Algorithms and Data Structures for Interactive Ray Tracing on Commodity Hardware

Stefan Popov

Computer Graphics Group
Saarland University
66123 Saarbrücken, Germany

Dissertation zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

UNIVERSITÄT
DES
SAARLANDES

Betreuender Hochschullehrer / Supervisor:
Prof. Dr.-Ing. Philipp Slusallek, Universität des Saarlandes, Saarbrücken, Germany

Gutachter / Reviewers:
Prof. Dr.-Ing. Philipp Slusallek, Universität des Saarlandes, Saarbrücken, Germany
Dr. Habil. Karol Myszkowski, MPI Informatik, Saarbrücken, Germany
Prof. Dr. Kun Zhou, Zhejiang University, Hangzhou, P. R. China

Dekan / Dean:
Prof. Dr. Mark Groves, Universität des Saarlandes, Saarbrücken, Germany

Eingereicht am / Thesis submitted:
13. Juli 2012 / July 13th, 2012

Datum des Kolloquiums / Date of defense:
18. September 2012 / September 18th, 2012

Prüfungskommission / Committee:
Vorsitzender / Chair:
Prof. Dr. Sebastian Hack, Universität des Saarlandes, Germany

Prüfer / Examiners:
Prof. Dr.-Ing. Philipp Slusallek, Universität des Saarlandes, Saarbrücken, Germany
Dr. Habil. Karol Myszkowski, MPI Informatik, Saarbrücken, Germany

Protokoll / Reporter:
Dr.-Ing. Tobias Ritschel, MPI Informatik, Saarbrücken, Germany

Stefan Popov
Lehrstuhl für Computergraphik, Universität des Saarlandes
Campus E 1 1
66123 Saarbrücken
Germany
popov@cg.uni-saarland.de

Abstract

Rendering methods based on ray tracing provide high image realism, but have been historically regarded as offline only. This has changed in the past decade, due to significant advances in the construction and traversal performance of acceleration structures and the efficient use of data-parallel processing. Today, all major graphics companies offer real-time ray tracing solutions. The following work has contributed to this development with some key insights.

We first address the limited support of dynamic scenes in previous work, by proposing two new parallel-friendly construction algorithms for KD-trees and BVHs. By approximating the cost function, we accelerate construction by up to an order of magnitude (especially for BVHs), at the expense of only tiny degradation to traversal performance.

For the static portions of the scene, we also address the topic of creating the "perfect" acceleration structure. We develop a polynomial time non-greedy BVH construction algorithm. We then modify it to produce a new type of acceleration structure that inherits both the high performance of KD-trees and the small size of BVHs.

Finally, we focus on bringing real-time ray tracing to commodity desktop computers. We develop several new KD-tree and BVH traversal algorithms specifically tailored for the GPU. With them, we show for the first time that GPU ray tracing is indeed feasible, and it can outperform CPU ray tracing by almost an order of magnitude, even on large CAD models.

Kurzfassung

Ray-Tracing basierte Bildsynthese-Verfahren bieten einen hohen Grad an Realismus, wurden allerdings in der Vergangenheit ausschließlich als nicht echtzeitfähig betrachtet. Dies hat sich innerhalb des letzten Jahrzehnts geändert durch signifikante Fortschritte sowohl im Bereich der Erstellung und Traversierung von Beschleunigungs-Strukturen, als auch im effizienten Einsatz paralleler Berechnung. Heute bieten alle großen Grafik-Firmen Echtzeit-Ray-Tracing Lösungen an. Die vorliegende Dissertation behandelt Beträge zu dieser Entwicklung in mehreren Kernaspekten.

Der erste Teil beschäftigt sich mit der eingeschränkten Unterstützung von dynamischen Szenen in bisherigen Verfahren. Hierbei behandeln wir zwei zur Parallelisierung geeignete Algorithmen zur Erstellung von KD-Bäumen und Bounding-Volume-Hierarchien. Durch Approximation von Kosten-Funktionen kann eine Verbesserung der Konstruktionszeit von bis zu einer Größenordnung erreicht werden (speziell für BVH-Strukturen), bei nur geringem Verlust von Traversierungs-Effizienz.

Mit Blick auf den statischen Teil einer Szene beschäftigen wir uns mit der Erstellung "perfekter" Beschleunigungs-Strukturen. Wir entwickeln einen Algorithmus zur BVH-Erstellung, der ein globales Optimum in polynomialer Zeit liefert. Dies führt zu einer neuartigen Beschleunigungs-Struktur, welche sowohl die hohe Leistung von KD-Bäumen, als auch den geringen Platzbedarf von BVH-Strukturen in sich vereinigt.

Abschließend betrachten wir Echtzeit-Ray-Tracing auf Desktop-Computern. Wir entwickeln neuartige KD-Baum- und BVH-Traversierungs-Algorithmen, die speziell auf den Einsatz von Grafikprozessoren zugeschnitten sind. Wir zeigen damit zum ersten Mal, dass GPU-Ray-Tracing nicht nur praktikabel ist, sondern auch mehr als eine Größenordnung effizienter sein kann als CPU basierte Ray-Tracing-Verfahren, selbst bei der Darstellung großer CAD Modelle.

Contents

Chapter 1

Introduction

Over the past few decades, computer graphics has become ubiquitous. We encounter its products on a daily basis, on television, billboards, magazines and even on our phones. We encounter it indirectly even much more, since almost any modern human-made item we come into contact with, is usually designed with the help of CAD software. That said, it is not surprising that the main driving forces behind the progress of computer graphics are the movie and entertainment industry, the advertisement industry, and the CAD industry.

One of the main tasks in computer graphics is the digital synthesis (a.k.a. *rendering*) of 2D images from a 3D virtual scene description. Based on its use, rendering can be divided into off-line and interactive. In the first case images are rendered with a very high and typically photo-realistic quality and the process can take minutes, hours or even days. In the second – quality is sacrificed in the name of interactive performance and an image is typically rendered in a fraction of a second. To achieve this, many light transport effects are approximated, faked, or often completely omitted.

The first case assumes that the image is rendered once and then viewed without changes multiple times. It is typically used to render images for printed media or frames in a movie. To achieve photo-realism, images are rendered by simulating the propagation of light in the virtual scene, according to the laws of physic. In most cases, the largest performance bottleneck of these simulations is the *ray tracing operator*, whose job is to find the first point intersected by a given ray.

Interactive rendering on the other hand assumes that the user can alter the scene and/or the viewpoint and can see the effects of this interaction instantly. Typical applications for this case include CAD software and games. Interactive applications today mostly rely on an algorithm known as *rasterization*, running on extremely fast specialized graphics chips (known as *GPU*s). Due to its limitations however, rasterization can only handle efficiently a small set of light interactions. Even though many applications based on rasterization look stunningly good today (especially games), this is usually because they fake the physically based effects or they pre-compute them offline (if the effect allows it). In turn this often requires significant manual effort by the scene designers. And even then, efficient simulation of effects as simple as real refractions and reflections from glass objects remains beyond the reach of rasterization.

In the years preceding the work in this thesis, ray tracing algorithms and hardware

had evolved to a point where interactive rendering became feasible [Wald, 2004; Benthin, 2006]. Nevertheless, even the most efficient ray tracers at that time required clusters of computers to show non-trivial light transport effects (e.g. refraction), which were not already possible with rasterization [Benthin, 2006]. Furthermore, they were mainly limited to static scenes and scenes with rigid body motion, eventually allowing local deformations in some cases (e.g. for skinning). The reason behind this limitation were the large times required to construct an efficient acceleration structure, which in turn is essential for achieving interactive performance with ray tracing.

The primary goal of the research in this thesis was to bring interactive algorithms based on ray tracing to the commodity desktop computer and in general to make ray tracing a viable alternative to rasterization. To this end we address three problems here: fast construction of acceleration structures, needed to support dynamic scenes; construction of optimal acceleration structures, with the aim to maximize the ray tracing performance in the static parts of the scene; and finally GPU ray tracing, where we aim to develop new ray tracing algorithms that make efficient use of the most powerful processor in today's computers – the GPU.

1.1 Thesis Outline

This thesis is structured into two independent parts, dedicated to our contribution to acceleration structure construction and GPU ray tracing respectively. They are preceded by a common background chapter, which introduces the basics of ray tracing (Chapter 2).

Part I starts with a detailed discussion of the previous KD-tree and BVH construction algorithms (in Chapter 3). Then, in Chapter 4 we show our contribution to fast construction of KD-trees. We present there a new algorithm based on cost function approximation, which greatly accelerates KD-tree construction, at the cost of only few percent slower traversal (following our paper [Popov et al., 2006]). The structure of this algorithm makes it also very friendly to parallel and lazy construction, which in turn gives large potential for future optimizations. Furthermore, we present a previously unpublished extension of this algorithm, which augments our algorithm with ideas from [Hunt et al., 2006]. In Chapter 5, we present our paper [Günther et al., 2007], which applies the same idea to BVHs. There, we also show that it performs even better for BVHs, accelerating their construction by up to an order of magnitude. Finally, in Chapter 6, we propose a non-greedy construction algorithm for BVHs. By studying its results, we develop a new acceleration structure and a new construction algorithm for it, which accelerates traversal up to 6 times, especially for non-axis aligned scene geometry. There, we also identify a hidden requirement of the SAH cost function and we propose how to "fix" the cost, when partitioning is not enforced by the construction algorithm. The research from this chapter was first published in our paper [Popov et al., 2009].

In Part II we show our contribution to interactive GPU ray tracing. We again start with an overview (Chapter 7), discussing the previous CPU traversal algorithms and the limitations of the GPU hardware. Next, in Chapter 8 we develop two new KD-tree traversal algorithms specifically tailored for the GPU, which rely on ropes to

avoid the traversal stack. The use of a traversal optimized acceleration structure, allowed our algorithm to increase the performance of GPU ray tracing 30 times, when compared to previous work. Furthermore, with this work we showed for the first time that GPU ray tracing is not only feasible, but can also outperform CPU ray tracing considerably. In the next chapter (Chapter 9), we develop a new shared-stack BVH packet traversal algorithm, which pushes the performance of GPU ray tracing even further. Also, since BVHs are smaller in size and since our algorithm requires no additional data (such as ropes), our implementation also pushed the limit of supported scene sizes for GPU ray tracing: We were able to achieve interactive frame rates for a scene with 40 times more primitives than anything shown before on the GPU. The research from chapters 8 and 9 was first published in our papers [Popov et al., 2007] and [Günther et al., 2007] respectively.

Finally, in Chapter 10 we summarize the research presented in this thesis, we discuss what lasting impact it has on today's state-of-the-art interactive ray-tracing, and we also show its evolution in follow-up work up until today.

Chapter 2

Background

In this chapter we give the necessary background for the rest of the thesis. To show the importance of the ray tracing operator and to motivate the need of acceleration structures, we start by briefly covering the topics of light transport and rendering algorithms in general. Then, we discuss Whitted style ray tracing in more detail, as it is the fastest ray tracing based rendering algorithm and the first one to achieve real-time performance. We also discuss rasterization and its limitations, as it is the preferred rendering algorithm in most of today's interactive applications. Finally, we focus on acceleration structures in general and their relation to interactive rendering. The latter two are in essence the topics that this thesis deals with.

2.1 Light Transport

Light interaction and distribution in a scene is described by the light transport theory. We will give a very brief introduction of the latter in this section. If the reader is interested in more in-depth details, we recommend reading [Dutre et al., 2006], [Veach, 1998], or [Pharr and Humphreys, 2004].

2.1.1 Radiometric Quantities

The fundamental radiometric quantity in light transport theory is radiant power also called radiant flux. It expresses how much electromagnetic radiation (including visible light) passes through a volume per unit time. Flux is measured in Watts and is usually denoted with Φ.

There are several important quantities in computer graphics which are derived from radiant power, including irradiance $E(x) = d\Phi/dA$, which gives the incident radiant power per unit surface area, radiosity $B(x) = d\Phi/dA$, which gives the exitant radiant power per unit surface area, and radiance $L(x, \omega) = \frac{d^2\Phi}{dwdA^\perp}$, which specifies the flux per unit solid angle per unit projected area. The last one is also the most important one, as it captures the "appearance" of objects in a scene.

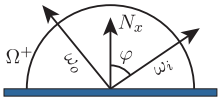
Radiance is a five dimensional function that varies with position x and direction ω. Since projected area can be expressed using surface area ($dA^\perp = \cos\varphi dA$), radiance becomes

$$L = \frac{d^2\Phi}{d\omega dA\cos\varphi}$$

where φ is the angle between $d\omega$ and the surface normal N_x at x. It can be proven that radiance remains unchanged along a straight line in vacuum [Dutre et al., 2006, p. 23].

2.1.2 The Rendering Equation

Light distribution in a scene is commonly modeled by using the rendering equation [Kajiya, 1986]. For a given point x on some surface and a given viewing (outgoing) direction ω_o, the outgoing radiance $L(x \to \omega_o)$ from x along ω_o can be expressed as:



$$L(x \to \omega_o) = L_e(x \to \omega_o) + \int_{\Omega^+} L(x \leftarrow \omega_i) f_r(x, \omega_i \to \omega_o) \cos\varphi \; d\omega_i \tag{2.1}$$

The equation states that the radiant energy flowing from x along w_o is equal to the energy $L_e(x \to \omega_o)$, emitted by the surface at x in direction ω_o, plus the incoming radiance from the whole hemisphere Ω^+ above x, weighted by the spatially varying bi-directional reflectance distribution function (a.k.a. BRDF) $f_r(x, \omega_i \to \omega_o)$ and the cosine of the angle φ between the geometric normal N_x at x and the incoming light direction ω_i. We use the notation $x \leftarrow \omega$ to denote incoming energy at x from direction ω and $x \to \omega$ to denote outgoing.

The rendering equation "works" locally on a point. It assumes that energy is either absorbed, emitted or reflected by a surface point. As a consequence, it can not model certain light phenomena, such as sub-surface scattering and diffraction. Even though more advanced models that can also capture such cases are available (e.g. [Jensen et al., 2001]), many rendering applications today still rely only on the rendering equation (especially the interactive ones). In this thesis we will only look at illumination algorithms based on the rendering equation.

A natural way of extending the rendering equation, is to integrate over the whole sphere of directions Ω around a point x [Veach, 1998, Section 3.6]. This formulation is known as the scattering equation and allows semi transparent materials, such as glass, to be simulated. The function $f_r(.)$ in this equation is known as the bi-directional scattering distribution function (BSDF).

To account for color in the general case, the functions L and L_E must be additionally parametrized by the wavelength λ of light. In this case the BSDF becomes a parametric linear operator over $L(\lambda) = L(x \leftarrow \omega_i, \lambda)$. In practice, the spectrum of light is usually represented as a weighted sum of delta functions and the functions L and L_E operate on vectors. Thus, the result of f_r is a matrix. For efficiency reasons, most rendering systems operate on only three wavelengths: red, green, and blue, which map directly to the RGB values of the pixels in a monitor. Furthermore, these systems assume that f_r operates on the different wavelengths independently, and thus the matrix of f_r is diagonal. The latter is true for most materials encountered in nature.

2.1.3 The Ray Tracing Operator

The rendering equation specifies how light interacts with surfaces, but leaves out the detail of how incoming and outgoing radiance relate. Due to the property of radiance being constant along a ray, in vacuum and "thin" air the relation is given by:

$$L(x \leftarrow \omega_i) = L(h(x, \omega_i) \rightarrow -\omega_i) \tag{2.2}$$

The ray tracing operator $h(x, \omega)$ in this equation returns the first intersection point of the ray with origin x and direction ω with the surfaces of the scene.

A close relative to the ray tracing operator is the binary visibility operator $V(x, y)$ which determines whether two points x or y are mutually visible. The visibility operator can be expressed as

$$V(x,y) = \begin{cases} 1 & \text{, if } \left| h(x, \frac{y-x}{|y-x|}) - x \right| \geq |y - x| \\ 0 & \text{, otherwise} \end{cases} \tag{2.3}$$

The rendering equation presented in (2.1) integrates over the incoming directions of light. Thus (2.1) is known as the directional formulation of the rendering equation. Using the visibility operator, the rendering equation can also be expressed as integration over the surfaces of the scene (a.k.a. the area formulation). The solid angle $d\omega$ as seen from the point x and subtended by a surface patch dA_y around a point y can be expressed as

$$d\omega = \cos(N_y, -\Psi)\frac{dA_y}{(x-y)^2}$$

with $\Psi = \frac{y-x}{|y-x|}$. The rendering equation written using the area formulation then becomes:

$$\begin{aligned} L(x \rightarrow \omega_o) &= L_e(x \rightarrow \omega_o) + \\ &\int_{\Omega^+} L(y \rightarrow \Psi) f_r(x, \Psi \rightarrow \omega_o) V(x,y) G(x,y) dA_y \end{aligned} \tag{2.4}$$

Here, the term $G(x, y)$ is known as the geometric term and is given by

$$G(x,y) = \frac{\cos(N_x, \Psi)\cos(N_y, -\Psi)}{(x-y)^2}$$

Note that the two forms of the rendering equation (2.1) and (2.4) are equivalent only if L_E, which is defined over the surfaces of the scene, is the only source of radiant energy. The directional form is more general as it can can also handle non-surface light sources. However, as we will see in Section 2.2.1, the two forms can be used together when calculating illumination.

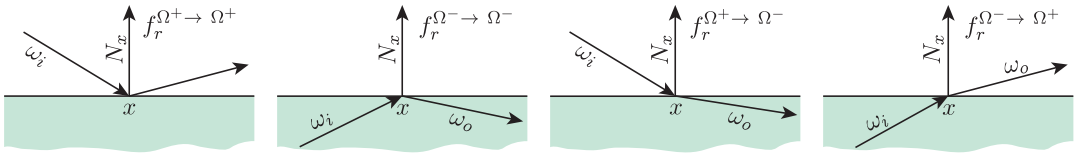


Figure 2.1: *Decomposition of f_r into $f_r^{\Omega^+\to\Omega^+}$, $f_r^{\Omega^-\to\Omega^-}$, $f_r^{\Omega^+\to\Omega^-}$, and $f_r^{\Omega^-\to\Omega+}$. The support of the base functions is disjoint.*

2.1.4 The Bi-Directional Scattering Distribution Function

The function $f_r(x, \omega_i \to \omega_o)$ defines the appearance of the objects in a scene. It is known as the bi-directional scattering distribution function (BSDF) when integrating over the whole sphere of directions and as the bi-directional reflectance distribution function (BRDF) when assuming opaque materials and integrating over the upper hemi-sphere only.

