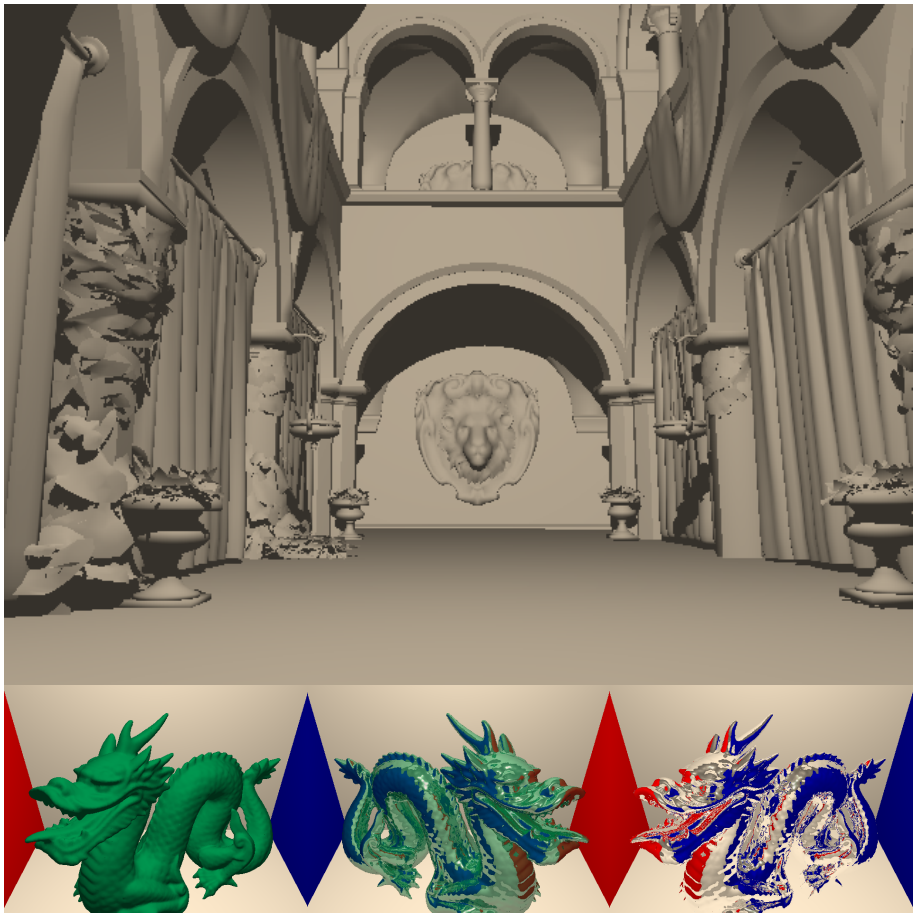
Efficient Ray Tracing of Dynamic Scenes on the GPU

Master's Thesis



Asger Dam Hoedt
asgerhoedt@gmail.com
20051770

Thomas Sangild Sørensen
sangild@cs.au.dk
Supervisor

Department of Computer Science
Aarhus University

February 2011

Abstract

The topic of this thesis is ray tracing dynamic scenes and doing that efficiently while harnessing the massive computational power of today's graphics cards. It is motivated by the ever increasing interest in ray tracing and global illumination for creating effects in movies, but also the increased usage of 2D and 3D ray tracing in modern computer games.
The thesis explores different techniques for creating hierarchical acceleration structures for ray tracing, specifically binary kd-trees, and how to create these efficiently on graphics processing units.
The quality of a kd-tree is defined as the speed with which it can be used to ray trace a scene and a lot of previous research has been focused on creating kd-trees of high quality, which usually results in an increased construction time. The goal of this thesis is to explore the relationship between construction speed and tree quality for kd-trees. I will focus on dynamic scenes, where the kd-tree must be rebuild before an image is rendered, and investigate if the overall time to ray trace a scene can be increased by producing acceleration structures faster, but at a lower quality.
As part of the thesis I will develop an optimized ray tracer to evaluate the quality of the kd-trees produced. The total time spent constructing and ray tracing the kd-tree will be used to estimate if there is a potential performance gain to be had by quickly producing kd-trees of lower quality.

Contents

Chapter 1

Introduction

Focus is a matter of deciding what things you're not going to do.
John Carmack

Rendering is the process of converting a scene description into an image and lies at the heart of *computer graphics*. The ability to render complex scenes realistically or distinctly is vital in several areas; computer games, movies and even medical imaging. A scene is made up of models, which can consist of several thousand geometric primitives, usually triangles. Information about these triangles are stored at the vertices and can be its position, a normal perpendicular to its surface or its color among other things. Such information stored at triangle vertices are called *vertex attributes*.

When real-time rendering is needed, the technique of choice for the last one and a half decade has been *rasterization*. In rasterization a geometric primitive's vertex attributes are mapped onto a *raster*[1] and used to calculate the color of individual raster cells. This technique is so popular in the gaming industry that processing units were created specifically for rasterization, the *Graphics Processing Unit* or *GPU* for short. Due to the industry's ever increasing demand for more detailed models and visual effects, the GPUs have seen a massive increase in both computational power and memory bandwidth over the last decade. Unfortunately, even with all this power, certain aspects of rendering are still not easily solved by rasterization. *Reflection* and *refraction* effects on non-flat surfaces are still notoriously hard to recreate. The reason is that the reflection of complex objects can not easily be mapped to a two dimensional grid, such as the raster. Reflections of complex objects can be approximated by *environment mapping* or *cube mapping*, of which a short describtion can be found on figure 1.1. A problem with cube mapping is that the scene must to be rendered once for each side of the cube map, which increases the cost of rendering a scene drastically. Another shortcoming of cube mapping is that it does not support *self reflection*, since it is only the surrounding environment that is rendered onto the map.

An alternative to rasterization is *ray tracing*, which elegantly solves reflection and refraction by tracing rays from the viewer's eye and into the scene, spawning and tracing new reflection- and refraction rays as needed when geometry is intersected. A comparisson of cube mapping and ray tracing can be seen on figure 1.2. The reflecting Stanford Dragon on figure 1.2 has been rendered by me, as have all images in this thesis unless explicitly stated Notice how the ray traced dragons backside reflects its neck, while the cube mapped version only reflects the surrounding box. Advanced lighting techniques that produce photorealistic images are also based on ray tracing. One such technique is *photon mapping*, which can accurately reproduce the effects of light bouncing of reflective surfaces, caustics and color bleeding.

The increased realism that can be achieved by using ray tracing does come at a high computational cost, which has previously made it unattractive for interactive applications or dynamic scenes. Nonetheless, the continued increase in computational power of both CPUs and GPUs, coupled with research into the area of *interactive ray tracing*, has yielded some remarkable results. Scenes of approximately 100k triangles can now be ray traced in real-time, even with effects such as shadows, reflection and refraction added. This makes ray tracing an increasingly more interesting alternative to rasterization. In this thesis I will examine *ray tracing of dynamic scenes*, with the express goal to minimize the time between modifying the scene and presenting a viewer with visual feedback of the change. This would be very

[1] A flat, 2D grid.

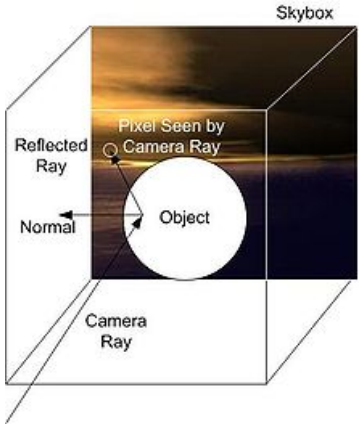
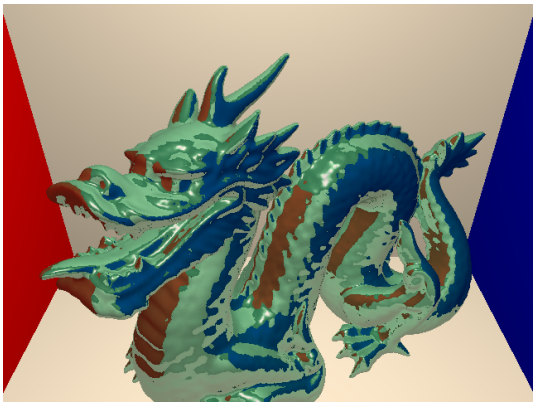
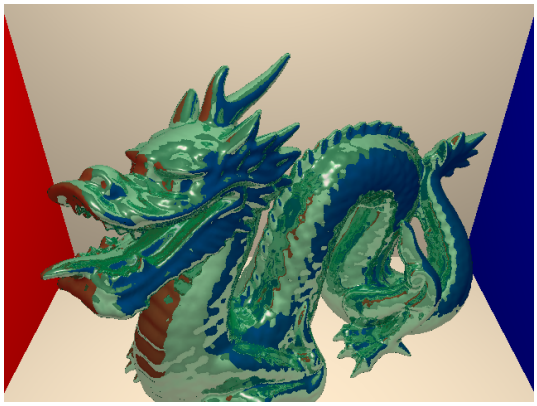


Figure 1.1: A visualization of cube mapping. The environment is rendered onto the sides of the cube and then mapped onto the object inside the cube map. The mapping is performed by using the calculated reflection vector as an index into the cube map.
Image taken from `en.wikipedia.org/wiki/Reflection_mapping`.



(a) A cube mapped reflecting dragon.



(b) A ray traced reflecting dragon. Notice the self reflection on the dragon's back and inside its mouth.

Figure 1.2: An example of the difference between using cube mapping or ray tracing for reflections.

interesting in the film industry, where ray tracers are used to create *CGI*[2] effects and 3D artists need fast visual feedback while modifying the scene.

To improve the performance of ray tracers, several acceleration structures have been developed. The most popular structures are *hierarchical data structures*, such as trees, and ray tracers that traverse these are called *hierarchical ray tracers*. When a ray traverses a hierarchical data structure it is in search of the nearest leaf node, which contains references to the geometry nearest to the ray. If the ray finds that it did not see, or *intersect*, any of the geometry in the current leaf node, it advances beyond that leaf and performs another traversal of the structure.

The structure employed in this thesis is the *kd-tree*, a binary, space partitioning tree-structure, which recursively subdivides k-dimensional geometry by splitting it with an *axis aligned splitting plane*. Each interior node in the tree contains a splitting plane and a reference to the location of its left and right children, while the leafs contain references to the geometry associated with them. If a leaf node is split by a splitting plane, the geometry associated with that leaf must be associated with its two newly constructed child nodes. The quality of a kd-tree is defined as the ease with which it allows a scene to be ray traced and is affected by the choice of splitting planes. A lot of computational power can go into finding optimal splitting planes, as there are infinitely many possible *splitting plane candidates*, or *split candidates*, to consider.

In order to facilitate dynamic scenes, the data structure must also be dynamic. It is possible to dynamically add and remove elements to a kd-tree, but doing so can degrade the quality of the kd-tree or its subtrees. If a subtree's quality has degraded too much, it will have to be restructured to facilitate fast ray tracing again. In the worst case scenario this restructuring needs to be performed on the entire tree and is equivelant to a complete reconstruction. Because of this, the algorithms for creating and restructuring kd-trees needs to be very fast, and might even have to sacrifice tree quality for improved construction speed. This tradeoff between speed and quality is also what makes dynamic scenes interesting compared to static scenes. Ray tracers rendering static scenes can use as much time as needed to produce acceleration structures of high quality. This may not be possible in dynamic scenes, where a user modifying the scene will want visual feedback of the modifications made as fast as possible.

In general there are three ways to optimize ray tracing with respect to dynamic scenes to achieve maximum efficiency:

1. **Building a higher quality acceleration structure -** An acceleration structure of higher quality will reduce the time it takes the ray tracer to find the nearest intersecting point between a ray and the scene. Algorithms for producing kd-tree's of different quality is the topic of section 3.1.

2. **Build the acceleration structure faster -** As described above, the acceleration structure may need to be entirely reconstructed each time a dynamic scene is rendered. Being able to rebuild it fast is therefore crucial. One way to ensure a faster reconstruction is by reducing the time spent deciding where to place the splitting plane, which can result in trees of lower quality.

3. **Create a faster ray tracer -** Several optimizations exist that improve the speed of a ray tracer without modifying the underlying acceleration stucture. Such optimizations will be discussed in section 4.2.

As observed at the beginning of this chapter, GPUs have grown quite powerful over the last decade, and since the introduction of programmable GPUs and NVIDIA's *CUDA*[3] framework, many algorithms have been succesfully ported to utilize the resources of the GPU. In this thesis I too will use the massive computational power of the GPUs to accelerate the creation of data structures and ray tracing them. The most compelling reason to do this, is that it leaves the CPU free to handle other aspects of a graphics application, such as user input and network communication.

[2] Computer-generated Imagery
[3] Compute Unified Device Architecture.

1.1 Previous Work

If you want to make an apple pie from scratch, you must first create the universe.
Carl Sagan

Arthur Appel[6] is credited as the being the first to describe the basic idea of ray casting, applying it to solve the *hidden surface problem* and to enhance the perception of depth by computing shadows in opaque polygonal scenes. Whitted[23] extended the idea of ray casting into the general recursive ray tracing algorithm still used today. If a ray would hit a surface, it could generate any number of new rays, reflection, refraction or shadow, depending on the surface's material proporties.

Since then a lot of time and effort has gone into improving the performance of ray tracing and several data structures have been proposed with this in mind. In 1976 Clark[7] was the first to suggest using *bounding volumes* to perform geometry culling and in 1986 Goldsmith and Salmon[11] extended this idea with an algorithm for automatically building *bounding volume hierarchies*, *BVH*'s, topdown. Fujimoto, Tanaka and Iwata[10] introduced the use of *uniform voxel grids* in 1986. Kaplan[14] introduced the use of kd-trees as a spatial partitioning scheme. To decide where to place the splitting plane he used the now standard *spatial median splitting* algorithm. Spatial median splitting places the splitting plane at the middle of a node's bounding box along some axis, usually either the longest or an axis chosen in a *round robin* fassion.

The idea of automatically creating hierarchical acceleration structures have since been revisited and improved upon countless times. One of the most important improvements was the introduction of the *Surface Area Heuristic*, *SAH*, generally attributed to MacDonald and Booth[15]. SAH estimates the expected cost of ray tracing a node's two child nodes with respect to some splitting plane. Given a list of splitting planes, the expected cost for all these planes can then be calculated and the plane with least cost is chosen as the splitting plane.

In his ph.d. thesis[12] from 2000, Havran argued that the kd-tree was the best known acceleration structure for ray tracing. While a lot of new data structures have since appeared, the kd-tree is still one of the most widely used structures due to its low memory footprint, fast ray/plane intersection test and countless research papers devoted to both optimal creation and traversal. For these same reasons this thesis will focus on the use of kd-trees as its acceleration structure.

Because of the lack of looping and branching on early programmable graphics hardware, the first completely GPU based ray tracers had to make use of non hierarchical acceleration structures like grids[18]. Grid's, however, do not scale aswell as hierarchical structures and, in large sparse scenes, fine-grained grids run the risk of wasting memory on a lot of empty cells, while more coursely grained grids might store most of the geometry in a few cells and thus not partition it effectively.

With the addition of branching and looping on graphics hardware, several GPU based hierarchical ray tracers were proposed. A known optimization to CPU based hierarchical ray tracers is to use a stack of neighbouring nodes that the ray could possibly traverse. This is used as a means to prevent restarting ray tracing from the root of the acceleration structure, if a ray does not intersect any primitives in its current leaf. Small amounts of available memory per thread on the graphics card made this optimization technique infeasable for GPU solutions. Instead Popov et al.[17] in 2007 introduced a stackless ray tracer rivalling CPU ray tracers in speed, which preprocessed the kd-tree and adds *ropes*, or references, between neighbouring nodes. These ropes would then allow the rays to quickly locate neighbouring nodes while traversing the kd-tree. Concurrently Horn et al.[13] proposed a different but equally effective solution. Instead of storing the entire stack of possible neighbouring nodes, a *short-stack* was stored in memory, containing only the N lowest nighbouring nodes that a ray could traverse. Horn et al. also introduced an optimization called *push-down*, where each ray did not restart at the root of the tree, but instead at the root of the smallest subtree enclosing the ray. Finally they showed how this could be implemented together with *packet tracing*, where several rays are traced in packets by one thread to amortize the cost of traversing the tree.

Although ray tracing on graphics hardware had been made as fast as its CPU counterparts by 2007, creating kd-trees on the GPU had still not been done efficiently.

This changed in 2008 when Zhou et al.[25] introduced *breath-first* tree creation on the GPU. Instead of creating kd-trees in *depth-first* manner, in which only one node was processed at a time, their breath-first algorithm made it possible to work on hundreds or thousands of nodes in parallel, allowing it to scale much better with the architecture of graphics cards. For the uppermost nodes in the tree they proposed

to parallelize the calculation of the node split cost across all geometric primitives, creating thousands of threads. For the lower part of the tree, where thousands of nodes needed to calculate their split cost, computations were simply parallelized over all available nodes.

1.2 Goals

In this thesis the goal is not to produce images of photorealistic quality, or create an interactive ray tracer [4]. Instead this thesis will explorer ray tracing acceleration structures, specifically the kd-tree, and its impact on ray tracing efficiency for dynamic scenes.

The main topic in this thesis is the relationship between the time spend constructing a kd-tree and its resulting quality, i.e. how fast can it accelerate ray tracing. Building on the kd-tree implementation presented by Zhou et al.[25], I will investigate different parts of the kd-tree construction phase for dynamic scenes and whether it is worthwile in dynamic scenes to sacrifice tree quality in order to gain faster kd-tree construction. The two parts of the kd-tree creation phase that will be investigated is the choice of splitting plane and how geometry is associated with child nodes after a split. As part of this investigation I will describe several different solutions and show how to implement them efficiently on dataparallel GPUs using CUDA.

The kd-trees created by the different methods above will be evaluated by how fast they can be constructed and how efficiently a ray tracer can traverse them to render a scene. During evaluation I will always perform a complete rebuild of the kd-tree before rendering a scene. This is done to ensure that my tests capture the worst case scenario for dynamic scenes, which is a complete update of the entire scene and thus a complete rebuild. The goal of this thesis is then to find a kd-tree configuration that minimizes the total time spent both recreating and traversing the acceleration structure and evaluate the tradeoff between construction speed and tree quality.

I will furthermore discuss the ray tracers used to evaluate the quality of the kd-trees. To provide a fair and useful comparison of a kd-tree's construction time and its quality, I will need to create a highly optimized ray tracer. These optimizations are important as a fully optimized ray tracer can render scenes up to 70% faster than a basic implementation and does so independent of the underlying kd-tree and its construction schemes. While I will be discussing and applying these optimizations, such as the short-stack optimization from Horn et al.[13], the topic is secondary to kd-trees and merely included for evaluation purposes.

Given the added overhead of continuously rebuilding the kd-trees, an interesting question is whether or not we even need them. I therefore present an *exhaustive ray tracer*, which does not use any acceleration structure and thus intersects every ray with every triangle. The exhaustive ray tracer will be compared to the hierarchical ray tracers and hopefully motivate the continued use of acceleration structures for dynamic scenes.

1.3 Overview

The rest of the thesis is structured as follows:

Chapter 2 introduces NVIDIA's CUDA framework. Here I will describe its thread and memory model. I will then go on to describe a new primitive proposed by Sengupta et al[19], which will be needed during kd-tree construction, e.g. when assigning triangles to child nodes after their parent node has been split. The last part of this chapter will focus on optimization techniques specific to CUDA and apply these incrementally in a case study of a global minima algorithm.

Chapter 3 is devoted to discussing the construction of kd-trees. The first part of this section deals with the general kd-tree construction algorithm. Here I will present several algorithms for choosing the splitting plane and discuss three different approaches used to associate triangles with leaf nodes. The second part of chapter 3 deals with the actual implementation of kd-trees on a GPU and will describe how the nodes are structured in memory and how to construct binary trees effectively on dataparallel hardware.

[4] This goal alone can be achieved trivially by reducing the complexity of the ray traced scene or lowering the image resolution.

