AMHH08, p. 744] [PH10, p. 193].

Usually every triangle is stored in only one leaf of the BVH. This can be done because it does not matter if the child nodes are overlapping each other on any level. The advantage is that each primitive is tested only once. The disadvantage

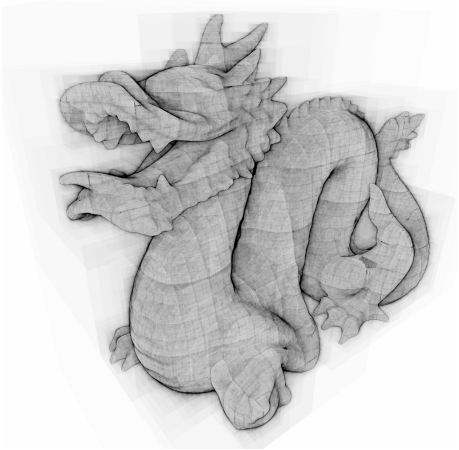


Figure 4.5 *3D BVH visualized by displaying the amount of work each pixel has to perform to find the closest intersection with the dragon model. The darker color means more work.*

is that when the first intersection is found, it is possible that the traversal cannot terminate. Given that there might be a closer triangle in another overlapping branch even though the branches are travelled in closest first order. Some of the overhead can be reduced with *Split Bounding Volume Hierarchy* SBVH, which tries to avoid overlapped sibling AABBs [SFD09]. The result is tighter fitted AABBs, but this requires references to the same primitive from multiple leaf nodes.

Despite the fact that BVH requires more computation and extra memory, it is very adaptable. The minimum and maximum values of the AABB of BVH nodes can be changed without rebuilding the entire data structure. Changing the values is required, for example, if there are animated objects in the 3D world. Of course, the quality of the tree decreases when values are changed from the optimal ones. For this reason, some ray tracers using BVHs update the values regularly and rebuild the tree only when the quality has decreased under a certain threshold. This way, both adaptability and good quality can be achieved.

4.1.5 Tree Construction with the Surface Area Heuristic

To generate high quality k-d trees and BVHs automatically, there needs to be a way of computing the quality of a single split. The *Surface Area Heuristic* (SAH) estimates how much work a ray tracer must do for a random ray to find out which triangles it intersects with in a given volume [MB90]. The basic equation of SAH can be as simple as

$$C = Sa \times N \times T \tag{4.1}$$

where C is the cost of the finding the closest triangle, Sa is the surface area of the volume in which the tests are done, N is the number of triangles in the volume and T is the cost of a single triangle intersection test. In order to make ray tracing faster, the space can be divided into smaller volumes. The size of the volumes can be decided based on the SAH cost.

The division into smaller volumes begins with the whole space. All the possible split points are tested for the SAH cost of both possible child roots. The possible split points are the ones where any of the primitives start or end, because the optimal split is always one of these points. After all the possible split points are tested, the position with the lowest combined cost for both child volumes is chosen as the actual split position. Both child volumes can then be divided with the same algorithm, and the splitting can continue to their children in a recursive manner. The result of

this splitting is a tree-like structure which contains the so-called inner nodes, where space is divided, and the so-called leaf nodes, where primitives are stored.

Due to the idea of SAH, it is also easy to determine when to stop splitting and leave a volume whole. The cost if a node is left as a leaf has already been calculated in the parent level SAH test. For the split to be made, the sum of the costs of the two child nodes should be less than the cost of not splitting the node. The final parameter needed for this test is the cost of a ray tracing tree branch traversal step S. The equation for choosing whether to split a node is defined as

$$C_{leaf} < S + C_1 + C_2 \tag{4.2}$$

where C_{leaf} is the cost if this node is made into leaf, C_1 the cost of the first child in the lowest combined total child cost and C_2 is cost of the second child in the lowest combined total child cost. The node is made into a leaf and the recursion is stopped if Eqn. (4.2) is true. In that case, the SAH cost of the leaf is lower than the SAH cost of two children.

There are difficulties with this approach, though, because both the children are calculated as if they were leaves. However, the first children after the root are often the roots of deep sub-trees. In the case of a sub-tree, the cost estimate is not the same as the actual cost of the final node. Calculating the actual cost in the construction phase would be too computationally expensive, because for each possible split on every level it would require recursive calculation up to the leaf nodes.

SAH tree construction is used as the golden standard in most of the research examining other construction methods. Usually these other methods try to make the construction somehow faster, because the SAH method is so computationally heavy.

4.1.6 Faster Tree Construction

There are many ways to speed up tree construction [Wal07]. For example, some parts of the tree can be built with faster methods that are based on a so-called Z-curve. The idea of the Z-curve is to give every position in 3D space an alternative single dimension coordinate. This coordinate is decided in a way so that objects close to each other in the 3D world also tend to be close in the Z-curve coordinate. If the whole tree is built using Z-curve without any SAH it is called *Linear Bounding Volume Hierarchy* (LBVH) [LGS*09] and if it is a combination of both methods, it is called *Hierarchical Linear Bounding Volume Hierarchy* (HLBVH) [PL10].

Another idea is to calculate the SAH values only for some points in the space, which is called *Binned SAH* [DPS10]. For example, it is possible to test only eight points inside the parent AABB. The best of these points is then used instead of the actual best SAH position. Parent AABBs can be divided equally based on the volume or, if the primitives are already sorted, the points can be chosen so that each bin contains as many of the primitives.

While implementing his own ray tracer the author tried improving binned SAH by applying parabola fitting to the values found for bins. The idea was that the lowest point of the parabola is closer to the actual best SAH position than best bin. This seemed good, because in the example figures of SAH costs the values form nice clean parabolas. In reality the SAH cost curve can be very complex, to the point that the parabola found by curve fitting was upside down. This meant that the fitted parabola disagreed wildly with the actual best SAH position. Even with dirty hacks for caching problematic cases, parabola fitting gave only minor or not at all improvements on the original binned SAH. A similar idea was introduced by Hunt et al.[HMS06], but their approach took more samples in the problematic areas without any curve fitting. They claim that the approach achieves some performance increase.

4.2 Ray Traversal Optimization

This section describes online optimizations which are the optimizations that are used during the ray traversal. These optimizations can be done on a general level. However, for efficient implementation they require as much knowledge of the targeted hardware as possible.

4.2.1 Inner Loop

The inner loop is the part of the ray tracer which is looped over and over again in the ray traversal phase. It includes both the data structure traversal and the primitive ray test. Optimizing the inner loop means reducing the total latency of it to the minimum. Reduction can be done by pre-calculating values which are going to be the same on every loop iteration and by avoiding unnecessary branching in the loop. Usually the compiler should do this type of optimization, but if the code is complex, it might be too hard for it to detect that the optimization is possible. One typical optimization is to calculate a piecewise vector value inversion of ray direction before the inner loop, replacing a costly piecewise division in the ray-AABB intersection test with a relatively fast piecewise multiplication.

There are some frameworks which can be used for writing and compiling efficient parallel inner loops quickly. Examples of these frameworks are *Compute Unified Device Architecture* (CUDA),*Open Computing Language* (OpenCL) and *Open Multi-Processing* (OpenMP). The idea is that they can use special instructions and optimize parameters based on the targeted hardware, which cannot be done with conventional C++ compiler. The advantages of these frameworks also include that the same code can be used for multiple platforms.

However, if the absolute optimal result for the targeted hardware is desired, it can be achieved by modifying or writing the inner loop using intrinsics or assembly language. Usually, this is quite slow work and the result is not portable to hardware using different instruction sets. On the other hand, the programmers can handle all of the resources themselves. This means that the resulting software can have better utilization of all resources than with the compiler deciding the utilization.