The BSDF is in general a 6 dimensional function (because x lies on a 2D surface). If Ω^- is the lower hemisphere (below x with respect to the normal N_x) then a BSDF can be decomposed into (see Figure 2.1):

$$f_r(x, \omega_i \to \omega_o) = f_r^{\Omega^+\to\Omega^+} + f_r^{\Omega^-\to\Omega^-} + f_r^{\Omega^+\to\Omega^-} + f_r^{\Omega^-\to\Omega^+}$$

where

$$f_r^{\Omega^1\to\Omega^2}(x, \omega_i \to \omega_o) = \begin{cases} f_r(x, \omega_i \to \omega_o) & \text{, if } \omega_i \in \Omega^1 \wedge \omega_o \in \Omega^2 \\ 0 & \text{, otherwise} \end{cases}$$

The support of the four functions is disjoint. The first two functions, namely $f_r^{\Omega^+\to\Omega^+}$ and $f_r^{\Omega^-\to\Omega^-}$, describe the reflectance properties of the outside and inside surfaces respectively. Thus, they can be viewed as outer and inner BRDFs respectively. Which surface is inside and which outside is determined by the normal, which is usually assumed to point to the outside.

The second pair of functions ($f_r^{\Omega^+\to\Omega^-}$ and $f_r^{\Omega^-\to\Omega^+}$) describe the transmittance of light respectively from outside to inside and vice versa. They are known as the bidirectional transmittance distribution functions (BTDFs). In practice, the majority of materials are opaque. For such materials, only the outside BRDF is non zero.

BSDFs that describe physically based materials conserve energy

$$\int_{\Omega+} f_r(x, \omega_i \to \omega_o)\, d\omega_o \leq 1$$

and physically based BRDFs are symmetric (i.e. they obey the Helmholtz reciprocity principle [Helmholtz, 1867])

$$f_r(x, \omega_i \to \omega_o) = f_r(x, \omega_o \to \omega_i)$$

Note that the latter is unique to reflection only and does not apply to the BTDF part of the BSDF in general. Further details on BSDFs can be found in [Veach, 1998].

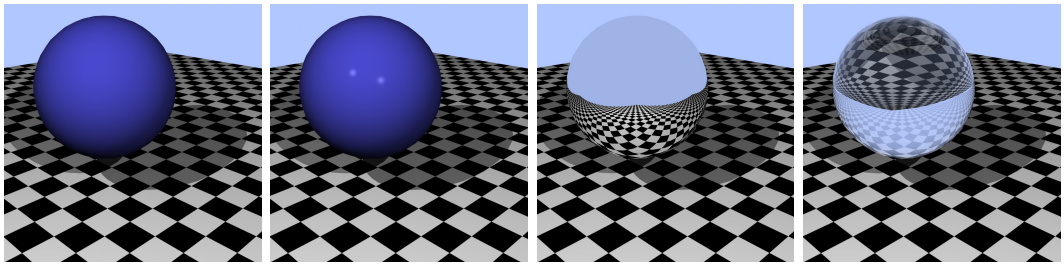


Figure 2.2: *Four different types of BSDF: Lambertian diffuse, Blinn-Phong, perfect mirror, and glass.*

The ray tracing applications presented in this thesis rely on four basic BSDF models (Figure 2.2), namely diffuse (a.k.a. Lambertian), Blinn-Phong, glass, and mirror. They are among the most widely used models in interactive rendering. Other popular models include Cook-Torrance [Cook and Torrance, 1982], Ward [Ward, 1992], and Lafortune[Lafortune et al., 1997].We will also use combinations of the above materials, to achieve effects such as a reflecting table top for example (by combining mirror and Blinn-Phong).

The mirror and glass materials belong to a special class known as specular materials. The latter includes all materials whose BSDF can be represented as a sum of delta functions. For a fixed x and ω_o, such materials have only a finite set of directions that can contribute to the outgoing energy. This set is encoded in the arguments of the delta functions, and is used by some rendering algorithms to optimize performance.

Diffuse Surfaces

Lambertian/diffuse surfaces are opaque diffuse surfaces that scatter light uniformly. They are characterized by an absorption coefficient $\rho \in [0, 1]$, which specifies how much of the total incoming radiant energy is absorbed by the surface. The BRDF of a Lambertian surface is constant, since the incoming radiance from one direction is scattered uniformly. It is given by

$$f_r(x, \omega_i \rightarrow \omega_o) = C_d(x) = \frac{\rho(x)}{2\pi}$$

Dividing by 2π is necessary since ρ is actually the integral over the upper hemisphere of $f_r(.)$ w.r.t. ω_o.

Blinn-Phong Surfaces

The Blinn-Phong model [Blinn, 1977] is an extension to the Lambertian diffuse model, that also accounts for glossy highlights. Again it is an opaque material, and thus only its outside BRDF is non-zero. The Blinn-Phong model is not physically correct, as it does not conserve energy. However, it is simple and fast to compute and gives plausible results. Thus it is the preferred model in many interactive and real-time rendering applications.

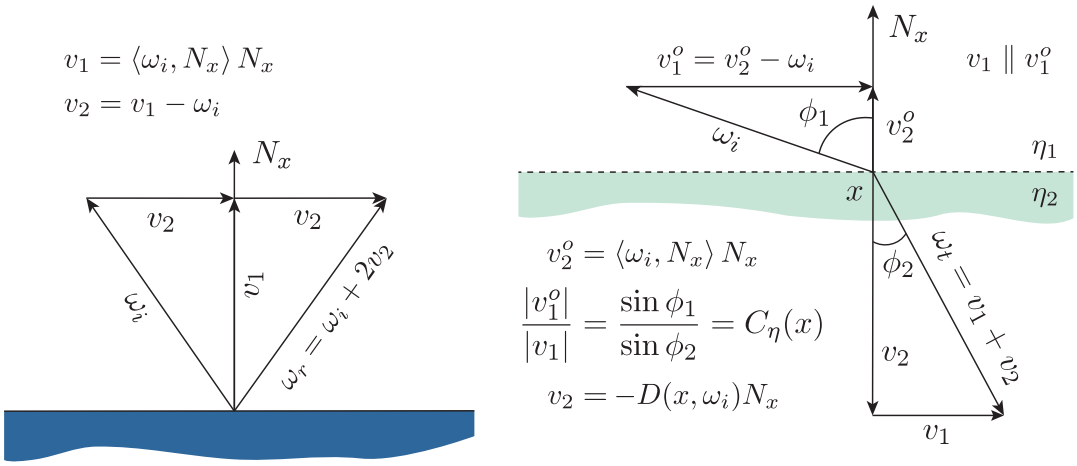


Figure 2.3: *A schematic derivation of the functions $\omega_r = R(x, \omega)$ and $\omega_t = T(x, \omega)$ respectively. Here η_1 and η_2 give the speed of light and $C_\eta(x) = \frac{\eta_2}{\eta_1}$*

The BRDF of the Blinn-Phong model is given by:

$$f_r(x, \omega_i \to \omega_o) = C_d(x) + C_s(x) \left\langle \frac{\omega_i + \omega_o}{|\omega_i + \omega_o|}, N_x \right\rangle^{C_e(x)}$$

As before, C_d specifies the absorption coefficient for the diffuse part. The coefficient C_e specifies how glossy the surface is, whereas C_s specifies the weight of the glossy part in the lighting calculations.

Mirror Surfaces

The mirror is a specular opaque material. Incoming radiance flowing along a direction ω_i is reflected in exactly one direction ω_o, equal to ω_i reflected around the normal N_x. Its BRDF is given by [Veach, 1998, Section 5.2.1.2]:

$$f_r(x, \omega_i \to \omega_o) = \frac{\delta(R(x, \omega_i) - \omega_o)}{\cos(N_x, \omega_i)}$$

where $\delta(.)$ is the Dirac impulse function and $R(x, \omega)$ is a function whose result is ω reflected around the normal N_x. The function expects that ω points away from x and is given by (see Figure 2.3 and [Pharr and Humphreys, 2004, Section 8.2])

$$R(x, \omega) = -\omega + 2 \langle \omega, N_x \rangle N_x$$

The set of directions that can contribute to the outgoing energy for a mirror material has exactly one element: $R(x, \omega_o)$.

Refractive Surfaces

Refractive surfaces, such as glass and water, are specular and semi transparent. Incoming radiance from a direction ω_i is both reflected in direction ω_r and transmitted

through the surface in direction ω_t, with the latter being determined by Snell's law. The fractions μ_r and μ_t of the incoming energy that is reflected/refracted along ω_r/ω_t are determined by Fresnel's equations. Additionally, at shallow incoming angles, total internal reflection can occur in the more dense media. In those cases $\mu_r = 1$ and $\mu_t = 0$.

Refractive surfaces are parametrized by the coefficient C_η giving the ratio of the speed of light inside the medium to its speed outside the medium. An additional parameter $C_\alpha \leq 1$ can be used to control the transparency (to simulate colored glass for example).

As with mirror surfaces, the reflected direction is calculated as $\omega_r = R(x, \omega_i)$. The refracted direction is given by (see [Pharr and Humphreys, 2004, Section 8.2] and Figure 2.3):

$$\omega_t = T(x, \omega_i) = C_\eta(x)\big(-\omega_i + \langle \omega_i, N_x \rangle N_x \big) - \sqrt{1 - \frac{1 - \langle \omega_i, N_x \rangle^2}{C_\eta(x)^2}}\, N_x$$

If we denote the expression under the square root with $D(x, \omega_i)$, then the function $T(.)$ is only defined if $D(x, \omega_i) \geq 0$. In case it is negative, total internal reflection occurs.

If we denote the angle between $-\omega_i$ and N_x with ϕ_1 and the angle between ω_t and $-N_x$ with ϕ_2 then, according to the Fresnel equations for unpolarized light, the transmittance and reflection coefficients are given by:

$$\mu_r(x, \omega_i) = \frac{1}{2}\left[\frac{C_\eta(x)\cos\phi_1 - \cos\phi_2}{C_\eta(x)\cos\phi_1 + \cos\phi_2} + \frac{C_\eta(x)\cos\phi_2 - \cos\phi_1}{C_\eta(x)\cos\phi_2 + \cos\phi_1}\right]$$

$$\mu_t(x, \omega_i) = 1 - \mu_r(x)$$

Thus, the BSDF of a refractive material becomes:

$$f_r(x, \omega_i \to \omega_o) = \begin{cases} \frac{\delta\left(R(x,\omega_i) - \omega_o\right)}{\cos(N_x, \omega_i)} & , D(x, \omega_i) \leq 0 \\ \frac{1}{\cos(N_x, \omega_i)}\Big[\mu_r(x)\ \delta\big(R(x, \omega_i) - \omega_o\big) & \\ \quad + C_\alpha(x)\ \mu_t(x)\ \delta\big(T(x, \omega_i) - \omega_o\big)\Big] & , D(x, \omega_i) > 0 \end{cases}$$

where $\delta(x)$ is again the Dirac impulse function. The set of directions that can contribute to the outgoing energy for a refractive material has either one or two elements. The first element is always $R(x, \omega_o)$ and in case no total internal reflection occurs, the second element is $T(x, \omega_o)$.

2.2 Rendering Algorithms

Rendering is the process of taking a picture on a virtual camera by solving the rendering equation. There are a vast amount of rendering algorithms available, the more popular of which include path tracing [Kajiya, 1986], bidirectional path-tracing

[Lafortune and Willems, 1993; Veach, 1998], radiosity [Goral et al., 1984], and photon mapping [Jensen, 1996]. In this thesis we are going to use Whitted style ray tracing [Whitted, 1980] as it currently presents the optimal trade-off between image quality and interactivity (among all ray-tracing based algorithms).

Besides discussing Whitted style ray tracing, we show in this section, how to define the input of a rendering algorithm (i.e. the scene) and we discuss rasterization. The latter is important, as most current interactive algorithms are built on top of it. Also, it has been the driving force behind the development of graphics hardware in the past decades.

2.2.1 Scene Description

The input data on which rendering algorithms operate is generally known as the scene description. It includes description of the scene's geometry, the materials, the camera parameters, and the light sources.

Geometry Description

There are various ways to describe the input geometry and the means of description also depend on the rendering algorithm itself. Especially for interactive applications, the most common case is to describe the scene through its surface, specified as a set of geometric primitives. This is also the representation that we will assume in this thesis.

In ray tracing, the primitives can be of any type, as long as the application knows how to compute the intersection of a ray with a primitive. For efficiency reasons, most ray tracing algorithms require further that the primitives are bounded and that there exists a way to compute those bounds. This requirement enables the use of an acceleration structure (see Section 2.3). Some acceleration structures require further the ability to clip a surface primitive with a bounding box (e.g. KD-trees). Finally, rendering algorithms can also operate efficiently on unbounded primitives, provided that their count is small.

The preferred surface type for representing geometry in most rendering systems is the triangle as it is easy to transform and manipulate, it can be efficiently intersected, has a well defined and efficient to compute bounding box, and higher order surfaces can be tessellated using triangles. Triangles are specified through their vertices, which beside position can contain arbitrary user defined attributes, such as color and surface normal. During shading, these attributes are linearly interpolated for each point from the triangle, using the barycentric coordinates as weights [Phong, 1975].

Animation

During rendering, the geometry can not change for consistency reasons, but it is free to do so between two frames. This changes are known as animation. They can be encoded in the input itself, but they can also come from other sources, such as user interaction or a physics simulation engine for example.

Rendering algorithms are not interested in the animation itself (as it is static while rendering), but rather in its structure. Knowing the latter, they can sometimes

cache certain scene-related data between frames and accelerate computations. In the context of ray tracing animation can be classified as static geometry, rigid body, skinning (or deformable animation), and fully dynamic. Each of the classes in this list is a superset of all classes that precede it. We will discuss how ray tracing based algorithms exploit the animation structure in Section 2.4.2.

Static geometry contains no animation at all. In rigid body animation, the scene consists of a (usually small) set of rigid bodies (i.e. static geometries), with an affine transformation applied over each of them. Animation is then performed by changing the affine transformations. Rigid body animation can be used for example to animate moving objects and skeletons. Deformable animation builds on top of rigid body animation and allows the triangles within each rigid body to move in a small constrained space. This type of animation can be used for skin or leaf animations. Fully dynamic (a.k.a. unstructured) animation allows arbitrary changes to the scene.

Materials

Besides geometry, the other major factor that determines the appearance of an object are its materials. In a surface based scene description, a material is usually equivalent to a BSDF and it is given as a property of the surface primitive. Conceptually, a material consists of two parts: a parametric model, which is uniform over the surface, and the actual parameters, which define the concrete appearance at a point and can vary arbitrary over the surface.

The parametric model is specified as a set of programs (a.k.a. *shaders* [Apodaca and Mantle, 1990; Pixar, 1989; Hanrahan and Lawson, 1990]), whose purpose is specific to the rendering algorithm. A material commonly provides two functions: for computing the BSDF at a point and for BSDF sampling. The shaders can be built into the rendering application, in which case we speak of *fixed-function shading*, or they can be supplied with the scene description (a.k.a. *programmable shading*).

Material parameters are specified in the scene description. Parameters that are uniform over a surface primitive are usually stored as a property of the latter. Parameters that vary linearly over the surface of a primitive are usually specified as attributes in the vertices of the primitive. Finally, finer control can be achieved by using *textures*, which are two dimensional maps of parameters. The mapping of surface position to texture position is usually specified by storing an attribute in each vertex, that gives the 2D position of the vertex onto the texture (a.k.a. *texture coordinates*).

Camera Description

To render a picture from a scene description, we first define a camera model. Like in the real-world, a virtual camera is used to capture an image of the virtual environment, which can then be displayed on a screen. A virtual camera consists of a sensor array (a.k.a. the image plane) and a lens system.

The sensors measure the radiant energy that falls onto them. They are usually arranged in a rectangular grid. To display an image taken from a virtual camera, it suffices to map the measured values to pixel intensities.

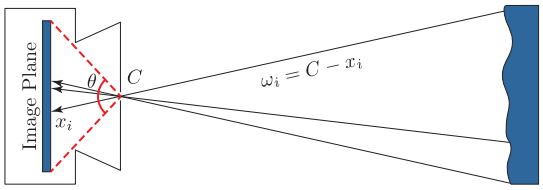


Figure 2.4: *A pinhole camera. Directions ω_i are mapped onto positions x_i on the image plane as light passes through the pinhole C. The vertical opening angle (red) is denoted by θ.*

The lens system provides the mapping between rays entering the camera and positions on the image plane. The simplest and most widely used lens system is the one from the pinhole camera. It consists of an obstacle in front of the image plane with an infinitely small hole (see Figure 2.4). This camera model captures an image as seen from the center C of the pinhole. It provides a one-to-one mapping between the direction of incoming rays at C and the position on the image plane. This is the model we are going to use throughout this thesis.

Rendering is the process of measuring the total radiant power falling on the sensors. For each sensor j of a pinhole camera, this power is given by the measurement equation [Dutre et al., 2006, Section 2.8]:

$$M_j = \int_{A_j} W(x \leftarrow \Psi) L(x \leftarrow \Psi) \cos\theta dx \tag{2.5}$$

where $\Psi = \frac{x-C}{|x-C|}$ and θ is the angle between the vector Ψ and N_x. The response function $W(.)$ is defined over all sensors (they are treated as part of the scene), and gives the sensitivity to radiance on their surfaces. The resulting 2D array of measured values forms the picture taken by the virtual camera.

To achieve color rendering, the measurement equation has to be parametrized by the wavelength λ of light. As in the case of the rendering equation, the measurement function becomes a linear parametric operator that works on L and the value of M_j becomes a function over the wavelength of light. Displaying of rendered images is achieved by convoluting the computed sensor values $M_j(\lambda)$ with the response functions of the display device's pixel channels.

In the common case, the response functions of the display device are assumed to be delta functions located at the wavelengths of red, green, and blue. In this case, the spectrum is approximated during rendering as a weighed sum of these three functions and $W(.)$ is taken to be identity.

A pinhole camera is specified in the scene data through the location of its center C, its orientation in space (i.e. the forward and up directions), its vertical opening angle (see Figure 2.4) and the sensor resolution.

Light Sources

The illumination in a scene is modelled using light sources, which are usually specified in the scene input. Their purpose is to define the emittance function L_e from the

rendering equation, but they are more general than that, as they can also define "detached" illumination that does not originate from a surface.

Each light source l occupies a certain area A_l of either space or directions. In the first case, i.e. *finite* light sources, each point from A_l emits light. In the second case (i.e. *infinite* light sources), light is emitted from infinity along the directions contained in A_l. Light sources are further characterized by their *intensity* function measured in watt per steradian (i.e. $\frac{d\Phi}{d\omega}$). For finite light sources, this function is given as $I_l(x \to \omega)$ with $x \in A_l$ and $\omega \in \Omega$, while for infinite – as $I_l(\omega)$ with $\omega \in A_l$.

2.2.2 Rendering with Non-Surface Light Sources

To account for the fact that light sources are not necessary attached to surfaces of the scene, we first reformulate the rendering equation in its incident radiance form [Dutre et al., 2006, Section 4.1.3]:

$$L(x \leftarrow \omega_i) = L_e(x \leftarrow \omega_i) + \int_\Omega L(y \leftarrow \omega) f_r(y, \omega \to -\omega_i) \cos(N_y, \omega) d\omega \tag{2.6}$$

with $y = h(x, \omega_i)$. The term $L_e(x \leftarrow \omega_i)$ represents incident emitted radiance and is equal to $L_e(h(x, \omega_i), -\omega_i)$.