Having introduced kd-trees, chapter 4 will deal with the algorithms for traversing such trees and ray tracing a scene. Here several optimizations to a basic hierarchical ray tracer will be discussed and incrementally added. First though, an exhaustive ray tracer is presented and will be used in the Conclussion to motivate the use of hierarchical data structures. This chapter also includes a discussion of two triangle intersection algorithms with respect to achieving maximal performance on GPUs.

In chapter 5 I will first evaluate the performance of several different ray tracers. I will then use the ray tracer that generally performs best to evaluate the quality of the different kd-trees created. The metric used to determine which kd-tree construction algorithm performs best will be the total rendering time, i.e. the sum of the construction time and ray tracing time.

I will then conclude my work by discussing my results and their implications for the future of ray tracing dynamic scenes. Finally in chapter 7 I will address future improvements based on the experience I have gained while working on this thesis.

Chapter 2

Understanding CUDA

Hardware: The parts of a computer system that can be kicked.
Jeff Pesis

Because of the evergrowing demand for new visual effects and more detail in computer games and other 3D graphics applications, GPUs have seen a massive increase in computational power and potential memory bandwidth over the last decade. This is evidenced by figure 2.1, which compares the development of NVIDIA's GPUs with that of Intel's CPUs. These figures illustrate the potential performance gain associated with implementing computationally heavy solutions on todays GPUs instead of CPUs. The first figure, figure 2.1a, compares the computational capacity and shows how the theoretical number of floating point operations have increase over the last decade, with GPUs being able to perform nearly 1500G floating point operations per second as of the GeForce 480 GTX series, while Intel CPUs only reach around 200G operations. The second figure, figure 2.1b, compares the memory bandwith of GPUs and CPUs. Memory bandwidth is the rate at which data can be read from or stored into memory by one of the processing units and is around 6 times higher for the GPU than the CPU.

Utilizing the GPU's computational power and memory bandwidth effectively, however, is not straightforward and in order to create algorithms for the GPU that are executed faster than their CPU counterparts, we need an understanding of how the GPU works.

The reason behind the difference in computational power and memory bandwidth seen in figure 2.1, is that the CPU is designed for efficient handling of control flow and with a large cache at its disposal. This enables it to handle all kinds of applications, not just computationally intense ones. The GPU on the other hand is designed for high throughput of small, arithmetically intense, *data parallel programs*[1] called *kernels*, which is exactly what is required of a graphics card that processes thousands of independent vertices and performing the same color calculations, or *shading*, on hundreds of thousands of *fragments*[2].

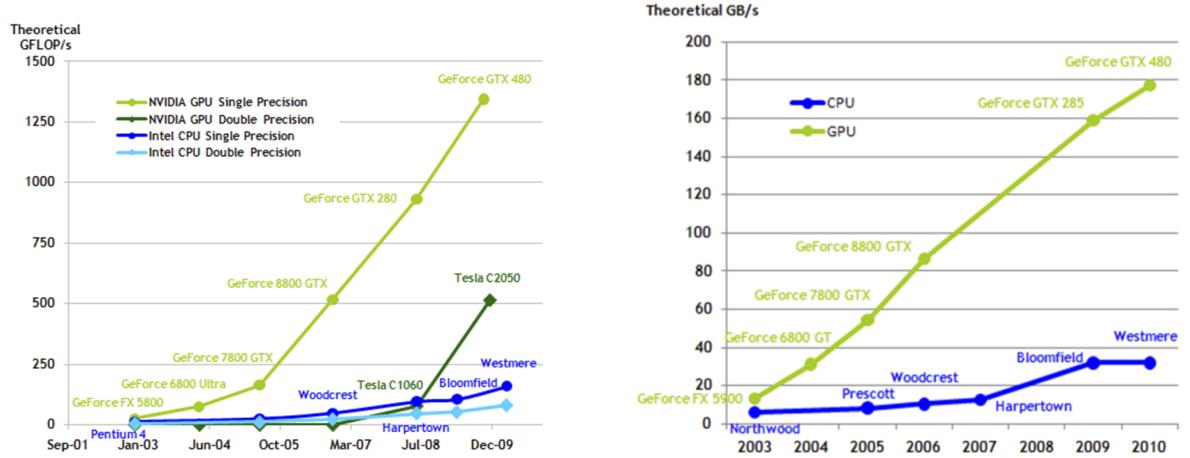
Since the graphics card is designed to perform vertex and fragment processing independently of their respective neighbouring threads, this means that no assumptions can be made about which thread is where in its execution when programming the graphics card. This presents a problem in cases where the n'th thread depends on information from all previous $n-1$ threads, e.g. when shuffling elements in a list or calculating the minimum or maximum values of a list of vertices. NVIDIA's CUDA framework remedies this somewhat by providing synchronization primitives, but these can only synchronize a subset of the running threads. The general problem of passing information between threads is therefore not solved by CUDA's synchronization primitive alone.

An obvious solution is of course to perform easily parallelisable operations on the graphics device and leave the rest for the CPU, or *host*. However, the following quote comes to mind:

> *It is important to include the overhead of transferring data to and from the device in determining whether operations should be performed on the host or on the device.*
>
> [8]

So once it has been decided to use the graphics card, data should not be transfered back and forth

[1] Each instruction is concurrently applied on multiple data elements.
[2] A fragment contains all the data needed to generate the color of geometry in a single raster cell.



(a) Comparison of computational capacity in billions of floating point operations per second.

(b) Comparison of memory bandwidth in gigabytes per second.

Figure 2.1: A comparison of the development of floating point operations per second and memory bandwidth of GPUs and CPUs over the last decade.
The figures are from chapter 1 in [9].

between the CPU and GPU too often. The transfer overhead would in many cases outweigh the performance increase gained by using the GPU in the first place.

Finding effective GPGPU[3] solutions to the above mentioned problems is the motivation for this chapter, which is structured as follows. First we shall look at how threads and memory are organized in CUDA. Understanding this will be critical in developing efficient GPGPU solutions. Then a section will present a scan primitive for GPU computing. In this section I will show why this new primitive is useful and present a case where the data in a list is shuffled by threads running in parallel. Something that could not have been done effectively without the scan primitive. In the chapters final section I will discuss a couple of general CUDA optimization techniques, while applying them to a case study.

2.1 The Architecture of CUDA

Understanding the architecture of CUDA requires us to first understand the relationship and layout of threads and the different kinds of memory available to these threads.

CUDA's architecture is based on *multiprocessors*. Unlike a single threaded CPU, a multiprocessor is able to execute hundreds of threads concurrently. Managing these threads is made easier by the multiprocessors' *SIMT*[4] architecture, in which a single instruction in a program is relayed to several threads in parallel. A multiprocessor is equipped with fast, but limited local memory, which can be used by threads currently executing on the multiprocessor. These multiprocessors enable the graphics card to handle execution of more than a million threads per kernel launched.

2.1.1 The Thread Hierarchy

At the lowest level, threads are scheduled and executed completely parallel in small groups called *warps*[5], with a warp size of 32 threads on current generation hardware. A thread in a warp has its own instruction counter and register state, and can therefore branch independently of neighbouring threads. However, a warp can only execute one specific instruction at a time. The following example should help clarify this.

if threadID $<$ 16 **then**
  $x \leftarrow threadID$
else

[3] General-Purpose computation on Graphics Processing Units.
[4] Single-Instruction, Multiple-Thread
[5] The term originates from weaving, the first parallel thread technology.[9]

$x \leftarrow 32 - threadID$
end if

The first 16 threads in the warp will evaluate the condition to true and thus perform the assignment $x \leftarrow threadID$, while the next 16 threads will evaluate it to false and execute the alternate statement $x \leftarrow 32 - threadID$. Since a warp can only perform one distinct instruction at a time, it will have to first execute $x \leftarrow threadID$, leaving the last 16 threads idle. It next executes the else branch, meaning the first 16 threads are now left idling. While this example shows how branching can hurt performance, when all threads in a warp do not take the same execution path, knowning that all threads in a warp are always executed synchronized can also be very beneficial, which I will discuss in section 2.3.5.

Warps are organized into 3 dimensional *blocks* and all threads in a block are expected to reside on the same multiprocessor, which provides a limit as to how many threads a block can contain. On current generation GPU's the limit can be up to 1024 threads per block or multiprocessor. The size of blocks is chosen before a kernel is launched and all blocks allocated for that kernel will have the same size. Threads can lookup their *thread index* inside a block through the 3 dimensional *threadIdx* CUDA built-in variable. Being able to lookup a threads index is important for working with data. In the one dimensional case the n'th thread will usually process the n'th data element. Without the thread index this would not be possible. Executing blocks on the same multiprocessor enables us to synchronize the threads inside that block at specific points in the kernel. This enables threads inside blocks to cooperate on solving problems and allows them to share data through the multiprocessor's fast local memory without fearing race conditions

Blocks are themselves arranged into the uppermost part of the thread hierarchy, a 2 dimensional *grid*. The amount of data being processed usually defines how large the grid will be. Just like a thread can access its index, it can also lookup its *block index* inside the grid through the built-in variable *blockIdx*. A two dimensional thread hierarchy can be seen on figure 2.2. The kernels in this thesis will usually process the data as one dimensional linear lists, where a threads global index can be calculated as

$$index = blockDim.x * blockIdx.x + threadIdx.x$$

2.1.2 The Memory Model

With the thread hierarchy explained, the different memory spaces made available through CUDA can be described.

CUDA provides the programmer with three overall types of memory:

- *global memory* - which can be accessed by any thread at any time and persists across kernel launches.
- *shared memory* - which is shared by all threads in a block and only persists for as long as that block is active.
- *registers* - which are local to each thread and only exists as long as its thread.

In the following I have devoted a subsection to each memory space.

Global Memory

Global memory persists across kernel launches and is therefore perfect for storing input data to kernels and their results. Unfortunatly it also has the highest access time of the three memory spaces on CUDA devices. To avoid making lots of memory transactions to and from global memory, a warp will try to *coalesce* its threads' global memory accesses into as few transactions as possible. How well global memory access can be coalesced depends on the *compute capability* of the graphics hardware, with the coalescence restrictions on newer GPUs being more flexible. Suffice it to say here that it is preferable to access global memory sequentially, i.e. the n'th thread in a warp accesses the n'th data element.

Another restriction on global memory is that data accessed must be of size 1, 2, 4, 8 or 16 bytes, otherwise memory transactions will be broken up into multiple requests with an interleaved access pattern. Accesseing global data with an interleaved access pattern hinders effective coalescence and should therefore be avoided. Due to this, global memory access can be performed more efficiently when using

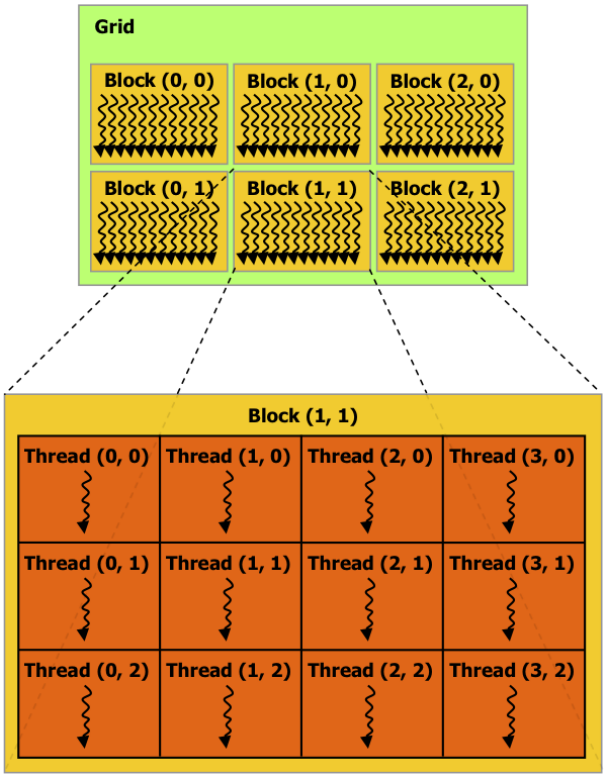


Figure 2.2: A figure of CUDA's thread hierarchy.
The figure is from chapter 2 in [9].

the CUDA built-in struct *float4* instead of *float3*, since *float3* takes up 12 bytes and will be split into two memory request with 8 and 4 bytes interleaved access patterns.

For more information on coalescence see Appendix G in [9], where the coalescence requirements for hardware of a specific compute capability is described.

The scheduler will also help with hiding latency from global memory access. If one warp is stalled while waiting for data, another warp that is resident on the same multiprocessor and ready to execute can be scheduled instead. Utilization of the graphics hardware is therefore heavily dependent on the number of resident warps and maximizing these can be important. Especially in cases where memory accesses are scattered, like they can become when the threads traverse trees, such as the kd-tree. One way to increase the number of resident warps will be introduced in the Registers and Local Memory section below.

Textures also reside in device memory space. Textures are read only 1D, 2D or 3D data arrays that provide a fast texture cache optimized for 2 dimensional spatial data locality. This can make texture access faster than pure global memory access, as a device memory read is only performed on a cache miss. On current generation hardware however, global memory can use shared memory as a cache, so textures will not be used in this project.

Shared Memory

Shared memory resides on the multiprocessor and is much faster than global memory. It is shared by all threads in the same block and allows them to share data.

A normal usecase for shared memory is local caching of global data, which is shared by multiple threads in the block. In this case the threads in a kernel will cooperate on loading the data into shared memory, then the kernel will perform a block-wide synchronization and proceed with operating on the data in shared memory. When the kernel has finished its computations, the data can be written back into global memory.

As mentioned above, on newer architectures multiprocessor memory can be used implicitly as a cache

for global memory.

Registers and Local Memory

Registers are part of the threads execution context and is the fastest kind of memory on the device. Since there is only a fixed amount of registers available per multiprocessor, the register usage of a kernel's threads can have a high impact on *occupancy*[6], which in turn can have an impack on the schedulers ability to hide memory latency.

The compiler employs different heuristics to minimize register usage, while ensuring that kernels run efficiently. Sometimes though, programmers may want to use even fewer registers for specific kernels, in order to maximize occupancy and global memory latency hiding. To this end they can aid the compilers heuristics by providing *launch bounds*. In CUDA two arguments can be given as launch bounds. The first tells the compiler the maximum number of threads per block that the kernel will ever be invoked with. The second argument tells the compiler how many blocks should be resident on a multiprocessor. The compiler can then use this information to derive upper bounds for the registers available per thread.

But the compiler cannot always simply reduce the number of registers to fit inside the launch bounds. If a programmer specifies that a kernel's threads only use 16 registers per thread, but each thread needs 20 registers to hold 20 distinct values, then those values have to be stored somewhere else. In that case the compiler can use *local memory*, which resides in device memory and thus has the same high aceess latency as global memory. Forcing a few registers into local memory to gain occupancy can be beneficial though, and since all non-idling threads will access sequentially stored local memory at the same time, there is a high probability that local memory access can be coalesced.

Kernel variables that the compiler will most likely place in local memory are:

- Any array that from the compilers point of view is dynamically indexed.
- Structures or arrays that are to large to fit inside the registers.
- Any variable in the kernel if the kernel has to many variables to place them all in register memory. This is referered to as *register spilling*.

2.2 The Scan Primitive

As explained above problems where the n'th thread requires information from all previous $n-1$ threads can be quite hard to solve on data parallel hardware. In this section I will outline a solution to the problem as presented by Sengupta et al[19] and give an example of how this solution can be used to sort a list of data into two lists. In section 3.2 we shall see that being able to perform such sorting is important when associating triangles with the children of a node that has been split.

An example taken from Sengupta et al[19] showcasing this problem is the calculation of a *prefix-sum*, which is a special case of *exclusive scan*. Exclusive scan takes as input a list of data, $[a_0, a_1, a_2, ...]$, and a binary operator, $\oplus$, with an identity element, i. The result of exclusive scan is then a new array with values $[i, a_0, a_0 \oplus a_1, a_0 \oplus a_1 \oplus a_2, ...]$. For prefix-sum the operator is $+$ and the identity element is 0. An example of prefix-sum can be seen below:

$in:$	3	1	7	0	4	1	6	3
$out:$	0	3	4	11	11	14	16	22	25

Calculating the prefix-sum for n elements on the CPU is trivial and can naïvely be done with $O(n)$ memory accesses by iterating over the data array from start to finish. A naïve implementation in the GPU however would require each thread to sum up every previous value on its own, which would require $O(n^2)$ memory accesses. Instead a data parallel algorithm has to be devised that allows threads to cooperatively solve the problem and doing that as efficiently as the CPU, i.e. with $O(n)$ memory accesses.

As can be seen on figure 2.3, Sengupta et al[19]'s work-efficient prefix-sum requires two passes over the data, one called *reduce* and another called *downsweep*. The object of the reduction phase is to calculate the sum of all elements in the input list. It does so by applying the *divide and conquer paradigm* and first solving the problem for subsections of the list. Since only threads belonging to the same block can

[6]The number of warps resident on a multiprocessor, relative to the maximum amount possible.

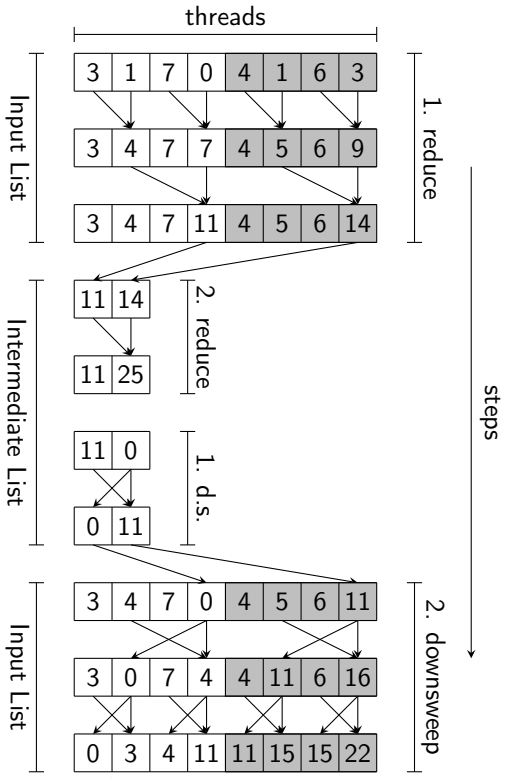


Figure 2.3: An example of segmented scan data flow. Cells with the same shading belong to the same block and the arrows represent data movement, either a copy or an application of $\oplus$.

only be synchonized, reduce only works on elements belonging to the same block. If the input list holds more elements than can be reduced by a single block, then reduce is simple applied to the result of the previous reduction until all values have been reduced. The downsweep phase then makes use of the values computed by reduce to produce the prefix-sum of the input list.

This is best described by the prefix-sum example on figure 2.3, where the prefix-sum of a list of 8 elements is computed. I have chosen to reduce two blocks to show how the first reduction result is combined into a new list and then reduced further. Cells with the same color represent threads belonging to the same block. The flow of data is represented as arrows, one arrow entering a cell represents the data being copied, while two arrows entering a cell represents an application of $+$ on the two data elements and the result of that application is then stored in the cell pointed to. After three steps each individual block has reduced its own elements and the results are therefore copied to an intermediate list for further reduction. When reduce is complete, the downsweep phase begins. Downsweep first sets the last element in the current list to the identity element, 0 in the case of prefix-sum, and then proceeds to iteratively compute the prefix-sum. The result of applying the reduce and downsweep is the prefix-sum of the input list, which can be seen in the final list in figure 2.3.

For specific details and further implementation optimizations see Sengupta et al[19].

As can be seen on figure 2.3, the parallel scan performs $2 \cdot O(n + n/2 + n/4 + ... + 1) = O(n)$ memory accesses, since the memory accessed is cut in half for each iteration of the reduce and down-sweep kernels.