4.2.2 Packet Tracing

The idea of packet tracing is to trace more than one ray at a time. Using SIMD (described in Section 3.2.1), this is relatively simple. Instead of one ray's origin and direction, the software loads groups of them and performs all of the same calculations for each one in the group in parallel [WGBK07].

But of course, there are some drawbacks. It is possible that only one ray requires further inspection of a space partition structure in which case an unnecessary inspection is carried out on all of the rays. If it is assumed that the hardware actually has the same width SIMD instructions as the vector width of the software, even in the worst case the computation should not be slower than when tracing each ray one by one. However, if the hardware is emulating wider SIMD instructions by running the code with narrower SIMD instructions multiple times, packet tracing might slow down the software.

The advantages of packet tracing are decreased if the rays are not coherent. Coherence means that the rays are going in almost the same direction. Coherence is high when the primary rays are sent from the camera, but with the reflected and refracted secondary rays the coherence is much lower. Notably, the coherence of shadow rays can be somewhat high even after many recursion steps, because they are all going to the same light source.

Another problem with packet tracing is the fact that it is hard to avoid parts where all the rays in the packet need to be processed sequentially. Depending on the

programming language, it might be difficult to store SIMD data in such a way that they can be accessed one by one, yet still store them in super-fast registers.

4.2.3 SIMD Tree Traversal

Another idea for exploiting SIMD instructions is if the software, instead of tracing multiple rays, tests for multiple children in parallel in every branch of the data structure [WBB08, DHK08]. In some sources this is called *Multi Bounding Volume Hierarchy* (MBVH) [EG08]. This is quite a good idea, because in the most of the algorithms all of the children are tested anyway. Or if they are not tested, why not modify the algorithm to take advantage of it. With SIMD, the computational cost of this would be almost the same as testing only for the first children. Additionally, in every algorithm it is quite likely that the other children are tested for as well. Another advantage is that, even though individual nodes with MBVH require more space from the memory, there are less nodes in total, which reduces memory usage.

The only drawback is that the data structure needs to support multiple children. For example, a k-d tree only stores the split axis and the split point of a volume, so it is hard to modify it to test for multiple children. On the other hand, BVH is a good candidate for SIMD tree traversal because it can have as many children on each branch as needed. Nevertheless, the number of children required on each level for SIMD tree traversal needs to be taken into account when the tree is constructed.

One idea is to build the tree as if every branch has only two children. Once the tree is ready, it is re-formed. During this reformation some levels are deleted and the children of those levels are designated as the children of the deleted level's parent. For example, if the deletion is done at every other level, the end result would be that the tree has four children in every branch node. Any power of two branching factors can be achieved with this method, by first building with two children and then removing some of the intervening levels. Luckily, both the SIMD widths and the easily achieved BVH children counts are powers of two. So it is easy to implement testing all the children with the SIMD instructions.

Similarly to testing multiple child nodes in parallel, multiple primitives in a leaf node can be tested in parallel with SIMD instructions. This is more beneficial if as many leaves as possible contain the same number of primitives as the SIMD instruction can compute. This can be achieved by modifying the SAH to prefer leaves with correct number of primitives [EG08]. Another idea is to build a normal binary tree and do some extra work in the re-forming phase. One concept is to recursively go through the tree and test if leaves should be combined based on the combination's

SAH cost [WBB08].

If the underlying hardware supports very wide SIMD instructions, it is possible to combine both SIMD tree traversal and packet tracing. This way, the increase in speed from both of these ideas can be exploited. The total SIMD width of the combined method is the product of the two SIMD widths used in the individual implementations of these techniques. Unfortunately, even if the hardware can do computations with really wide vector instructions it is possible that it runs low on vector registers. This may reduce how many SIMT threads there can be in parallel execution. If there are not enough registers the data needs to be stored into caches or memory, which increases the latency of fetching the data. The latency can be so high that it is faster to compute with narrower SIMD instructions.

4.3 Thread Parallelism

In traversal, every pixel is independent of each other. Therefore, as many pixels as possible can be computed in parallel. Usually there are more than 10^6 pixels in an image. Depending on the ray tracing algorithm there can be from one to infinity rays per pixel. This means alongside SIMD parallelism, it is a good idea to have a thread pool with an optimum amount of threads for the targeted hardware to handle the rays.

Algorithms especially targeted for GPUs exploiting SIMT execution should divide the ray tracing work into smaller pieces (in other words into smaller kernels) [LKA13]. This way it is more likely that the threads are performing similar work without considerable branch divergence, which results in better thread occupancy. On CPUs and GPUs exploiting just SIMD divergence is not problematic since the threads are independent from each other. Without SIMT, splitting kernels into smaller pieces might even harm performance, because in between the kernels all results need to be ready, which leads into overfilled caches.

Another idea for better thread occupancy is that the traversal does not examine the leaf nodes immediately after they are found [LKA13]. Instead it pushes them into a stack and starts to examine them once every thread in the warp has enough similar work. This results in much better thread occupancy than conventional task switching straight after a leaf node is found.

Multiple parallel threads can also be used in the tree construction. If the tree is built or refitted for every frame, there can be separate threads for traversal and tree modifications. Using multiple threads just for SAH tree building is somewhat

troublesome. To explain, in the beginning there is the whole space which needs to be divided into two halfs. Although this work can be split for multiple threads, it is not that easily done. After the first division there can easily be two threads working on their own halves. After these two have found their splits there can be four threads and so forward. The problem is that there is only one thread in the beginning, which is reducing performance. In order to fix this problem, HLBVH can be used. First the lower levels of the three are built using a less accurate Z-curve method, which can be thread parallized. There are even ways of constructing Z-curve based trees completely in parallel with SIMT [Kar12]. Multiple small trees made with the Z-curve method are then used as primitives for the more accurate SAH construction. This ensures better quality closer to the root of the tree, which is traversed by the most of the rays. This way the tree can be built in parallel and still achieve quite good quality.

4.4 Bounding Volume Compression

Bounding volume hierarchies can be compressed in at least two ways. Firstly, instead of storing some of a child AABB's bounds, they can be inferred from the parent AABB. Secondly, AABBs can be stored with a less accurate data type which uses fewer bits for each bound value. The focus of this thesis is on the latter approach, since it is of special interest in mobile systems. Less accurate data types might be natively supported by mobile systems, because they can be used for built-in sensor data manipulations with less power.

4.4.1 Inferred Bounds

In a typical BVH implementation, each inner node stores the AABBs of its children as six single-precision floating-point numbers [Pur04, WBS07, DHK08]. Also a pointer to each child node is needed, which consists of one integer value, resulting in a total memory footprint of $7 \times 4 = 28$ bytes per AABB. This may be padded to 32 bytes per child in order to improve the cache access pattern. In a conventional BVH, this means that a complete BVH inner node contains 64 bytes.

BVH data can be compressed by inferring half of the child AABB coordinates from the parent AABB [FD09]. This can be done because in binary BVH each of the parent bounds is also a bound of one or the other children. In other words, the parent's six bound values and six new values define the bounds for the two children. In addition to the six new single-precision float values, one bit for each of them is

needed to define which of the two children is using the value. The total bit count is then $32 \times 6 + 6 = 198$, which is not that cache line friendly. For this reason, [FD09] reduced the bit count of the child pointer so that the whole inner node fitted into the cache line. This means 32 bytes for whole conventional BVH inner node.

Bounding Interval Hierarchies (BIH) [WK06] further reduce the memory footprint by restricting the shape of the child AABBs. In this format children are defined by only two planes. This means that both of the children inherit both the upper and the lower bounds on two of the axes. The volume is then only reduced on one axis. Even on the reduced volume axis, the lower bound of one child and upper bound of the other is inherited. In addition to just two single-precision values, BIH needs 2 bits to tell which axis is the one where the volume is reduced. Then one 30-bit pointer is used to point to the location where the pair of children is located. All this results in a total node size of 12 bytes, but it requires many restrictions in choosing the AABBs.