The formulation (2.6) is more powerful than (2.1) as it gives the freedom to incorporate non-surface emittance. To do so, we rewrite (2.6) in its direct illumination formulation [Dutre et al., 2006, Section 2.6.3]:

$$L(x \leftarrow \omega_i) = L_e(x \leftarrow \omega_i) + L_r(x \leftarrow \omega_i) \tag{2.7}$$

$$L_r(x \leftarrow \omega_i) = L_{direct} + L_{indirect} \tag{2.8}$$

$$L_{direct} = \int_\Omega L_e(y \leftarrow \omega) f_r(y, \omega \to -\omega_i) \cos(N_y, \omega) d\omega \tag{2.9}$$

$$L_{indirect} = \int_\Omega L_r(y \leftarrow \omega) f_r(y, \omega \to -\omega_i) \cos(N_y, \omega) d\omega \tag{2.10}$$

with $y = h(x, \omega_i)$. According to the description of the intensity function, L_{direct} and $L_e(x \leftarrow \omega_i)$ can be written as:

$$\begin{aligned} L_{direct} = &\sum_{l \in E_P} \int_{A_l} I(\xi_l \to \vartheta) f_r(y, -\vartheta \to -\omega_i) V(y, \xi_l) G(y, \xi_l) d\xi_l \\ &+ \sum_{l \in E_D} \int_{A_l} I(\omega_l) f_r(y, -\omega_l \to -\omega_i) \cos(N_y, -\omega_l) d\omega_l \end{aligned} \tag{2.11}$$

$$\begin{aligned} L_E(x \leftarrow \omega_i) = &\sum_{l \in E_D} I_l(-\omega_i) V^\infty(x, \omega_i) \\ &+ \sum_{l \in E_P} \int_{A_l} \frac{\delta\left(\frac{x - \xi_l}{|x - \xi_l|} + \omega_i\right) I_l(\xi_l, -\omega_i) \cos(N_{\xi_l}, -\omega_i) V(x, \xi_l)}{(x - \xi)^2} d\xi_l \end{aligned} \tag{2.12}$$

with $\vartheta = \frac{y-\xi_l}{|y-\xi_l|}$. Here E_P denotes the set of finite light sources, E_D – the set of infinite ones, and $V^\infty(x,\omega)$ is a function that has a value 0 if the ray (x,ω) intersects anything and 1 otherwise. We also assume that $I(\omega)$ is defined for any direction and is 0 if $\omega \notin A_l$.

Light Sources Classification

There are several common types of light sources used in today's rendering systems. Surface (or area) lights are finite light sources for which A_l is a 2-manifold. The normal at each point of the light source coincides with the surface normal at that point. Such lights are used to simulate surface emission, such as fluorescent lamps for example. A special case of an area light source is the virtual point light (VPL). It's A_l contains a single point only, it has a fixed normal, and its intensity is a delta function. VPLs are used to approximate surface area lights for algorithms that only support delta light sources.

Volumetric lights are again finite and their A_l is a 3-manifold. While integrating in (2.12), the normal to each point is assumed to coincide with the direction towards x and thus $\cos(x-\xi, N_\xi) = 1$. Volumetric lights are used to simulate volumetric lighting such as fire for example. Point light sources are a special case for which A_l is a point and $I_l(x \to \omega)$ is a delta function. They are used to simulate light from e.g. incandescent light bulbs.

Environment lights are infinite light sources such that their A_l has non-zero measure. They are usually used to simulate distant illumination. Directional lights are a special case of an environment light, where A_l contains a single direction only and $I_l(\omega)$ is a delta function. They are typically used to simulate light coming from the sun.

2.2.3 Whitted Style Ray Tracing

In order to render a picture, one has to solve the measurement equation (2.5) for each sensor of the camera. This also involves solving the rendering equation (2.1). Except in trivial cases, the two equations can not be solved analytically. Instead, rendering algorithms use numerical methods.

Probably the simplest and most efficient rendering algorithm is Whitted style ray tracing [Whitted, 1980]. To render a picture it samples the incoming radiance at the image plane by tracing paths from the camera towards the light sources.

For efficiency reasons, Whitted style ray tracing only works with delta light source (i.e. point and directional). The rationale behind this limitation is to be able to deterministically and efficiently compute the direct illumination at any point of the scene. It can be extended to support arbitrary light sources if required, by using Monte-Carlo integration for example.

Furthermore, Whitted style ray tracing simplifies the lighting model by assuming that any indirect illumination at a point can only come from the specular part of a BSDF. Since the latter is a finite sum of delta distributions, the algorithm can limit its search for incoming energy to the directions encoded in those functions only. This

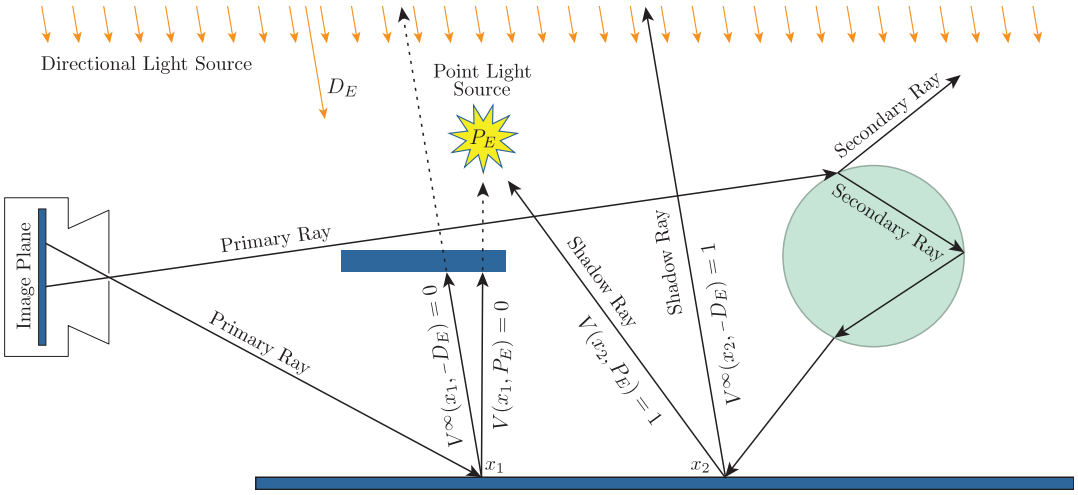


Figure 2.5: *Whitted style ray tracing. The figure shows the types of rays shot during rendering. One of the primary rays intersects a glass sphere and generates secondary rays. The hit point x_2 sees both light sources, where as x_1 sees neither, being occluded by the blue rectangle. Note that not all shot shadow rays are shown on the picture.*

severely limits the directions that Whitted style ray tracing needs to explore and thus leads to a much greater efficiency.

Under these assumptions, and taking into account that the emittance of point and directional light sources is a delta function, equations (2.11), and (2.10) become:

$$\begin{aligned} L_{direct} = &\sum_{l \in E_P} I_l(\xi_l \to \vartheta) f_r(y, -\vartheta \to -\omega_i) V(y, \xi_l) G(y, \xi_l) \\ &+ \sum_{l \in E_D} I_l(\omega_l) V^\infty(x, -\omega_l) f_r(y, -\omega_l \to -\omega_i) \end{aligned} \tag{2.13}$$

$$L_{indirect} = \sum_{\omega \in \Omega(y, -\omega_i)} L_r(y \leftarrow \omega) f_r(y, \omega \to -\omega_i) \cos(\omega, N_y) \tag{2.14}$$

with $\vartheta = y - \xi_l$. Here, ξ_l denotes the position in space of the point light l, ω_l denotes the direction of the directional light l, and $\Omega(x, -\omega_i)$ is the finite set of all directions ω, for which the specular BSDF $f_r(x, \omega \to -\omega_i)$ is not zero.

Equations (2.13) and (2.14) form the core of the ray tracing algorithm (Algorithm 2.1), which computes the incoming radiance for a given point and direction. Rendering an image is then achieved by sampling the sensors of the image plane, computing the incoming radiance at each sample, and summing the obtained values according to the sample weights and the response function. Ray tracing is usually viewed as a backward algorithm, as it actually starts at the image plane and works its way back to the light sources. Thus, the paths connecting a light source to a sample on the image plane are usually depicted as flowing towards the light source (see Figure 2.5).

Algorithm 2.1 The Ray Tracing Algorithm

function $L_{direct}(y, \omega_i)$ ▷ Direct illumination (2.13)
  $ret \leftarrow 0$
  for all $l \in E_P$ **do**
    if $V(y, \xi_l)$ **then**
      $\vartheta \leftarrow y - \xi_l$
      $ret \leftarrow ret + I_l(\xi_l \rightarrow \vartheta) f_r(y, -\vartheta \rightarrow -\omega_i) G(x, \xi_l)$
    end if
  end for
  for all $l \in E_D$ **do**
    if $V\infty(y, -\omega_l)$ **then**
      $ret \leftarrow ret + I_l(\omega_l) f_r(y, -\omega_l \rightarrow -\omega_i) \cos(N_y, -\omega_l)$
    end if
  end for
  return ret
end function

function $L_{indirect}(y, \omega_i)$ ▷ Indirect illumination (2.14)
  if recursion depth > maximum allowed **then**
    return 0
  end if
  $ret \leftarrow 0$
  for all $\omega \in \Omega(y, -\omega_i)$ **do**
    $\tau \leftarrow L_r(y, \omega)$ ▷ Recursion
    $ret \leftarrow ret + \tau f_r(y, \omega \rightarrow -\omega_i) \cos(N_y, \omega)$
  end for
  return ret
end function

function $L_r(x, \omega_i)$ ▷ See (2.7)
  $y \leftarrow h(x, \omega_i)$
  return $L_{direct}(y, \omega_i) + L_{indirect}(y, \omega_i)$
end function

function $L(x, \omega_i)$ ▷ Computes incoming radiance
  return $ret \leftarrow L_r(x, \omega_i)$
    ▷ We ignore L_e, since this function is only used for the measurement equation and the probability of directly hitting a delta light source is 0
end function

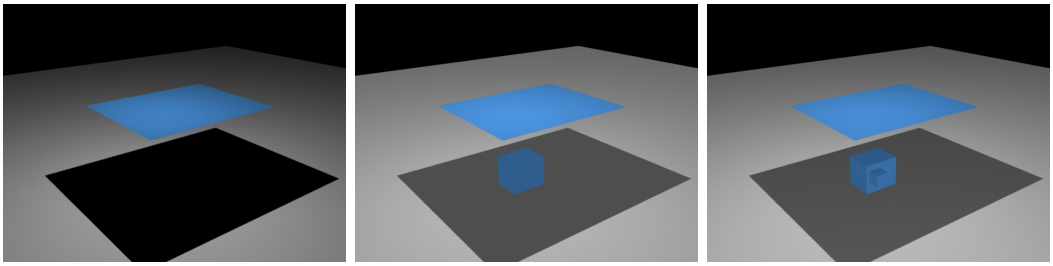


Figure 2.6: *Ambient Illumination. The same scene rendered without ambient illumination (left), with standard ambient illumination (middle) and with our improved version(right). Adding ambient illumination reveals a cube under the plane. Adding eye-light reveals one further cube.*

In the context of Whitted style ray tracing, we distinguish three types of rays. Primary rays are those that enter the camera and hit the image plane. Shadow rays are used to determine the visibility of the light sources. All other rays are known as secondary (see Figure 2.5). Primary and shadow rays can be computed more efficiently than secondary. In the first case, they have a very specific distribution (see Section 2.4). In the second case, we are interested in any intersection and not the closest one, which can be used to optimize the binary visibility function $V(x, y)$

Approximation Quality

Whitted style ray tracing is clearly an approximation. It captures the most important illumination (direct illumination and specular reflections), but ignores the rest and the images it produces tend to look darker. As a workaround, many Whitted style ray-tracing implementations use a technique known as ambient illumination. The latter is a non-physically correct form of uniform illumination present in the whole scene. It is usually defined by its intensity I_a, and through a surface coefficient $C_a(x)$, specifying the ambient reflectivity at x. The contribution $I_a C_a(x)$ of the ambient illumination is then added to the outgoing radiance of a point.

For the ray tracers presented in chapters 8 and 9, we use an improved model for ambient illumination. Objects illuminated with the standard model look "flat" (Figure 2.6) as it does not account for the surface geometry. To avoid this, we further add eye lighting. We define a second ambient illumination constant I'_a and further add $I'_a \ f_r(x, \omega_o \rightarrow \omega_i)$ to the outgoing radiance of the hit point x. This allows us to reveal important geometry details, that would otherwise remain hidden in regions with no direct lighting.

2.2.4 Rasterization

The currently fastest rendering algorithms are not based on ray tracing, but rather on a technique known as *rasterization*. The latter was initially designed as an efficient approach for intersecting a regular batch of primary rays with the scene. Today, it is commonly implemented in dedicated hardware, known as a graphics processing unit (GPU), which is present in all modern computers.

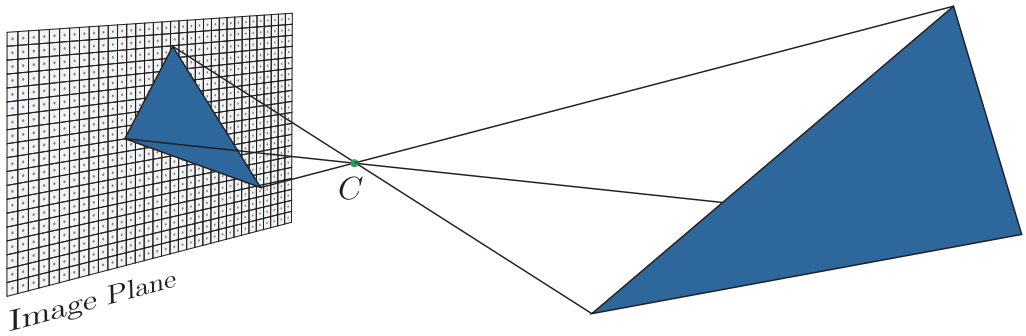


Figure 2.7: *The triangle is projected onto the image plane through C. The primary rays of all samples (with origins depicted as dots) covered by the projection will intersect the triangle.*

Rasterization expects that all rays in the batch originate from a pinhole camera. Thus, it expects that their origins are arranged on a rectangular planar grid (a.k.a. *the image plane*) and all of them pass through a point C (Figure 2.7).

Instead of tracing rays, rasterization works by projecting each triangle onto the image plane, using the pin hole C as a center. If a point on the image plane is covered by a projection, then the ray originating from this point will intersect the triangle. Thus, rasterization loops over all triangles of a scene and for each triangle it enumerates all samples that are covered by its projection. For each covered sample, rasterization executes a small program and stores the result with the sample. We will refer to this value as the *color* of the sample, although its purpose can be arbitrary. The executed program is uniform over all samples and is known as a *fragment program* or *pixel shader*.

To account for occlusion, rasterization also stores the distance from each sample to the closest intersection found so far. If a triangle covers a sample that has been covered by a previous triangle, the depth and color values are only updated if the sample's stored distance is smaller than incoming one. The buffers where those values are stored are known as the depth- or Z-buffer [Catmull, 1974] and the color buffer.

The worst case complexity for rasterizing a scene is $O(MN)$, where N is the number of primitives and M is the number of pixels. In practice however, rasterization is much faster than that. For most reasonable smaller scenes, each pixel is covered by only a few primitives. Furthermore, methods such as hierarchical frustum culling [Clark, 1976], portal culling [Teller, 1992], level-of-detail [Erikson, 2000], and occlusion culling, enable efficient rasterization of huge depth-complex models as well.

2.2.5 Ray Tracing vs. Rasterization

Rasterization is primarily used for rendering. Simple lighting, such as ambient illumination and eye light, is easy to simulate with rasterization, by specifying the proper pixel shader. With the help of algorithms such as shadow mapping [Williams, 1978] and shadow volumes [Crow, 1977], rasterization can render scenes with direct illumination from point lights and directional lights.

Figure 2.8: *The same scene rendered with rasterization (left), Whitted style ray tracing (middle), and photon-mapping (right). Rasterization can not handle refractions and reflections in general, while Whitted style ray tracing can not handle global light interaction, such as the caustics on the rightmost image.*

In the recent years, many techniques have been developed to simulate even more advanced light transport effects with rasterization. Instant radiosity [Keller, 1997] for example, as well as its faster but more inaccurate variant of using imperfect shadow maps [Ritschel et al., 2008], can simulate multi-bounce diffuse inter-reflections. Environment maps have been used to fake specular effects such as reflection and refraction [Watt, 1993, Section 5.6]. Various shadow map techniques, such as percentage closer filtering and variance shadow maps [Donnelly and Lauritzen, 2006], can be used to blur shadow edges, which gives the impression that the scene is illuminated from a small area light source.

Despite those advances however, the types of light transport that can be efficiently simulated using rasterization remains limited. The main issue comes from the fact that rasterization can solve only one very specific type of visibility query and in order to solve it efficiently, the query must contain a very large amount of rays.

Thus, even simple effects such as simulating light interaction with glass, which are more or less trivial for a Whitted style ray tracer, become impractically slow to do with rasterization in the general case. And so do all advanced light transport algorithms such as bi-directional path tracing [Veach, 1998] or photon mapping [Jensen, 2001]. Figure 2.8 illustrates the limits of rasterization, comparing it to Whitted style ray tracing and photon mapping.

2.3 Acceleration Structures Background

The primary mechanism for transferring energy between two points for most rendering algorithms is the ray tracing operator. Thus, it is also usually their major performance bottleneck. A typical rendering algorithm will need to trace millions to even billions (e.g. path tracing) of rays for just one frame.

The naive way to solve a ray tracing query is to loop over all primitives in the scene, intersect each with the ray and select the closest intersection to the ray's origin. Such an algorithm has a linear time complexity and except for trivial scenes is unsuitable for rendering, due to its excessive running time.

The most common strategy to reduce the time complexity of intersection is to use an *acceleration structure*. The latter is a form of spatial index over the primi-

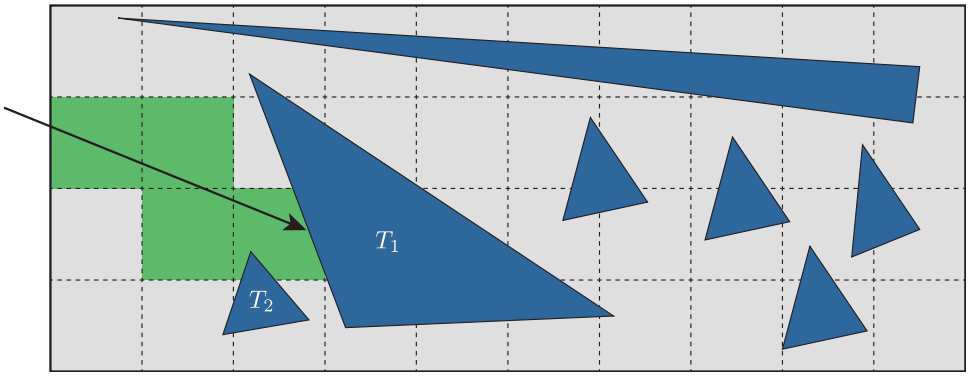


Figure 2.9: *A regular grid. The ray will traverse the voxels in green and will only test triangles T_1 and T_2 for intersection (without even considering the rest).*

tives, which allows the ray tracing operator to only examine primitives near the ray, without ever touching the rest. In this section we introduce the main types of acceleration structures. We will briefly discuss their respective traversal algorithms and construction algorithms, leaving the details for chapters 3 and 7.