Now the question is: *why is this useful?*. The answer is that the prefix-sum is vital to shuffling elements in lists on the GPU. Imagine a list of triangles that have been split by a splitting plane and must now be sorted into a new array. All the triangles in front of the plane will be sorted to the left and the triangles behind it sorted to the right.

$$\begin{array}{rcccccc} \mathit{triangles}: & [t_0 & t_1 & t_2 & t_3 & t_4 & t_5] \\ \mathit{side}: & [r & l & r & l & l & r] \end{array} \Rightarrow \begin{array}{cccccc} [t_1 & t_3 & t_4 & t_0 & t_2 & t_5] \\ [l & l & l & r & r & r] \end{array}$$

This needs to be done every time a tree node is split into two child nodes, so being able to do this efficiently on the GPU is imperative to creating kd-trees efficiently.

Obviously the individual threads in a split kernel will have no idea which address to move the triangle to, since that depends on every other thread. E.g. t_0 must move its data to entry 3, which it can only do if it knows that it holds the first r and that there are three l's. However, by first computing the prefix-sum of the *side* list, while adopting the convention that $r = 0$ and $l = 1$, splitting the triangles become quite easy. Using the example above the prefix-sum becomes

$$\begin{aligned} side: &\quad [0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0] \\ prefix\text{-}sum: &\quad [0 \quad 0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 3] \end{aligned}$$

where the last element in $prefix\text{-}sum$ represents the total number of triangles moved left, denoted nf.

The observant reader will have noticed that the prefix-sum actually calculates the addresses where the triangles on the left side should be moved to. All that remains is then to calculate the addresses of the triangles moved right. This is done using $right_i = i - prefix\text{-}sum_i + nf$.

$$right: \quad [3 \quad 4 \quad 4 \quad 5 \quad 5 \quad 5]$$

The address that a thread should move its triangle to is then simply $address_i = side_i == l \; ? \; prefix_i : right_i$ and becomes

$$address: \quad [3 \quad 0 \quad 4 \quad 1 \quad 2 \quad 5]$$

which will divide the triangles into their respective sides, just like we wanted.

Sometimes the splitting planes used to split the nodes of a kd-tree will intersect with some of the triangles in the scene and the triangle needs to be associated with both the left and right side. In section 3.2.1 I will show how this specific sorting problem can be solved.

2.3 Optimization Techniques: Reduction

In the previous section we saw that it is important to be able to perform *reductions* efficiently on the GPU. A reduction is the processing of a list of data in some order while building a return value.

In this section I present a naïve reduction algorithm and incrementally apply CUDA specific optimizations to it. At the end of the section I present a table summarizing the decrease in execution time after each applied optimization. The reduction example will process the data using the binary operator **min**, with identity element ∞, and return the smallest value from the input list, but the algorithm presented can easily be generalized to any binary operator and its identity element. The naïve algorithm is based on the reduction algorithm presented above and will perform $O(n)$ memory accesses, which is the best we can hope for. Instead of improving the time complexity, the following optimizations will instead focus on making better use of the GPU by hiding latency, using faster memory where available and reduce the number of calculations made.

The reduction algorithm will reduce values for individual blocks. If the input list is too large to be reduced by one block, then the algorithm can be run recursively on its own output, until a single result is found.

The algorithm presented assumes that the length of the input list is a power of two. If that is not the case then either the data could be preprocessed or the kernel itself can pad the data with the binary operators identity element.

2.3.1 Naïve implementation

The naïve algorithm, presented in algorithm 1, is a straight forward implementation of the interleaved access pattern shown in figure 2.3. The algorithm uses the modulo operator to distinguish which threads are done and which should continue to perform reductions. All intermittent values are written back into global memory, to allow other threads to access the reduced values when needed. When the block has finished the reduction, the result is returned by the first thread.

Algorithm 1 Naïve reduction

```
procedure Reduce0
    in values : Number List; id : Integer; elements : Integer
    out result : Number
begin
  offset ← 1
  while id + offset < elements do
    if id mod (offset * 2) = 0 then
      values[id] ← min( values[id] , values[id + offset] )
    end if
    offset ← offset * 2
    synchronize
  end while
  if id = 0 then
    result ← values[0]
  end if
end
```

2.3.2 Coalesced Memory Access

Of course there are several inefficiencies to correct in algorithm 1. To start with we will focus on global memory access and update it to allow the warp to perform coalesced memory access. Since we are interested in a global reduction it will not matter in which order values are compared. This allows the interleaved access pattern to be exchanged with a sequential one, which then lets the warp perform coalesced memory access.

Two pleasent side effects to this change, which is shown in algorithm 2, is that the slow modulo operator has disappeared. Warp utilization has also increased dramatically, since the first $offset$ threads in a block perform the same computations per iterations.

Algorithm 2 Coalesced reduction

```
procedure Reduce1
    in values : Number List; id : Integer; elements : Integer
    out result : Number
begin
  offset ← elements/2
  while offset > 0 do
    if id < offset then
      values[id] ← min( values[id] , values[id + offset] )
    end if
    offset ← offset/2
    synchronize
  end while
  if id = 0 then
    result ← values[0]
  end if
end
```

2.3.3 Working from Shared Memory

Even with global memory access coalesced into fewer memory transactions, continuously accessing global memory is still quite slow. To remedy this the implementation in algorithm 3 first copies the data into shared memory before performing any reductions. As can be seen the threads are working together to perform the copy. Instead of all threads having to copy the entire data, each thread only fills the id'th cell in the shared list and leaves the other cells to be filled by the other threads. The subsequent

synchronization ensures that all the data has been copied before proceding with the reduction. Since the threads are still accessing data sequentially, this change preserves the coalesced memory fetches.

Algorithm 3 Shared memory reduction

```
procedure Reduce2
    in values : Number List; id : Integer; elements : Integer
    out result : Number
begin
  sValues : shared Number List
  sValues[id] ← values[id]
  synchronize
  offset ← elements/2
  while offset > 0 do
    if id < offset then
      sValues[id] ← min( sValues[id] , sValues[id + offset] )
    end if
    offset ← offset/2
    synchronize
  end while
  if id = 0 then
    result ← sValues[0]
  end if
end
```

2.3.4 Using Registers

Shared memory is quite fast, but registers are even faster. Examining the statement

$$sValues[id] \leftarrow \textbf{min}(\ sValues[id]\ ,\ sValues[id + offset]\ )$$

we can see that the *id*'th thread will always access the value stored in $sValues[id]$ and is the only thread writting to that cell. This provides us with the possibility to use a register to store this value in instead of shared memory. We still need to store the reduced value in shared memory though, for when another thread needs the value in its next iteration. The resulting changes can be seen in algorithm 4.

2.3.5 Loop unrolling

The final optimization that will be applied to the reduction algorithm is *loop unrolling*. As stated above one of the assumptions made is that the kernel knows beforehand how many elements a block will need to reduce. We now extend this assumption with the condition that all blocks reduce the same number of elements. Again this can be accomplished quite easily by padding the input values when copying them to shared memory.

There are several reason as to why loop unrolling can increase performance. Firstly looping requires indirection and even though most of the threads in our warps will loop the same amount of times, the warps still incur an overhead by having to perform the indirection. Secondly unrolling removes the overhead of updating the $offset$ variable, which is a quite significant portion of the work performed by the kernel when the body inside the loop is this small. Thirdly we can take advantage of the synchronized execution of threads in the same warp and remove explicit synchronization invocations when offset becomes less than the warp size.

After having unrolled the loop, it is now also possible to move thread 0's last reduction, **min**($rValue$, $sValues[1]$), down to where the result is output. We are also able to inline he first reduction, where the data is copied from global memory to shared memory, reducing the shared memory requirements by half per block. Algorithm 5 shows the inlining of the first and final reduction. Unrolling the loop is quite straightforward, but takes a lot of space and has therefore been omitted here.

Algorithm 4 Register reduction

```
procedure Reduce3
    in values : Number List; id : Integer; elements : Integer
    out result : Number
begin
  sValues : shared Number List
  rValue ← sValues[id] ← values[id]
  synchronize
  offset ← elements/2
  while offset > 0 do
    if id < offset then
      rValue ← sValues[id] ← min( rValue , sValues[id + offset] )
    end if
    offset ← offset/2
    synchronize
  end while
  if id = 0 then
    result ← rValue
  end if
end
```

Algorithm 5 Unrolling reduction loops

```
procedure Reduce4
    in values : Number List; id : Integer; elements : Integer
    out result : Number
begin
  offset ← elements/2
  sValues : shared Number List
  rValue ← sValues[id] ← min( values[id] , values[id + offset] )
  synchronize
  offset ← offset/2
  while offset > 1 do
    if id < offset then
      rValue ← sValues[id] ← min( rValue , sValues[id + offset] )
    end if
    offset ← offset/2
    synchronize
  end while
  if id = 0 then
    result ← min( rValue , sValues[1] )
  end if
end
```

Algorithm:	*Total execution time:*	*Decrease in execution time:*
Reduce0:	59822.1μs	0.0%
Reduce1:	26515.1μs	55.7%
Reduce2:	13410.3μs	77.6%
Reduce3:	12474.4μs	79.1%
Reduce4:	8839.62μs	85.2%

Figure 2.4: This table shows the total execution time of the different reduce kernels presented in section 2.3. The final column shows the reduction in execution time compared with Reduce0.

2.3.6 Optimization results

The reduction algorithms above have been tested as part of a kd-tree construction on a scene consisting of 70k triangles. During kd-tree construction the reduce kernels were executed 20 times and their total execution time can be seen in figure 2.4 aswell as the decrease in execution time for each optimized kernel relative to the performance of Reduce0. The final Reduce4 algorithm decreased execution time by 85.2% compared to the naïve implementation. This shows us that knowledge of how the GPU works is essential to creating efficient GPU implementations. In the rest of the thesis it can be assumed that the above optimizations have been applied to kernels that could benefit from them. In some cases the very nature of an algorithm makes it impossible to apply certain optimizations, such as coalescing data fetches for several threads traversing different parts of the kd-tree. In these cases I will note the lack of optimization and may present other solutions that can be applied instead.

Chapter 3

KD-Trees

With todays's fast ray tracers, the difference between a "good" and a naïvely built kd-tree is often a factor of 2 or more.
Ingo Wald and Vlastimil Havran

Since Arthur Appel described ray tracing four decades ago, several spatial acceleration structures have been developed to increase the speed of ray tracers. This chapter will focus on one of the the most popular acceleration structures, namely the kd-tree.

In his ph.d. dissertation [12], Vlastimil Havran did an extensive study of spatial acceleration structures, including grids, octress and kd-trees. In chapter 3 of the dissertation he concludes that in most cases the kd-tree will outperform other well-known acceleration structures.

Since Zhou et al.[25], it has been possible to efficiently construct kd-trees entirely on the GPU. In the paper they show that their approach even allows ray tracing dynamic scenes on the GPU, where the kd-tree is reconstructed for each new image rendered. Building on their approach, I will in section 3.2.2 present alternatives to their method for choosing splitting planes. I will also investigate two different methods for determining when a triangle and a node overlaps, this is done in section 3.1.3.

Though the different methods and their implementation is presented in this chapter, the methods will not be evaluated until chapter 5. This is postponed to allow the introduction of the ray tracers used when evaluating the quality of the kd-trees in chapter 4.

Before diving into specifics about implementing kd-trees on the GPU, I will first use the next section to illustrate the recursive kd-tree construction algorithm in general terms and describe the representation of tree nodes in memory. The following subsections will then describe different established algorithms for choosing a splitting plane and how to associate geometry with child nodes after a split has occured. The latter part of this chapter deals with converting a single-threaded recursive kd-tree construction algorithm into a dataparallel algorithm.

3.1 Building KD-trees

A kd-tree is a binary tree that recursively subdivides k-dimensional geometry into smaller tree nodes. Algorithm 6 describes a general recursive kd-tree construction scheme.

CreateNode takes as arguments a list of triangles and a bounding volume describing the volume of the node. Usually this bounding volume is an *axis aligned bounding box*, *AABB*, which is a box whose planes are aligned with the axes of the coordinate system. An axis aligned bounding box, b, can thus be described by its maximum and minimum values along each axis and I therefore define it as, $b = \{b_{min}, b_{max}\} \in \{R^3, R^3\}$. All bounding volumes used in this thesis are axis aligned bounding boxes and I will therefore use the descriptions bounding volume, bounding box and axis aligned bounding box interchangeably. CreateNode first checks if the triangles and the bounding box satisfy the requirements for becoming a leaf node. If so, then a leaf is produced and the recursion terminates. If not, then the node's bounding box is divided by a chosen splitting plane, *plane*. How to determine when to end recursion and choose a splitting plane is the topic of section 3.1.2. Dividing the bounding box produces two new axis aligned bounding boxes, $voxel_L$ and $voxel_R$. Triangles are then associated with these new

Algorithm 6 Recursive kd-tree constructor

```
procedure CreateNode
    in T : Triangle List; voxel : AABB
    out node : Node
begin
  if IsLeaf(T, voxel) then
    node ← Leaf(T)
  else
    // Determining the splitting plane will be discussed in section 3.1.2
    plane ← DeterminePlane(T, voxel)
    (voxel_L, voxel_R) ← Split(voxel, plane)
    // How to associate geometry with a voxel will be the topic of section 3.1.3
    T_L ← AssociateGeometry(T, voxel_L)
    T_R ← AssociateGeometry(T, voxel_R)
    node ← Node(plane, CreateNode(T_L, voxel_L), CreateNode(T_R, voxel_R))
  end if
end
```

bounding volumes by an association algorithm, which will be discussed in section 3.1.3. Finally a new node is created that will contain references to its two children.

An example of an iteration of CreateNode can be seen in figure 3.1. The box surrounding the scene is the bounding volume of the root node. Before the scene has been subdivided by splitting planes, the root node itself is a leaf node and therefore contains a list of all triangles associated with it. In figure 3.1b a line has been drawn down the middle of the scene. This line represents a chosen splitting plane and in the tree's root node it can be seen that the plane splits along the x-axis at $x = 4$, which corrosponds to what is seen in the scene.

3.1.1 Tree Representation

Each interior node in a kd-tree must contain three pieces of information.

- *Split axis* - The axis that an interior node is split along.
- *Split position* - The position of the splitting plane along the split axis.
- *Child references* - Information about how to find the node's children.

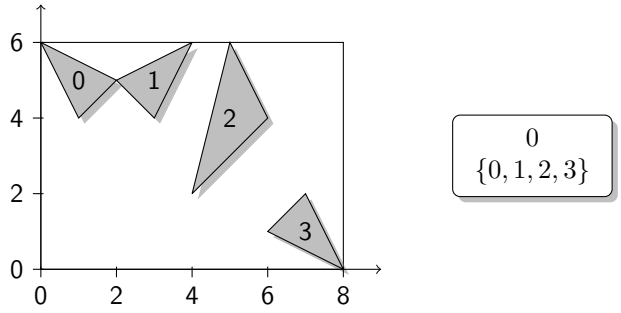
The first two items are pretty straightforward. The choice of axis, x, y or z, can be stored inside two bits and the position of the splitting plane should be stored as a floating point number. How an interior nodes should reference its child nodes, however, is not as straightforward.

In general there are two ways a node can reference its child nodes. The first is through the use of *balanced trees*, where children of a node can be addressed implicitly without the use of pointers. The reason for this is that nodes at the same tree level are placed sequentially in memory and the start address of some tree level, l is given by $2^l - 1$. The left and right child of node n can then be indexed using $2n + 1$ and $2n + 2$. The parent of a node is indexed with $\lceil n/2 \rceil - 1$.

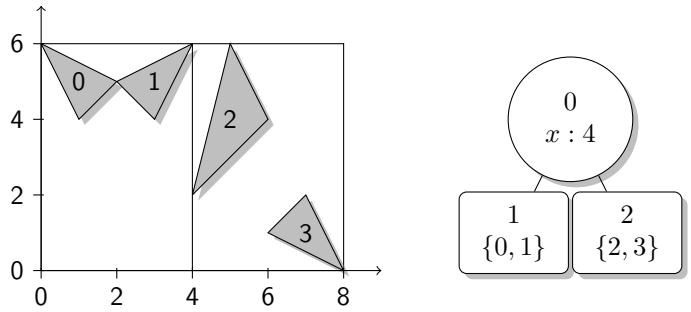
Wald et al.[21] has the following to say about balanced trees.

> *Balancing is optimal only for binary searching, and if all nodes have equal access probabilities. Neither of these two prerequisites are fulfilled for range queries (such as ray traversal and kNN queries), nor for unevenly distributed primitives such as photons or triangles.*
>
> [21]

To avoid balanced trees we can instead use pointers to reference the children. This allows for more flexibility when creating the tree. Unfortunately it also means storing more data per node and in the case of slow memory access it can cause a memory latency bottleneck. However, when experimenting



(a) A simple scene and its corrosponding kd-tree. The entire scene is contained in the root node.



(b) The same scene as in figure 3.1a. The kd-tree has split the geometry down the middle and created two new leaf nodes, with which the geometry has been associated.

Figure 3.1: One iteration of the kd-tree construction algorithm.

with different techniques for creating kd-trees, such as is done in this thesis, the added flexibility can be a benefit. In section 3.2.1 we shall see how this flexibility can be used to add the *Empty Space Maximization* optimization, without modifications to the existing kd-tree construction implementation. This would not have been possible had the tree been balanced.

The only information that must be stored in a kd-tree's leaf node is how to reference the triangles associated with it. In this thesis I have choosen to store a reference to the triangles associated with a leaf as one large list of indices. I have adopted the convention that all triangle indices associated with a given leaf must be stored sequentially in memory and a leaf therefore only needs to contain a reference to its first triangle index and the amount of triangles it is associated with. Using indices instead of actual triangles makes it cheaper to perform sorting after a node has been split and its triangles have been associated with its children. However to simplify the notation and keep the algorithms simple, in the rest of the thesis I will assume that a leaf has direct access to the triangles associated with it. When the number of triangles associated with a node goes below a certain threshold I will switch the representation to an index and a bit mask. How this will work and the advantages of the bit mask representation will be explained in section 3.2.2.

3.1.2 Choosing the Splitting Plane

Looking again at algorithm 6, we can see that the brilliance associated with constructing a high quality kd-tree is knowing where to place the splitting plane and when to end the recursion and create a leaf.

In the following section I present two algorithms that solves this problem and then I extend them with Empty Space Maximization, which will improve the quality of the tree by allowing rays to quickly skip large subtrees and thus find the geometry they intersect faster.

Spatial Median

A quite simple method for choosing the splitting plane is to place it at the spatial median of a node's bounding box. There are two ways to choose which dimensions spatial median to use. A *round robin*

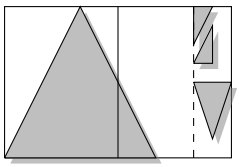


Figure 3.2: A poor split produced by median splitting. The solid line is a median split. The dashed one is a more optimal splitting plane, since it divides the large triangle from the small.

approach can be used, where the dimensions are cycled every iteration; e.g. when creating the first node the plane will lie perpendicular to the x-axis, that node's children will then be split along the y-axis, in the next iteration the nodes will be split along the z-axis and then the cycle restarts again at the x-axis. Another way of choosing the dimension is to split along the largest axis of the node's bounding box. This will be more costly than the round robin approach since more axes must be analysed, but there is also a higher probability that it will produce trees of higher quality.

The termination criteria for spatial median splitting is equally as simple as choosing the splitting plane. If the number of triangles associated with a node falls below a certain threshold, then that node becomes a leaf.

This method is refered to as naïve in [22] and rightly so, since spatial median splitting does not take the distribution of geometry inside a node's bounding box into account. On figure 3.2 an example is given where a node is split in a non-intuitive way. Spatial median splitting does, however, make up for its suboptimal splitting planes by being quite fast.

Surface Area Heuristic

A far better splitting plane position can be obtained by applying the *Surface Area Heuristic*, *SAH*. Instead of only considering the bounding volume surrounding the geometry, SAH considers the entire geometry associated with a node. In essence SAH computes the *expected cost*, C_{SAH}, of traversing a node, n, that has been split into the child nodes l and r.

$$C_{SAH}(n \to \{l, r\}) = C_{trav} + \frac{C_l A_l}{A_n} + \frac{C_r A_r}{A_n}$$

where C_{trav} is the cost of traversing an interior node and is independent of the splitting plane, C_l is the cost of traversing the left child node and C_r is the cost of traversing the right child node. A_k is the summed surface area of the geometry associated with node k. Choosing the most optimal splitting plane amounts to applying SAH to all possible splitting planes and then choosing the one with lowest cost. Since the above cost evaluation does not take ray directions into account, SAH assumes that rays are uniformly distributed, infinite lines.