These formats are less applicable to MBVHs since the same number of parent AABB coordinates are shared between an increased number of child AABBs. For example, in MBVH with a branching factor of 4 there are only 6 parent values and 24 child values. This means that only one fourth of the child values can be inferred. Since MBVHs use SIMD instructions and, therefore, make better use of all computation hardware, it is preferable to find compression strategies, which work with MBVHs without setting any restrictions.

4.4.2 Data Types with Lower Accuracy

Several other compression ideas which work with MBVHs have been proposed. In an extreme solution only two bits are stored in a BVH node [BEM10]. This is done by leaving the pointers out, which requires a fixed number of primitives in leaf nodes and a fixed tree layout so that child nodes can be found. Furthermore, the axis on which the parent AABB is split is chosen in a fixed order and the split can be made only in three different fixed positions. This kind of representation also requires inner nodes that are not branches of the tree because it might not be beneficial to have a split in the current axis, which is determined by the fixed order. The fourth possible bit pattern, which can be represented with two bits, is reserved for this.

Compression to 16, 8 or 4 -bit integers has also been proposed [MWDI05, MW06]. If the model is already located within the range of the compressed data type, this can be done by just changing the six single-precision float coordinates to the compressed data type. The accuracy of the lower levels of the BVH can be increased by making

the system hierarchical. In this case, the child level's coordinates are calculated using the parent level's coordinates. The range of the data type used in the child starts from the lower bound of the parent and ends at the upper bound of the parent. However, this kind of compression is always a trade-off between the compression ratio, the quality of decompressed data and the computational overhead needed for decompression. Some of the overhead can be avoided by leaving the top region of the tree uncompressed [BEM10], which can be beneficial since the root of the tree is traversed by all rays.

Decompression can be avoided completely if the calculation can be performed directly in the compressed data type. Mahovsky et al. [MWDI05] simulated calculation with compressed data type using a different sized integers (12, 16, 20, 24 bits). They also proposed theoretical hardware which is able to perform intersection tests with these data types. In their approach, the world is quantized into fixed equally sized volumes. Prior to quantization, the scene is resized in such a way that it lies within the range of the used data type. Calculations in lower accuracy data type are beneficial, because the lower precision hardware can be designed so that it is faster. On some hardware, halving the bits in the data type doubles the SIMD width that can be computed in the same time frame. Furthermore, lower accuracy values take less space from vector registers and from caches, which could be the bottleneck for the whole ray traversal.

The disadvantage with lower precision calculations is that the origin and the direction are altered when they are quantized to low precision data type. This can lead into false positive intersections with tree branches or in worst case missing an actual intersection. Fortunately, there are ways to handle these situations. For example the origin of the ray can be moved along the traversal path to keep it accurate enough for the next intersection test [Kee14].

5. BVH COMPRESSION WITH HALF-PRECISION FLOATING-POINT NUMBERS

Storing AABBs of a BVH in halfs decreases the amount of bytes required for each inner node. Unfortunately, more inaccurate AABBs may lead the ray traversal to examine sub-trees unnecessarily, increasing the number of ray-AABB and ray-triangle tests performed. The goal of the techniques described in this chapter is to reduce memory bandwidth while increasing computations as little as possible.

Ray tracing is usually a memory bound operation, therefore, reducing the memory footprint is of high interest. BVHs of complex scenes have a large memory footprint. The optimal BVH node count is around half the number of triangles in the scene [MW06]. Consequently, with MBVH the optimal node count is roughly the number of triangles divided by the MBVH's branching factor. The BVH inner node memory requirement might seem small compared to the memory requirements of the triangles. However, the BVH inner node memory is the most frequently accessed memory area during the traversal. For example, the root of the BVH is traversed by all of the rays, whereas each triangle is traversed by only a small portion of the rays. This means that it is more beneficial for the cache hit ratio, if the size of the inner nodes are reduced.

The smaller memory footprint also means that bigger 3D scenes fit into memory. Otherwise, large scenes need to be streamed from hard disk to memory (and maybe from the main memory to the GPU memory), during traversal. This is great drawback to performance and power consumption.

The drawback of BVH compression is that decompression requires extra instructions in the performance crucial inner loop. Additionally, the quality of the decompressed BVH may be worse than the original BVH, which means that the traversal needs to examine extra tree branches. This occurs because the AABBs have to be enlarged. Then rays that are in reality not intersecting with nodes child geometry are unnecessarily intersecting with the enlarged AABB. This is called *false positive*

intersection.

Prior work on BVH node compression often uses integers for storing the bounds, but floats are interesting due to their greater dynamic range. If the camera is positioned near the origin of the scene, objects with similar on-screen size have a similar numeric accuracy. 3D scenes are often designed so that they have more details on closer objects. Furthermore, real-time graphics applications often use *Level-of-Detail* (LoD) algorithms which reduce the complexity of distant geometry. Future interactive ray-tracing applications might conceivably make use of LoD algorithms to reduce the complexity of distant geometry.

5.1 Half-Precision Floating-Point Numbers

Half-precision floating-point numbers are defined in the IEEE 754-2008 standard [IEE08] Halfs are just like singles, but they use only 16 bits instead of 32 bits. These 16 bits are divided into one sign bit, five exponent bits and 10 fraction bits. This means that the quantization step with halfs is much greater than with singles. In addition, halfs round to positive infinity after 65504 and negative infinity after -65504. Like singles, halfs have subnormal values for better accuracy close to zero.

In this chapter, the BVH is compressed to 16-bit half-precision floating-point numbers with and without the hierarchical format. Compared to the integers' uniform interval quantization points, it is expected that floating-point numbers' greater dynamic range works better with the teapot-in-a-stadium problem [Hai88]. Given that with integers the entire teapot might need to be within a couple of quantization points, but with floats the teapot can be located near the origin where it can be divided into much smaller AABBs.

Recent CPUs and GPUs provide hardware-accelerated instructions, at least for conversions between single and half-precision floats [AMD07, ARM10, Kon12, nVi15], allowing decompression to be performed efficiently with existing hardware. Despite the available instruction-set support, to the best of the author's knowledge, the use of half-precision for storing AABBs in BVHs was for the first time systematically studied in [KVJ*15].

5.2 Storing Bounding Volume Hierarchies in Half-Precision

If the storage required by the bounding volumes can be halved, the memory footprint will decrease significantly. BVH inner nodes would use 12 bytes for the bounding

volume and 4 bytes for the child pointer. This means 16 bytes in total for each child, which should already be cache line friendly without any padding. In a conventional BVH this reduces 64-byte inner nodes to 32 bytes and, in the example of testing 4 children in parallel in a MBVH, 128-byte nodes are reduced to 64 bytes. Moreover, the amount of data loaded from the memory for a single BVH traversal step is halved.

It is straightforward to modify a ray tracer to use halfs instead of singles by just adding conversion instructions. Furthermore, most of the traversal and intersection code is unaffected and there is no need to run complex decompression code. Only a couple of additional details need to be taken into account.

5.2.1 Hierarchical encoding

Half floats can be used as plain world coordinates just like single floats are used. If inaccurate world coordinates are causing too many computations and, therefore, slowing the traversal down, halfs can be used hierarchically. The idea of the hierarchical compression is that the child AABB limits are defined by using the parent AABB limits. This can be done by setting the half's minimum value (-65504) to be equal to the parent AABB's lower bound P_{lower} and half's maximum (65505) value to be equal to the parent AABB's upper bound P_{upper}.

The basic idea for compression to the hierarchical encoding can be seen in Alg. 1 In the algorithm, all values are in single-precision floats, except the return value is a half-precision float. The algorithm first converts the value that is between parent lower bound P_{lower} and parent upper bound P_{upper} to a value that is between 0.0 and 1.0. The result is then used for deciding a point between minimum and maximum of targeted compressed data type. In the case of half-precision floats, $H_{min} = -65504$ and $H_{range} = 131008$.