There are three major types of acceleration structures used today: grids, space partitioning hierarchies, and bounding volume hierarchies. Each of the types has several variations. A good in-depth study of acceleration structures for ray tracing can be found in [Havran, 2000].

2.3.1 Grids

Probably the simplest acceleration structures is the regular grid [Fujimoto et al., 1986]. It partitions the space of the scene uniformly using a rectangular regular grid. Each voxel of the grid contains a list of primitives that overlap with it.

To compute $h(x, \omega)$, the grid voxels along the ray are incrementally enumerated using a digital difference analyzer (Figure 2.9). The primitives contained in each enumerated voxel are intersected with the ray. If at least one intersection is contained in the current voxel, the enumeration stops and the closest intersection is returned. If the grid is constructed properly, a visibility query requires $O(\sqrt[3]{|S|})$ time on average [Cleary and Wyvill, 1988].

The main disadvantage of grids is that they can not adapt to non-uniform primitive distributions due to their regular subdivision of space. To alleviate the problem, hierarchical grids have been introduced. A hierarchical grid, like a regular grid, contains lists of primitives in most of its voxels. However, some voxels can contain a nested (hierarchical-) grid instead. The construction algorithm decides whether to create a nested grid or a primitive list based on criteria such as how much geometry is contained in a voxel. A description of the types of hierarchical grid structures and further references can be found in [Havran, 2000].

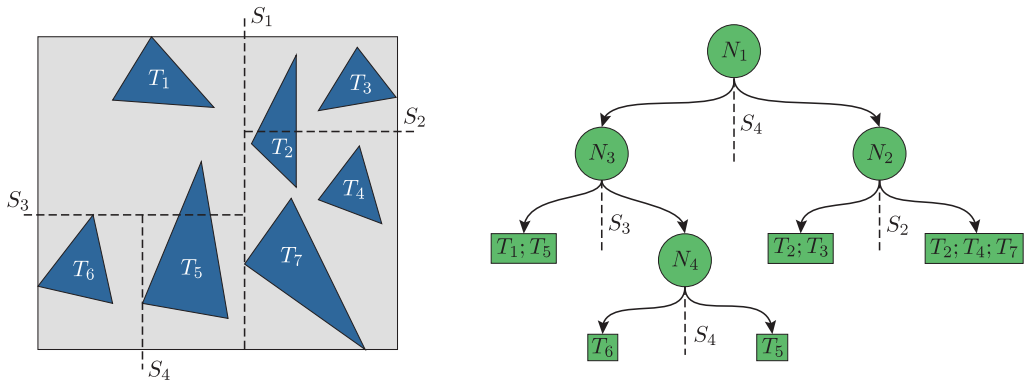


Figure 2.10: *A KD-tree. The left part of the picture shows how the scene space is partitioned by the tree on the right. The split planes are denoted with S_i, the primitives – with T_i, the nodes with N_i, and the leafs are represented as rectangles whose content is the list of incident primitives.*

2.3.2 Space Partitioning Hierarchies

Space partitioning hierarchies, the second major type of acceleration structure, are tree structures. Each node N of the tree corresponds to a region of space $R(N)$. The root node corresponds to a region bounding all primitives of the scene. The region corresponding to a node is always partitioned by its children. Thus, the set of regions corresponding to the leafs of the tree forms a full partitioning of the space corresponding to the root. Each leaf contains a list of all primitives that overlap with its region.

We discuss construction strategies for space partitioning hierarchies in Chapter 3 and their traversal in Chapter 7. Furthermore, we give a sketch of the traversal algorithm here, as it is required for the rest of the discussion in this chapter. To solve a visibility query, a space partitioning hierarchy is traversed in a recursive manner. Starting from the root, the children of each node are enumerated in an ascending order, using as a key the distance from the ray's origin to the entry point of the ray in that child. An intersection is then searched for in each child recursively in the above order. If a node is a leaf, the ray is tested for intersection with all of the leaf's incident primitives. If such an intersection is found, the traversal stops and the closest intersection in the leaf becomes the result of the visibility query.

Two major types of space partitioning hierarchies, that have been commonly used in ray tracing, are octrees [Glassner, 1984] and KD-trees (see Figure 2.10). Even though the first are still used in some rendering software today (such as [Blender]), they are considered inefficient. KD-trees are preferred due to their potential to adapt quicker to non-uniform geometry. Thus, we will not discuss Octrees here, but will rather concentrate on KD-trees.

A KD-tree is a binary space partitioning tree. The children of each each internal node of the KD-tree are separated by an axis aligned (split-) plane. Again, a leaf of the KD-tree contains the list of all primitives that overlap with it. KD-trees are constructed top-down and the different construction strategies differ in the choice

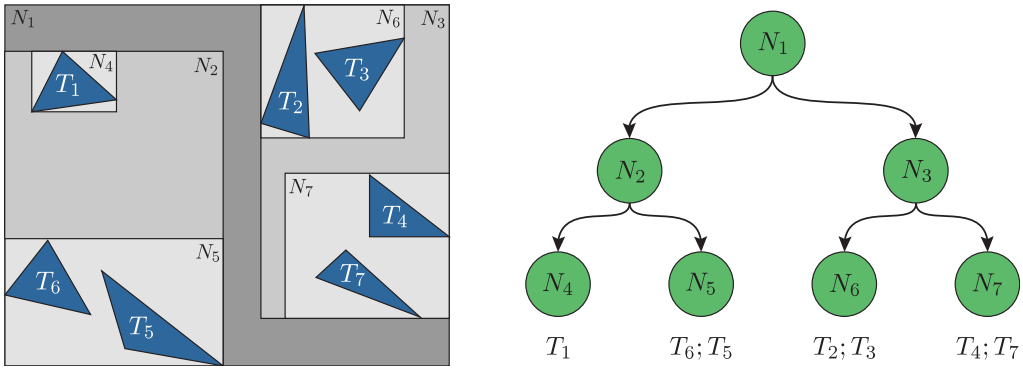


Figure 2.11: *A Bounding Volume Hierarchy. The left part of the picture shows the scene and how the nodes of the BVH are formed. The right part is the BVH tree. The lists of primitives overlapping the leafs (N_4, N_5, N_6, and N_7) are presented under them on the right picture.*

of split planes. Practical results show that suitably chosen split planes can make an order of a magnitude difference in the average time needed for computing an intersection [Havran, 2000, Appendix E].

KD-trees are a special case of binary space partition (BSP) trees, and the difference is that the latter allow non-axis aligned split planes as well. While this could help to build better acceleration structures in theory, the added cost of intersecting with an arbitrary plane makes BSP traversal slower in practice [Havran, 2000, Section 4.2.1]. As a consequence, they are not used for ray tracing in practice.

2.3.3 Bounding Volume Hierarchies

Bounding volume hierarchies or BVHs are tree structures with a usually fixed arity (Figure 2.11). Again, each node of the tree has a corresponding region of space which fully bounds the regions of space of its children. In the case of a leaf, the corresponding region of space fully bounds the primitives contained in the leaf. The main difference between a BVH and a space partitioning hierarchy is that while the first partitions space, the second partitions sets of objects. As a consequence, the children of a BVH node can overlap and also some empty space regions of a node can be left uncovered by any of its children. Furthermore, each primitive is referenced in exactly one leaf.

Similar to a space partitioning hierarchy, a BVH is traversed in a recursive manner. Starting from the root, the children of each node are ordered along the ray (according to their entry point). Children that are not intersected at all by the ray are skipped. In contrast to space partitioning hierarchies however, the traversal does not stop once an intersection is found. Rather, the distance to the intersection is remembered and only nodes that have a ray entry distance smaller than the remembered one are traversed. Traversal stops when there no more nodes are available for traversal.

Except in some architecture specific cases [Dammertz et al., 2008; Ernst and Greiner, 2008], the arity of a BVH is most commonly chosen to be two. The shape

of the corresponding region of a node can be arbitrary, but in practice it is usually chosen to be an axis aligned bounding box (AABB).

2.4 Interactive Ray Tracing

Until recently, the only technique capable of handling general purpose interactive 3D visualization was rasterization (Section 2.2.4). Unfortunately, due to its limitations, rasterization can not simulate many light transport effects (e.g. reflections and refractions). Furthermore, rasterization techniques for orthogonal light effects are often incompatible or combining them is very hard. On the positive side, rasterization is very fast. Ray tracing based techniques on the other hand have none of the above problems, but they have been historically slow and only used for off-line rendering. Due to hardware and algorithmic development in the recent years, algorithms based on ray tracing have become interactive (especially Whitted style ray tracing) and are increasingly viewed as an alternative to rasterization.

In this section, we try to summarize briefly the important work that has lead to the current state of the art in interactive ray tracing. We first look at the development of traversal algorithms. We pay special attention to those that exploit thread and data level parallelism, as they are the base and inspiration of our work in Part II of this thesis. We than discuss the limitations of previous work on interactive animation with ray-tracing, as those limitations were the motivation behind our work in Part I.

2.4.1 Interactive Traversal

Undoubtedly, the largest speed improvements in ray tracing based algorithms have come through the use of acceleration structures. More notable in this area is the work of Cleary and Wyvill [Cleary and Wyvill, 1988], for their contribution on regular grid construction and traversal for ray tracing, the work of Goldsmith and Salmon [Goldsmith and Salmon, 1987], for the introduction of the surface area model and the automatic construction of BVHs, the work of MacDonald and Booth [MacDonald and Booth, 1989], for the development of the surface area heuristic and applying it to KD trees, and finally the PhD thesis of Vlastimil Havran [Havran, 2000] for summarizing all relevant acceleration structures research and comparing the properties of all known (at that time) acceleration structures.

After the introduction of SAH, there has been little algorithmic improvement of the speed of single ray visibility queries. Traversal has been accelerated by statistical optimizations (e.g. [Havran et al., 1998b] or [Mahovsky, 2005, Section 4.5]) and there have been attempts to come up with better cost metrics (including our own, presented in Chapter 6). Most of the speed-up, has come through the use of parallelism, architecture specific optimizations, and through approximation techniques.

Notable approximation techniques include the *Render Cache* [Walter et al., 1999] and the *Holodeck* [Ward and Simmons, 1999]. While undoubtedly interesting, approximation techniques will be left out of the discussion here, as they are orthogonal to the rendering algorithms and to ray-tracing. We will rather focus on the core ray-tracing techniques.

In most rendering algorithms, the paths passing through different sensors of the image plane are independent. Thus, they can be processed in parallel, which makes most ray tracing based algorithms inherently parallel. One of the first such implementations (and probably the first interactive ray tracer) was due to Muuss [Muuss, 1995], who used an array of SGI PowerChallange computers (with 96 processors in total). Later, Parker et al. [Parker et al., 1999] developed an interactive ray tracing system on a shared-memory supercomputer. Like Muuss, they used a "brute force" approach, in the sense that their system simply executed the serial code for different pixels in parallel. In both systems, the major challenge was the load balancing and the synchronization. Since then, there have been many other parallel implementations, including Manta [Bigler et al., 2006], and OpenRT [Wald et al., 2003].

While the implementations mentioned above are parallel at task level, parallelism can also be exploited at data level. Most current processors offer extensions that allow efficient processing of packets of data, in the form of instructions that operate on vectors (a.k.a. *Single Instruction Multiple Data* or *SIMD*). Since the cost of a vector instruction is usually the same as the cost of a scalar one, researchers have tried to parallelize serial code, by emulating task level parallelism using the SIMD extensions. The challenge in such emulations comes from code divergence. It can happen for example, that one thread executes the if part of an if-else construct, while the other one executes the else part. In this case, both threads will have to execute both parts (due to the SIMD model). To obtain correct results, implementations keep a mask of the currently active threads and discard results of the inactive. SIMD parallel traversal of acceleration structures has first been demonstrated in [Wald et al., 2001] for KD-trees, in [Wald et al., 2007] for BVHs, and in [Wald et al., 2006a] for grids.

A very important factor for the efficient execution of any program today is its cache friendliness. Due to the multi-level memory hierarchy of modern processors, an incoherent memory access pattern can lead to an order of magnitude slower execution. Thus, many ray tracing implementations, including [Wald et al., 2001, 2006b, 2007; Reshetov et al., 2005; Overbeck et al., 2008], process even larger packets in SIMD manner, for rays known to be coherent (e.g. primary and shadow rays). Since the coherent rays typically visit the same elements of the acceleration structure and intersect the same triangles, the cost of an operation to main memory is amortized in this way over the whole packet. The idea of large packets has been furthermore extended to frustum traversal [Reshetov et al., 2005; Wald et al., 2007, 2006b]. In some cases (e.g. primary and sometimes shadow rays), the rays in a packet can be bound by a frustum, which can be used to efficiently cull elements of the acceleration structure that will not be traversed by any ray.

The efficiency of packet traversal techniques is limited by the coherency of the rays to be traversed. Thus, in scenarios such as path tracing, packet traversal becomes less useful. In order to be able to exploit SIMD parallelism in this cases, several approaches [Dammertz et al., 2008; Wald et al., 2008; Tsakok, 2009] use higher arity acceleration structures and intersect in parallel one ray with a SIMD-width number of acceleration structure elements.

All of the above discussed work is CPU based. Recognizing that the CPU is a general purpose processor, which is not specifically optimized for graphics, re-

searchers have attempted to accelerate ray tracing by using hardware implementations (e.g. [Woop et al., 2005]). Furthermore, there were several attempts prior to our work to use the GPU for ray-tracing. They were rather unsuccessful, despite the GPU being the most powerful processor in modern computers. This is where our contribution lies in. In the second part of this thesis, we develop several traversal algorithms specifically tailored for the GPU, which show that GPU ray tracing is indeed feasible and can considerably outperform the CPU based one. We leave the in-depth discussion of prior GPU implementations for there (in Chapter 7).

2.4.2 Animation

Until now we have regarded the scene as a "soup" of primitives. Even though convenient, this representation poses a great challenge to a ray tracing pipeline, since the acceleration structure has to be rebuilt every frame. Depending on the scene size, a rebuild can take from few milliseconds to even hours, which effectively disallows interactive rendering of non-trivial animated scenes. This has inspired a lot of research on fast acceleration structure construction lately, and chapters 4 and 5 present our contribution to the field.

An alternative to fast rebuilding is to exploit the structure of the animation. The first proposed solution was to separate the geometry into static and dynamic, use an acceleration structure for the static part and intersect the rays separately with the dynamic one using a naive intersection algorithm [Parker et al., 1999]. A disadvantage of this approach is that the complexity of the dynamic geometry needs to be low in order to achieve interactive frame rates. One could as well build an acceleration structure for the dynamic geometry at each frame, but again the amount of dynamic geometry has to be low.

A further solution is to exploit the structure of rigid body animation. Since the rigid bodies are composed of static geometry, their acceleration structures can be built in advance. During animation a new top-level acceleration structure is built for each frame over the transformed AABBs of the rigid bodies. The approach works under the assumption that the count of top level objects is relatively low. During ray tracing, the top-level structure is used to enumerate the rigid bodies pierced by a ray. For each object in the enumeration, the ray is transformed with the object's inverse transformation and then it is intersected with the scene using the object's acceleration structure. This approach has been introduced in [Lext and Akenine-möller, 2001; Wald et al., 2003].

Later on, the approach was extended to also support deformable geometry. In some cases, a bounding box relative to the rigid object can be computed for each primitive, such that it bounds all possible positions in space that the primitive can take. In the context of KD-trees, this fact can be exploited to build fuzzy KD-trees (e.g. for skinning animations) by using this AABB instead of the primitive's own AABB [Günther et al., 2006a]. Furthermore, unstructured key-frame animations can be automatically analyzed to determine the rigid body structure and the AABBs that constrain the movement of the primitives [Günther et al., 2006b]. Supporting deformable geometry for BVHs is simpler, as they allow for fast refitting [Wald et al., 2007; Lauterbach et al., 2006]. Hybrid data structures that allow refitting have been

proposed as well [Woop et al., 2006].

Even though rigid body and deformable techniques achieve their goal (i.e. interactive animation), they limit the types of scene interactions that are allowed. In the context of games and physical simulations for example, two interactions that become impossible are breaking of objects and explosions. Furthermore, with those techniques there is a penalty to traversal performance, which can be as high as 40% (compared to a full rebuild). These two issues have motivated us to look at new methods for fast construction of acceleration structures from scratch, which we present in Part I of this thesis.

2.5 Summary

In this chapter, we gave a brief overview of the topics from computer graphics relevant to the rest of the thesis. Since part I of the thesis will deal mainly with the issues of improving the speed of acceleration structure construction (in Chapter 4 and Chapter 5) and improving the quality of the constructed trees (in Chapter 6), we will revisit acceleration structure construction in much more detail in Chapter 3. Part II will deal with interactive ray tracing on the GPU. Thus, we will look closer at acceleration structure traversal, the GPU architecture, and prior work on GPU ray tracing in Chapter 7. We will then use this knowledge in Chapter 8 and Chapter 9 to develop several techniques for KD-tree and BVH traversal on the GPU.

Part I

Construction of Acceleration Structures

Chapter 3

Construction Background

Part I of this thesis is dedicated to our contribution in acceleration structure construction. To gain better understanding of the topic, we first present the necessary background in this chapter. We introduce the common structure of a construction algorithm, followed by a discussion on the surface area cost model. The latter is an essential tool for building traversal-optimized trees. We then focus on the previous state of the art KD-tree and BVH construction algorithms (which we refer to as "classical"). We explore them in detail, since our work derives from them.

We leave grids out of the discussion, mainly because they are not connected to our work, but also because their single-ray ray tracing performance is inferior to kd-trees and BVHs (see [Havran, 2000]). Recent results show that grids can be indeed an alternative to KD-trees and BVHs for unstructured animation [Kalojanov et al., 2011; Kalojanov and Slusallek, 2009; Wald et al., 2006b], due to their low construction times. On the other hand, current fast BVH and KD-tree construction algorithms [Lauterbach et al., 2009; Pantaleoni and Luebke, 2010; Zhou et al., 2008; Wald, 2007; Günther et al., 2007; Danilewski et al., 2010; Shevtsov et al., 2007], achieve similar construction speeds but with much faster traversal performance. Thus, the usefulness of grids in general remains questionable in our opinion. More details on grid construction can be found in [Cleary and Wyvill, 1988; Havran, 2000].