SAH can also automatically determine when to stop splitting. The cost of traversing a leaf node is given as

$$C_{leaf}(n) = \|T_n\| C_i$$

where $\|T_n\|$ is the number of triangles associated with n and C_i is the cost of testing intersection between a triangle and a ray. SAH's termination criteria is then $C_{leaf}(n) <= C_{SAH}(n)$, i.e. SAH terminates when the expected cost of splitting n becomes higher than the cost of keeping n as a leaf node.

Calculating a globally optimal solution using SAH is infeasable for complex scenes, as that would require evaluating the cost of all possible subtrees. A *local greedy approximation* is used instead, where it is assumed that the created child nodes are leafs. This assumption simplifies the SAH calculation to

$$C_{SAH}(n \to \{l, r\}) = C_{trav} + \frac{C_{leaf}(l) A_l}{A_n} + \frac{C_{leaf}(r) A_r}{A_n}$$

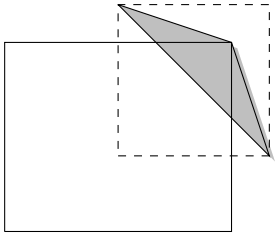


Figure 3.3: An example of how the sides of a triangle's bounding box, the dashed box, does not represent the most optimal splitting planes.

The termination criteria can now be simplified into

$$
\begin{aligned}
& C_{leaf}(n) <= C_{SAH}(n) \\
\Updownarrow & \\
& \|T_n\|C_i <= C_{trav} + \tfrac{C_{leaf}(l)A_l}{A_n} + \tfrac{C_{leaf}(r)A_r}{A_n} \\
\Updownarrow & \\
& \|T_n\|C_i <= C_{trav} + \tfrac{\|T_l\|C_iA_l}{A_n} + \tfrac{\|T_r\|C_iA_r}{A_n} \\
\Updownarrow & \\
& A_n(\|T_n\| - \tfrac{C_{trav}}{C_i}) <= \|T_l\|A_l + \|T_r\|A_r
\end{aligned}
$$

Deciding on the optimal splitting plane is therefore reduced to finding the plane with the least weighted area $\|T_l\|A_l + \|T_r\|A_r$ and choosing appropriate values for the constants C_{trav} and C_i is reduced to determining C_{trav}/C_i.

Inspite of the assumptions made by SAH about ray distribution and that in practice a local greedy approximation is used, it is still considered one of the best heuristics and can generally produces trees of the highest quality.

Split Candidates As mentioned above, the SAH cost needs to be evaluated for *all* available splitting planes. Since there are infinitely many potential planes along either axis, some method is needed to distinguish the useful splitting planes from unimportant ones. The interesting splitting planes are those finitely many planes, where the geometry association in the resulting left and right child nodes change. These splitting planes are called *split candidates*.

The obvious choice of axis aligned split candidates are the 6 planes defined by a triangle's axis aligned bounding box. These fulfil the requirement for becoming split candidates, as they represent the exact location where the geometry association for the left and right side changes. While using the bounding box' sides is a simple and fast solution, it is not always the most optimal. The reason for this is that a triangle may not be located entirely inside a node and therefore its bounding box does not represent the most optimal split. An example of this can be seen in figure 3.3, where slightly moving the split candidates given by the sides of the triangle's bounding box would not change the triangle association in the resulting two leaf nodes.

Another problem with choosing the sides of a triangle's bounding box as split candidates is that bounding boxes of small nodes may be completely contained inside the geometry's bounding box. Thus none of the planes are actually useful split candidates as they can not split the node, which is demonstrated on figure 3.4.

A solution to this is to continuously *clip* the bounding box of the split geometry to fit the part of the geometry inside the tree nodes bounding box, which creates optimal split candidates or *perfect split condidates*, as demonstrated on figure 3.5.

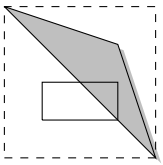


Figure 3.4: The kd-tree node's bounding box is completely contained inside the triangles bounding box.

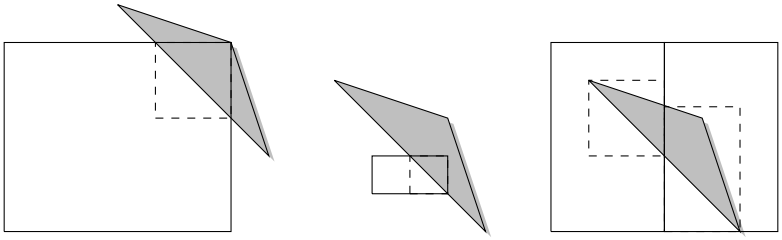


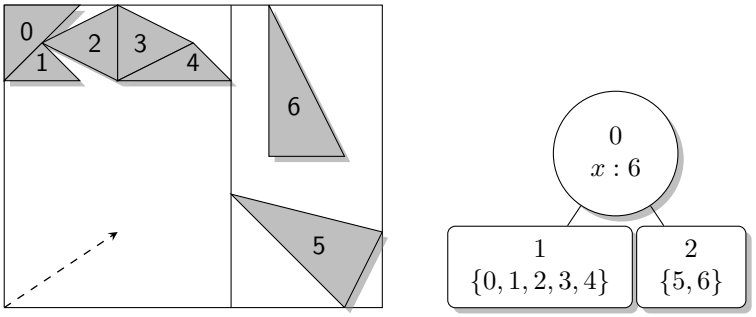
Figure 3.5: The triangles' bounding boxes have all been clipped to fit the part of the triangle contained in the nodes' bounding boxes.

Empty Space Maximization

An effective optimization to the quality of kd-trees is *Empty Space Maximization* and it can be applied to both trees created with Spatial Median Splitting or SAH. The idea behind the optimization is to cut away large empty parts of the scene near the top of the tree. This is done by injecting empty leaf nodes into the tree at interior nodes where the distance from their parent's bounding box to the associated geometry is above a certain threshold. The empty nodes will then provide rays traversing the tree with an early out option, allowing them to skip a large portion of the geometry.

Figure 3.6 shows a simple scene without empty space cut away and its corrosponding kd-tree. Without Empty Space Maximization the ray entering the scene from the lower left cornor is forced to perform intersection tests with every triangle in leaf 1. In contrast the kd-tree in figure 3.7 has been created with Empty Space Maximization enabled and the ray entering the scene now ends up in leaf 4. Leaf 4 is empty and allows the ray to advance into the scene without testing for intersection with any of the triangles in leaf 1.

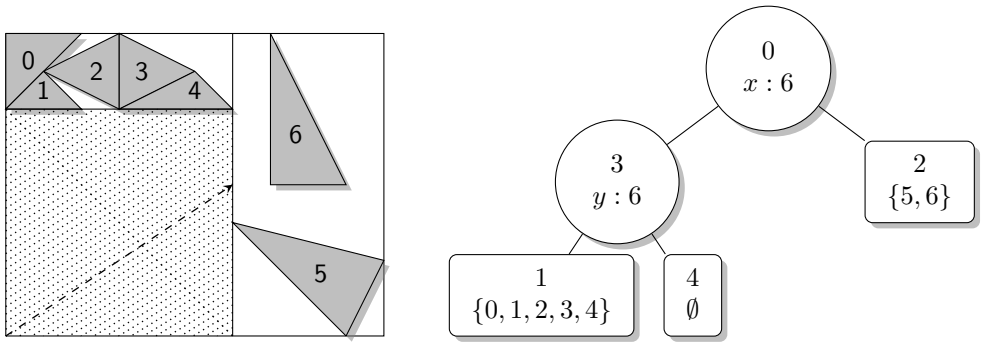
Unfortunately, the percentage of a node that should be empty space before it is cut away is highly scene specific, making this optimization less usefull in the general case. Zhou et al.[25] found that cutting



(a) A simple scene without empty space cut away.

(b) The tree corrosponding to the scene in figure 3.6a.

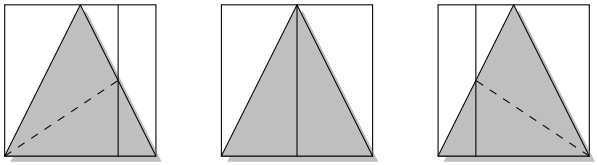
Figure 3.6: A kd-tree constructed around a simple scene. The kd-tree does not use the Empty Space Maximization optimization.



(a) The scene from figure 3.6a with empty space cut away.

(b) The tree from figure 3.6b with the empty space nodes 3 and 4 injected.

Figure 3.7: An example of Empty Space Maximization. The dotted region represents an empty node and allows the ray to leap across it, thus skipping intersection tests with the five triangles above.



(a) Two vertices to the left of the splitting plane.

(b) One vertex intersected by the splitting plane.

(c) Two vertices to the right of the splitting plane.

Figure 3.8: The three different triangle splitting cases. A dashed line represents an additional split needed to keep representing geometry as triangles.

away 25% or more empty space produced optimal trees, but instead of simply using their threshold I will test it with threshold of 15%, 25% and 35% in chapter 5.

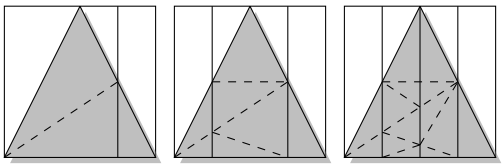
3.1.3 Triangle/Node Association Schemes

When a splitting plane has been chosen, the interior node's associated triangles needs to be associated with the child nodes whose bounding box they overlap. Triangles located entirely in front of the splitting plane are associated with the left child node and triangles entirely behind are associated with the right child. The association schemes discussed in this section present different methods for handling triangles intersected by the splitting plane. The scheme employed is important in dynamic schemes. On one hand a fast approximating association scheme may assign triangles to nodes that they do not necessarily overlap, resulting in larger trees and thus slower ray tracing times. On the other hand a scheme that rigorously checks every triangle and only assigns triangles to nodes that they definitely overlap will result in a smaller tree and faster ray tracing time, but will also incur a performance penalty when constructing the acceleration structure.

Triangle Splitting

The most common approach when a triangle is intersected by a splitting plane, is to split the triangle up into new triangles. New triangles produced by triangle splitting will always be located entirely within the bounding box of the kd-node they are associated with. When performing triangle splitting there are three cases that must be handled, as can be seen in figure 3.8.

The first case in figure 3.8a illustrates the split when two vertices are located on the left of the splitting plane and one on the right. This case is mirrored by figure 3.8c where two of the vertices are located to



(a) One split. **(b)** Two splits. **(c)** Three splits.

Figure 3.9: Triangle splitting performing excessive splits on a triangle.

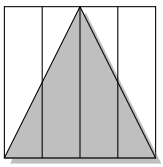


Figure 3.10: Dividing a triangle among the leafs associated with it. Notice that each leaf only contains one reference to the original triangle.

the right of the splitting plane. In both of these cases a triangle split will produce three new triangles. The last case in figure 3.8b shows one of the vertices located inside the splitting plane and splits the triangle perfectly into two new triangles. This last case however is highly unlikely and in general it can be assumed that a split triangle always produces three new triangles.

Triangle/Node Overlap

Since splitting a triangle almost always leads to three new triangles being created, this can result in excessively many new triangles, as evidenced by figure 3.9 where the same triangle is split three times. The problem with excessive splitting is that each new triangle actually represents the original triangle and on figure 3.9 nodes can be seen containing up to 6 triangles, which all represents the exact same original and are cluttering up the tree with redundant geometry. A more preferable situation is seen in figure 3.10, where each leaf node only contains one reference to the original triangle.

This is the central idea behind performing a *Triangle/Node Overlap* test. Instead of splitting a triangle by the splitting plane and creating new triangles, the original triangle is associated with a leaf node if the triangle and the leaf node's bounding box overlap. How to test this is described in Möller[2].

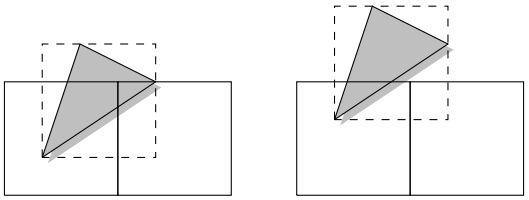
Box Inclusion

Performing a triangle/bounding box overlap test required for Triangle/Node Overlap can be a computationally heavy task. A cheaper splitting scheme is to test if the triangle's axis aligned bounding box, t, overlaps with the axis aligned bounding box of the node, n. Performing the overlap test between these axis aligned bounding boxes is simply

$$\text{overlap}(n, t) = n_{min} < t_{max} \wedge t_{min} < n_{max}$$

where the infix operator $<: \{R^3, R^3\} \rightarrow bool$ is defined as $n < t = n.x < t.x \wedge n.y < t.y \wedge n.z < t.z$.

Figure 3.11 shows two examples of a node being split down the middle. In both examples the triangle will be associated with both new nodes, as the triangle's bounding box overlaps both nodes. But in figure 3.11b we clearly see that the triangle itself does not overlap the right node's bounding box. This illustrates the drawback of only testing a triangle's bounding box and how Box Inclusion can produce *false positives*, where geometric primitives are associated with nodes they do not overlap. Such false postives will increase the size of the tree and thus slow down ray tracers. However, Box Inclusion makes



(a) Box Inclusion correctly associates the triangle to both nodes.

(b) Box Inclusion will incorrectly associate the the triangle with the left node.

Figure 3.11: Box inclusion examples.

up for this by performing fast node/triangle association, cutting down on the time spent constructing the tree.

3.2 Adopting the Algorithms for CUDA

Having explored the different methods for deciding which splitting plane to use and how to associate geometry with a node in the kd-tree, it is time to look at the actual implementation of a kd-tree construction algorithm on the GPU.

The general kd-tree construction method presented in algorithm 6 recursively constructs *one* tree node at a time in a *depth-first* manor. In order to efficiently utilize the GPU, as many of its multiprocessors as possible must be fully occupied. This means that algorithm 6 must be restructured to work on multiple nodes in parallel. Fortunately this is exactly what Zhou et al.[25] did by changing the kd-tree constructor to work on a list of nodes and recursively create the tree in *breadth-first* order, as outlined in algorithm 7.

Algorithm 7 BFS recursive kd-tree constructor

```
procedure CreateNodes
    in activeNodes : Node List //list of kd-nodes not processed yet
    out nextNodes : Node List
begin
  for each node in activeNodes in parallel do
    if IsLeaf(node) then
      node ← Leaf(node)
    else
      // Determine the splitting plane.
      plane ← DeterminePlane(node)
      (node_L, node_R) ← Split(node, plane)
      // Associate the geometry with a node and add it to the next list.
      node_L.geometry ← AssociateGeometry(node.geometry, node_L)
      nextNodes.Add(node_L)
      node_R.geometry ← AssociateGeometry(node.geometry, node_R)
      nextNodes.Add(node_R)
    end if
  end for
end
```

At the bottom levels, where thousands of nodes are created in parallel, breadth-first creation allows full utilization of the GPU. But for the upper levels of the tree this approach will still not fully exploit the graphics hardware, as only a couple of nodes are active per iteration, leaving the multiprocessors underutilized.

To remedy this the tree construction will be split into two phases, an upper and a lower phase. A

node is said to belong to the upper part of the tree if the number of triangles associated with it is above a given threshold. In this thesis I will be using the thresholds 32 and 64. During the upper tree creation phase, the choice of splitting plane is parallelized over all geometric primitives, of which there can be hundreds of thousands. When creating the lower tree, there will be thousands of nodes and computations can effectively be structured as in algorithm 7.

The overall construction of the tree is presented in algorithm 8. First the axis aligned bounding box for each triangle is precomputed. These are used to effectively compute the bounding boxes of nodes and for Box Inclusion, if that association scheme is used. Then the upper part of the tree is constructed iteratively until there are no more new nodes to process. How this is achieved is the topic of section 3.2.1. Once the upper part of the tree is constructed, the leaf nodes are processed to prepare for the lower tree construction phase. The lower part of the tree is then iteratively constructed and section 3.2.2 details three different algorithms that produces lower trees with varying construction speed and quality.

Algorithm 8 Construct kd-tree

```
procedure ConstructKDTree
    in Triangles : Triangle List
    out root : Node
begin
  for each t in Triangles in parallel do
    Compute axis aligned bounding box for t.
  end for

  activeNodes, leafs, nextNodes : Node List
  // Upper node construction phase.
  activeNodes.Add(rootNode)
  while activeNodes.NotEmpty do
    nextNodes.Clear
    (newLeafs, nextNodes) ← CreateUpperNodes(activeNodes)
    leafs.Append(newLeafs)
    swap(activeNodes, nextNodes)
  end while

  // Lower node construction phase.
  PreprocessLowerNodes(leafs)
  CreateLowerNodes(leafs, activeNodes)
  while activeNodes.NotEmpty do
    nextNodes.Clear
    nextNodes ← CreateLowerNodes(activeNodes)
    swap(activeNodes, nextNodes)
  end while
end
```

3.2.1 Upper Tree Creation

At the uppermost levels of the kd-tree there will not be many nodes over which computations can be parallelized. But each node can be associated with thousands of geometric primitives and parallelizing the choice of splitting plane across the geometry will utilize the GPU effectively.

The issue is then which algorithm to use when choosing the splitting plane. The Surface Area Heuristic provides an algorithm that can easily be parallelized over the geometry. Each triangle would then be responsible for computing the expected cost of splitting a node with its 6 bounding box planes and afterwards the best possible plane could be reduced with the method described in section 2.3. Unfortunately there can be hundreds of thousand of triangles associated with the tree's upper nodes, so comparing a split candidate with all of them becomes computationally heavy. SAH also assumes that the created children will be leaf nodes in the final tree and therefore that their cost can be computed using C_{leaf}, which is almost never true at the top of the tree. This makes SAH an undesirable algorithm

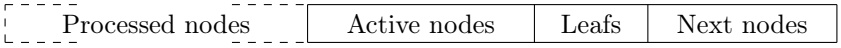


Figure 3.12: The structure of the kd-tree's nodes in memory.

for choosing splitting planes for the upper tree, since we need to be able to decide on which splitting plane to use fast to accomodate dynamic scenes. Instead Zhou et al.[25] proposes to use spatial median splitting with Empty Space Maximization. Parallizing the splitting plane decision over the geometry then becomes reducing an axis aligned bounding box from a node's associated triangles, again as described in section 2.3, and then each node can use that bounding box to decide where to place its splitting plane.

To associate triangles and nodes in the upper parts of the tree, both schemes described in section 3.1.3 can be employed. As with the splitting plane decision, the triangle/node association scheme will be parallelized over all triangles. Since I have made the assumption that all triangles associated with a node must be placed sequentially in memory, the triangles need to be sorted after the triangle/node association has been determined. To parallelize this sorting over all triangels, I again make use of the scan primitive described in section 2.2. Since a triangle can be associated with both child nodes, the left-right sorting in section 2.2 is not directly applicable. Instead I will be using a variant where the prefix-sum calculation is applied to a list, $associateSide$, of size $2 \cdot \|triangles\|$. In the first half of $associateSide$ I will then use 0's and 1's to represent whether the t'th triangle overlapped its current node's left child. In the last half of the list the same is done but for the right child instead. Computing the prefix-sum of this list then yields the addresses, $associateAddr$, where the triangles should be copied to. Below is an example of this

$$
\begin{array}{rl}
triangles: & [t_0\ \ t_1\ \ t_2\ \ t_3\ \ t_4\ \ t_5]\ \ [t_0\ \ t_1\ \ t_2\ \ t_3\ \ t_4\ \ t_5] \\
associateSide: & [1\ \ 1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 1\ \ 0\ \ 1\ \ 0\ \ 0\ \ 1] \\
associateAddr: & [0\ \ 1\ \ 2\ \ 2\ \ 3\ \ 4\ \ 4\ \ 5\ \ 5\ \ 6\ \ 6\ \ 6\ \ 7] \\
sorted: & [t_0\ \ t_1\ \ t_3\ \ t_4\ \ t_0\ \ t_2\ \ t_5]
\end{array}
$$

where the $triangles$ list is duplicated to easily show the association between the entries in $associateSide$ and the triangles. With the $associateSide$ and $associateAddr$ lists it is possible to parallelize the triangle sorting effectively over all triangles using the algorithm in algorithm 9.