The basic idea for decompressing the hierarchical encoding can be seen in Alg. 2. In the algorithm, the input *hier* is in the compressed data type and all other values are in single-precision floats. Note that the decompression is equivalent to compression

Algorithm 1 Compression of hierarchical data.

half16 WorldToHierCoord($world$, P_{lower}, P_{upper})
$tmp \leftarrow world - P_{lower}$
$P_{range} \leftarrow P_{upper} - P_{lower}$
$hier \leftarrow H_{range} \times (tmp \div P_{range}) + H_{min}$
return $convertTo16BitHalf(hier)$

Algorithm 2 Decompression of hierarchical data.

float32 HierToWorldCoord($hier$, P_{lower}, P_{upper})
$tmp \leftarrow convertTo32Bitfloat(hier) - H_{min}$
$P_{range} \leftarrow P_{upper} - P_{lower}$
$world \leftarrow P_{range} \times (tmp \div H_{range}) + P_{lower}$
return $world$

except all the H and P values are vice versa and actual conversions are done so that all intermediate results are in single-precision.

Because of the nature of the hierarchical encoding, it actually has better accuracy on lower levels than single-precision floats have. With hierarchical encoding, on every level the size of quantization step is determined by the size of the parent AABB. On the lowest levels the size of the parent AABB can be close to quantization step of the single-precision, yet the volume is divided with the full range of the data type. Unfortunately, this extra precision is not useful, since the triangles are usually designed in single-precision.

Another thing, which comes from small parent AABB in the lower levels, is that the dynamic range of the half-precision floats is not that beneficial. There is no need to have more precision in the center of the small parent AABB and less on the edges. For this reason it might seem like much better idea to use integers with hierarchical encoding. In fact in Section 6.2 it is shown that the both data types are almost equally good with the hierarchical encoding.

5.2.2 Problems in quantization

With both plain world coordinates and hierarchical encoding, the quantization to half-precision causes errors in the bounding volumes, which lead to visible holes in the model. This occurs when the size of the bounding volume is shrunk during the quantization so that it does not actually cover all of its children's geometry. In this case, some rays may miss their intersection with the AABB and terminate the traversal of that branch prematurely, even though they would intersect with some of the triangles in the leaf. This is called *false negative* intersection. The ray tracer needs to make sure that every upper bound of an AABB is rounded towards positive infinity and that every lower bound is rounded towards negative infinity.

The problem can be seen in Fig. 5.1, in which blue lines represent triangle bounds in single-precision and numbered black lines show possible quantization points in half-precision. In the default rounding mode, the blue lower bound is correctly rounded

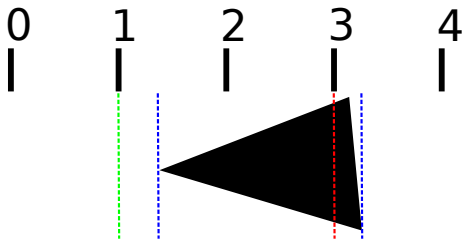


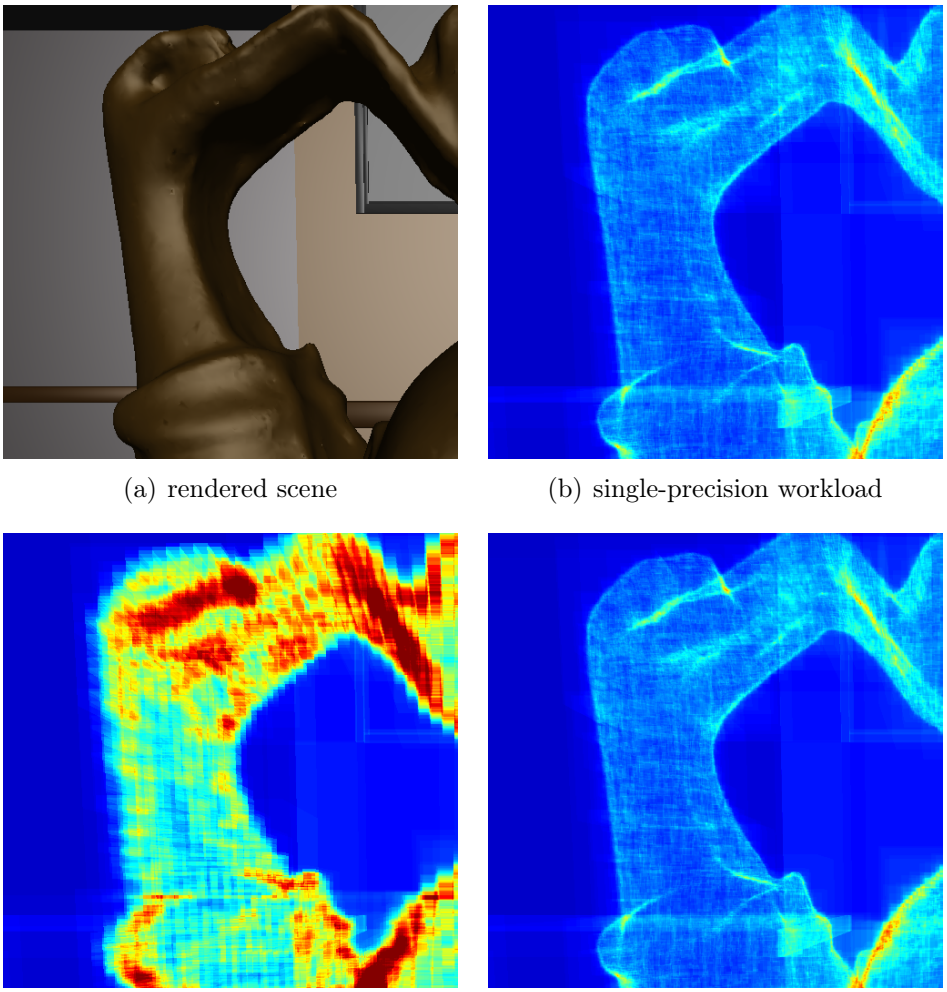
Figure 5.1 *Quantization of triangle bounds*

to the quantization point 1, but there is some extra space around the triangle. This causes the rays intersecting with the area between quantization point 1 and the triangle's lower bound to continue traversal unnecessarily. Similarly, the blue upper bound is rounded to quantization point 3. In this case, the rays intersecting with the area between the triangle's actual upper bound and quantization point 3, would not continue traversal into the leaf and, therefore, would fail to detect some intersections with the triangle, possibly causing visual errors. This issue is avoided by rounding the upper bounds toward positive infinity, but the extra space around the triangle causing the extra traversal cannot be avoided. This extra work is also visualized in Fig. 5.2, where the Buddha's hand far away from origin of the scene is shown. The visualization was done in a similar manner to Fig. 4.5, but this time the color was set so that warmer colors mean more tests. This is a difficult case for the world coordinate half-precision BVH, but the hierarchical version is indistinguishable from the single-precision version.

Fortunately, many of the instruction sets have native support for different rounding modes, which can be used to achieve the desired effect [ARM10, Kon12]. The correct rounding modes should be taken into account already while constructing the tree.

5.2.3 Half-Precision Tree Construction

In order to ensure the best SAH tree construction result, it has to use the same precision as the resulting compressed BVH. Incorrect AABBs (due to high precision in the construction phase) would lead to decision points which would be quantized to different values when the precision is changed after the construction. Even if these values are rounded correctly, this would lead to a suboptimal tree for the actual precision. Extra volumes around primitives in half-precision are causing shallower trees to be the most optimal ones. To explain, when bounds are rounded correctly and, therefore, volumes are enlarged, they overlap each other even more. This enlarged amount of overlapping makes all splitting choices less appealing for the SAH based tree construction.