3.1 Basics

Since kd-trees and BVHs are in essence trees, there are two ways in general to construct them: top-down recursive and bottom-up. The first approach starts at the root and recursively constructs the sub-trees of all child nodes. The second – starts at the leafs, which are sub-trees with height 1, and iteratively groups sub-trees into new sub-trees, until only one is left.

3.1.1 Top-down Construction

In top-down construction each to-be-constructed sub-tree with root node N has an associated enclosed region of space $R(N)$ and a set of primitives $S(N)$. At the root, $S(N)$ is the set of all primitives and $R(N)$ fully contains each of them. In order to

construct valid acceleration structures, construction algorithms have to ensure that if a primitive p overlaps with $R(N)$ for some node N, then $S(N)$ will contain p.

A construction step involves three sub-steps . First, the set of primitives is split into two $S(N) = S(N_L) \cup S(N_R)$, with N_L and N_R being the left and right child of N respectively. Next, the regions of space enclosed by N_L and N_R are computed based on $S(N_L)$, $S(N_R)$, and $R(N)$. Finally, the construction step is recursively repeated for the sub-trees rooted at N_L and N_R, using the computed $S(N_L)$ and $R(N_L)$ for the left node, and $S(N_R)$ and $R(N_R)$ for the right one. The recursion terminates according to some boolean criteria $T\left(S(N), R(N)\right)$, known as the termination criteria. In this case, N becomes a leaf and $S(N)$ is stored in it (usually as a list). From now on we will denote $S(N_L)$ with S_L and $S(N_R)$ with S_R.

Notice, that we have assumed arity 2 for the trees in the above explanation. In general, the arity can be larger. Especially for BVHs, applications of trees with higher arity have proven practical [Tsakok, 2009; Dammertz et al., 2008; Wald et al., 2008; Ernst and Greiner, 2008]. However, n-ary BVHs are currently constructed by folding nodes from an existing binary BVH, and we are not aware of algorithms able to directly construct BVHs with higher arity.

If we assume at most linear complexity for both the part that determines how to split $S(N)$ and for the computation of $T\left(S(N), R(N)\right)$, then the algorithm above runs in $O(|S| \log |S|)$ on average [Havran, 2000, Section 4.9].

3.1.2 Bottom-up Construction

Bottom-up construction first partitions the set of primitives according to some criteria and forms a set $\mathcal{S}$, whose elements are the subsets of the partition. Each element of $\mathcal{S}$ also holds a pointer to a corresponding node in the future tree. Initially, the elements in $\mathcal{S}$ correspond to the leafs. Construction then proceeds by repeatedly taking out two or more elements $s_1, s_2, \ldots$ from $\mathcal{S}$, merging them, and putting back the resulting element into $\mathcal{S}$. The set of primitives contained in the new element is simply the union of the primitives of all s_i, while the corresponding node is a new node, with children – the nodes of s_i. The process continues until the size of $\mathcal{S}$ becomes 1. The bounding volume of each new node is computed while creating it, by computing a possibly tight volume, containing the volumes of all children.

The above described approach applies to BVHs only. We are not aware of any bottom-up construction algorithms for space partitioning hierarchies. The major advantage of this approach is its potential to perform faster than top-down recursive construction. With suitably chosen criteria, the whole construction process can become linear, thus beating the $O(N \log N)$ complexity of top-down construction.

The only bottom-up approach with decent tree quality that we are aware of is presented in [Walter et al., 2008]. Unfortunately, it is actually slower than "gold standard" BVH construction [Wald et al., 2007]. Also, the authors do not compare their BVHs to [Wald et al., 2007], and it thus remains unclear whether the traversal performance is higher and whether their method is useful in practice. We believe that the choice of fast and efficient grouping criteria for bottom up builds is still an open question.

Figure 3.1: *Exact vs boxed overlap. The exact overlap test will fail unless the triangle T and the bounding box $\mathcal{B}$ overlap (left). The boxed one however will not fail as the bounding boxes $\mathcal{B}(T)$ and $\mathcal{B}$ overlap (right).*

3.1.3 The Overlap Test

Another topic relevant to the construction of a hierarchies is how to determine if a primitive p is incident with the region of space $R(N)$ of some node N. For efficiency reasons [Havran, 2000, Section 4.2.1], in the context of ray tracing $R(N)$ is always chosen to be an axis aligned bounding box. Thus, we will assume from now on that $R(N)$ is indeed an AABB and we will denote it with $\mathcal{B}(N)$.

The overlap test boils down to determining whether some part of the primitive is inside the bounding box or not (see Figure 3.1). An exact result requires clipping of the primitive against a bounding box, which has linear time complexity w.r.t. the number of vertices of polygonal objects. Thus, even with the simplifying assumption of $R(N)$ being AABB, the test might still take arbitrary long in the general case and it can be slow even for simple primitives such as a triangle. For efficiency reasons, primitives are sometimes approximated with the tightest AABBs around them. Construction based on such an approximation is known as *boxed* construction. We will refer to non-boxed construction as *exact* from now on. The trees produced from exact construction can be up to 40% faster w.r.t. ray tracing than the ones from boxed construction [Havran, 2000, Section 4.10.4]. On the other hand, boxed construction can be much faster.

The common way to test if a triangle is incident to an AABB is to use a clipping algorithm such as the one by Sutherland and Hodgeman [Sutherland and Hodgman, 1974]. Testing if two bounding boxes overlap (for boxed construction) is performed through interval intersection in 1D [Akenine-Möller et al., 2008, p. 765]. More object/AABB intersection routines can be found in [Akenine-Möller et al., 2008], though usually primitives more complex than a triangle are always approximated by AABBs.

3.1.4 Termination Criteria

The choice of good termination criteria is important as it influences the subsequent speed of the visibility queries. Termination criteria can be fixed or automatic.

Fixed (or ad-hoc) criteria were introduced in [Glassner, 1984]. They track two characteristics of the current node N: depth of the node from the root and the number of primitives associated with it. If this characteristics exceed respectively fall below pre-determined thresholds (e.g. if a tree gets too deep or a node has too few primitives), the termination criteria signals the construction process to create a leaf. The thresholds are specified by the user and they are scene dependent.

Automatic termination criteria [Havran, 2000, Section 4.5] on the other hand use a cost model and compare the cost of creating a leaf vs. the cost of creating a node. As their name implies, they are designed to be scene independent, without the need of user-specified parameters. While the term itself was first introduced in [Subramanian and Fussell, 1991], the first working automatic termination criteria was described in [Havran, 2000]. As the latter is based on the surface area heuristic, which we will introduce in the next section, we will leave its discussion for later on.

3.2 The Surface Area Cost Model

The ray tracing performance of a tree can be predicted using the surface area cost model introduced by [MacDonald and Booth, 1989] and later refined by [Havran, 2000]. The expected cost of a tree T is given by

$$Exp(T) = \mathcal{C}_T \sum_{N \in S_I} P\left(N \mid T\right) + \mathcal{C}_I \sum_{L \in S_L} P\left(L \mid T\right) |L| \tag{3.1}$$

where S_I is the set of all inner nodes of the tree, S_L is the set of all leafs, $P\left(.|T\right)$ is the geometric probability of a random ray hitting a node/leaf given that it hits the root node of the tree, $|L|$ is the number of primitives in the leaf L, and $\mathcal{C}_T$ and $\mathcal{C}_I$ are the traversal and intersection costs respectively. Assuming that the regions of space corresponding to the tree nodes are convex, $P(.|T)$ can be expressed as the ratio of the surface area of the node/leaf to the surface area of the root node [Santaló, 1976]. Thus $P(.|T) = \mathcal{A}(.)/\mathcal{A}(T)$, with $\mathcal{A}(.)$ being a function that returns the surface area of its argument.

$Exp(T)$ gives the expected cost for traversing the tree with a uniformly randomly chosen ray. It assumes that the traversal does not terminate before traversing all nodes incident to the ray. Even though this assumption rarely holds, since traversal typically stops once it finds the closest intersection, in practice there is a strong correlation between $Exp(T)$ and traversal performance. Thus, construction algorithms aim at minimizing $Exp(T)$.

For binary trees, equation (3.1) can be expressed recursively:

$$Exp(N) = \begin{cases} \mathcal{C}_T + P\left(N_L \mid N\right) E(N_L) + P\left(N_R \mid N\right) E(N_R) & , \ N \text{ node} \\ \mathcal{C}_I \ |N| & , \ N \text{ leaf} \end{cases} \tag{3.2}$$

where N_L and N_R are the left and right children of N, and $|N|$ is the number of primitives in N. It is easy to see that $Exp(T) = Exp(Root(T))$. Again, $P(.|N) = \mathcal{A}(.)/\mathcal{A}(N)$, assuming that the regions of space associated with the nodes and leafs of the tree have convex shape.

3.2.1 The Surface Area Heuristics

One of the primary uses of the surface area cost model is to construct trees optimized for traversal. To create a tree T with optimal cost $Exp(T)$, the children N_L and N_R of each node N have to be chosen so that $Exp(N)$ is minimized.

Unfortunately, this process is NP hard as $Exp(N)$ is recursively dependent on $Exp(N_L)$ and $Exp(N_R)$. Thus, for each tested configuration of N_L and N_R, the full

trees of N_L and N_R need to be built. The surface area heuristic, or *SAH* [MacDonald and Booth, 1989], tries to overcome this limitation by using an approximation.

To compute $Exp(N)$, the SAH replaces $Exp(N_L)$ and $Exp(N_R)$ with the number of primitives contained in each node (i.e. $|S_L|$ and $|S_R|$). This corresponds to the upper bound of $Exp(N)$, which would be reached if the construction algorithm creates leafs from the left and right sub-trees. In this case, the expected cost becomes

$$Exp(N) \approx \underset{SAH}{Exp(N)} = \mathcal{C}_T + \mathcal{C}_I\big(P(N_L \mid N)\ |S_L| + P(N_R \mid N)\ |S_R|\big) \tag{3.3}$$

3.2.2 Automatic Termination Criteria

Another benefit of using a cost model is the ability to use an automatic termination criteria. The basic idea is to compare the minimum cost of a node as an internal tree node to the cost of the node as a leaf [Havran, 2000, Section 4.5]. The quality of a split is defined as the following ration

$$r_Q(N) = \frac{Exp(N)}{\mathcal{C}_I|S(N)|} \approx \frac{\mathcal{C}_T + \mathcal{C}_I\big(P(N_L \mid N)\ |S_L| + P(N_R \mid N)\ |S_R|\big)}{\mathcal{C}_I|S(N)|}$$

If $r_Q(N) \geq 1$ the recursion has to terminate as it is more beneficial to create a leaf. Furthermore [Havran, 2000] proposes to introduce a second threshold r_Q^{min} and to consider a spilt as partly successful if $r_Q^{min} < r_Q(N) < 1$. If the count of consecutive partly unsuccessful steps from N towards root exceeds a scene specific threshold F_{max}, the construction should create a leaf. The intuition behind that approach is that even though a split can be locally "bad" things might get better further down the tree. The proposed way of computing F_{max} is based on the maximum leaf depth of the tree d_{max}

$$F_{max} = K_{fail}^1 + K_{fail}^2\, d_{max}$$

The constants are empirically chosen (based on practical test) to be $K_{fail}^1 = 1.0$, $K_{fail}^2 = 0.2$, $r_Q^{min} = 0.75$. Since KD-trees and BVHs are more or less balanced, the maximum depth d_{max} can either be computed as $d_{max} = \log N$, where N is the number of primitives in the tree, or it can be a fixed user-specified constant, in case the automatic termination criteria is combined with ad-hoc termination.

3.2.3 Plane Sweep Algorithms

Most top down construction algorithms, including the new algorithms presented in this thesis, rely on an approach from the field of computational geometry known as the *plane sweep algorithm* [de Berg et al., 2008], which uses a conceptual sweep (hyper-) plane to solve various problems in Euclidean space.

A sweep plane moves continuously along the direction of its normal. The movement is defined by the distance v of the plane from the origin O of the coordinate system. The sweep plane also has an associated *status*. The latter can be any property of the geometry (such as for example the number of primitives to the left/right of the plane), but with one restriction: the number of plane positions at which the status is of interest to the solved problem has to be finite. These positions are called *events*.

To work efficiently, the plane sweep algorithm requires the following knowledge: 1) the first event to visit; 2) how to compute the next event, based on data from past events; 3) a way to determine if an event is the last one; and 3) how to incrementally compute the status for a new event, based on the status of the previously visited events. The algorithm then visits the events in order, incrementally computes the status of each of them, and carries out a user specified operation on the computed status.

The sweep plane algorithm is used commonly to search for properties of geometry when either the events can not be computed efficiently in advance, or when computing the status incrementally is more efficient than computing it directly.

A good example for the first case is finding all intersections of a given set of line segments in 2D [de Berg et al., 2008, Section 2.1]. The sweep line in this case moves perpendicular to the axis Ox and the events are the start and end points of the segments as well as the intersections. The status of the sweep line is the set of segments intersecting it and the user specified operation outputs an event if it is an intersection point. The events are initially only the start and end points of the segments. Once an event is reached, new (intersection-) events are calculated based on the current status. With this algorithm, all intersections can be found in $O(N \log N)$ as opposed to the $O(N^2)$ time of the naïve one.

In the context of acceleration structure construction, the sweep plane algorithm is used to efficiently compute separating hyper planes that minimize a given cost function over the geometry of a scene. We will use it to compute a median split in Section 3.3.1 and SAH based splits in Section 3.3.2 and Section 3.4.1. We will use the following notation: v will denote the offset of the sweep plane from the origin of the coordinate system, the events (which are known in advance) will be $v_{start} = v_0 < v_1 < \ldots < v_k = v_{end}$ and the objective will be to find the event v_{opt} that minimizes some cost function $cost(v)$.

3.3 Construction of KD-trees

KD-trees are always constructed in a top-down manner. At each node N, the construction process searches for a split plane $P(N)$ and uses it to partition the region $R(N)$ among the two children of N. There are various methods for constructing a KD-tree and their main differences lie in the process of choosing the split plane, in the termination criteria used, and on whether the construction is boxed or exact. The different methods present different trade-offs between construction and traversal speed.

Before going into details about these methods, we will introduce some common notations. Besides the set of corresponding primitives $S(N)$ and the the bounding box $\mathcal{B}(N)$, a construction process with exact overlap testing also keeps for each node N a set

$$S_{\mathcal{B}}(N) = \{\mathcal{B}\left(p \cap \mathcal{B}(N)\right) \mid p \in S(N)\} \tag{3.4}$$

The latter contains the AABBs around the part of each primitive from $S(N)$ that is also in $\mathcal{B}(N)$ (see Figure 3.2). For a construction process with a boxed overlap test

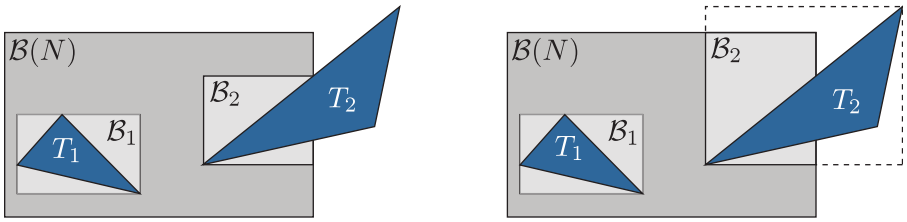


Figure 3.2: *The set $S_\mathcal{B} = \{\mathcal{B}_1, \mathcal{B}_2\}$, when using an exact overlap test (left) and a boxed one (right). In the first case $\mathcal{B}_2 = T_2 \cap \mathcal{B}(N)$ whereas in the second $\mathcal{B}_2 = \mathcal{B}(T_2) \cap \mathcal{B}(N)$*

this set is given by:

$$S_\mathcal{B}(N) = \{\mathcal{B}(p) \cap \mathcal{B}(N) \mid p \in S(N))\} \tag{3.5}$$

Assuming that the split plane $P(N)$ is already computed, the construction process has to split the sets $S(N)$ and $S_\mathcal{B}(N)$ into $S(N) = S_L \cup S_R$ respectively $S_\mathcal{B}(N) = S_\mathcal{B}(N_L) \cup S_\mathcal{B}(N_R)$. To do so (see Algorithm 3.1), it looks at each element in $\mathcal{B} \in S_\mathcal{B}(N)$ and its respective primitive $p \in S(N)$. If $\mathcal{B}$ is entirely to the "left" of $P(N)$, it and p are put in $S_\mathcal{B}(N_L)$ and S_L respectively. If it is entirely to the right – the two are put in $S_\mathcal{B}(N_R)$ and S_R respectively. Finally, if $\mathcal{B}(p)$ is incident to $P(N)$ the primitive p is put in both S_L and S_R. The corresponding AABBs that go into $S_\mathcal{B}(N_L)$ respectively $S_\mathcal{B}(N_R)$ are computed according to (3.4) and (3.5). With exact overlap testing, they are $\mathcal{B}(p \cap \mathcal{B}(N_L))$ and $\mathcal{B}(p \cap \mathcal{B}(N_R))$ respectively, and for boxed – $\mathcal{B}(p) \cap \mathcal{B}(N_L)$ and $\mathcal{B}(p) \cap \mathcal{B}(N_R)$ respectively . The above described process is known as *sifting*. A bounding box $\mathcal{B}$ is considered to the "left" of a plane P iff all its vertices are contained in the negative half-space of P and to the "right" if all its vertices are contained in the positive one. The positive half-space of P is along the normal of P, which in this case is parallel to one of the major axes (i.e. two of its components are 0 and the third one is 1).

3.3.1 Split in the Middle and Median Split

The simplest strategy for computing $P(N)$ is to split $\mathcal{B}(N)$ in the middle along one of the three major axes. The latter can be chosen by taking the one that is parallel to the longest side of $\mathcal{B}(N)$ or in an alternating way: if the parent node uses axis $d \in \{0, 1, 2\}$, the current one will use $(d + 1) \bmod 3$. This strategy is known as split in the middle.

A similar strategy is to choose $P(N)$ so that the number of primitives to the left and right of the plane is approximately the same. Again, the dimension along which the split happens can be chosen to be parallel to the longest side of $\mathcal{B}(N)$ or in an alternating manner. This strategy is known as *median splitting*.