Algorithm 9 Parallel triangle sorting.

```
procedure TriangleSort
    in triangles : Triangle List, associateSide : Bit List, associateAddr : Integer List
    out sorted : Triangle List
begin
  sorted : Triangle List
  for each t in triangles in parallel do
    if associateSide[t] then
      addr ← associateAddr[t]
      sorted[addr] ← t
    end if
    if associateSide[t + ||triangles||] then
      addr ← associateAddr[t + ||triangles||]
      sorted[addr] ← t
    end if
  end for
end
```

Like the triangles, the nodes themselves are placed in one large array with the structure shown in figure 3.12. The already processed nodes are placed in one sequential chunk from index 0. The currently active nodes are located right after them. The child nodes created by algorithm 10 are then placed after the active nodes, with the leaf nodes to the left and next set of active nodes to the right. This preserves the invariant that the n currently active nodes, if any, are always the n last nodes in the array after an iteration of the construction algorithm. This is a desirable property as it means I do not need to maintain a list of indices to active nodes, but can instead describe the list $activeNodes$ by a range and

an index into the node list. This also improves coalescence when subsequent threads can access nodes sequentially.

The algorithm for constructing the upper parts of the kd-tree can be seen in algorithm 10. It takes a list of non processed nodes, $activeNodes$ as input and returns a list of leaf nodes, $leafs$, and a list of new nodes, $nextNodes$, that needs to be processed in the next iteration.

The first thing that algorithm 10 does is split the triangles associated with each active node into fixed-size segments. This may seem odd at first, but recall that the upper nodes are split along the spatial median, that finding the spatial median means computing a tight bounding box around the geometry associated with a node, and that to do this we need to apply the reduction described in section 2.3. Generally having fixed-size segments will also make it easier to choose a kernels block size, so that each segment is processed by one block. A segment contains information about which triangles it contains, which node its triangles are associated with and, incase there are not enough triangles to fill it, it also stores how many triangles it actually contains. The reduction kernel can use this information to pad the data loaded into shared memory with identity elements.

After having segmented the triangles, the bounding boxes of the individual segments can be computed as shown in section 2.3 by using the operators **min** and **max** with their identity elements. Performing a segmented reduction on the result, as described in Algorithm 3 in Zhou et al.[25], will then compute a tight bounding box for each node in activeList. A kernel working on all active nodes will then use this bounding box to place the nodes splitting planes along their spatial median.

With the splitting plane determined for each node, the triangles will be able to determine their triangle/node association with the nodes children and sort themselves into a new list as described above.

Lastly the nodes in $activeList$ are split into their child nodes in parallel. For node n, the values in $associateAddr$ can be used to calculate the triangle index and range of the child nodes, $left$ and $right$, as seen in algorithm 11. The idea behind algorithm 11 is that using the triangle index of the parent nodes we are able to extract the triangle index of its left and right child nodes. Then by adding the number of triangles in the parent node to its index, we can lookup the starting index of the next child node's triangles. Subtracting these two values yields the range of triangles spanned by a child node.

Adding Empty Space Maximization

To experiment with the Empty Space Maximization optimization it needs to be implemented as a solution that can be turned on and off at will. This has been achieved with the design found in algorithm 12, which injects new nodes created by Empty Space Maximization into the tree. In order for a node to perform Empty Space Maximization it needs a *loose bounding box*. The loose bounding box is computed by splitting a parents bounding box with its splitting plane. Since the geometry inside a child might not touch all sides of a parents bounding box, the loose bounding box provides a loose upper bound on the geometry associated with a child.

The algorithm for performing Empty Space Maximization is inserted into algorithm 10 right after computing the nodes' tight bounding boxes and before the new child nodes are created. Whether or not Empty Space Maximization should be performed on any of the nodes in $activeNodes$ is then checked in parallel for all nodes. The sides of a node's bounding box is checked sequentially to see if the empty space between the loose and the tight bounding box is above a certain threshold, which means that the empty space should be cut away. These sides are added to the list $emptySides$, which will be used later when the nodes are split. Each such split results in two new nodes, one node that refers to the empty space and one node injected between the parent node and its child node in active list. This injection is depicted in figure 3.6 on page 23 and figure 3.7 on page 24. The number of new nodes created per node in $activeList$ by performing Empty Space maximization is stored in $emptySplitNodes$.

Once the sides that should have empty space cut away has been determined, where to place the nodes created by these splits are computed by calculating the prefix-sum of $emptySplitNodes$. Empty Space Maximization nodes are injected into to the list of nodes shown in figure 3.12 as described on figure 3.13, which again preserves the invariant that the next batch of active nodes are located at the end of the list. The arrows in figure 3.13 symbolize the parent→child relationship and very accuratly show how the new nodes are injected into the tree.

With the addresses of the nodes produced by Empty Space Maximization computed, a kernel can be launched that creates the actual empty space nodes and rewires the parent node to point to the empty space nodes instead of its current child node.

Algorithm 10 KD-Tree upper node creator

procedure CreateUpperNodes
  in $activeNodes$: Node List
  out $leafs, nextNodes$: Node List
begin
  // First split all triangles into segments.
  $segmentList$: List
  for each *node* **in** $activeNodes$ **in parallel do**
    Split all triangles contained in *node* into fixed sized segments and store those in $segmentList$.
  end for

  // Then compute each triangles bounding box using reduction as described in section 2.3.
  for each *segment* **in** $segmentList$ **in parallel do**
    Compute the bounding box of the triangles in each segment.
  end for
  Use segmented reduction as described in Zhou et al.[25] to compute each nodes bounding box.
  for each *node* **in** $activeNodes$ **in parallel do**
    Split *node* along its spatial median.
  end for

  // Perform Empty Space Maximization.
  // See algorithm 12.
  ...

  // Compute the addresses to sort the triangles to by comparing them to the splitting plane.
  $associateSide, associateAddr$: List
  for each *segment* **in** $segmentList$ **in parallel do**
    for each *triangle* **in** *segment.triangles* **in parallel do**
      $associateSide[triangle.id] \leftarrow$ AssociateLeft(*segment.node*, *triangle*)
      $associateSide[triangle.id + \|triangles\|] \leftarrow$ AssociateRight(*segment.node*, *triangle*)
    end for
  end for
  $associateAddr \leftarrow$ **Prefix-Sum**($associateSide$)
  // Sort the triangles.
  $triangles \leftarrow$ TriangleSort($triangles, associateSide, associateAddr$)

  // Split nodes.
  for each *node* **in** $activeList$ **in parallel do**
    Split the nodes into child nodes. $associateSide$ and $associateAddr$ is used to directly calculate the child nodes triangle index and range.
    // Sort child nodes into the leaf and nextNodes list. This is achieved just like the example from section 2.2.
    if *node.child*.size $<$ *threshold* **then**
      $leaf$.Add(*node.child*)
    else
      $nextNodes$.Add(*node.child*)
    end if
  end for
end

Algorithm 11 Compute child node triangle index and range

procedure ChildTriangleInformation
  in $parent, left, right$: Node, $associateAddr$: List
  out $left$: Node, $right$: Node
begin
  $left.triangleIndex \leftarrow associateAddr[parent.triangleIndex]$
  $leftEndAddr \leftarrow associateAddr[parent.triangleIndex + parent.triangleRange]$
  $left.triangleIndex \leftarrow leftEndAddr - left.triangleIndex$
  $right.triangleIndex \leftarrow associateAddr[parent.triangleIndex + \|triangles\|]$
  $rightEndAddr \leftarrow associateAddr[parent.triangleIndex + parent.triangleRange]$
  $right.triangleIndex \leftarrow rightEndAddr - right.triangleIndex$
end

Algorithm 12 Calculate Empty Space Maximization

procedure CreateUpperNodes
  in $activeNodes$: Node List
  out $leafs, nextNodes$: Node List
begin
  // First split all triangles into segments.
  ...
  // Then compute each triangles bounding box using reduction as described in section 2.3.
  ...

  // Perform Empty Space Maximization.
  if $performEmptySpaceMaximization$ **then**
    $emptySplitNodes, emptyAddr$: List
    $emptySides$: Plane List
    for each *node* **in** *activeList* **in parallel do**
      $emptySplits[i] \leftarrow 0$
      for each *side* **in** *node.boundingBox* **do**
        if *node* has more than C_e empty space on *side* **then**
          $emptySplitNodes[i] \leftarrow emptySplitNodes[i] + 2$
          $emptySides[node]$.Add($side$)
        end if
      end for
    end for
    $emptyAddr \leftarrow$ **Prefix-Sum**($emptySplitNodes$)
    for each *node* **in** *activeList* **in parallel do**
      Cut of empty space of sides in $emptySides$ and place new nodes sequentially at addresses specified in $emptyAddr$. Then rewire parent nodes to point to new nodes.
    end for
  end if

  // Compute the addresses to move the triangles to by comparing them to the splitting plane.
  ...
  // Move the triangles.
  ...
  // Split nodes.
  ...
end

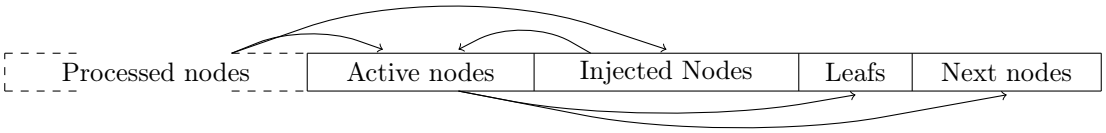


Figure 3.13: The structure of the kd-trees node's in memory with Empty Space Maximization. The arrows symbolize the parent→child relationship.

This construction is very flexible and all computations needed by Empty Space Maximization can be turned on and off at will, which allows me to easily compare the quality and construction speed of trees with and without this optimization.

3.2.2 Lower Tree Creation

Compared with the upper tree creation phase, the lower tree creation is quite simple and only requires a few kernels. Since there can be thousands of active nodes at the lower node phase, parallization can effectively be done over nodes instead of triangles.

Zhou et al.[25] proposes that the surface area heuristic is used when deciding which splitting candidate to use and that the set of splitting candidates is precomputed from the geometry's bounding volumes. Precomputing the splitting candidates instead of adjusting them after each split, can cause SAH to not choose the most desireable splitting plane. This has already been discussed in section 3.1.2 in the Split Candidates paragraph and figure 3.3 on page 22 presents an example. Zhou et al.[25]'s reasoning for doing it anyway is

> *While clipping is effective for large nodes by preventing false postives from accumulating over future splits, our experiments indicate that clipping rarely improves ray tracing performance.*
> [25]

Lower Tree Split Candidates

Before splitting any nodes however, I first introduce the split candidate structure used in the lower node phase. Recall from section 3.1.1 that a node stores the reference to its associated triangles as an index, i, and a range, r, into the list of all triangles. To efficiently compute the number of triangle associated with a node and performing triangle/node association in one instruction, I adopt Zhou et al.[25]'s novel approach of storing a node's triangle association using an index, i, and a bit mask, b. If the k'th bit in b is set, this means that the $i + k$'th triangle in the list of all triangles is associated with the node. The bit mask corrosponding to r would then simply have all its first r bits set. A more thorough example of how bit masks are used is given in figure 3.14.

Since the split candidates are local to the lower subtree, they too benefit from the bit mask representation. Each split candidate needs to store two triangle bit masks, the triangles in front of the splitting plane and the triangles behind. The method for computing the triangle bit masks of the precomputed split cnadidates can be seen in algorithm 13, which only computes them along the x-axis, but is trivially extended to other dimensions. The algorithm uses a triangle's axis aligned bounding box, $aabb_t$ to determine on which side of the splitting plane the triangle is located. If $aabb_t$'s minimum cornor is in front of the splitting plane, then the t'th bit in the triangle bit mask in front of the plane is set. If the maximum cornor of $aabb_t$ is behind the plane, then the triangle bit mask for the triangles behind the plane will have its t'th bit set.

Algorithm 13 loops over the list of triangles twice, once in parallel and once for each thread. This results in a lot of global memory access, which can potentially slow down the procedure. This is optimized the same way as in section 2.3.3, where the threads in a warp worked together to load data into shared memory and then accessed that instead of global memory. To better convey the intent of algorithm 13 this optimization has been left out.

With the split candidates precomputed and their left and right triangle sets stored as bit masks, it becomes trivial to associate a triangle with a child node after a split. E.g. a new left child's triangle bit mask would simply be the bitwise **AND** of its parent's bit mask and the bit mask for the triangle set

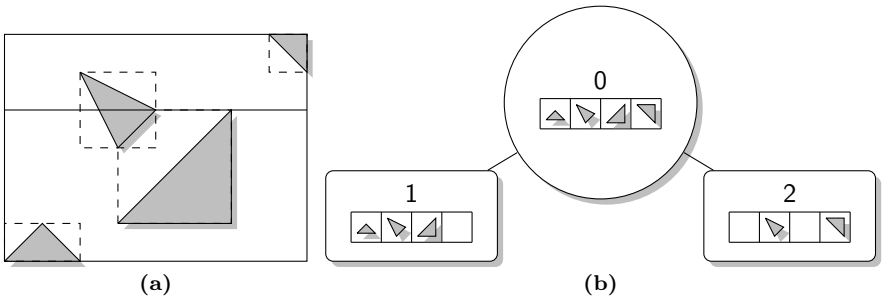


Figure 3.14: A description of storing triangles as bit masks. The bit masks are represented by a grid containing the triangles. If a grid cell contains a triangle, then that bit is set. Node 0 contains all triangles, as evidenced by its bit mask. Node 0 is then split along the precomputed split candidate drawn in the scene, which has the left triangle bit mask $[1, 1, 1, 0]$ and right mask $[0, 1, 0, 1]$. The result of splitting node 0 is leafs 1 and 2, whose triangle bit masks are the result of a bitwise **AND** between node 0's bit mask and the split candidate's left and right bit mask respectively.

in front of the splitting plane. Figure 3.14 also gives an example of this, as Node 0 is split by a splitting plane with the triangle bit mask $[1, 1, 1, 0]$ in front and $[0, 1, 0, 1]$ behind. Hereafter the triangle bit mask in front of a split candidate will be refered to as its left triangle bit mask and the bit mask behind is its right.

Algorithm 13 Preprocess Lower Nodes.

procedure PreprocessLowerNodes
  in $upperLeafs$: Node List
begin
  for each *node* **in** $upperLeafs$ **in parallel do**
    for each *triangle* **in** $node.Triangles$ **in parallel do**
      $aabb_p \leftarrow$ GetAabb($triangle$)
      // Create the splitting plane sets along the x-axis.
      x_{low}, x_{high} : Plane
      for each t **in** $node.Triangles$ **do**
        $aabb_t \leftarrow$ GetAabb(t)
        $x_{low}.left[t] \leftarrow aabb_t.min.x <= aabb_p.min.x$
        $x_{low}.right[t] \leftarrow aabb_p.min.x < aabb_t.max.x$
        $x_{high}.left[t] \leftarrow aabb_t.min.x <= aabb_p.max.x$
        $x_{high}.right[t] \leftarrow aabb_p.max.x < aabb_t.max.x$
      end for
      $node.splitCandidates$.Add(x_{low})
      $node.splitCandidates$.Add(x_{high})
      ...
      // Perform similar plane constructions for the y and z axis.
      ...
    end for
  end for
end

SAH Tree Construction

With the splitting planes defined and computed, it is now possible to split the nodes. Zhou et al.[25] used SAH to chose a splitting plane, so this will also be our first approach. While SAH did not fit the upper tree construction phase because of too many triangles per node and the assumption that all splits resulted in leaf nodes, these are the very reasons that it fits so much better for lower tree construction. As noted previously in section 3.2, lower trees are constructed from leaf nodes maximally referencing

32 or 64 triangles. This means the time complexity for computing C_{SAH} per node is now bounded by $O(32^2) = O(1)$ or $O(64^2) = O(1)$, instead of $O(t^2)$ in the upper tree phase, where t is the number of triangles in an upper node and can be hundreds of thousands. Also the assumption that a split will result in two leafs in the final tree is far more likely to be true at the lower tree phase than the upper phase.

Algorithm 14 Calculate SAH cost

```
procedure ComputeSAHCost
    in activeNodes : Node List
    out splittingPlanes : Plane List, leafs : Boolean List
begin
  // Calculate SAH cost for each split candidate and return the split candidate with minimal cost.
  for each node in activeNodes in parallel do
    for each candidate in node.splitCandidates do
      set_l ← node.triangles ∩ candidate.left
      C_l ← ‖ set_l ‖
      A_l ← summed surface area of set_l
      set_r ← node.triangles ∩ candidate.right
      C_r ← ‖ set_r ‖
      A_r ← summed surface area of set_r
      weightedArea ← C_l · A_l + C_r · A_r
    end for
    splittingPlanes[node] ← p ← Split candidate with minimal weighted area
    C_SAH ← C_trav + p.weightedArea/node.summedArea
    leafs[node] ← C_SAH < C_leaf
  end for
end
```

The algorithm for computing the SAH cost of a splitting plane is given in algorithm 14. As described earlier the algorithm is parallelized over all nodes. A node must then iterate over all useful split candidates, i.e. those split candidates that were created from bounding boxes of the node's triangles. For each split candidate the weighted area, $C_l \cdot A_l + C_r \cdot A_r$, is computed and the one with the lowest is stored as the proposed splitting plane. The SAH value of the best split candidate is then computed and compared to the cost of leaving the node as a leaf node. The result of this comparison is returned at the end of the algorithm along with the best splitting plane.

A kernel then creates children to the nodes where $SAH < C_{leaf}$. In order to do this however, we must again first apply the prefix-sum. This time we apply it to the $leafs$ list and use the result to find the addresses where a nodes children should be created. An example is given below, where nodes n_0 and n_2 should have leafs created and these should be created at address 0 and 2 respectively.

$$\begin{array}{rcccc} node: & [n_0 & n_1 & n_2 & n_3] \\ leafs: & [1 & 0 & 1 & 0] \\ prefix\text{-}sum: & [0 & 1 & 1 & 2] \\ address: & [0 & 2 & 2 & 4] \end{array} \quad //prefix\text{-}sum \cdot children\ per\ node$$

Simplified SAH Tree Construction

While SAH's time complexity per node is $O(1)$, the iteration over all triangles assocated with a node to compute A_l and A_r is still quite expensive. For dynamic scenes I propose the following simplifying assumptions: That all triangles have the same surface area, specifically the surface area 1. Calculating A_n for a given node n could then be reduced from a loop to one single instruction $A_n = \|set_n\|$ and the entire SAH cost computation can be simplified as follows.

$$\begin{array}{l} C_{SAH}(n \to \{l, r\}) = C_{trav} + \frac{C_l A_l}{A_n} + \frac{C_r A_r}{A_n} \\ \Downarrow \\ C_{SSAH}(n \to \{l, r\}) = C_{trav} + \frac{C_i \|set_l\|^2}{\|set_n\|} + \frac{C_i \|set_r\|^2}{\|set_n\|} \end{array}$$

Algorithm 15 Calculate simplified SAH cost

procedure ComputeSimpleSAH
  in $activeNodes$: Node List
  out $splittingPlanes$: Plane List, $leafs$: Boolean List
begin
  // Calculate the simplified SAH cost for each split candidate and return the split candidate with minimal cost.
  for each *node* **in** $activeList$ **in parallel do**
    for each *candidate* **in** $node.splitCandidates$ **do**
      $set_l \leftarrow node.triangles \cap candidate.left$
      $set_r \leftarrow node.triangles \cap candidate.right$
      $weightedArea \leftarrow \|set_l\|^2 + \|set_r\|^2$
    end for
    $splittingPlanes[node] \leftarrow p \leftarrow$ Split candidate with minimal weighted area
    $C_{SSAH} \leftarrow C_{trav} + p.weightedArea/\|node.triangles\|$
    $leafs[node] \leftarrow C_{SSAH} < C_{leaf}$
  end for
end

The algorithm for computing this *Simplified Surface Area Heuristic* or *SSAH* can be seen in algorithm 15 and works similarly to algorithm 14. Just as with the SAH approach, child nodes are created in a seperat kernel after prefix-sum has been applied to the list $leafs$.