(a) rendered scene (b) single-precision workload

(c) half-precision workload (d) half-precision hierarchy workload

Figure 5.2 *Workloads for the Buddha's hand far away from origin of the scene.*

In order to construct even more optimal trees for the scene without any hierarchy, the author emphasized the half-precision floating-point numbers' better precision close to the origin by centering the coordinates used in the BVH on the camera. This is beneficial because the AABBs closer to the camera are more accurate. Inaccurate AABBs close to camera would lead to more unnecessary tree traversal than inaccurate AABBs further from camera. AABBs far from the camera are likely to be behind closer geometry in which case they can be left out of the traversal. This also works well with the idea of LoD used in computer games, which reduces the precision of distant geometry. Although LoD is rarely seen in conventional ray tracing, it might conceivably be used in future interactive ray tracing applications. There is already some work in this direction [YLM06, PBD*10].

Camera-centered trees perform well with the teapot-in-a-stadium problem. If the teapot is close to the camera, there is sufficient precision to divide it into multiple

AABBs. In addition, if it is far away, only a few rays will intersect with it and suffer from the oversized AABBs. Of course, if the camera is moving and the tree is constructed only once before rendering, moving the origin is not an option. However, in animated scenes, the tree is reconstructed as refitting is not sufficient to restore the quality of the tree. Recentering the tree during the refit or the construction only requires looping through every triangle once, which is asymptotically inexpensive compared to the rest of the process.

The author also tested other heuristics for choosing the origin point, such as the closest scene point and points in front of the camera, but the camera position seems to work best in the general case. This is because the result of each heuristic is often unrepresentative of the rest of the image: for example, the nearest point in the camera's view direction might be in the horizon, while the sides of the image have nearby objects. In contrast the closest object of the whole scene might be in the corner of the image, in which case almost as close objects on the other corner would suffer from poor precision. In both of these examples, there are inaccurate AABBs close to the camera which are bad for performance. Centering on the camera assures that the nearby objects are at least somewhat accurate. The author sometimes observed small speed improvements when using the closest object in the camera's view direction as the origin, but the speed improvements were so small that they may have been caused by choosing different AABBs in the SAH tree construction.

6. EVALUATION

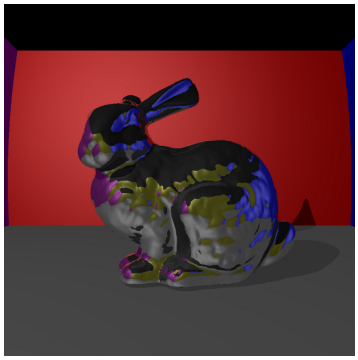
The test set-ups are explained in Section 6.1. Storing BVHs in half-precision is evaluated with hardware independent measurements in Section 6.2 and practical evaluations are in Section 6.3.

6.1 Theoretical Evaluation Set-Up

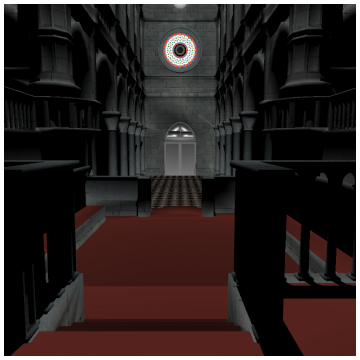
For evaluation purposes, an OpenCL-based ray tracer was implemented which uses Whitted-style ray tracing [Whi79] with primary, secondary and shadow rays. The ray tracer used MBVH4 for testing all 4 children of a inner node and up to 4 triangles of a leaf node simultaneously. Scenes in Fig. 6.1 were rendered with a resolution of 1024 x 1024. The scenes were shiny Bunny(7K triangles and 1 point light), Sibenik (75K triangles and 1 point light), Sponza(262K triangles and 2 point light), Teapot inside Sponza(278K triangles and 2 point light), Conference (331K triangles and 1 point light), Dragon (871K triangles and 1 point light), Buddha inside Conference (1 419K triangles and 1 point light) and Buddha far from camera inside Conference (1 419K triangles and 1 point light). The latter two test scenes were designed to be a showcase for the teapot-in-a-stadium effect.

Both non-hierarchical and hierarchical half-precision float formats were tested. In the non-hierarchical format, all bounds were stored in world coordinates with half-precision floats. In the hierarchical format, child bounds were stored in a way that the minimum of half-precision float (-65504) was equivalent to lower bound of the parent AABB and the maximum of the half-precision float (65504) was equivalent to upper bound of the parent AABB. The results were compared to the single-precision and the integer representations with and without hierarchy as proposed by Mahovsky et al. [MW06].

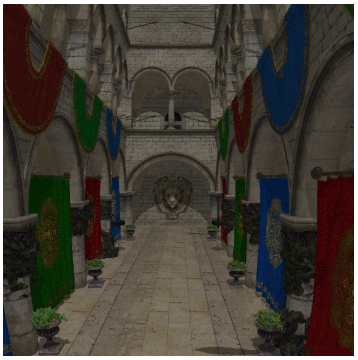
With half-precision, values were stored in half-precision, but converted to single-precision before the calculations. Integers were simulated with single-precision floating-point numbers, by choosing AABB bounds from quantization points supported by the data type. This works because Hierarchical formats' better precision in lower levels of the tree is useless since triangles are stored in single-precision.



(a) Bunny

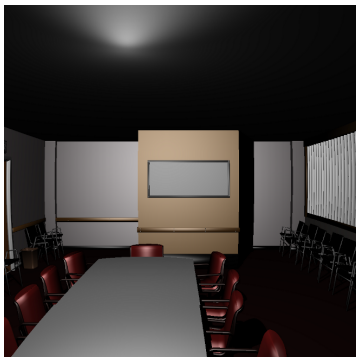


(b) Sibenik



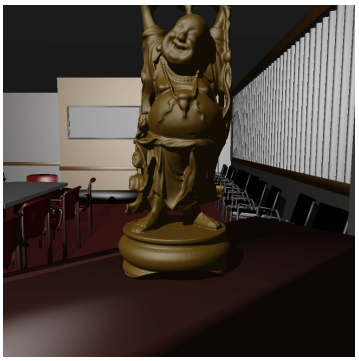
(c) Sponza

(d) Teapot inside Sponza



(e) Conference

(f) Dragon



(g) Buddha inside Conference

(h) Buddha inside Conference far from origin

Figure 6.1 *Rendered test scenes*

Table 6.1 *The number of ray-AABB tests with BVHs stored in different precisions*

	AABB tests (M)					
Model	F32	F16	F16 H	I16	I16 H	I8 H
Bunny	22	1.2%	0.1%	0.1%	0.0%	-4.2%
Sibenik	58	1.6%	2.0%	2.1%	0.3%	-1.1%
Sponza	129	0.9%	0.3%	0.4%	0.0%	-0.1%
Teapot	114	1.7%	1.7%	3.3%	1.5%	3.4%
Confer.	23	5.9%	3.2%	5.2%	0.7%	5.8%
Dragon	43	5.4%	-0.1%	0.1%	0.0%	2.1%
Buddha	47	4.1%	2.5%	5.1%	1.6%	-1.3%
FarBud.	30	6.5%	5.3%	5.9%	1.8%	6.0%
average	58	2.5%	1.5%	2.4%	0.7%	1.2%

Table 6.2 *The number of ray-triangle tests with BVHs stored in different precisions*

	Triangle tests (M)					
Model	F32	F16	F16 H	I16	I16 H	I8 H
Bunny	4	33%	0%	14%	0%	2%
Sibenik	9	12%	3%	10%	1%	6%
Sponza	27	10%	1%	7%	0%	7%
Teapot	22	5%	0%	15%	0%	3%
Confer.	6	31%	7%	28%	1%	20%
Dragon	10	16%	0%	1%	0%	-1%
Buddha	12	21%	7%	25%	5%	11%
FarBud.	9	24%	14%	21%	1%	21%
average	12	15%	3%	14%	1%	8%