The position of $P(N)$ in this case is best computed by using a sweep plane, which moves along the chosen dimension d. The status of the sweep plane, when positioned at offset v, is composed of two numbers: $|S_L(v)|$ and $|S_R(v)|$. Here, $S_L(v)$ and $S_R(v)$ denote the sets of primitives belonging to the potential left and right sub-trees respectively, in case v is used as a split plane. Determining the median split

Algorithm 3.1 Generic KD-tree Construction

```
1: function Sift(S, S_B, B, splitInfo)
2:     S_L ← ∅, S_R ← ∅
3:     S_B(N_L) ← ∅, S_B(N_R) ← ∅
4:     (B_L, B_R) ← SplitAABB(B, splitInfo.plane)
5:     d ← splitInfo.plane.axis
6:     v ← splitInfo.plane.offset
7:     for p ∈ S do
8:         X ← AABB corresponding to p from S_B
9:         if X^d_max ≤ v then                          ▷ X^d_max ≡ X.max[d]
10:            S_L ← S_L ∪ p, S_B(N_L) ← S_B(N_L) ∪ X
11:        else if X^d_min ≥ v then
12:            S_R ← S_R ∪ p, S_B(N_R) ← S_B(N_R) ∪ X
13:        else
14:            S_L ← S_L ∪ p, S_B(N_L) ← Clip(p, B_L)
15:            S_R ← S_R ∪ p, S_B(N_L) ← Clip(p, B_R)
16:        end if
17:    end for
18:    return (S_L, S_R, S_B(N_L), S_B(N_R), B_L, B_R)
19: end function
20:
21: function ConstructSubTree(S, S_B, B)
22:    splitInfo ← FindSplit(S, S_B, B)
23:    if T(splitInfo, S, S_B, B) then              ▷ T(.) – termination criterion
24:        return CreateLeaf(S, S_B, B)
25:    else
26:        (S_L, S_R, S_B(N_L), S_B(N_R), B_L, B_R) ← Sift(S, S_B, B, splitInfo)
27:        N_L ← ConstructSubTree(S_L, S_B(N_L), B_L)
28:        N_R ← ConstructSubTree(S_R, S_B(N_R), B_R)
29:        return CreateNode(splitInfo.plane, N_L, N_R)
30:    end if
31: end function
32:
33: function ConstructTree(S)                        ▷ S is the set of all primitives
34:    B ← emptyAABB                                 ▷ the AABB of the root node
35:    S_B ← ∅                                       ▷ the set of primitive AABBs
36:    for p ∈ S do
37:        B ← B ∪ B(p)                              ▷ extend B to contain B(p)
38:        S_B ← S_B ∪ B(p)
39:    end for
40:    return ConstructSubTree(S, S_B, B)
41: end function
```

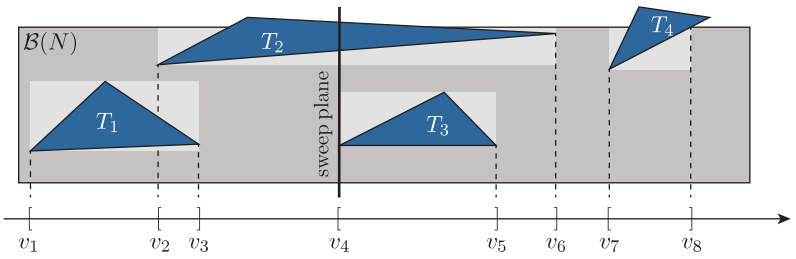


Figure 3.3: *Plane sweep for KD-tree construction. The sets determined by the split plane are* $S_L(v) = \{T_1, T_2\}$ *and* $S_R(v) = \{T_2, T_3, T_4\}$*. With exact overlap testing, the sets of events are* $S_{start} = \{v_1, v_2, v_4, v_7\}$*,* $S_{end} = \{v_3, v_5, v_6, v_8\}$ *and* $S_{flat} = \emptyset$

is equivalent to determining the position v_{opt} of the sweep plane that minimizes $cost(v) = \big||S_L(v)| - |S_R(v)|\big|$.

Another way of interpreting $|S_L(v)|$ and $|S_R(v)|$ is that they are the number of primitives whose corresponding bounding box $\mathcal{B}$ from $S_\mathcal{B}$ starts on the left of the sweep plane or respectively ends to the right of the latter (see Figure 3.3). In this case, the counts can be computed as:

$$|S_L(v)| = \Big|\Big\{\mathcal{B} \in S_\mathcal{B} \mid \mathcal{B}^d_{min} < v \vee \mathcal{B}^d_{min} = \mathcal{B}^d_{max} = v\Big\}\Big|$$

$$|S_R(v)| = \Big|\Big\{\mathcal{B} \in S_\mathcal{B} \mid \mathcal{B}^d_{max} > v\Big\}\Big|$$

with $\mathcal{B}^d_{min}$ and $\mathcal{B}^d_{max}$ being the minimum respectively maximum offsets of $\mathcal{B}$ from the origin O along the axis d. They can also be written as $\mathcal{B}^d_{min} = \min\limits_{x \in \mathcal{B}} x^d$ and $\mathcal{B}^d_{max} = \max\limits_{x \in \mathcal{B}} x^d$, where x^d is the scalar component of x along the axis d.

This alternative interpretation shows how to choose the events for the sweep plane and how to update the status. We distinguish between three types of events: *start* events, *end* events and *flat* events. They are defined as follows (figure 3.3):

$$S_{start} = \Big\{\mathcal{B}^d_{min} \mid \mathcal{B}^d_{min} \neq \mathcal{B}^d_{max} \text{ and } \mathcal{B} \in S_\mathcal{B}\Big\}$$

$$S_{end} = \Big\{\mathcal{B}^d_{max} \mid \mathcal{B}^d_{min} \neq \mathcal{B}^d_{max} \text{ and } \mathcal{B} \in S_\mathcal{B}\Big\}$$

$$S_{flat} = \Big\{\mathcal{B}^d \mid \mathcal{B}^d_{min} = \mathcal{B}^d_{max} \text{ and } \mathcal{B} \in S_\mathcal{B}\Big\}$$

The final set of events is the union $S_{start} \cup S_{end} \cup S_{flat}$. As required by the sweep plane algorithm, the events have to be sorted based on their offset from the origin O. At this point, we assume that no two events can be on the same offset. We will deal with the case of coinciding events in Section 3.3.4.

The status corresponding to the sweep plane requires two variables C_L and C_R, initially set to $C_L = 0$ and $C_R = |S(N)|$, that keep track of $|S_L(v)|$ and $|S_R(v)|$. At each event the status is updated by looking at the event type. A start event signals

that the sweep plane is stepping into the bounding box $\mathcal{B} \in S_{\mathcal{B}}$ of a new primitive. Thus, C_L needs to be increased by one to account for that. Similarly, C_R needs to be decreased by one on an end event and both C_L needs to be increased by one and C_R decreased by one on a flat event.

It is easy to see that the choice of events is correct for minimizing $cost(v)$, since the latter is piece wise constant with discontinuity points at the boundaries of the bounding boxes in $S_{\mathcal{B}}$. Special care needs to be taken when computing the cost at a flat event, since the primitive of a flat event can be put in either of the sets S_L and S_R. Thus, the cost needs to be calculated twice: once before processing the event, and once after that. A similar procedure needs to be carried out during sifting for those primitives.

3.3.2 Cost Based Splitting

The two approaches presented above, namely split in the middle and median split, are similar in the sense that they attempt to create "balance": split in the middle attempts to divide space more uniformly, whereas median split attempts to create a balanced tree. While balance is important for range searches and nearest neighbor searches, much better trees (w.r.t. to traversal performance) can be obtained by solving a minimization problem using SAH as a cost function [Havran, 2000]. In fact, both split in the middle and median split have the same SAH cost and SAH achieves its minimum between the two [MacDonald and Booth, 1989].

The most efficient approach to solve the above minimization problem exactly is currently known to be the plane sweep algorithm. The latter is executed independently for each axis (Ox, Oy, and Oz), resulting in three candidate solutions. Among them, the solution with minimal cost is chosen as the split plane.

To define the sweep plane events, we look into the definition of SAH, namely equation (3.3). We are interested in how the cost of a split changes when we move the sweep plane. Without loss of generality, we assume that the latter moves along the Ox axis and its movement is defined through the distance v from the origin. Similar to Section 3.3.1, we denote the potential children of the node w.r.t. the split plane position with $N_L(v)$ and $N_R(v)$. The surface area $\mathcal{A}(N_L(v))$ of the AABB of the left child is given by:

$$\begin{aligned}\mathcal{A}(N_L(v)) = {} & 2\mathcal{B}^y_{size}(N)\ \mathcal{B}^z_{size}(N) + \\ & 2\left(v - \mathcal{B}^x_{min}(N)\right)\left(\mathcal{B}^y_{size}(N) + \mathcal{B}^z_{size}(N)\right)\end{aligned} \tag{3.6}$$

where $\mathcal{B}(N)^d_{size}$ is the size of the bounding box along the axis d. Since $\mathcal{B}(N)$ is fixed, it can be seen from (3.6) that $\mathcal{A}(N_L(v))$ is a linear function with respect to v. Similarly, $\mathcal{A}(N_R(v))$ is also linear w.r.t. v and is given by

$$\begin{aligned}\mathcal{A}(N_R(v)) = {} & 2\mathcal{B}^y_{size}(N)\mathcal{B}^z_{size}(N) + \\ & 2\left(\mathcal{B}^x_{max}(N) - v\right)\left(\mathcal{B}^y_{size}(N) + \mathcal{B}^z_{size}(N)\right)\end{aligned} \tag{3.7}$$

The two surface areas can be written in a more compact form:

$$\mathcal{A}(N_L(v)) = C_1^L + vC_2^L \tag{3.8}$$

$$\mathcal{A}(N_R(v)) = C_1^R + vC_2^R \tag{3.9}$$

with $C_{1|2}^L$ and $C_{1|2}^R$ being the coefficients of the linear functions from (3.6) and (3.7) respectively. We assume here that v is the distance of the event from $\mathcal{B}_{min}(N)$ and not from the origin. Taking into account that $P(.|N) = \frac{\mathcal{A}(.)}{\mathcal{A}(N)}$, equation (3.3) becomes

$$\underset{SAH}{Exp}(v) = \mathcal{C}_T + \mathcal{C}_I \frac{(C_1^L + vC_2^L)|S_L(v)| + (C_1^R + vC_2^R)|S_R(v)|}{\mathcal{A}(N)} \tag{3.10}$$

If we assume for a moment that $|S_L(v)|$ and $|S_L(v)|$ are constant, equation (3.10) becomes a linear function. From Section 3.3.1 we know that $|S_L(v)|$ and $|S_R(v)|$ only change at the boundaries of the bounding boxes in $S_{\mathcal{B}}(N)$. Thus, it follows that the SAH cost is a piecewise linear function of v, and the discontinuity points are exactly those boundaries. Most importantly, the objective function reaches its minimum on at least one discontinuity point, since this function is a monotonic between the discontinuity points.

The above observation shows how to design a sweep plane approach for computing the optimal split plane according to SAH. Since the minimum is located on a boundary of a bounding box from the set $S_{\mathcal{B}}$, the events are chosen as in Section 3.3.1. As before, the counts $|S_L(v)|$ and $|S_R(v)|$ belong to the sweep plane status, and we also keep track of the surface areas $\mathcal{A}(N_L(v))$ and $\mathcal{A}(N_R(v))$ of the potential children. Again, $|S_L(v)|$ and $|S_R(v)|$ can be computed incrementally, whereas $\mathcal{A}(N_L(v))$ and $\mathcal{A}(N_R(v))$ can be computed directly. The objective function is $cost(v) = \underset{SAH}{Exp}(v)$ from (3.10) and it can be computed directly from the corresponding status of the sweep plane. We are going to leave the discussion of how to handle flat primitives and what to do with coinciding events for Section 3.3.4. There, we will also give pseudo code for the discussed algorithm.

3.3.3 Construction Complexity: $O(N \log N)$ vs $O(N \log^2 N)$

Searching for the split plane involves a plane sweep, which in turn requires the events to be sorted along the sweep axis. There are two options to do the latter: either extract and sort the events for each node independently, or extract them and sort them once in the beginning and keep them sorted while sifting.

The first method requires at least $O(N \log N)$ time to sort the events for each node, resulting in $O(N \log^2 N)$ construction. Since the events are floating point numbers with finite size (4 bytes), the sorting can also be done using radix sort on their bit representation. In theory, this would result in the better construction time of $O(N \log N)$. According to our experience however, construction based on the $O(N \log N)$ introspection sort [Musser, 1997] always outperforms construction based on the $O(N)$ radix sorting in practice. The reason is that on one side radix sort is not cache friendly, and thus large data sets lead to cache thrashing, and on the other, the real complexity of radix sort is $O(N \log_M N + M)$ where N is the number

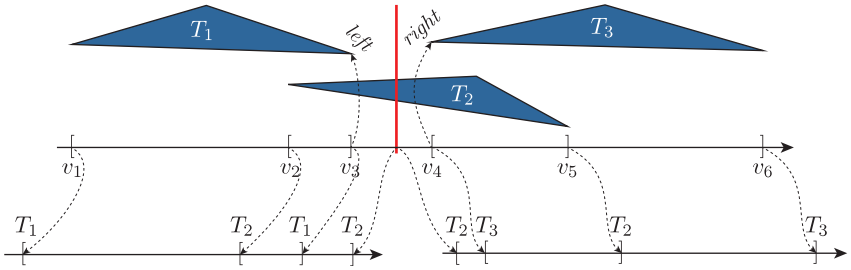


Figure 3.4: *Sifting along the split axis. Triangle T_1 is marked by the end event v_3 as belonging to the left child; T_3 – by the start event v_4 as belonging to the right child. Triangle T_2 is never marked and thus belongs to both children. During sifting, an end event is inserted in the left list for T_2 (coinciding with the split plane) and an opening one – in the right list.*

of elements and M is the radix. Thus, radix sort is efficient only if $N >> M$ (i.e. on the top tree nodes, which are only a fraction of all nodes). Choosing a smaller radix to allow for efficient sorting of small data sets alleviates the problem, but not by much, as it also increases $\log_M N$.

The second method has been proposed in [Havran, 2000, Section 4.9] and later presented in detail in [Wald and Havran, 2006]. During construction, each to-be-constructed subtree is associated with three additional lists of events L^x, L^y, and L^z (one for each axis). Each of this contains the sorted sweep plane events of all primitives in the node along the corresponding axis. During sifting, the lists are split among the children in linear time, thus achieving $O(N \log N)$ construction complexity. The lists for the root are constructed in a pre-processing step with the method discussed in Section 3.3.1. Then, again in a pre-processing step, the three lists for the root are sorted in $O(N \log N)$ time.

Sifting of the lists is achieved as follows: Without loss of generality we assume the split plane to be perpendicular to the axis Ox and at offset v from the origin. We first split the list L^x and simultaneously mark each primitive as belonging to the left, to the right, or to both children. The events in L_x are processed in two stages: those before v and those afterwards. All events from the first stage go into the list L^x_L of the left child (in the same order), and all events from the second stage – into the list L^x_R of the right child. Any end event encountered during the first stage marks the corresponding primitive as belonging to the left node only, while any start event during the second stage – as belonging to the right node only (see Figure 3.4). Primitives that are left unmarked belong to both nodes.

After stage one, all primitives that straddle the split plane will have their opening event in L^x_L and their closing event in L^x_R. Thus, sifting adds for them new closing events at the end of L^x_L and new opening ones at the beginning of L^x_R and all new events have offset v from O (see Figure 3.4).

The events of L^y are sifted by putting those, whose primitives fully belong to the left child, in the left child's list L^y_L and those whose primitives belong to the right child – in the right child's list L^y_R (see Figure 3.5). When using a boxed overlap test, the events of all primitives straddling the split plane are duplicated in both children's

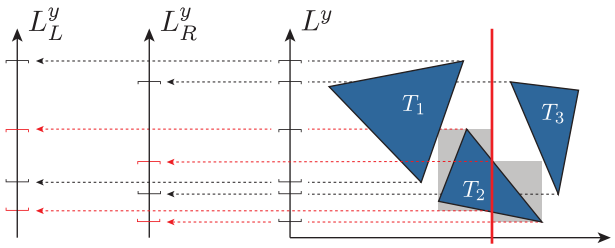


Figure 3.5: *Sifting of events along an axis parallel to the split plane (Oy in this case). The events for triangle T_1 are put into the left list and those for T_3 – into the right one. Since T_2 straddles the split plane, it is clipped with the AABB of the left/right child. Its events (red) are stored in separate lists, which are later merged with the children's lists.*

lists. The events in L^z are processed in a similar manner.

When using an exact overlap test, the events of the primitives that straddle the split plane can change their offset during sifting (see Figure 3.5). Thus, such events are initially ignored. In a second step, each primitive that straddles the split plane is clipped to the AABB of the left and right child respectively, and two sets $S_{\mathcal{B}}^L$ and $S_{\mathcal{B}}^R$ of bounding boxes are formed. Each of the two sets is used to create two lists of events (one for each axis parallel to the split plane), resulting in the four lists M_L^y, M_L^z, M_R^y, M_R^z. These lists are then sorted. To compensate for the events ignored during sifting, L_L^y, L_L^z, L_R^y, and L_R^z are merged with M_L^y, M_L^z, M_R^y, and M_R^z respectively. Since the number of primitives that straddle the split plane is expected to be $O(\sqrt{N})$ [Wald and Havran, 2006], the total time for sorting and merging will not exceed $O(N)$.

3.3.4 Implementation Details

We will now discuss how to proceed with a plane sweep in the "non-ideal" case, i.e. when there are flat events present and when two or more events can be at the same offset from the origin. We will follow the ideas presented in [Wald and Havran, 2006].

We define three sets of primitives: $P_L(v)$, $P_R(v)$, and $P_F(v)$. The first one contains all primitives from $S(N)$ whose corresponding bounding box $\mathcal{B} \in S_{\mathcal{B}}(N)$ is either entirely to the left of the split plane, or is not flat and straddles the split plane. The second one is defined analogically, but for the right side. The third set, namely $P_F(v)$, contains all flat primitives that coincide with the split plane.

For a fixed position of the split plane, if the set $P_F(v)$ is not empty, its primitives can go to either child. To decide where to put them, we look at the cost function. Since SAH depends linearly on the expression $\mathcal{A}(N_L(v))|S_L(v)| + \mathcal{A}(N_R(v))|S_R(v)|$, the lowest cost is obtained if all primitives from $P_F(v)$ are put in the node with smaller surface area. Thus, if v is before the middle of $\mathcal{B}(N)$ (along the split axis), all primitives in $P_F(v)$ have to be counted on the left side by adding their count to C_L. Otherwise their count has to be added to C_R.