No Tree Construction

The final tree construction algorithm I will propose for the lower tree phase is extremely simple: Do not create any tree!

Recall from section 2.1.2 that the latency of global memory access becomes harder for the multiprocessor to hide if threads' global memory access can not be coalesced. This can happen for threads in the same warp traversing a tree, since at some point they are likely to diverge and traverse different parts of the tree, which results in scattered global memory accesses. Favoring leaf nodes with more triangles associated would reduce the size of the tree and thus the thread divergence. Reducing thread divergence would then in turn allow the warp to better coalesce global memory access and thereby reducing memory transaction and easier hide memory latency. I therefore propose not to create a lower tree, but instead terminate after having created the upper tree. This would save both the time spent preprocessing the nodes to compile the list of split candidates and the time spent actually creating the subtree.

Which of the lower tree construction methods proves the best will be evaluated in section 5.3.

Chapter 4

Ray Tracing

Some argue that in the very long term, rendering may best be solved by some variant of ray tracing, in which huge numbers of rays sample the environment for the eye's view of each frame. And there will also be colonies on Mars, underwater cities, and personal jet packs.
Tomas Möller and Eric Haines

In this chapter we shall look at how to implement a ray tracer and how it can be optimized to run efficiently on GPUs using NVIDIA's CUDA framework. The result will be an optimized ray tracer, which will be used in chapter 5 to evaluate the quality of kd-trees.

A ray, r, is defined as a line which starts at a point called *origin*, r_{ori}, and can be traced infinitely along a certain *direction*, r_{dir}. Using t to describe the distance traveled along the ray, r can then be described as $r(t) = t \cdot r_{dir} + r_{ori}$. Tracing a ray into the scene to find the nearest intersecting triangle, then means finding the triangle that the ray intersects at the lowest value of t. Before the actual ray tracing can be performed, the primary rays traced from the camera and into the scene must be computed and stored as a list of rays, *rays*. How these rays are generated is a large topic in itself and lies outside the scope of this thesis. Chapter 6 in [16] provides good insight into the theory needed for creating the primary rays.

The general structure for a ray tracer that handles reflective surfaces can be seen in algorithm 16. Algorithm 16 takes as input a list of rays to be traced, *rays*, the scene that should be ray traced, *scene*, and the image, *pixels*, on which the ray traced scene should be drawn. Each ray is independent of the other rays and can therefore be traced in parallel by the GPU. The algorithm first determines which of the triangles intersected by the ray is closest to the ray's origin, i.e. the triangle first *seen* by the ray. It then computes the color of the triangle intersected by the ray and blends that color with the color that has been accumulated so far in $pixels[r]$. How the color is computed is not relevant to the kd-tree evaluations in chapter 5 and is therefore not discussed in this thesis. Suffice it to say that the ray tracer produced as part of the thesis uses the *general lighting equation* described on page 83 in [3]. Finally the algorithm checks if a triangle is reflective and if so then a reflection ray is generated and added to the *nextRays* list. A reflection ray's direction, ref_{dir} is calculated from the triangle's normal, n, and the incoming rays direction, ray_{dir}

$$ref_{dir} = -2(n \cdot ray_{dir})n + ray_{dir}$$

The new origin of the reflection ray is simply the intersection point of the incoming ray and the triangle. As long as one or more reflection rays are generated by the traced rays, the ray tracing process is repeated.

In this chapter I will present two methods for determining which triangle in a scene a ray intersects. In section 4.1 I present an exhaustive ray tracer, which finds the closest intersection point by intersecting each ray with every triangle. In section 4.2 I then present a hierarchical ray tracer, which will use the kd-tree to minimize the number of triangles each ray needs to be intersected with. In the same section I will describe three optimizations that can be applied to hierachical ray tracers running on GPUs. The optimizations will be added incrementally to the hierarchical ray tracer and implementation details to each specific optimization is therefore discussed alongside the theory. The optimized ray tracer will use the *short-stack* optimization from Horn et al.[13], a *packet scheme* inspired by Wald et al.[20] and an

Algorithm 16 A general ray tracer.

```
procedure RayTracer
    in rays : Ray List, scene : Scene Description, pixels : Image
    out pixels : Image
begin
  for each r in rays in parallel do
    // The scene description can be either a list of triangles in the scene, or a kd-tree created from those
    triangles.
    triangle ← ClosestIntersectingTriangle(r, scene)
    pixels[r] ← ComputeColor(pixels[r], r, triangle)
    nextRays : Ray List
    if IsReflective(triangle) then
      nextRays.Add(GenerateReflectionRay(triangle, ray))
    end if
  end for
  if nextRays.NotEmpty then
    pixels ← RayTracer(nextRays, scene, pixels)
  end if
end
```

optimization inspired by Empty Space Maximization, that utilizes a leaf node's bounding box, which is computed while creating the kd-tree.

In section 4.1 and section 4.2 I will merely have assumed that a method exists for determining if and where a ray and a triangle intersects. In the final section of this chapter I will discuss two such methods with respect to the GPU's memory hierarchy and maximizing occupancy.

The ray tracers and their optimization will be evaluated in chapter 5 and hopefully show that the hierarchical ray tracer is faster than the exhaustive ray tracer. This will be a very important result, as the time spent rebuilding acceleration structures for dynamic scenes would otherwise be wasted.

4.1 Exhaustive Ray Tracing

Before directing our focus on hierarchical ray tracers, we will first take a look at an exhaustive ray tracer.

The reason that an exhaustive ray tracer is interesting is that the GPU comes with high computational power, but little tolerance for branching, as was discussed in section 2.1.1. An exhaustive ray tracer fits very well onto such an architecture, since intersecting each ray with every triangle certainly requires a lot of computational power, and every ray in a warp will loop over the same triangles and therefore not branch independently.

The ClosestIntersectingTriangle procedure used by algorithm 16 is implemented in algorithm 17 using an exhaustive ray tracer. The actual ray tracers implemented in this thesis also compute and return a ray's barycentric coordinates on the triangle it intersects, which is explained in section 4.3. The barycentric coordinates are used to perform lighting calculations, but in order to keep the algorithms simple and instructive, this has been omitted and is assumed to be performed by the ComputeColor method invoked in algorithm 16.

The downside to the exhaustive approach is that intersecting a ray with m triangles, yields a time complexity of $O(m)$.

4.2 Hierarchical Ray Tracing

Hierarchical ray tracers provide a better upper bound than exhaustive ray tracers. Given a hierachical data structure over m geometric primitives in a scene, the leaf node that a ray should traverse next can be found in $O(\log m)$ time. This is significantly better than the exhaustive ray tracer from section 4.1.

When ray tracing a scene using a kd-tree as acceleration structure, each ray traverses the tree in search of the leaf node closest to the rays origin, while still being in front of the ray. The leaf nodes is

Algorithm 17 An exhaustive implementation of ClosestIntersectingTriangle

```
procedure ClosestIntersectingTriangle
    in R : Ray, Ts, Triangle List
    out t_near : Triangle
begin
  t_near ← NULL
  for each t in Ts do
    t_near ← ClosestHit(t, t_near)
  end for
end
```

hereafter refered to as the *nearest leaf*. At each interior node, the ray needs to determine which of the child nodes it should traverse next. This process is continued until a leaf node is reached. The ray then needs to intersect each triangle in the leaf and determine which intersected triangle is nearest, if any. If the ray intersects a triangle, then it reports this intersection, otherwise it needs to continue traversing the tree. How this is done is the topic of section 4.2.1 and section 4.2.2. The general algorithm for rays traversing a hierarchical acceleration structure can be seen in algorithm 18.

Algorithm 18 A general algorithm for rays a traversing hierarchical acceleration structure.

```
procedure ClosestIntersectingTriangle
    in ray : Ray, tree : kd-tree
    out t_near : Triangle
begin
  t_near ← NULL
  while ray inside scene do
    // Traverse the tree until the leaf node nearest the ray origin is found and store the distance to the
    nearest splitting plane the ray intersects in d_split.
    (leaf, d_split) ← TraverseTree(ray, tree)
    // Intersect triangles in leaf and return the nearest intersected triangle, if any.
    t_near ← Intersect(ray, leaf.triangles)
    // Break if a closest intersection is found and return it.
    if t_near ≠ NULL then
      break
    else
      ray.origin ← ray.origin + d_split * ray.direction
    end if
  end while
end
```

The method used to determine which of an interior node's children to visit next can be seen in algorithm 19. For each interior node the ray first needs to determine which of the two children are placed *nearest* and *farthest* along the rays direction. It then computes the signed distance from the ray to the nodes splitting plane. If this distance is below 0 then the plane is located behind the ray and the farthest child must be visited. If the distance is above 0 then the nearest node needs to be visited next. In addition the distance, d_{split}, to the nearest splitting plane in front of the ray needs to be updated. d_{split} is used to advance the ray beyond a visited leaf node if it did not intersect any geometry.

In figure 4.1a a simple scene consisting of four triangles is shown. The corrosponding kd-tree matching the axis aligned splitting planes is shown in figure 4.1b. A tree traversal of the ray, $r(t) = t\,[2,1]^T + [0,1]^T$, entering the scene in the lower left would look like this: Upon entering the scene the ray first visits the root node, 0. Node 0 splits the scene along the x-axis at $x = 4$ and the rays distance to that plane is then $d_{split} = (4-0)/2 = 2$. The ray therefore proceeds to the childnode $2 > 0\ ?\ 1 : 2 = 1$. The signed distance to node 1's splitting plane is $d_{node} = (4-1)/1 = 3$ and the ray proceeds to node $1 > 0\ ?\ 3 : 2 = 3$, where it finds no primitives to intersect. The ray is then advanced beyond the nearest splitting plane by advancing it d_{split} along its direction. The new ray becomes $r'(t) = r(t) + d_{split} \cdot r_{dir} =$

Algorithm 19 A basic kd-tree traversal algorithm

```
procedure TraverseTree
    in ray : Ray, tree : kd-tree
    out leaf : Node, d_split : Number
begin
  d_split ← ∞
  node ← tree.root
  while node ≠ LEAF do
    // The signed distance to the nodes splitting plane.
    node_near ← ray.direction[node.axis] > 0 ? node.left : node.right
    node_far ← ray.direction[node.axis] > 0 ? node.right : node.left
    d_node ← (node.splitValue − ray.origin[node.axis])/ray.direction[node.axis]
    if 0 < d_split then
      node ← node_near
      d_split ← min(d_split, d_node)
    else
      node ← node_far
    end if
  end while
end
```

$r(t) + 2\,[2,1]^T = t\,[2,1]^T + [4,3]^T$. Since the ray did not intersect any geometry and has not advanced beyond the scene, the traversal process is restarted for $r'(t)$.

4.2.1 KD-restart

The approach presented above, where kd-tree traversal is restarted at the root node, is known as *kd-restart* and is one of the simplest algorithms for ray tracing kd-trees. The reason for the name is that if a ray does not intersect any triangles in the nearest leaf node, then it advances past that leaf node using d_{split} and restarts traversal from the root of the tree.

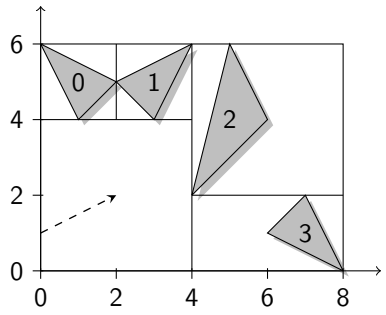
The algorithm in its entirety is presented in algorithm 20. Instead of advancing the ray by updating its origin, a signed distance, d_{min}, is used to determine how far a ray has traveled.

4.2.2 Short-stack

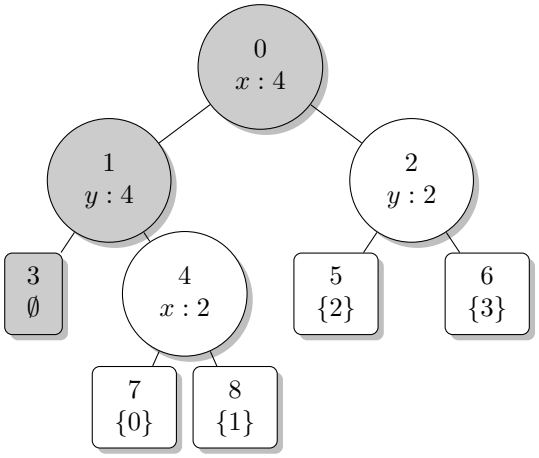
When a ray triggers a restart in kd-restart, it will almost always traverse many of the same interior nodes as it did in its previous traversel. The reason for this is that kd-trees will often store spatially local nodes as local nodes in the tree. CPU ray tracers utilize this property by pushing the not-visited child of an interior node onto a stack and then resuming from the first node on that stack instead of restarting. This can save a lot of resources otherwise spent on traversing the tree, since the ray will now skip interior nodes already visited and resume closer to the next leaf node.

CUDA's memory model, however, is not flexible enough for this approach, which requires either dynamically allocating more memory if the stack is filled or pre-allocating a *large enough* stack in local memory to handle any given kd-tree. This would require quite large amounts of memory for huge scenes and might even require too much memory for practical use on today's graphics cards. The solution proposed by Horn et al.[13] was to use a *short-stack*, a fixed-size, circular stack of N elements, and then revert to using kd-restart if the stack underflows. The circular nature of the short-stack means that the stack will favor nodes near the leafs and overwrite nodes traversed at the top of the kd-tree. Since the nodes near the leafs are the ones with the highest associated traversal cost, these are exactly the ones we would like to be able to resume from in future traversals. In their research Horn et al.[13] found that a short-stack approach only visited 3% more nodes compared to the unlimited stack in CPU approaches.

Obviously all child nodes can not be pushed onto the short-stack, as that would force the rays to visit the entire kd-tree. Therefore restrictions need to be formulated as to which nodes should be pushed onto the short-stack. The first restriction is not to push nodes located behind the ray, since the ray can never intersect these. The second restriction is not to push nodes where the distance to the splitting plane is



(a) A simple scene subdivided by a kd-tree.



(b) The kd-tree matching the scene. The gray shaded nodes represent the ray's traversal of the kd-tree.

Figure 4.1

Algorithm 20 A kd-restart implementation of ClosestIntersectingTriangle

```
procedure ClosestIntersectingTriangle
    in ray : Ray, tree : kd-tree
    out t_near : Triangle
begin
  t_near ← NULL
  d_min ← 0
  while d_min < ∞ do
    d_split ← ∞
    node ← tree.root
    // Traverse the tree until a leaf node is reached
    while node ≠ LEAF do
      d_node ← (node.splitPosition − ray.origin[node.axis])/ray.direction[node.axis]
      if d_min < d_node then
        node ← ray.direction[node.axis] > 0 ? node.left : node.right
        d_split ← min(d_split, d_node)
      else
        node ← ray.direction[node.axis] > 0 ? node.right : node.left
      end if
    end while
    // Test intersection with the leafs primitives
    t_near ← Intersect(ray, leaf.triangles)
    if t_near ≠ NULL then
      break
    else
      // Advance the ray beyond the next splitting plane.
      d_min ← d_split
    end if
  end while
end
```

greater than d_{split}, as these would never be visited in a kd-restart traversal. The reasoning for this is that if a ray advances beyond the splitting plane of one of its parent nodes, it effectively means the ray should never visit that nodes nearest subtree again and therefore no nodes from that subtree should be stored in the short-stack. An implementation of ClostestIntersectingTriangle for algorithm 16 using the short-stack optimization is shown in algorithm 21.

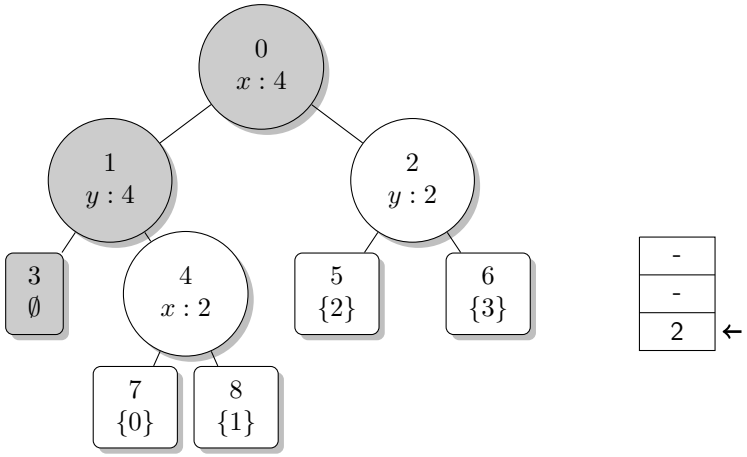
Algorithm 21 A short-stack implementation of ClosestIntersectingTriangle

```
procedure ClosestIntersectingTriangle
    in ray : Ray, tree : kd-tree
    out t_near : Triangle
begin
  t_near ← NULL
  d_min ← 0
  while d_min < ∞ do
    if stack.IsEmpty then
      node ← tree.root
      d_split ← ∞
    else
      (node, d_split) ← stack.Pop
    end if
    // Traverse the tree until a leaf node is reached
    while node ≠ LEAF do
      d_node ← (node.splitPosition − ray.origin[node.axis])/ray.direction[node.axis]
      if d_min < d_node then
        node ← ray.direction[node.axis] > 0 ? node.left : node.right
        if d_node < d_split then
          stack.push(upperChild, d_split)
        end if
        d_split ← min(d_split, d_node)
      else
        node ← ray.direction[node.axis] > 0 ? node.right : node.left
      end if
    end while
    // Test intersection with the leafs primitives
    t_near ← Intersect(ray, leaf.triangles)
    if d_min < t_split then
      break
    else
      // Advance the ray beyond the next splitting plane.
      d_min ← d_next
    end if
  end while
end
```

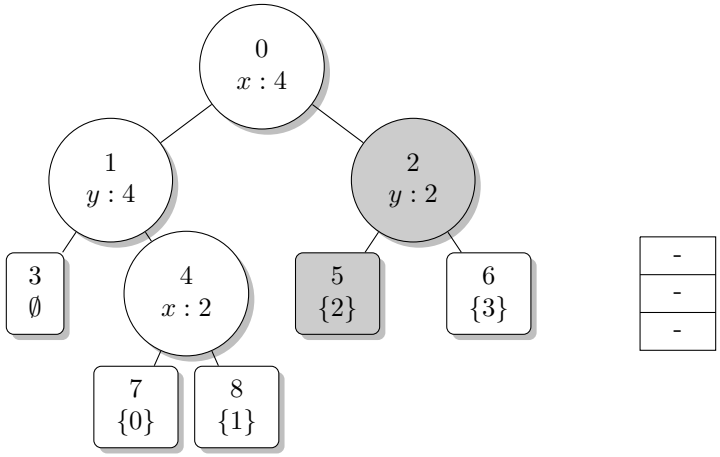
Let us look at a simple example of how the short-stack will speedup traversal. Using the scene in figure 4.1 again, we saw previously that the ray, $R(t) = t\,[2,1]^T + [0,1]^T$, will traverse nodes 0, 1, and 3. At node 0 the distance to the splitting plane was 2, meaning the splitting plane is in front and the child not traversed should be pushed to the short-stack. At node 1 the distance was 3, so the ray intersects node 1's splitting plane after node 0's. Node 1's other child therefore does not need to be pushed to the stack. The result of the first traversal can be seen on figure 4.2a.

So far the short-stack traversal and kd-restart traversal have both chosen the exact same path through the tree. But where kd-restart would have had to restart its traversal from the root node, the short-stack allows the ray to start the second traversal from node 2 and proceed to leaf 5, where the ray intersects triangle 2.

In this simple example having a short-stack only allows the ray to skip one node at the cost of



(a) The short-stack algorithm's first traversal of the scene from figure 4.1. When the ray traversed node 0 it pushed node 2 onto the short-stack.



(b) The short-stack algorithm's second traversal of the scene from figure 4.1. Traversal is resumed from the first node on the stack, which is 2.