Table 6.3 *The memory usage for storing only the BVH inner nodes. The 8-bit integer was not padded at all, which meant 2,5 bytes for each BVH node. This can be bad for cache performance.*

	Memory used for BVH inner nodes (MB)					
Model	F32	F16	F16 H	I16	I16 H	I8 H
bunny	7	-51%	-50%	-50%	-50%	-69%
sibenik	7	-51%	-50%	-50%	-50%	-69%
sponza	25	-51%	-50%	-50%	-50%	-69%
teapot	26	-52%	-50%	-51%	-50%	-69%
conference	29	-79%	-50%	-71%	-50%	-69%
dragon	100	-54%	-50%	-50%	-50%	-69%
buddha	148	-60%	-50%	-61%	-50%	-69%
farBuddha	148	-81%	-50%	-61%	-50%	-69%
hairball	176	-51%	-50%	-50%	-50%	-69%
average	74	-61%	-50%	-56%	-50%	-69%

Table 6.4 *The total SAH costs*

	Total SAH cost					
Model	F32	F16	F16 H	I16	I16 H	I8 H
bunny	7	1.5%	0.0%	0.1%	0.0%	1.0%
sibenik	26	2.0%	0.2%	0.2%	0.0%	1.0%
sponza	46	1.2%	0.0%	0.1%	0.0%	0.9%
teapot	46	1.7%	0.0%	0.1%	0.0%	1.0%
conference	24	2.9%	0.1%	0.2%	0.0%	1.1%
dragon	19	14.0%	0.1%	0.6%	0.0%	1.2%
buddha	25	2.6%	0.0%	0.2%	0.0%	0.7%
farBuddha	25	3.0%	0.1%	0.1%	0.0%	0.9%
hairball	362	4.5%	0.0%	0.3%	0.0%	0.9%
average	64	3.9%	0.1%	0.3%	0.0%	0.9%

6.2 Hardware Independent Evaluation Results

With the set-up described in previous section, performance with hardware- and platform-independent metrics was measured. The metrics were counts of required ray-AABB and ray-triangle intersection tests (Table 6.1 and 6.2). In addition, required memory for BVH inner nodes (Table 6.3) was calculated. This was done based on the count of inner nodes and the amount of bits each inner node required. This means that the simulated integers' calculated results should be equivalent to actually using the data types. Finally the computed SAH cost was measured (Table 6.4).

With compression to 16 bits, the amount of data loaded for each AABB test is halved compared to 32-bit single-precision. However, the number of ray-AABB tests and ray-triangle tests both increased as the result of the enlarged AABBs. Increased numbers are caused by rays examining subtrees that they actually do not need to.

As expected, due to the dynamic range, plain halfs perform better than plain integers with teapot-in-a-stadium scenes *Teapot* and *Buddha*. However, with other scenes integers perform better, which leads to about the same average performance for both of them.

With hierarchical encoding, the 16-bit integer's performance is the best on average, but half-precision floats performance is only 1.0% worse. The two are so close to each other that the order is dependent on the targeted hardware. For example, on Intel's Haswell CPU architecture, the machine instruction to convert a vector of eight halfs to eight singles (VCVTPH2PS) has a latency of 4 cycles, while its throughput is one conversion per cycle [Fog14]. In contrast, the instructions to convert eight 16 or 8-bit integers to 32-bit (VPMOVSX) and to convert 32-bit integers into floats (VCVTDQ2PS) both have latencies of 3 cycles and throughputs of one per cycle

[Fog14]. Since halfs can be converted with one-half the instructions and one-third less total latency, they are expected to be somewhat faster.

There are so few quantization points for the 8-bit hierarchical integers that sometimes it accidentally manages to do the traversal with fewer tests compared to the ground truth single-precision. This is the case, even the SAH readings of the 8-bit integers are always worse than the single-precision. As mentioned in Subsection 4.1.5, SAH is only a heuristic and it is calculated for a random ray.

When looking at calculated memory savings, the variation in plain world coordinate methods seems high. The variation comes from the fact that with less precision, sibling nodes in BVH are more overlapping. This makes splitting a volume into subvolumes a less appealing alternative to SAH construction and, therefore the trees are shallower. The memory readings are for all inner nodes and a shallow tree contains less inner nodes. This is why there are much greater memory savings than 50% even though the amount of bits in a single node is just halved. The hierarchical systems' total inner node memory savings are always roughly 50%. This is explained with the fact that the accuracy in the lower levels is more than enough to choose similar nodes than single-precision tree construction would choose. Because triangles are stored in single-precision and it is wise to choose triangle bounds as split points, similar splits to single-precision are made with hierarchical systems.

The ray tracer was run on an ARM Mali-T628 GPU on a Odroid XU3 board, which can measure the power usage. The Mali-T628 is used in development boards as well as in consumer mobile systems like tablets and phones. The special characteristic of the Mali-T628 is that both the CPU and the GPU use the same memory [ARM13, 7.3. Memory optimizations]. Additionally, bigger kernels should work better on the Mali-T628 [ARM13, 7.4. Kernel optimizations]. This is probably due to the small caches which would overfill between the kernels.

To isolate the interesting part for the power measurements the rendering and most of the shading were disabled. On average the memory used 10% less power with half-

Table 6.5 *The average power usages of different components*

	Power usage (W)		
Component	F32	F16	I16
A15 CPUs	1.0	-0.7%	0.5%
A7 CPUs	0.2	-1.0%	0.0%
memory	0.2	-10.0%	-2.1%
GPU	1.3	-1.4%	0.3%
total	2.7	-1.7%	0.2%

precision than with single-precision. Furthermore, the GPU used 1.4% less power while the rays per second count increased 0.8%. This results in 1.7% less power in total, since total power usage is dominated by the GPU usage. In contrast, 16-bit integers' improvement on average was not measurable. The author believes that this is because the Mali-T628 is more optimized for half floats than 16-bit integers. Breakdown of the usage of the different components can be seen in Table 6.5.

6.3 Practical Evaluation

In order to evaluate memory savings in a practical renderer, a half-precision extension without hierarchy to Intel's Embree ray tracing kernel collection [WWB*14] was implemented. This was done because it was possible that the author's own ray tracer was not representative enough to state of the art ray tracers. The typical structure of a 4-wide MBVH with a maximum of 4 triangles per leaf node was used. In addition, the original 64-bit addresses were replaced with 32-bit offsets so as to fit each node into exactly 64 bytes, however, this requires some additional computations. The viewpoints in Fig. 6.1 were rendered with the Embree example path tracer at 16 samples per pixel. Tracing was performed with 4 threads on an Intel Core i7-4500U CPU. We measured tracing performance and total memory consumption including both the BVH tree and the leaf primitives. Moreover, data cache misses were measured using the Linux *perf* utility.

The results are shown in Table 6.6. Half-precision floats reduce the total memory footprint by 7% and cache misses by 16% on average. This shows that, though the triangles take up more memory than nodes, the nodes receive much more traffic. However, due to conversion overheads and the additional node and primitive tests incurred by the non-hierarchical format, performance was approximately 7% lower

Table 6.6 *Evaluation results with Intel Embree. Memory readings contain the whole BVH including the triangles stored in leaf nodes.*

	Mem (MB)		Speed (Mrays/s)		L1 miss (%)	
Model	F32	F16	F32	F16	F32	F16
Bunny	5	-8.0%	7.9	-8.5%	3.3	-12%
Sibenik	6	-5.5%	5.5	-7.9%	7.4	-22%
Sponza	19	-6.4%	5.3	-7.2%	6.4	-26%
Teapot	20	-6.6%	5.1	-6.2%	4.9	-23%
Confer.	24	-6.3%	5.7	-5.4%	6.3	-10%
Dragon	66	-7.3%	5.3	-6.6%	5.4	-16%
Buddha	107	-7.3%	5.1	-9.4%	6.5	-11%
FarBud.	107	-7.4%	5.1	-6.8%	6.5	-10%
average	44	-6.8%	5.6	-7.2%	5.8	-16%

than the baseline. Furthermore, an 8-wide MBVH which maps better to Intel's conversion intrinsics, was implemented to reduce the conversion overhead. This improved performance, but not enough to match the baseline.