The above observation requires a small modification to the sweep plane procedure from Section 3.3.1: We need to keep track of $|P_L(v)|$, $|P_R(v)|$, and $|P_F(v)|$. To do so, we modify the concept of an event: we interpret the group of all (elementary) events

Algorithm 3.2 Optimal Split Plane for Axis d for KD-tree

function LOCATESPLIT($\mathcal{E}, d, \mathcal{B}, |S(N)|$)
▷ finds the split plane along d
▷ $\mathcal{E} \equiv$ the set of events
▷ $\mathcal{B} \equiv$ the AABB of the node
▷ $|S(N)| \equiv$ number of primitives in the node
  bestSplit $\leftarrow$ (cost $= \infty$)
  $\lambda \leftarrow -\infty$ ▷ the previous event offset
  $C_L^E \leftarrow 0,\ C_R^E \leftarrow 0,\ C_F^E \leftarrow 0$
  $C_L \leftarrow 0,\ C_R \leftarrow |S(N)|$
  $M \leftarrow \mathcal{B}_{size}^{d+1}\mathcal{B}_{size}^{d+2},\quad A \leftarrow \mathcal{B}_{size}^{d+1} + \mathcal{B}_{size}^{d+2}$
  $C_1^L \leftarrow 2\left(M - \mathcal{B}_{size}^{d}A\right),\ C_L^2 \leftarrow 2A$ ▷ see (3.8)
  $C_1^R \leftarrow 2\left(M + \mathcal{B}_{size}^{d}A\right),\ C_L^2 \leftarrow -2A$ ▷ see (3.9)
  for all $e \in \mathcal{E} \cup \{(\text{flat event at } +\infty)\}$ **do** ▷ enumerate in order
    if $\lambda \neq e.offset$ **then**
      $C_L \leftarrow C_L + C_L^E,\quad C_R \leftarrow C_R - C_R^E$
      if $\lambda < \frac{\mathcal{B}_{min}+\mathcal{B}_{max}}{2}$ **then**
        $C_L \leftarrow C_L + C_F^E$
      end if
      $C_L^E \leftarrow 0,\ C_R^E \leftarrow 0,\ C_F^E \leftarrow 0$
      $v \leftarrow \lambda - \mathcal{B}_{min}$
      $cost \leftarrow C_L\left(C_1^L + vC_2^L\right) + C_R\left(C_1^R + vC_2^R\right)$ ▷ see (3.10)
      if $\lambda \geq \frac{\mathcal{B}_{min}+\mathcal{B}_{max}}{2}$ **then**
        $C_R \leftarrow C_R - C_F^E$
      end if
      if $cost <$ *bestSplit*.cost **then**
        bestSplit $\leftarrow \left(\text{plane} = (d, \lambda), \text{cost} = \mathcal{C}_T + \mathcal{C}_I \frac{cost}{\mathcal{A}(\mathcal{B})}\right)$
      end if
      $\lambda = e.offset$
    end if
    if e is *start* event **then**
      $C_L^E \leftarrow C_L^E + 1$
    else if e is *end* event **then**
      $C_R^E \leftarrow C_R^E + 1$
    else
      $C_F^E \leftarrow C_F^E + 1$
    end if
  end for
end function

Algorithm 3.3 Find Optimal Split Plane w.r.t. SAH for KD-tree

```
1: function FindSplit(S, S_B, B)                          ▷ the O(N log² N) variant
2:     bestSplit ← (cost = ∞)
3:     for d ∈ {x, y, z} do                                ▷ The split axis
4:         E ← ∅                                           ▷ The set of events along d
5:         for B ∈ S_B do
6:             if B^d_min = B^d_max then
7:                 E ← E ∪ (B^d_min, flat)
8:             else
9:                 E ← E ∪ {(B^d_min, start), (B^d_max, end)}
10:            end if
11:        end for
12:        SortWrtOffset(E)                                ▷ Use the event's offset as key
13:        curSplit ← LocateSplit(E, d, B, |S|)            ▷ see Algorithm 3.2
14:        if curSplit.cost < bestSplit.cost then
15:            bestSplit ← curSplit
16:        end if
17:    end for
18:    return bestSplit
19: end function
```

at a position v as a single new event. Thus no two new events can be at the same position.

We process the new events in increasing order (w.r.t. their distance to the origin). For each grouped event, we process the elementary events that form it and we count the number of start, end, and flat events. We store these counts in C_L^E, C_R^E, and C_F^E. We update C_L and C_R by adding C_L^E and respectively subtracting C_R^E. After the update, it holds that $C_L = |P_L(v)|$ and $C_R = |P_R(v)|$. As discussed above, we need to add the "flat" primitives to the set corresponding to the node with smaller probability. Thus, we add C_F^E to C_L if v is before the middle of $\mathcal{B}(N)$ and we subtract it from C_R otherwise. At this point, it holds that $C_L = |S_L(v)|$ and $C_R = |S_R(v)|$ and we can use these to compute the SAH cost.

Pseudo code for the above algorithm can be found in Algorithm 3.2. Furthermore Algorithm 3.3 shows how the $O(N \log^2 N)$ construction algorithm extracts the events for a node. Note that Algorithm 3.2 is the same for both the $O(N \log N)$ and the $O(N \log^2 N)$ construction algorithms.

3.4 Construction of BVHs

Construction of BVHs follows a recursive pattern similar to the one of KD-trees. The major difference is that the children of a node in a BVH partition the set of objects associated with that node, whereas in spatial hierarchies the children of a node partition its corresponding region of space space. This leads to a substantial difference in the problem that the two construction processes have to solve at each node.

Space partitioning implies that the children of a node are separated by the surface they contact at. Thus, the job of a construction algorithm for a spatial hierarchy is to find that surface. The KD tree construction algorithms described above search for a separating hyperplane according to a cost criteria. To find that plane, they need to explore $O(|S(N)|)$ possible locations.

A BVH on the other hand has to partition the set of primitives corresponding to the node. It needs to find S_L and S_R, such that $S(N) = S_L \oplus S_R$. Furthermore, in the context of ray tracing, this partitioning has to minimize a cost function (typically SAH). Since there are exactly $2^{|S(N)|-1}$ ways to partition $S(N)$ into two sets, finding an optimal partitioning is in the general case NP-hard, as the algorithm needs to evaluate the cost function for each possible partition.

One way to construct BVHs automatically is proposed in [Goldsmith and Salmon, 1987]. This method builds the BVH by adding the primitives incrementally to it. The leaf where the primitive is inserted is chosen such that the total surface area grows least. This method is based on heuristics and measurements show that the BVHs produced by it are inferior to other acceleration structures [Havran, 2000, Appendix E]. Thus, the method is seldom used in practice.

To overcome the exponential nature of cost-based construction, Wald et al. [Wald et al., 2007] propose to replace each primitive with the center of its AABB (a.k.a. *centroid*) and to transform the problem into a space partitioning one. To do so, they search for a plane that is perpendicular to one of the major axes and separates the centroids spatially. Primitives with centroid to the left of the plane are put in S_L (in the left child) and those with a centroid to the right – in S_R. The goal is to find the plane position that minimizes the SAH, although other cost functions (such as median split and split in the middle) could be used as well. The intuition behind the method is that if a centroid is on one side of the plane then more than half of the primitive (more precisely of its AABB) will be on that side too. This is an approximation of the problem, which however yields good results in practice. Furthermore, with the exception of the above methods and our new algorithms discussed in Chapter 6, we are not aware of any other method that can construct cost-based BVHs in reasonable time. From now on we will refer to this approach as classical BVH construction.

3.4.1 Searching for the Optimal Split Plane

The procedure of searching for the optimal split plane position for BVHs is very similar to the one introduced in Section 3.3.1. As in KD tree construction, best results are again obtained by using the SAH as cost function and all three dimensions need to be examined.

For efficiency, we again use the plane sweep algorithm. If the sweep plane is positioned at offset v along the axis d we need to know $|S_L(v)|$, $|S_R(v)|$, $|\mathcal{A}(N_L(v))|$, and $|\mathcal{A}(N_R(v))|$ in order to compute the SAH cost. The first two can be computed by incrementally counting the number of centroids to left respectively right of the sweep plane. To compute the second pair, we need to keep track of $\mathcal{B}(S_L(v))$ and $\mathcal{B}(S_R(v))$ – the tightest AABBs around all primitives in $S_L(v)$ and $S_R(v)$ respectively. As we will see, we can also do this incrementally. Thus, the status of the sweep plane is composed of $C_L = |S_L(v)|$, $C_R = |S_R(v)|$, $\mathcal{B}_L = \mathcal{B}(S_L(v))$, and $\mathcal{B}_R = \mathcal{B}(S_R(v))$. Since

Algorithm 3.4 Optimal Split Plane w.r.t. SAH for BVH

1: **function** LocateSplitPlaneBVH($d, S_C, S_\mathcal{B}$)
▷ finds the split plane along d
▷ $S_C \equiv$ the set of centroids sorted along d
▷ $S_\mathcal{B} \equiv$ the AABBs of the primitives
2: $bestSplit \leftarrow (\text{cost} = \infty)$
3: $C_L \leftarrow 0, C_R \leftarrow |S_C|$
4: $\mathcal{B}_L, \mathcal{B}_R \leftarrow$ empty AABB
5: $\mathcal{B}_A \leftarrow$ array holding $|S_C|$ AABBs
6: **for** $i = |S_C|..1$ **do**
7: $\mathcal{B}_A[i] \leftarrow \mathcal{B}_R$
8: $\mathcal{B}_R \leftarrow$ ExtendAABB $(\mathcal{B}_R, S_\mathcal{B}[i])$
9: **end for**
10: **for** $i = 1..|S_C|$ **do**
11: $cost \leftarrow \mathcal{A}(\mathcal{B}_L)C_L + \mathcal{A}(\mathcal{B}_A[i])C_R$
12: **if** $cost < bestSplit.\text{cost}$ **then**
13: $bestSplit \leftarrow \left(\text{plane} = (d, S_C^d[i]), \text{cost} = \mathcal{C}_T + \mathcal{C}_I \frac{cost}{\mathcal{A}(\mathcal{B}_R)}\right)$
14: **end if**
15: $C_L \leftarrow C_L + 1$
16: $C_R \leftarrow C_R - 1$
17: $\mathcal{B}_L \leftarrow$ ExtendAABB $(\mathcal{B}_L, S_\mathcal{B}[i])$
18: **end for**
19: **return** $bestSplit$
20: **end function**

all four variables of the status depend only on $S_L(v)$ and $S_R(v)$, and the latter two only change when the sweep plane passes through a centroid, the events are chosen to be the offsets $C^d(p)$ of the centroids $C(p)$ along the axis d. The objective function is $\underset{SAH}{Exp}(v)$ from equation (3.3).

To design the rules for updating the status of the sweep plane, we will look at what happens when the sweep plane moves beyond an event v_i. Let v_i correspond to the centroid of the primitive p. If the sweep plane is before v_i, p belongs to S_R. Once the sweep plane passes through v_i, p is moved from S_R into S_L. Thus, C_L is increased by one and C_R is decreased by one when the sweep plane passes over an event. The initial values are again $C_L = 0$ and $C_R = |S(N)|$.

Since p is added to S_L at an event, $\mathcal{B}_L$ can be updated by taking the union of its current value and $\mathcal{B}(p)$. Updating $\mathcal{B}_R$ can not be done incrementally as there is no way to predict the effect of removing p from S_R on $\mathcal{B}_R$. Computing it efficiently requires an additional pre-processing plane sweep pass. This time, the plane moves in the opposite direction (i.e. from $\mathcal{B}^d_{max}(N)$ towards $\mathcal{B}^d_{min}(N)$), the events are the centroids, and the status is $\mathcal{B}' = \mathcal{B}(S_R(v))$. In this direction, passing over an event corresponding to the primitive p results in p being taken out from S_L and being put in S_R. Thus the bounding box $\mathcal{B}'$ can again be incrementally computed through an union. The computed values for $\mathcal{B}'$ at each event in this pass are stored inside the

event for later use. During the main plane sweep, C_L, C_R, and $\mathcal{B}_L$ are updated as described above, whereas $\mathcal{B}_R$ becomes the value of $\mathcal{B}'$ that has been stored inside the event.

The pseudo code for the above described algorithm can be found in Algorithm 3.4. The latter expects a list of centroids, sorted along the split axis. As with KD-trees, this list can be sorted independently for each node and each dimension, resulting in $O(N \log^2 N)$ construction. Alternatively, three lists of centroids can be created and sorted prior to construction, and their order can be kept during sifting (similar to Section 3.3.3).

3.5 Summary

We presented several construction algorithms for KD-trees and BVHs in this chapter. Our presentation was rather detailed, as we will build our algorithms in the next three chapters on the ones presented here.

The two presented KD-tree algorithms are still known to produce the trees with the lowest ray intersection cost. Their disadvantage however is the low construction speed. In Chapter 4, we will present our new KD-tree construction algorithm, that is significantly faster then both of them. It will run in $O(N \log N)$ and will be based on an approximation of SAH. Also, the ray tracing performance of its trees will be mostly indistinguishable compared to the two presented algorithms.

Using a similar approach, we will also significantly improve the performance of BVH construction in Chapter 6. Finally, in Chapter 6, we will analyse the construction of acceleration structures in general and we will present a new generic construction algorithm that can build any type of cost based binary tree acceleration structure, including KD-trees and BVHs. There, we will also present a new BVH construction algorithm that builds BVHs with significantly higher traversal performance at a speed comparable to that of KD-tree construction.

Chapter 4

Fast Construction of KD-Trees

In the recent years, ray tracing has become a popular technique for interactive visualization. It has been shown (e.g. [Wald, 2004; Reshetov et al., 2005]) that real-time ray tracing is feasible even on a single computer. Nonetheless, interactive frame rates can only be achieved if a suitable acceleration structure is used (i.e. SAH based). Because of the high cost of acceleration structure construction, interactive ray tracing was previously limited to rigid body scenes mostly, eventually allowing for local deformations (e.g. skinning) in some cases. With our work, initially published in [Popov et al., 2006] and further extended in this chapter, we strive to remove this limitation by accelerating the core construction process. We target construction of high-quality SAH based KD-trees. Together with the similar approach from [Hunt et al., 2006], our approach has served as a basis of a number of fast construction algorithms (see Chapter 10).

4.1 Background

One way of achieving interactive ray tracing of dynamic scenes is to avoid the costly rebuilds in every frame. As discussed in Section 2.4.2, this can be accomplished by exploiting the structure of the animation.

The alternative is to use acceleration structures that are cheap to build, such as for example split-in-the middle KD-trees, grids [Wald et al., 2006b; Reinhard et al., 2000], or bounding interval hierarchies [Wächter and Keller, 2006]. Unfortunately, all these structures are not SAH based. Thus, they can be up to an order of magnitude less efficient in traversal than SAH based structures [Havran, 2000, Appendix E]. They present a trade-off between construction and traversal performance and are useful only for scenarios with relatively low number of rays per frame. Since the construction and traversal times are independent, SAH based structures will always outperform non-SAH ones on animated scenes, above a certain number of rays per frame.

The goal of our work presented here is to accelerate SAH based construction of KD-trees to the point where it becomes competitive to the non-SAH one (i.e. split-in-the-middle). Of course, since such trees will be built according to SAH, they will not be subject to the trade-off discussed above. They will actually be both fast to construct and fast to traverse.

4.2 Bottlenecks in KD-Tree Construction

In this section we analyse the bottlenecks of the previous construction methods, namely the $O(N \log N)$ and $O(N \log^2 N)$ ones, by looking into their two basic components: searching for the split plane and sifting. Based on those analysis, we will develop our new method for fast construction of KD-trees. The exposition will follow our paper [Popov et al., 2006].

Instead of analyzing and trying to improve the worst-case complexity of KD-tree construction, which can not fall below $O(N \log N)$, we will try to improve its execution speed. Thus, we will look into issues such as "cash friendliness" and number of executed instructions. They are commonly ignored in traditional algorithms literature, but they are indeed the reason why algorithms with larger complexity sometimes execute faster than such with lower complexity. Examples related to ray tracing include the $O(N \log N)$ versus the faster in practice $O(N \log^2 N)$ KD-tree construction as well as radix sorting vs. introspection sorting.

In both previous construction algorithms, searching for the split plane involves sorting of events. In the $O(N \log^2 N)$ algorithm, the events are created and sorted at each node. Even though sorting is really efficient, it still consumes more than half of the processor time during construction. The plane sweep itself also takes a considerable fraction of the execution time, mainly because of the cost of evaluating the SAH at each event. In this algorithm, sifting takes only a fraction of the execution time.

The second algorithm, namely the $O(N \log N)$ one from [Wald and Havran, 2006], again sorts the events, but only once at the beginning. Besides the cost computations, what really consumes processor time there is the sifting of events. On one side, the code is complex and requires many operations per event, including expensive dependent memory accesses. On the other, even though the events are sorted along the three axes, their corresponding primitives can be in arbitrary order in memory. Thus, labeling the primitives as belonging to the left/right sub-tree (see Section 3.3.3), is an operation with a random memory access pattern. On modern architectures, such a pattern can slow down a program's execution by an order of magnitude, since the CPU cannot find the data in the caches and has to wait for it to arrive from the "slow" RAM.

Another disadvantage of this algorithm is related to parallelism. Since the label of each primitive has to be stored inside the latter for performance reasons, a parallel implementation becomes relatively complicated. Multiple events can share the same primitive and care has to be taken to duplicate the associated label or to avoid processing those events in parallel.

As discussed above, both construction approaches spend a lot of time in the plane sweep itself. We regard this as a waste of resources, since all of the computed SAH values, with the exception of one, are actually thrown away. To avoid the issue, we first observe that choosing a "wrong" split plane will still create a valid tree, but with a potentially worse quality. Combining that with experimentally derived information about the shape of the SAH cost function, we will develop our new construction algorithm in the next section. It will search for an *approximate* minimum of the SAH and will sample the latter at only few locations. Since our algorithm is not be based

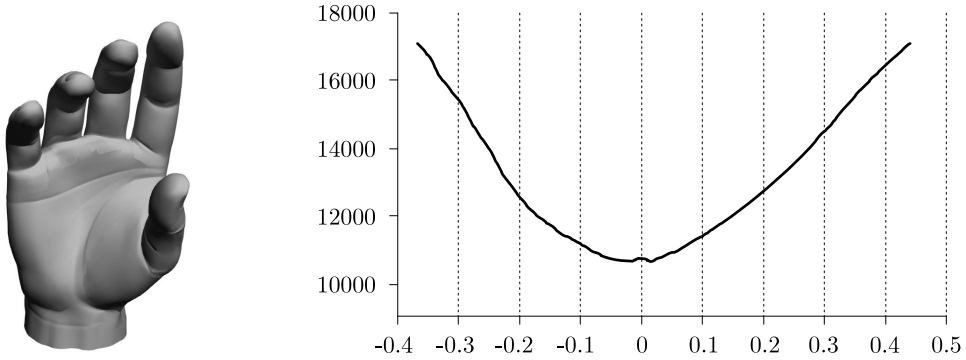


Figure 4.1: *The* HAND *model and a plot of the SAH cost for the root node along the axis Ox.*

on a plane sweep at all, it requires neither sorting, nor an expensive book keeping of sorted event lists.

4.3 Binned Cost Function Sampling

Our new algorithm relies on the fact that the SAH cost is a rather smooth function for nodes with many primitives. To come to this conclusion, we have observed the behaviour of the latter on various scenes, by plotting it for a set of randomly chosen nodes of an already built KD-tree. One such plot can be seen on figure Figure 4.1

Because of the smoothness, we can find an *approximately* optimal location for the split plane through sampling. To do so, we place $M+1 << |S(N)|$ samples regularly in the domain of $\underset{SAH}{Exp}$. The location of sample $i \in \{0, 1, \dots M\}$ is

$$v_i = \mathcal{B}^d_{min}(N) + i \ \frac{\mathcal{B}^d_{max}(N) - \mathcal{B}^d_{min}(N)}{M}$$

with d being the split axis. To evaluate $\underset{SAH}{Exp}(v_i)$, we need to know $|S_L(v_i)|$ and $|S_R(v_i)|$. One way to compute the latter two is by testing each primitive with each sample location. This approach, which we name *brute force*, needs $O(MN)$ time to find the split plane. Thus, for reasonable number of samples, using it would result in slow construction.