Figure 4.2

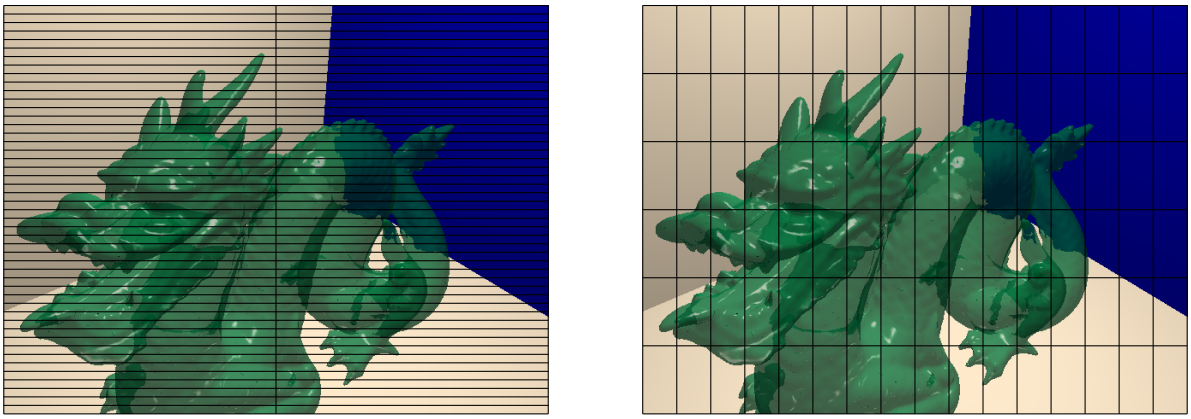
maintaining and using a short-stack, which will cause a significant overhead. Consequently having a short-stack will probably not provide any speedup in this small example. But imagine this simple tree as a subtree in a much larger scene, with perhaps ten levels of nodes above it. Then a short-stack implementation allows the ray to skip those first eleven nodes, which can yield quite a performance improvement.

4.2.3 Packets

Using some form of *packets* to accelerate ray tracing is quite a standard technique. Packets were introduced to CPU ray tracers by Wald et al.[20], who utilized *SIMD*[1] instructions to trace rays in packets of four rays at a time. Horn et al.[13] extended their short-stack implementation with another form of packet tracing, where individual threads would trace multiple rays to amortize the cost of traversing the tree. Unfortunately tracing several rays in one thread increases thread incoherence across threads in a warp and Aila and Laine[1] concludes that

> *It is worth noticing that [kd-restart] is faster than packet traversal in all cases, and with diffuse rays the difference is approximately 2X.*
>
> [1]

[1]Single instruction, multiple data.



(a) Sequential ray tracing. Notice how almost all of the warps overlap both the dragon and the background.

(b) Spatially coherent ray tracing. Fewer warps will now ray trace the dragon and the spatial coherence between rays results in entire warps raytracing only the background.

Figure 4.3

While Horn et al.[13]'s approach to packet tracing might not have yielded the intended increase in performance, the GPU is still tracing rays in warp-sized packets and an effective ray tracer needs to address this.

On the GPU the rays are stored in rows from left to right in a linear list. For an image with a resolution of 64x48, processing the n'th ray with the n'th tread results in the ray/warp classification seen in figure 4.3a, which I will call *sequantial ray tracing*. As seen on the figure, a large number of warps are intersecting the detailed dragon geometry. The threads in those warps who quickly intersect their rays with the simple background will then have to idle, while waiting for the rays intersecting the detailed dragon to finish.

Instead I propose to take advantage of the spatial coherence between neighbouring rays and organize spatially coherent rays into warpsized packets as done in figure 4.3b, where the rays are organized into packets with a width, p_{width}, of 4 and a height, p_{height}, of 8. As the figure demonstrates, not only does fewer warps ray trace the detailed dragon, but the spatial coherence between the rays in a warp causes more warps to ray trace the same spatially local geometry, which results in fewer global memory transactions per warp.

The calculations needed to transform sequentially ordered ray ids into spatially local ray ids are quite simple and can be seen in algorithm 22. The performance benefits gained by tracing spatially coherent rays and fewer global memory transactions are enormous though, as seen in figure 5.2 on page 49.

It should be noted that a possibility exists that rays reflected multiple times may eventually diverge completely and no longer be spatially coherent, but this is no more of a problem for spatially local ray packets, than it is with the sequential ray tracing approach and the spatially local ray packets will still speed up primary rays.

4.2.4 Skipping Leaf Nodes

The final ray tracer optimization presented in this thesis is inspired by the early out option provided by Empty Space Maximization.

Since the only information available to rays traversing a kd-tree is the position of the splitting plane and not the bounding volume of the node, the ray can easily get sidetracked and end up in leaf nodes, whose bounding box it does not even intersect. Figure 4.4 is a simple example of this. Extending the ray, $R(t) = t\,[1,1]^T + [-4,0]^T$, we can clearly see that it will intersect leaf node 2. Unfortunately, when using kd-trees, the only information available to the ray is that the node is split along the y-axis at $y = 3$, which in the above example causes the ray to get sidetracked into leaf 1. This is a problem in scenes with large empty areas that a ray has to traverse. Each time the ray intersects a splitting plane in those empty areas, it can get sidetracked and traverse subtrees, whose associated geometry the ray does not

Algorithm 22 Converting a thread id to a ray id.

procedure PacketID
  in *thread* : id
  out *ray* : id
begin
  // Calculate the index of the packet cell.
  $p_{id} \leftarrow thread/(p_{width} * p_{height})$
  // Then compute the location of that cell in the grid of packets.
  $grid_x \leftarrow p_{id} \mod (image_{width}/p_{width})$
  $grid_y \leftarrow p_{id}/(image_{width}/p_{width})$
  // Compute the location of the ray inside that packet cell.
  $p_x \leftarrow threads \mod p_{width}$
  $p_y \leftarrow (threads \mod (p_{width} * p_{height}))/p_{width}$
  // Finally compute the location of the ray inside the grid and return the ray id.
  $ray_x \leftarrow grid_x * p_{width} + p_x$
  $ray_y \leftarrow grid_y * p_{height} + p_y$
  $ray \leftarrow ray_x + ray_y * screen_{width}$
end

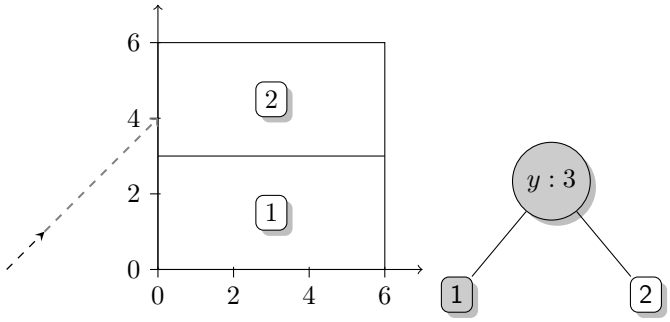


Figure 4.4: A ray traversing a leaf node, whose bounding box it does not intersect.

intersect.

Empty Space Maximization has shown us that providing an early out option for rays that would otherwise get sidetracked can increase performance substantially. Since the leaf's bounding boxes are known from the tree creation phase, they can be used to provide such an early out option and quickly skip leaf nodes that the ray does not intersect. This saves a ray a lot of triangle intersection tests.

This optimization is only possible because I've stored the leafs bounding box and will not be applicable in scenes where only a kd-tree is available.

4.3 Ray/Triangle Intersection

The inherent diverging behaviour of rays traversing the kd-tree makes them hard to parallize efficiently on the graphics card, since data access will be nearly impossible to coalesce. The solution is to have enough active warps to effectively hide the latency from global data fetching. This means that each rays register usage must be kept to a minimum, in order to have enough memory for as many active warps as possible.

Deciding which ray/triangle intersection method to use when ray tracing can lower register usage and thus yield a significant performance boost.

Ray/triangle intersection can be broken down into two parts: A *ray/plane intersection* test, which computes the signed distance, t, a ray must travel to intersect the plane spanned by a triangle, and a *triangle inclusion* test, which checks if the intersection point is located inside the triangle. The inclussion test is usually performed by calculating the intersection point's *barycentric coordinates* within the triangle.

Every point on a triangle, T, with vertices T_a, T_b and T_c, can be described as

$$T(u,v) = (1-u-v)T_a + uT_b + vT_c, 0 \leq u, 0 \leq v, u+v \leq 1$$

where (u, v) are the aforementioned barycentric coordinates. For a barycentric coordinate to be inside T it must therefore fulfill the requirements $0 \leq u$, $0 \leq v$ and $u + v \leq 1$.

Just as barycentric coordinates are used to linearly interpolate the vertex positions across the triangle, they are also important for interpolating other vertex attributes, such as normals or colors.

The following two ray/triangle intersection methods both compute the signed distance and barycentric coordinates.

4.3.1 Möller-Trumbore

The algorithm presented in Möller-Trumbore[5] is one of the most popular algorithms for ray/triangle intersection, probably in no small part due to its appearence in chapter 16.8 of Real-Time Rendering[4], but also because it is one of the fastest ray/triangle intersection algorithms that do not rely on extra memory or preprocessing of the triangle before ray tracing.

Möller-Trumbore[5] exploits the fact that computing the intersection between a triangle, $T(u, v)$ and ray, $R(t) = tR_{dir} + R_{ori}$ is equivalent to solving the linear system of equations, the *ray/triangle intersection equation*

$$\begin{aligned} & R(t) = T(u,v) \\ \Updownarrow & \\ & tR_{dir} + R_{ori} = (1-u-v)T_a + uT_b + vT_c \end{aligned}$$

Solving the ray/triangle intersection equation amounts to determining the signed distance from the ray to the triangle, t, and the barycentric coordinates, (u, v). In [5] this is done by rearranging the terms and applying Cramer's Rule, and the solution then becomes

$$\begin{bmatrix} t \\ u \\ v \end{bmatrix} = \frac{1}{p \cdot e_1} \begin{bmatrix} q \cdot e_2 \\ p \cdot r \\ q \cdot R_{dir} \end{bmatrix}$$

where $e_1 = T_b - T_a$, $e_2 = T_c - T_a$, $r = R_{ori} - T_a$, $p = R_{dir} \times e_2$ and $q = r \times e_1$.

Implementation

The implementation of this method is pretty straightforward using CUDA's vector primitives and can be seen in algorithm 23. The implementation provided in [5] was optimized for the CPU and therefore provided a lot of early out options as soon as the algorithm detected that a ray had missed the triangle. Due to the synchonized behaviour of threads in the same warp, providing as many early out possibilities as [5] causes a branching overhead and results in reduced performance. This is understandable since only one ray needs to intersect its triangle in order for every thread in the warp to have to wait for it to finish. For the ray tracers implemented as part of this thesis the optimal compromise was found to be only providing one early exit after the ray/plane intersection and before computing the barycentric coordinates.

4.3.2 Woop

While Möller-Trumbore's ray/triangle intersection approach requires minimal storage, in the sense that it takes as input the vertex positions already stored in global memory, it does unfortunately require quite a lot of registers, since it needs to both store the ray's parameters aswell as the determinant and the matrix $[q \cdot e_2, p \cdot r, q \cdot R_{dir}]^T$. The high register usage can be remedied by performing precalulations per triangle as described in Chapter 5 of Sven Woops diploma thesis[24]. The trade-off is a higher per triangle storage requirement for the scene's geometry.

He observed that the triangle, T, can be transformed into the unit triangle

$$U(u,v) = (1-u-v)\,[0,0,0]^T + u\,[1,0,0]^T + v\,[0,1,0]^T$$

Algorithm 23 Möller-Trumbore ray/triangle intersection test

procedure Möller-Trumbore
  in T : Triangle, R : Ray
  out hit : Bool, t : Distance, (u, v) : Barycentric Coordinates
begin
  $e_1 \leftarrow T_b - T_a$
  $e_2 \leftarrow T_c - T_a$
  $r \leftarrow R_{ori} - T_a$
  $p \leftarrow R_{dir} - e_2$
  $q \leftarrow r - e_1$
  $determinant \leftarrow p \cdot e_1$
  $t \leftarrow (q \cdot e_2)/determinant$
  // Provide early out if the triangle is behind the ray.
  if $0 < t$ **then**
    $u \leftarrow (p \cdot r)/determinant$
    $v \leftarrow (q \cdot R_{dir})/determinant$
    $hit \leftarrow 0 \leq u$ **and** $0 \leq v$ **and** $u + v \leq 1$
  else
    $hit \leftarrow false$
  end if
end

by applying an affine transformation, consisting of a linear transformation matrix, M, and a translation vector, n, on each of T's vertices T_a, T_b and T_c.

$$U_v = MT_v + n, v \in \{a, b, c\}$$

The inverse affine transformation is the one that transforms the unit triangle into T.

$$T_v = M'U_v + n', v \in \{a, b, c\}, M' = M^{-1}, n' = -M^{-1}n$$

Given that we already know T and U, M' and n' can be constructed as follows

$$M' = [T_a - T_c, T_b - T_c, (T_a - T_c) \times (T_b - T_c)]$$
$$n' = T_c$$

We can then derive M and n from M' and n'.

$$M = (M')^{-1}, n = -Mn'$$

This requires M' to be invertable, which it always is for *non-degenerate triangles*[2].

Applying the affine transformation described by M and n to the ray, $MR(t)+n = R'(t) = R'_{ori}+tR'_{dir}$, it is transformed into the same space as the unit triangle. Computing the distance, t, and barycentric coordinates, (u, v) then becomes trivial.

$$t = -R'_{dir,z}/R'_{ori,z}$$
$$u = tR'_{dir,x} + R'_{ori,x}$$
$$v = tR'_{dir,y} + R'_{ori,y}$$

Since each dimension in R' can be computed independently, we get

$$t = -(M_z \cdot R_{dir} + n_z)/(M_z \cdot R_{ori})$$
$$u = t(M_x \cdot R_{dir} + n_x) + M_x \cdot R_{ori}$$
$$u = t(M_y \cdot R_{dir} + n_y) + M_y \cdot R_{ori}$$

where M_i is the i'th row of M.

[2] A degenerate triangles is a triangles that has collapsed into a line or a point.

Implementation

The implementation is straightforward. M and n are computed and stored in 3 four component vectors as $w_i = [M_i, n_i]$ prior to ray tracing. During ray/triangle intersection each vector w_i can then be loaded into register memory one vector at a time. In practice this saves three registers compared with the Möller-Trumbore approach and can yield a sizeable performance increase, as can be seen in figure 5.2 on page 49.

As in the implementation of Möller-Trumbore, the Woop implementation also only has one early out option, which is placed right after the distance calculation.

Finally, the extra memory required for storing the precalculated m and n can be mitigated if the triangles vertex positions are replaced by m. This is only possible however, if the vertices are not used elsewhere in the ray tracer.

Chapter 5

Results

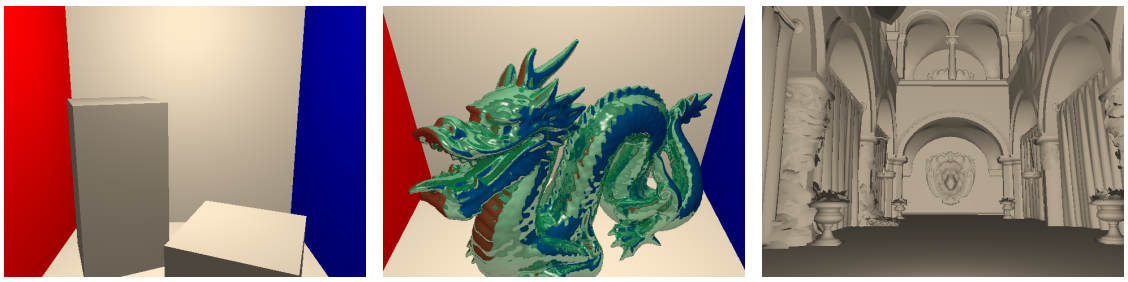
It's hardware that makes a machine fast. It's software that makes a fast machine slow.
Craig Bruce

In this chapter I will compare the construction speed and quality of different kd-tree construction schemes implemented as part of this thesis.

The implementations have been tested on a dual core Intel Core i7 2.66GHz CPU with 4GB RAM. The GPU used is an NVIDIA 330M with CUDA Compute Capability 1.2, 6 multiprocessors running at 1.1GHz and 512MB RAM. CUDA 3.1 was installed on the machine. All tests where performed with a screen resolution of 640x480, which equals 307200 primary rays traced into the scene. The timing results presented below where obtained by rendering the scene at least 20 times. Due to a lack of precision in the method used to time the results, they could vary by approximately 2% between rendered images. In an attempt to prevent this variance from affecting the results, the timings presented below are the lowest measured over those 20 renderings. Fortunately the variance will not affect the evaluation of the kd-tree implementation, as the performance difference between the kd-tree implementations far exceed 2%.

The tests will be conducted on three different test scenes shown in figure 5.1. *The Cornell Box* scene was chosen for its simplicity. With only 36 triangles the scene should be very fast to render and allows me to compare the performance of the ray tracer implementations on a simple scene. The *Sponza* scene was chosen for its complexity. At 279k triangles it will test both the quality and the construction speed of the kd-tree constructors in large scenes. The last scene tested is *The Reflecting Stanford Dragon* consisting of 203k triangles. The reflecting geometry in the scene spawns 189572 reflection rays and these will be used test the importance of the kd-tree's quality when more rays than just the primary are traced.

In section 5.1 I will evaluate the different ray tracer implementations and the impact of the optimizations described in section 4.2. I have chosen to implement the short-stack optimization as a seperat ray tracer, instead of applying it to the kd-restart ray tracer. This is merely a design decision and has no impact on the actual results presented below. The overall fastest ray tracer will be used in the following sections to evaluate the quality of the kd-trees constructed. In section 5.2 I will compare quality and



(a) The Cornell Box - 36 triangles. **(b)** The Reflecting Stanford Dragon - 203k triangles. **(c)** Sponza - 279k triangles.

Figure 5.1: The three test scenes used in this chapter.

Ray Tracer:	*Ray/triangle intersection:*	*Packets:*	*Leaf skipping:*	Cornell Box 1.0ms (3)	Reflecting Dragon 150ms (49909)	Sponza 263ms (98779)
Exhaustive[a]	Moeller-Trumbore	None	N/A	11.2ms	1118ms	964ms
		4x8	N/A	11.6ms	1114ms	964ms
	Woop	None	N/A	**9.3ms**	**898ms**	**650ms**
		4x8	N/A	9.7ms	903ms	651ms
KD-Restart	Moeller-Trumbore	None	No	17.9ms	790ms	592ms
			Yes	18.1ms	486ms	287ms
		4x8	No	17.2ms	579ms	424ms
			Yes	17.4ms	**355ms**	**196ms**
	Woop	None	No	15.5ms	690ms	549ms
			Yes	15.8ms	500ms	287ms
		4x8	No	**14.6ms**	495ms	342ms
			Yes	14.8ms	**355ms**	**198ms**
Short-Stack	Moeller-Trumbore	None	No	20.0ms	808ms	595ms
			Yes	20.2ms	503ms	266ms
		4x8	No	19.5ms	590ms	421ms
			Yes	19.8ms	**360ms**	**167ms**
	Woop	None	No	18.5ms	738ms	560ms
			Yes	18.7ms	522ms	270ms
		4x8	No	**17.8ms**	523ms	348ms
			Yes	18.1ms	368ms	171ms

[a]The exhaustive ray tracer only ray traces the first 4096 triangles in a scene. Even with this hard upper limit it is clear that an exhaustive approach will not work for detailed scenes.

Figure 5.2: The table shows the time it took different ray tracer configurations to render each of the three test scenes. The result of the ray tracer configurations that performed the best on a scene are written boldfaced, while the results in gray performed the worst. The kd-trees for the scenes were constructed with Empty Space Maximization enabled and a threshold of 25%, the triangle/node association scheme employed was Triangle/Node Overlap. The threshold for the lower tree was 32, but no lower tree was constructed. The number of nodes in the kd-tree is reported in parenthesis below the name of the scene. The size of the short-stack was 4 elements, which has been found to perform well in practice.

construction speed of the different construction schemes for the upper part of the kd-tree. The upper tree configuration that performs best, will be used in section 5.2, where the kd-tree's lower parts will be constructed using SAH, SSAH and no construction scheme.