6.4 Half-Precision Inner Nodes Conclusions

In this chapter, it was shown that when BVHs are stored in half-precision floating-point instead of single-precision, the amount of memory bandwidth is halved for a ray tracing traversal step. This is not that obvious because the pointer to the next inner node needs to be loaded as well. Reducing the memory bandwidth usage might be especially beneficial on mobile systems. It also decreases the memory footprints of the BVH inner nodes by 50% – 81%, reducing cache misses and allowing a larger scene to fit in the main memory. Savings larger than 50% are possible because SAH based construction makes shallower BVHs for the lower precision. This contributes to an average of 18% memory savings of complete BVH containing also the triangles. In a practical renderer, halving the BVH inner nodes' storage area reduces L1 cache misses by 16% and average of 6.8% memory from the whole BVH containing the triangles in the leaf nodes. In addition, an average of 1.7% total power saving in ray traversal was measured on a consumer mobile system GPU.

The downside is the need for 2.5% more ray-AABB tests on average and 15% more ray-triangle tests on average. If hierarchical encoding of half-precision floats is used, the average test counts are almost indistinguishable from single-precision. With hierarchical encoding the minimum value of the data type is set to the lower bound of the parent and maximum of the data type is set to upper bound of the parent. However, hierarchical encoding requires more computation during the ray traversal.

Due to the dynamic range, halfs perform better than 16-bit integers with small triangles close to the origin of the scene. In contrast, integers work better with scenes that consist of evenly sized triangles. With hierarchical encoding the two are so close to each other that superiority is determined by the targeted hardware's speed of converting the data types to single-precision floats. At least, some hardware is optimized for floating-point operations rather than integers. In addition, half-precision storage is straightforward to implement with currently available hardware because most of the hardware supports conversions from half-precision to single-precision and back.

7. CONCLUSIONS

Ray tracing is a method of producing images of a virtual 3D world. These images are produced by transmitting rays and determining which objects in the 3D world they intersect with. This requires an enormous amount of computation power. However, complicated real life effects can be achieved which are difficult to achieve with the conventional GPU rasterization pipeline. Ray tracing scales more on the number of rays and, therefore, the number of pixels rather than the number of primitives. This makes it interesting to use on mobile devices, which in general have smaller screens.

There are many ways of optimizing ray tracing. The most important method is using a data structure for storing the 3D world. This reduces the work of finding the intersections from looping through all the objects in the world to examining just some parts of the data structure.

Other methods are making sure that the inner loops are as small as possible and using different kind of parallelisms. SIMD instructions can be used for examining multiple parts of the data structure at the same time and for tracing multiple rays at the same time. Threads can be used for tracing multiple rays in parallel, since most of the rays are independent from each other.

There are many factors affecting the effectiveness of compression data structures in ray tracing. First of all, compression reduces the memory footprint of each data structure node and therefore reduces memory bus usage and cache misses as well as enables bigger models to fit into memory. On the other hand, compression reduces the quality of the data structure, which leads to unnecessary traversal. In addition, decompression introduces more instructions into the performance crucial traversal inner loop.

Compressed data types can be used hierarchically when the data structure achieves roughly the same precision as with non-compressed data structures. Unfortunately this introduces even more instructions into the inner loop.

It is more likely that compression is beneficial on mobile systems since they may have native support for less accurate data types. Furthermore, compression reduces

memory bandwidth and power usage and mobile systems typically have smaller memory bandwidth and stricter memory budget.

In conclusion, it depends on the ray tracing application and the targeted hardware whether data structure compression is beneficial. If the application is heavily memory bound like ray tracing usually is, then compression might be a good idea. In contrast, if the application is computation bound, adding more computation cannot make it faster. The most important thing is to measure what is the bottleneck and decide on optimization based on facts.

In the future, the author is interested in finding new ways to make ray tracing faster and more energy efficient. One could be calculating intersection tests in half-precision, which on some hardware could double the amount of tests done with one set of SIMD instructions. Other current ideas include optimizing the SAH function for BVHs which are targeted to wide half-precision SIMD instructions.

BIBLIOGRAPHY

[AMD07] AMD: *AMD64 Technology: 128-Bit SSE5 Instruction Set*, 3.1 ed., August 2007. Publ. No. 43479.

[AMHH08] Akenine-Möller T., Haines E., Hoffman N.: *Real-Time Rendering*, 3rd ed. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[AMS08] Akenine-Möller T., Ström J.: Graphics processing units for handhelds. *Proceedings of the IEEE 96*, 5 (2008), pp. 779–789. doi:10.1109/JPROC.2008.917719.

[ARM10] ARM: *RealView Compilation Tools: Assembler Guide*, 4.0 ed., December 2010.

[ARM13] ARM: *Mali-T600 Series GPU OpenCL: Developer Guide*, 1.1.0 ed., February 2013.

[BEM10] Bauszat P., Eisemann M., Magnor M. A.: The minimal bounding volume hierarchy. In *Proceedings of the International Symposium Vision Modeling Visualization* (2010), pp. 227–234. doi:10.2312/PE/VMV/VMV10/227-234.

[Bou05] Boulos S.: Notes on efficient ray tracing. In *Proceedings of the ACM SIGGRAPH Courses* (2005), pp. 10:1–10:7. doi:10.1145/1198555.1198749.

[CH10] Carroll A., Heiser G.: An analysis of power consumption in a smartphone. In *Proceedings of the USENIX annual technical Conference* (2010), vol. 14.

[CPC84] Cook R. L., Porter T., Carpenter L.: Distributed ray tracing. *ACM SIGGRAPH Computer Graphics 18*, 3 (1984), pp. 137–145. doi:10.1145/964965.808590.

[DHK08] Dammertz H., Hanika J., Keller A.: Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. *Computer Graphics Forum 27*, 4 (2008), pp. 1225–1233. doi:10.1111/j.1467-8659.2008.01261.x.

[DPS10] Danilewski P., Popov S., Slusallek P.: *Binned SAH Kd-Tree Construction on a GPU.* Tech. rep., Saarland University, Germany, June 2010.

[EBA*11] Esmaeilzadeh H., Blem E., Amant R. S., Sankaralingam K., Burger D.: Dark silicon and the end of multicore scaling. In *Proceedings of the IEEE Symposium on Computer Architecture Conference* (2011), pp. 365–376.

[EG08] Ernst M., Greiner G.: Multi bounding volume hierarchies. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing* (2008), pp. 35–40. doi:10.1109/RT.2008.4634618.

[FD09] Fabianowski B., Dingliana J.: Compact BVH storage for ray tracing and photon mapping. referenced: 7/13/2015. URL: http://www.researchgate.net/publication/239575907.

[Fog14] Fog A.: *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*, 2014-12-07 ed. Technical University of Denmark, December 2014.

[Hai88] Haines E.: Spline surface rendering, and what's wrong with octrees. *Ray Tracing News 1*, 2 (1988).

[HMS06] Hunt W., Mark W. R., Stoll G.: Fast kd-tree construction with an adaptive error-bounded heuristic. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing* (2006), pp. 81–88. doi:10.1109/RT.2006.280218.

[IEE08] IEEE standard for floating-point arithmetic. IEEE Std 754-2008, Aug 2008. doi:10.1109/ieeestd.2008.4610935.