As explained in the introduction, the worst-case scenario for a dynamic scene is a complete reconstruction of the acceleration structure. To ensure that my implementations are tested against the worst-case scenario, the acceleration structures are completely rebuild for each image rendered. The best kd-tree configuration is therefore the one that minimizes the total time spent constructing the kd-tree and afterwards ray tracing it.

5.1 Evaluate Ray Tracers

Figure 5.2 shows the time it took the different ray tracer configurations to render each of the three test scenes. The kd-tree configuration used to produce kd-trees for the hierarchical ray tracers is described in the figure's caption.

From figure 5.2 it can be seen that the exhaustive ray tracer generally performs worst of all the ray tracers. This was to be expected, since the exhaustive ray tracers time complexity $O(m)$ per ray and the hierarchical ray tracers' is $O(\log m)$, for m triangles. As expected the implementation of Woop's simpler ray/triangle intersection performs better than Moeller-Trumbore's for all exhaustive ray tracer configurations. The packet optimization has also been applied to the exhaustive ray tracer, but generally

causes a small overhead instead of a performance improvement. This was also to be expected, since all rays intersect the triangles in the same order, and therefore it does not matter which rays are traced together. In the Reflecting Dragon scene we see a small speed increase when using packets. This may be the result of fewer warps tracing reflection rays, but can also be caused by a deviation in timing precision.

In the Cornell Box scene, the kd-restart implementation performed worse than the exhaustive ray tracer, even without taking the kd-tree construction time of 1ms into account. This is not entirely unexpected, since the exhaustive ray tracer only needs to intersect every ray with 36 triangles, where the kd-restart implementation needs to first traverse the root node, before it can begin ray/triangle intersection tests in leafs that can reference up to 32 triangles. In the more complex scenes, kd-restart outperformed exhaustive ray tracing by a wide margin. Unsurprisingly, the configuration consisting of Moeller-Trumbore intersection, no packets and no leaf skipping performed the worst. It was a bit more surprising to see that with packets and leaf skipping enabled, it did not matter much if Moeller-Trumbore's or Woop's intersection method was used. Especially since Woop's intersection test generally outperformed Moeller-Trumbore. However, this makes sense since the rays in the same warp are spatially close and are therefore able to skip most of the leaf nodes they visit, and thus a lot of intersection computations, before reaching their intersection point.

The packet size 4x8 was found to perform best and is therefore the only packet size represented in figure 5.2 to keep the table simple. In general enabling Packets decreased rendering time by 27-37%.

Enabling Leaf Skipping decrased rendering time by 28-54%. The largest decrease in rendering time was seen in configurations that used Moeller-Trumbore's ray/triangle intersection tests, again confirming that Moeller-Trumbore is the slowest ray/triangle intersection compared to Woop's.

The short-stack implementation performed very similar to kd-restart. In the simple Cornell Box scene it performed slightly worse, due to the overhead of maintaining a stack. In the Reflecting Dragon scene, it performed roughly similar to kd-restart and in the Sponza scene it outperformed both the exhaustive ray tracer, kd-restart and was about 70% faster than an unoptimized kd-restart implementation.

Overall the short-stack implementation with Moeller-Trumber intersection, packets and leaf skipping enabled performed the best. While it may have been outperformed in the Cornell Box, it was by far the best ray tracer in the complex Sponza scene. This ray tracer configuration will therefore be used in the following sections to evaluate the quality of kd-trees.

5.2 Evaluate Upper Tree Creation

Since the Cornell Box scene contains so few triangles and only shows that the exhaustive ray tracer is fastest for extremely simple scenes, I will not be using it to evaluate the kd-tree constructor configurations. Instead I will only use the Reflecting Dragon and Sponza scenes.

The results from creating the upper parts of the kd-tree with different configurations can be seen in figure 5.3. The entries in the table shows kd-tree construction time, the time used to render the scene and the number in parenthesis is the amount of nodes in the constructed kd-tree.

From the results it is clear that using the Triangle/Node Overlap association scheme produces trees of higher quality than Box Inclusion. In the Reflecting Dragon scene, trees build with Triangle/Node Overlap are generally ray traced 10ms, or 2.5%, faster and contain 3k fewer nodes than their Box Inclusion counterparts. In the Sponza scene the difference is even more noticeable at up to 8.7%. Unfortunately producing kd-trees using the Triangle/Node Overlap scheme is a lot slower than using Box Inclusion. In the above scenes this means that while Triangle/Node Overlap produces the highest quality trees, Box Inclusion makes up for its slower ray tracing results by producing trees up to 19% faster.

As can be seen on the table, enabling Empty Space Maximization adds 4-6ms to the kd-tree construction time and a few thousand extra nodes to the tree. Enabling it also increase the tree quality however and ray tracing improvements lie in the range of 8% for the Reflecting Dragon and 24% in Sponza's great hall. The substantial increase in Sponza is a direct consequence of all the empty space in the middle of the scene that the rays have to traverse. As for setting the Empty Space Threshold, 35% seems to be marginally better than 25%. They achieve roughly the same tree quality, but with a 35% Empty Space Threshold the tree becomes 4-10% smaller than a tree with a 25% threshold. The tree will thus take up less memory, which can become important on GPUs that have limited memory.

Association scheme:	*Empty Space Maximization:*	*Empty Space Threshold:*	Reflecting Dragon	Sponza
Triangle/Node Overlap	No	N/A	146/385ms (38k)	258/220ms (87k)
	Yes	15%	150/376ms (56k)	263/167ms (104k)
		25%	**150/355ms** (50k)	**263/167ms (99k)**
		35%	150/357ms (45k)	**263/167ms (95k)**
Box Inclusion	No	N/A	116/397ms (41k)	213/241ms (112k)
	Yes	15%	121/386ms (59k)	219/182ms (130k)
		25%	121/373ms (53k)	219/182ms (125k)
		35%	**121/366ms (48k)**	**219/180ms (120k)**

Figure 5.3: The entries in the table show the time it took to create the kd-tree in milliseconds, the time it took to render the scene and the number of nodes in the kd-tree is shown in parenthesis. The scene was ray traced using the short-stack configuration that performed best in previous tests. To keep the lower tree construction phase from having as little influence on the results as possible, no lower tree was constructed and the lower tree threshold was set to 32. As in figure 5.2 the boldfaced text denotes the fastest configuration and gray the slowest.

Overall configuring the tree constructor to use Box Inclusion and Empty Space Maximization with a 35% threshold yielded the best tradeoff between construction speed and ray tracing time.

5.3 Evaluate Lower Tree Creation

For the lower tree creation phase, using a bit mask of size 32 generally performs better than one of size 64. One of the reasons for this is the added register usage from having larger bit masks. In the case of the ray tracers it means storing an extra 32bit of information per thread, which results in more register spilling. The real performance hit however comes when creating the lower tree. The reason for this is that the register usage of the kernels implementing algorithm 13, algorithm 14 and algorithm 15 in section 3.2.2 increase drastically when switching to 64bit bit masks. An example is algorithm 13 which increase from a register usage of 30 per thread when 32bit bit masks are used, to 53 registers per thread at 64 bits. Inputting this register usage into the *CUDA Occupancy Calculator*[1] we can see that 53 registers per thread results in an occupancy of 25%, which is not enough to properly utilize the GPU's multiprocessors, nor enough to efficient hide global memory latency. Trying to minimize the register usage by storing more computations in global memory only increases the global memory latency and does little for performance.

Looking at the results for kd-trees with 32bit bit masks, constructing no lower tree performs best overall. Obviously kd-tree constructors that do not construct any lower tree will have the fastest construction time, but it was surprising that they also had the fastest ray tracing time and therefore the highest tree quality. This data seems to verify my idea from the No Tree Construction subsection in section 3.2.2, that favoring larger leaf nodes on GPUs can reduce thread divergence and increase performance.

Using the Surface Area Heuristic to construct the lower trees instead of no construction increases construction time by up to 253% in the worst case, while all the time yielding kd-trees of similar quality.

Using my proposed Simplified Surface Area Heuristic instead only increases construction time by 31-45% compared to no tree construction, but also reduces the tree quality by up to 27%, which makes it unsuitable compared to no tree construction.

In all cases constructing no lower tree performed best, and only when the bit masks allowed the lower tree to associate up to 64 triangles per leaf was SAH and SSAH able to produce trees of higher quality than a solution not producing any lower trees. However, the cost of producing these trees were to high to be of any practical use in a dynamic scene.

[1] `http://developer.download.nvidia.com/compute/cuda/3_2_prod/sdk/docs/CUDA_Occupancy_Calculator.xls`

Bit Mask:	*Splitting Scheme:*	C_{trav}/C_i :	Reflecting Dragon	Sponza
32bit	None	N/A	**121/357ms (48.1k)**	**219/179ms (120k)**
	SAH	24	273/358ms (48.2k)	616/180ms (121k)
		16	273/359ms (48.3k)	648/180ms (129k)
		8	280/358ms (50k)	775/181ms (167k)
	SSAH	24	159/382ms (126k)	305/201ms (287k)
		16	162/393ms (143k)	309/207ms (313k)
		8	165/401ms (168k)	318/227ms (394k)
64bit	None	N/A	**98/464ms (19k)**	**160/193ms (38k)**
	SAH	40	1113/462ms (20k)	2393/194ms (41k)
		32	1113/463ms (20k)	2530/193ms (43k)
		24	1126/467ms (20k)	2816/190ms (48k)
	SSAH	40	326/416ms (92k)	609/200ms (162k)
		32	328/423ms (102k)	613/204ms (177k)
		24	331/426ms (110k)	616/206ms (191k)

Figure 5.4: Like in figure 5.3, the entries in the table show the time it took to create the entire kd-tree in milliseconds, the time it took to render the scene and the number in parenthesis is the number of nodes in the kd-tree. To create the upper part of the tree the best configuration from section 5.2 was used and the overall best ray tracer from section 5.1 was used to evaluate the quality of the tree. As in figure 5.2 the boldfaced text denotes the fastest configuration and gray the slowest.

Chapter 6

Conclusion

The best thing about a boolean is even if you are wrong, you are only off by a bit.
Anonymous

In this thesis I have presented several different algorithms for constructing kd-trees and compared their construction speed and their quality, using an optimized hierarchical ray tracer and applying the kd-tree constructors to dynamic scenes.

In chapter 5 I tested an exhaustive ray tracer and compared it against hierarchical ray tracer solutions. The result was that the exhaustive ray tracer did not scale nearly as well with scene complexity as the hierarchical approaches and I can therefore conclude that acceleration structures should still be used when ray tracing dynamic scenes.

Both Packets and Leaf Skipping proved to be reliable optimizations that only provided a small overhead in the very simple Cornell Box scene, but otherwise sped up ray tracing. The short-stack optimization did not provide such a stable increase in speed. Even for the fairly complex Reflecting Dragon scene, the overhead from maintaining a stack was still too much to provide any performance boost. However in the Sponza scene, where the tree was twice as large compared to the dragon scene, adding a short-stack allowed the rays to avoid re-traversing large parts of the tree and thus improve their overall performance. I can therefore conclude that while the short-stack optimization may not always provide a decreased rendering time, using it allows the ray tracer to perform better in complex scenes.

I have during this thesis presented four different algorithms for splitting the nodes in a kd-tree and two schemes for associating triangles with the resulting child nodes.

During the the upper tree creation phase I have shown that using Empty Space Maximization adds very little overhead to the tree creation phase, but significantly improves the quality of the tree. I have also shown that while kd-trees produced with Box Inclusion are larger and slower to ray trace compared with the Triangle/Node Overlap scheme, they can be produced much faster and therefore decreases the overall time spend ray tracing the scene.

In the lower tree creation phase I was surprised to see that applying the Surface Area Heuristic during tree construction yielded trees of roughly the same quality as simply ending tree construction. It was a welcome surprise though as trees created with SAH had a very high construction time, which is an undesireable property for kd-tree constructors working on dynamic scenes. My proposed Simplified Surface Area Heuristic was not a success. While it was faster than SAH, the sum of its construction time and ray tracing time was slower than the combined construction and ray tracing time of simply producing no lower tree.

Generally, I have shown that when creating kd-trees to accelerate ray tracing of dynamic scenes, one should always consider the tradeoff between constructing a kd-tree of high quality or quickly constructing a lower quality acceleration structure.

Chapter 7

Future Work

Software is like entropy: It is difficult to grasp, weighs nothing, and obeys the Second Law of Thermodynamics; i.e., it always increases.
Norman Augustine

While I am quite content with the performance of the kd-trees and ray tracers produced as part of this thesis, there is still room for optimizations.

First of all the layout of the individual nodes can be optimized. As stated in section 3.1.1 my nodes contain a pointer to each of its children, which provided me with the flexibility I needed to inject nodes produced by Empty Space Maximization into the tree. The standard layout of nodes in a kd-tree is to place sibling nodes next to each other, so they can both be referenced by one pointer. Compared to the standard layout, the interior nodes of my kd-tree therefore take up four extra bytes per node, which must be loaded each time a ray traverses a node. It is therefore quite possible that converting my current kd-tree constructor to work with the standard layout would improve ray tracing performance, since less data would have to be fetched from global memory per node visited.

In figure 5.3 we saw that the Triangle/Node Overlap association scheme produced kd-trees of superior quality compared with Box Inclusion. Therefore improving the execution time of Triangle/Node Overlap to produce trees at a speed comparable to Box Inclusion would be worthwile. This might be done by utilizing the knowledge we have about the splitting axis. When splitting an interior node by a splitting plane, the resulting childrens bounding box will consist of five of the same sides as their parent node's bounding box. This knowledge can be utilized to skip some of the intersection tests described in [2] and thus decrease construction time. It would be implemented by a switch with a case for each axis and since all triangles in a segment belong to the same node, all threads in a warp would follow the same execution path. Unfortunately my preliminary attempts at implementing this optimization has been hindered by the CUDA compiler, which seem to inline the switch in order to avoid branching and thus creates a large overhead.

Finally, I might be able to speed up Empty Space Maximization by caching the created nodes in a list in shared memory and then writting that list coalesced to global memory once every thread in a warp is done creating nodes. Since the time spend creating those nodes is already relative low, 4-6ms in the test scenes, this optimization has a very low priority.

Bibliography

[1] Timo Aila and Samuli Laine, *Understanding the efficiency of ray traversal on gpus*, Proceedings of the Conference on High Performance Graphics 2009 (New York, NY, USA), HPG '09, ACM, 2009, pp. 145–149.

[2] Tomas Akenine-Möller, *Fast 3d triangle-box overlap testing*, ACM SIGGRAPH 2005 Courses (New York, NY, USA), SIGGRAPH '05, ACM, 2005.

[3] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman, *Real-time rendering*, 2nd ed., A. K. Peters, Ltd., Natick, MA, USA, 2002.

[4] ———, *Real-time rendering 3rd edition*, A. K. Peters, Ltd., Natick, MA, USA, 2008.

[5] Tomas Akenine-Möller and Ben Trumbore, *Fast, minimum storage ray-triangle intersection*, journal of graphics, gpu, and game tools **2** (1997), no. 1, 21–28.

[6] Arthur Appel, *Some techniques for shading machine renderings of solids*, Proceedings of the April 30–May 2, 1968, spring joint computer conference (New York, NY, USA), AFIPS '68 (Spring), ACM, 1968, pp. 37–45.

[7] James H. Clark, *Hierarchical geometric models for visible surface algorithms*, Commun. ACM **19** (1976), 547–554.

[8] NVIDIA Corporation, *NVIDIA CUDA C Best Practice Guide*, 3.1 ed., 2010.

[9] ———, *NVIDIA CUDA C Programming Guide*, 3.2 ed., 2010.

[10] Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata, *Artsccelerated ray-tracing system*, IEEE Comput. Graph. Appl. **6** (1986), 16–26.

[11] Jeffrey Goldsmith and John Salmon, *Automatic creation of object hierarchies for ray tracing*, IEEE Comput. Graph. Appl. **7** (1987), 14–20.

[12] Vlastimil Havran, *Heuristic ray shooting algorithms*, Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

[13] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan, *Interactive k-d tree gpu raytracing*, I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games (New York, NY, USA), ACM, 2007, pp. 167–174.

[14] M. R. Kaplan, *The uses of spatial coherence in ray tracing*, (1985).

[15] David J. MacDonald and Kellogg S. Booth, *Heuristics for ray tracing using space subdivision*, Vis. Comput. **6** (1990), no. 3, 153–166.

[16] Matt Pharr and Greg Humphreys, *Physically based rendering, second edition: From theory to implementation*, 2nd ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[17] Stefan Popov, Johannes Günther, Hans-Peter Seidel, and Philipp Slusallek, *Stackless kd-tree traversal for high performance GPU ray tracing*, Computer Graphics Forum **26** (2007), no. 3, 415–424, (Proceedings of Eurographics).

[18] Timothy J. Purcell, Craig Donner, Mike Cammarano, Henrik Wann Jensen, and Pat Hanrahan, *Photon mapping on programmable graphics hardware*, HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (Aire-la-Ville, Switzerland, Switzerland), Eurographics Association, 2003, pp. 41–50.

[19] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens, *Scan primitives for gpu computing*, Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware (Aire-la-Ville, Switzerland, Switzerland), Eurographics Association, 2007, pp. 97–106.

[20] Ingo Wald, Carsten Benthin, Markus Wagner, and Philipp Slusallek, *Interactive rendering with coherent ray tracing*, Computer Graphics Forum (Proceedings of EUROGRAPHICS 2001 (Alan Chalmers and Theresa-Marie Rhyne, eds.), vol. 20, Blackwell Publishers, Oxford, 2001, available at http://graphics.cs.uni-sb.de/ wald/Publications, pp. 153–164.

[21] Ingo Wald, Johannes Günther, and Philipp Slusallek, *Balancing Considered Harmful – Faster Photon Mapping using the Voxel Volume Heuristic*, Computer Graphics Forum **22** (2004), no. 3, (Proceedings of Eurographics, to appear).

[22] Ingo Wald and Vlastimil Havran, *On building fast kd-trees for ray tracing, and on doing that in O(N log N)*, Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, 2006, pp. 61–69.

[23] Turner Whitted, *An improved illumination model for shaded display*, SIGGRAPH Comput. Graph. **13** (1979), 14–.

[24] Sven Woop, *A Ray Tracing Hardware Architecture for Dynamic Scenes*, Tech. report, Saarland University, 2004.

[25] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo, *Real-time kd-tree construction on graphics hardware*, ACM Trans. Graph. **27** (2008), no. 5, 1–11.

List of Algorithms

List of Figures

Appendix A

Obtaining the Project

A short demo of the project can be found on `http://www.youtube.com/watch?v=h9gecC5woxs` or the root of the accompanying CD.

The source code for the application developed as part of this master's thesis is also located on the accompanying CD in the folder RayTracingDynamicScenes. The application has been developed in the open source 3D engine framework OpenEngine and I have therefore not implemented everything found in RayTracingDynamicScenes. The implementation of my ray tracer can be found under the project/RayTracing folder, which contains the scene setup, and under extensions/PhotonMapping, where the code for construction kd-trees and ray tracing them is located. The project has been compiled and tested on Max OS X and Ubuntu.

If the CD is not present, then the project can be obtained by installing OpenEngine and then installing the RayTracing project. OpenEngine requires Python, CMake, Boost and the version control system Darcs to be installed on the system. My project additionally requires that SDL, CUDA 3.1 and FreeImage be installed on the system.

To install OpenEngine simply run the following command in a terminal

```
% darcs get http://openengine.dk/code/openengine
```

This will place the OpenEngine source in a folder named *openengine* in the current working directory. cd into that directory and execute the commands

```
% chmod +x dist.py
% chmod +x make.py
```

The project and its dependencies can then be installed with the command

```
% ./dist.py install proj:RayTracing
```

The project can be compiled by running

```
% ./make.py
```

and executed with the command

```
% ./build/RayTracing/RayTracing
```