[Kaj86] Kajiya J. T.: The rendering equation. *ACM SIGGRAPH Computer Graphics 20*, 4 (1986), pp. 143–150. doi:10.1145/15886.15902.

[Kar12] Karras T.: Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the High-Performance Graphics Conference* (2012), pp. 33–37. doi:10.2312/EGGH/HPG12/033-037.

[Kee14] Keely S.: Reduced precision for hardware ray tracing in GPUs. In *Proceedings of the High-Performance Graphics Conference* (2014), pp. 29–40. doi:10.2312/hpg.20141091.

[KK86] Kay T. L., Kajiya J. T.: Ray tracing complex scenes. *ACM SIGGRAPH Computer Graphics 20*, 4 (1986), pp. 269–278. doi:10.1145/15886.15916.

[Kon12] Konsor P.: Performance benefits of half precision floats, August 2012. referenced: 3/26/2015. URL: `https://software.intel.com/en-us/articles/performance-benefits-of-half-precision-floats`.

[Kos13] Koskela M.: *Real Time Depth of Field Rendering.* BSc thesis, Tampere University of Technology, Finland, December 2013.

[KVJ*15] Koskela M., Viitanen T., Jaaskelainen P., Takala J., Cameron K.: Using half-precision floating-point numbers for storing bounding volume hierarchies. In *Proceedings of the 32nd Computer Graphics International Conference* (2015).

[LGS*09] Lauterbach C., Garland M., Sengupta S., Luebke D., Manocha D.: Fast BVH construction on GPUs. *Computer Graphics Forum 28*, 2 (2009), pp. 375–384. `doi:10.1111/j.1467-8659.2009.01377.x`.

[LKA13] Laine S., Karras T., Aila T.: Megakernels considered harmful: Wavefront path tracing on GPUs. In *Proceedings of the High-Performance Graphics Conference* (2013), pp. 137–143. `doi:10.1145/2492045.2492060`.

[LNOM08] Lindholm E., Nickolls J., Oberman S., Montrym J.: nVidia Tesla: A unified graphics and computing architecture. *IEEE Micro 28*, 2 (2008), pp. 39–55. `doi:10.1109/MM.2008.31`.

[LSL*13] Lee W.-J., Shin Y., Lee J., Kim J.-W., Nah J.-H., Jung S., Lee S., Park H.-S., Han T.-D.: SGRT: A mobile GPU architecture for real-time ray tracing. In *Proceedings of the High-Performance Graphics Conference* (2013), pp. 109–119. `doi:10.1145/2492045.2492057`.

[Lux15] LuxRender team: LuxRender, GPL physically based renderer v1.4, 2015. referenced: 6/17/2015. URL: `http://www.luxrender.net/en_GB/index`.

[MB90] MacDonald J., Booth K.: Heuristics for ray tracing using space subdivision. *The Visual Computer 6*, 3 (1990), pp. 153–166. `doi:10.1007/BF01911006`.

[Mea82] Meagher D.: Geometric modeling using octree encoding. *Computer Graphics and Image Processing 19*, 2 (1982), pp. 129–147. `doi:10.1016/0146-664X(82)90104-6`.

[MW06] Mahovsky J., Wyvill B.: Memory-conserving bounding volume hierarchies with coherent raytracing. *Computer Graphics Forum 25*, 2 (2006), pp. 173–182. doi:10.1111/j.1467-8659.2006.00933.x.

[MWDI05] Mahovsky J., Wyvill B., Davis A., Ibrahim A.: Robust ray-bounding volume hierarchy traversal with reduced precision integer arithmetic. referenced: 7/13/2015. URL: http://www.researchgate.net/publication/267206360.

[nVi15] nVidia: *CUDA C Programming Guide: Design Guide*, 7.0 ed., March 2015.

[PBD*10] Parker S. G., Bigler J., Dietrich A., Friedrich H., Hoberock J., Luebke D., McAllister D., McGuire M., Morley K., Robison A., et al.: OptiX: A general purpose ray tracing engine. *ACM Transactions on Graphics 29*, 4 (2010), pp. 66:1–66:13. doi:10.1145/1778765.1778803.

[PH10] Pharr M., Humphreys G.: *Physically Based Rendering: From Theory To Implementation*, 2nd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[PH11] Patterson D. A., Hennessy J. L.: *Computer Organization and Design: The Hardware/Software Interface*, 4th ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[PL10] Pantaleoni J., Luebke D.: HLBVH: Hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proceedings of the High-Performance Graphics Conference* (2010), pp. 87–95. doi:10.2312/EGGH/HPG10/087-095.

[Pur04] Purcell T. J.: *Ray Tracing on a Stream Processor*. PhD thesis, Stanford University, CA, USA, March 2004.

[SFD09] Stich M., Friedrich H., Dietrich A.: Spatial splits in bounding volume hierarchies. In *Proceedings of the High-Performance Graphics Conference* (2009), pp. 7–13. doi:10.1145/1572769.1572771.

[SWW*04] Schmittler J., Woop S., Wagner D., Paul W. J., Slusallek P.: Realtime ray tracing of dynamic scenes on an FPGA chip. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware* (2004), pp. 95–106. doi:10.1145/1058129.1058143.

[Wal07] Wald I.: On fast construction of SAH-based bounding volume hierarchies. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing* (2007), pp. 33–40. doi:10.1109/RT.2007.4342588.

[WBB08] Wald I., Benthin C., Boulos S.: Getting rid of packets – efficient SIMD single-ray traversal using multi-branching BVHs. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing* (2008), pp. 49–57.

[WBS07] Wald I., Boulos S., Shirley P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1 (2007), pp. 6:1–6:18.

[WGBK07] Wald I., Gribble C. P., Boulos S., Kensler A.: *SIMD Ray Stream Tracing-SIMD Ray Traversal with Generalized Ray Packets and On-the-fly Re-Ordering.* Tech. rep., Scientific Computing and Imaging Institute University of Utah, USA, August 2007.

[WH06] Wald I., Havran V.: On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the IEEE Symposium on Interactive Ray Tracing* (2006), pp. 61–69. doi:10.1109/RT.2006.280216.

[Whi79] Whitted T.: An improved illumination model for shaded display. *ACM SIGGRAPH Computer Graphics 13* (1979), pp. 343–349. doi:10.1145/1198555.1198743.

[WIP08] Wald I., Ize T., Parker S. G.: Fast, parallel, and asynchronous construction of BVHs for ray tracing animated scenes. *Computers & Graphics 32*, 1 (2008), pp. 3–13. doi:doi:10.1016/j.cag.2007.11.004.

[WK06] Wächter C., Keller A.: Instant ray tracing: The bounding interval hierarchy. In *Proceedings of the 17th Eurographics Conference on Rendering Techniques* (2006), pp. 139–149. doi:10.2312/EGWR/EGSR06/139-149.

[WMG*09] Wald I., Mark W. R., Günther J., Boulos S., Ize T., Hunt W., Parker S. G., Shirley P.: State of the art in ray tracing animated scenes. *Computer Graphics Forum 28*, 6 (2009), pp. 1691–1722. doi:10.1111/j.1467-8659.2008.01313.x.

[WSS05] Woop S., Schmittler J., Slusallek P.: RPU: A programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics 24*, 3 (2005), pp. 434–444. doi:10.1145/1073204.1073211.

[WWB*14] Wald I., Woop S., Benthin C., Johnson G. S., Ernst M.: Embree: A kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics 33*, 4 (2014), pp. 143:1–143:8. doi:10.1145/2601097.2601199.

[YLM06] Yoon S.-E., Lauterbach C., Manocha D.: R-LODs: Fast LOD-based ray tracing of massive models. *The Visual Computer 22*, 9-11 (2006), pp. 772–784. doi:10.1007/s00371-006-0062-y.