Instead, we create M bins, each of which corresponds to the space between two adjacent sampling points: bin B_i with $i \in \{0, 1, \dots M-1\}$ occupies the space between the samples i and $i+1$ (Figure 4.2). We use the bins to count how many primitives start respectively end between two consecutive sampling locations. For this purpose, we store two counters in each bin: B^i_{start} and B^i_{end}. Again, we work only with the part of each primitive that is inside the current node. Thus, we populate the counts by enumerating all bounding boxes $\mathcal{B} \in S_{\mathcal{B}}(N)$ and for each of them we increase B^i_{start} and B^j_{end}, with i and j being such that $v_i \leq \mathcal{B}^d_{min} < v_{i+1}$ and $v_j < \mathcal{B}^d_{max} \leq v_{j+1}$. Primitives that are flat along the split axis are handled by increasing both B^i_{start} and B^i_{end}, with $v_i \leq \mathcal{B}^d_{min} = \mathcal{B}^d_{max} < v_{i+1}$ if the primitive is in the left part of the node's bounding box $\mathcal{B}(N)$ and $v_i < \mathcal{B}^d_{min} = \mathcal{B}^d_{max} \leq v_{i+1}$ otherwise.

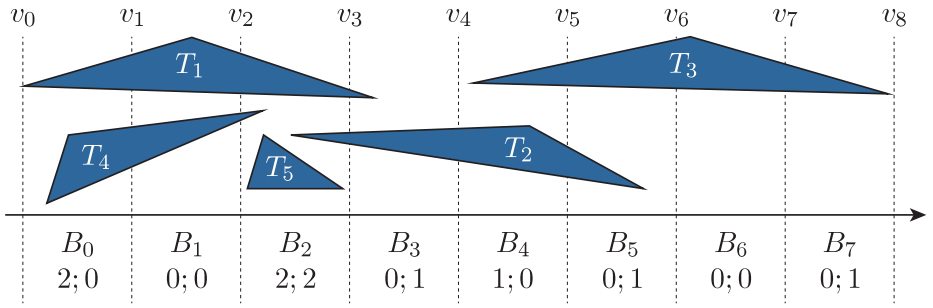


Figure 4.2: *Sampling the cost function. The bins $B_0 \dots B_7$ are located in-between the events $v_0 \dots v_8$. The respective number of primitives that start/end in a bin are given under the bin label, separated with a semicolon.*

To compute $|S_L(v_i)|$ and $|S_R(v_i)|$ for a sampling location v_i, we now observe that the first is actually equal to the number of primitives that start to the left of v_i, whereas the second one is equal to the number of primitives that end to the right of v_i. Thus, it holds that

$$|S_L(v_i)| = \sum_{j=0}^{j<i} B^j_{start}$$

$$|S_R(v_i)| = \sum_{j=i}^{j<M} B^j_{end}$$

We compute the above sums for all sampling locations simultaneously, by using two prefix sums. For $S_L(v_i)$ we do a prefix sum in increasing order over B^i_{start} and for $S_R(v_i)$ – in decreasing order over B^i_{end}. Once we know the two counts for each event position, we can easily calculate the SAH costs there, and we can place the split plane at the sampling position with minimal cost.

Pseudo code for the above described algorithm can be found in Algorithm 4.1. Clearly, our approximation avoids the problems discussed in Section 4.2. We do not need sorting at all, there is no complex book-keeping, and our algorithm performs much less operations per primitive, as the SAH cost is evaluated on a fixed-size independent set of locations. A valid question is how much do we loose in terms of ray tracing performance. As we will see in Section 4.6 the speed loss is negligible as it is always in the range of only few percent.

4.3.1 Sampling Accuracy

By doing measurements on a variety of different scenes, we found that 1024 regularly placed samples are more than enough to approximate the SAH cost with almost perfect quality. Nevertheless, there might be scenes which require greater accuracy. Instead of blindly increasing the sample count, we show here how to adaptively identify the interval that needs more sampling. We base our approach on a *conservative* estimate of the approximation error.

Algorithm 4.1 Optimal Split Plane for KD-tree through Binning

function LocateSplitPlane($d, S_{\mathcal{B}}, \mathcal{B}(N), M$)
▷ finds the split plane along d
▷ $S_{\mathcal{B}} \equiv$ the AABBs of the primitives
▷ $\mathcal{B}(N) \equiv$ the AABB of the node
▷ $M+1 \equiv$ the number of sample locations
  $B_{start},\ B_{end} \leftarrow$ arrays $[-1 \dots M]$ initialized to 0
  for all $\mathcal{B} \in S_{\mathcal{B}}$ **do**
    $i = \left\lfloor M \dfrac{\mathcal{B}^d_{min} - \mathcal{B}^d_{min}(N)}{\mathcal{B}^d_{max}(N) - \mathcal{B}^d_{min}(N)} \right\rfloor$
    $j_f = M \dfrac{\mathcal{B}^d_{max} - \mathcal{B}^d_{min}(N)}{\mathcal{B}^d_{max}(N) - \mathcal{B}^d_{min}(N)}$
    $j = \lfloor j_f \rfloor$
    if $j = j_f$ **then** $j = j - 1$ ▷ ensure that $v_j < \mathcal{B}^d_{max} \leq v_{j+1}$
    if $\mathcal{B}^d_{min} \neq \mathcal{B}^d_{max}$ **then** ▷ primitive is not flat
      $B^i_{start} = B^i_{start} + 1;\ B^j_{end} = B^j_{end} + 1$
    else ▷ flat primitive
      if $i < \frac{M}{2}$ **then** $k = j$ **else** $k = i$
      $B^k_{start} = B^k_{start} + 1;\ B^k_{end} = B^k_{end} + 1$
    end if
  end for
  for $i = 0 \dots M$ **do** $B^i_{start} = B^i_{start} + B^{i-1}_{start}$
  for $i = M-1 \dots -1$ **do** $B^i_{end} = B^i_{end} + B^{i+1}_{end}$

  $M \leftarrow \mathcal{B}^{d+1}_{size}(N)\mathcal{B}^{d+2}_{size}(N),\quad A \leftarrow \mathcal{B}^{d+1}_{size}(N) + \mathcal{B}^{d+2}_{size}(N)$
  $C^L_1 \leftarrow 2\left(M - \mathcal{B}^d_{size}A\right),\ C^2_L \leftarrow 2A$ ▷ see (3.8)
  $C^R_1 \leftarrow 2\left(M + \mathcal{B}^d_{size}A\right),\ C^2_L \leftarrow -2A$ ▷ see (3.9)
  $bestSplit \leftarrow (\text{cost} = \infty)$
  for $i = 0 \dots M$ **do**
    $v \leftarrow B^d_{min}(N) + i\ \dfrac{B^d_{max}(N) - B^d_{min}(N)}{M}$
    $cost \leftarrow B^{i-1}_{start}\left(C^L_1 + vC^L_2\right) + B^i_{end}\left(C^R_1 + vC^R_2\right)$ ▷ see (3.10)
    if $cost <$ *bestSplit*.cost **then**
      $bestSplit \leftarrow \left(\text{plane} = (d, v), \text{cost} = \mathcal{C}_T + \mathcal{C}_I \dfrac{cost}{\mathcal{A}\left(\mathcal{B}(N)\right)}\right)$
    end if
  end for
  return *bestSplit*
end function

We look again at (3.10) – the definition of SAH w.r.t. to the split plane position v. This function can be expressed as

$$\begin{aligned} \underset{SAH}{Exp}(v) &= \mathcal{C}_T + \mathcal{C}_I \frac{f_L(v) + f_R(v)}{\mathcal{A}(N)} \\ f_L(v) &= \big(C_1^L(N) + vC_2^L(N)\big)|S_L(v)| \\ f_R(v) &= \big(C_1^R(N) + vC_2^R(N)\big)|S_R(v)| \end{aligned}$$

By looking at (3.6) and (3.7), it can be seen that $f_L(v)$ is a monotonically increasing function whereas $f_R(v)$ is monotonically decreasing and both functions are positive. Thus, for each pair of numbers $v' < v''$, it holds that

$$\min_{v\in[v',v'']} \left(\underset{SAH}{Exp}(v) \right) \geq \mathcal{M}(v', v'') = \mathcal{C}_T + \mathcal{C}_I \frac{f_L(v') + f_R(v'')}{\mathcal{A}(N)} \tag{4.1}$$

The above inequality follows from the fact that $f_L(v) > f_L(v')$ and $f_R(v) > f_R(v'')$ for each $v : v' \leq v \leq v''$.

We can now use (4.1) to improve the sampling accuracy of our method. Assume that we have already found an initial value $\mathcal{M}$ for the minimum, by regularly placing M samples. Then, when we place the additional samples, there is no need to regard intervals where $\mathcal{M}(v_i, v_{i+1}) > \mathcal{M}$ with $\{v_i\}$ being the set of old samples. Since we need to place the new samples regularly, we identify two numbers $i < j$, such that $\mathcal{M}(v_0, v_i) > \mathcal{M}$, $\mathcal{M}(v_j, v_M) > \mathcal{M}$, $\mathcal{M}(v_i, v_{i+1}) \leq \mathcal{M}$, and $\mathcal{M}(v_{j-1}, v_j) \leq \mathcal{M}$. We then place the new samples regularly in the domain $[v_i, v_j]$.

In our tests, the average value of $j - i$ for $M = 1024$ over all nodes was below 2 for all scenes and the maximum did not exceed 10. Thus, with two step refinement, we were able to obtain accuracy comparable to placing approximately 1 million samples on average.

4.4 Processing the Lower Tree Levels

The approach from Section 4.3 becomes inefficient once the ratio of sampling positions to primitives in a node rises above a certain threshold. In this case, we can either reduce the number of samples, by calculating it as a function of the number of primitives, or we can switch to classic construction as we did in our paper [Popov et al., 2006].

4.4.1 Improving Classical Construction

At the point of switching from sampling to exact SAH evaluation, the set of primitives we have to work with is rather small. Most importantly, it is small enough to fit in the caches of the CPU. To keep the $O(N \log N)$ complexity of the algorithm, in our paper we used the $O(N \log N)$ construction method from Section 3.3.3 . Because all

data fits in the cache of the processor, the method does not result in cache trashing. This, combined with the fact that the number of primitives was in the order of few thousand, allowed us to use efficient radix sorting for the event lists. We also proposed a scheme for incrementally evaluating the cost function, which reduced the number of operations required to evaluate it at an event. The improvement was rather small, so we are not going to discuss it here. Rather, we are going to show a new alternative method, which uses sampling all the way down to the leafs and thus achieves better construction times. This method, as well as the in-place sifting from 4.5.3, have not been previously published.

4.4.2 Brute-Force Sampling

Profiling the algorithm from our paper, suggested that most of the construction time is spent in the second phase (i.e. after switching to classical construction). On the other hand, the results presented in [Hunt et al., 2006], suggest that using sampling all the way down to the leafs still leads to well optimized KD-trees. Thus, we are going to present a new algorithm here that only relies on sampling.

The first change that we introduce is that the number of samples is no longer a constant. Instead, we use a bounded linear function

$$\textit{sample-count}(x) = \min(C_{max},\ \max(C_{min},\ Ax + B))$$

with x being the number of primitives in the node.

The second change is that once the number of primitives falls below C_{min}, we switch to exact SAH evaluation. For this, we create samples at the boundaries of the primitives inside the node and then we use *brute-force* sampling. For every sample, we loop over all primitives in the node, keeping track of the count of primitives to left and right of the sample position. We then use these counts to compute the SAH at the sample.

By running measurements on a variety of scenes, we have discovered $A = 1$, $B = 0$, $C_{max} = 256$, and $C_{min} = 23$ to be good values for an Intel Core2 Duo architecture. The threshold $C_{min} = 23$ for switching to brute force was chosen to be the point where brute-force sampling becomes more effective than binned sampling. We chose $C_{max} = 256$ instead of the 1024 from our paper, so that the sample counters can fit in the L1 caches of the CPU. As we will see in Section 4.6, this construction method is considerably faster than the one proposed in our paper, and the traversal cost is almost the same.

4.5 Implementation Details

An important issue while implementing KD-tree construction is the memory management. During construction, an implementation has to keep for each to-be-constructed node N at least two sets in memory: $S(N)$ and $S_{\mathcal{B}}(N)$. Since they both are split in two while processing a node, an implementation has to allocate new memory for every node. This could be achieved through the operating system's memory allocation routines. However, since these allocations happen at *each* node (and their count is $O(N)$), relying on the OS will become the major bottleneck for non-trivial scenes.

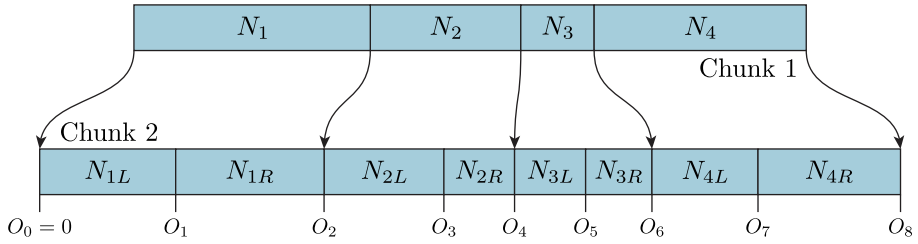


Figure 4.3: *Memory layout for breadth-first construction. The offsets O_j in chunk 2 are computed as $O_i = O_{i-1} + |S_{i-1,R}|$ for i even, and as $O_i = O_{i-1} + |S_{i,L}|$ for i odd.*

The usual approach of most implementations is to allocate a "big enough" chunk of memory and to manage it on their own. As general-purpose memory management can be quite complex and slow, we will describe below several strategies for efficient memory management for breadth-first and depth-first construction.

4.5.1 Memory Management for Breadth-First Construction

In breadth-first construction, the unprocessed nodes are kept in a queue. Whenever a node is created, its children are scheduled for construction, by adding them to the queue. Thus, a node at depth d in the tree is only processed if all nodes with depths smaller than d have also been processed. We use this fact to devise an efficient memory management scheme. We process the nodes in groups, one tree level at a time. We keep two memory chunks. The first one contains the data (e.g. $S(N)$ and $S_{\mathcal{B}}(N)$) necessary for processing the nodes of the current level. We use the second one to store the data required to construct the nodes for the next level. Once we finish processing a level, we continue to the next level, using the data from the second chunk.

Processing of a level proceeds in three phases (see Figure 4.3). First, we compute the optimal split plane, the cost of the split and the number of primitives $C_L(N)$ and $C_R(N)$ in the left and right child respectively, for each node of the current level. Then, we create the memory layout for the next level, by performing an exclusive prefix sum over $C_L(N)$ and $C_R(N)$, assuming that the two are zero for leaf nodes. Finally, we sift the primitives of each node N of the current level, and we store the required data (e.g. $S(N)$ and $S_{\mathcal{B}}(N)$) according to the computed layout.

With this scheme, we still need to re-allocate memory, however only once per level (resulting in $O(\log N)$ allocations). To avoid memory fragmentation (which can lead to out-of-memory errors), we further improve our allocation scheme by pre-allocating the two chunks of memory. Then, after a level is complete, we simply swap the pointers to the chunks. We only re-allocate a chunk if its size is not large enough to hold the new data and we only do that after the layout phase. We have found experimentally, that that chunks with initial size $5N$ for N primitives do not cause re-allocations for the tested scenes.

4.5.2 Memory Management for Depth-First Construction

In our paper [Popov et al., 2006], we used breadth-first construction with the above scheme for managing memory. The alternative is depth first construction, which has the advantage that all of the data required to build a sub-tree will fit in the caches after some level. We present below three efficient memory management schemes, including our new and unpublished in-place sifting approach.

To manage allocations, we again allocate a chunk of memory and we define two operation on it: tail allocation and tail de-allocation. We keep an integer variable v_s, giving the size of allocated memory from the chunk. All data in the memory at offsets between 0 and v_i is marked as "in-use", while all memory after v_s is marked as free. A tail allocation of N bytes reserves the memory area $v_s \dots (v_s + N)$ and increments v_s by N. If there is not enough space left in the chunk, we reallocate it, increasing its size by at least the required amount of space, but with no less than 50% of its previous size. A tail de-allocation of N bytes simply decrements v_s.

With those two operations, we can significantly reduce the number of required allocations. Whenever we process a node, we tail-allocate the space required for its children, we process the children recursively, and upon return, we tail-deallocate the children's memory.

Since the number of primitives straddling the split plane of a node $\mathcal{N}$ is on average $O(\sqrt{|S(\mathcal{N})|})$ (see [Wald and Havran, 2006]), the maximum memory of the chunk should not be larger than $T(N)$, with N being the number of primitives in the scene and $T(x) = 2x + T(\frac{x}{2}) + \sqrt{x}$. Solving the recurrent relation, and assuming that $N > 1200$, we obtain

$$T(N) = -5 - \sqrt{2} + \left(2 + \sqrt{2}\right)\sqrt{N} + 4N < 4.1N$$

Thus, using a size for the initial chunk, large enough to hold the data for $4.1N$ primitives, is likely to avoid re-allocations completely in the general case. We have also confirmed that experimentally for all scenes we have tested.

The memory required by the above described approach can effectively limit the size of the scenes that can be constructed in memory. We can reduce this memory by observing the following fact. In normal construction, the data required to build the sub-trees of a node has to stay until both sub-trees are built. If we however reverse the order in which we process the nodes (i.e. right node first), we can tail-deallocate the memory for the right sub-tree immediately after it is processed. Note that this only holds, if we have tail-allocated the left sub-tree's data before the right one's. The recurrence relation is than changed to $T(x) = T(x) = \frac{3}{2}x + T(\frac{x}{2}) + \sqrt{x}$ and the memory requirements (assuming that $N > 1200$) are reduced to

$$T(N) = -4 - \sqrt{2} + \left(2 + \sqrt{2}\right)\sqrt{N} + 3N < 3.1N$$

4.5.3 In-Place Sifting

To further reduce the required memory, we propose a new quick-sort-style in-place sifting. With it, we completely reuse the memory space of a node's data, overwriting it with the data of both children.

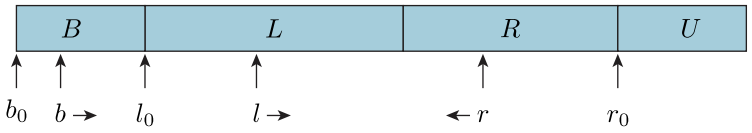


Figure 4.4: *The desired memory layout targeted by in-place sifting, the pointers used while sifting (see also Algorithm 4.2) and the direction they move in.*

If no primitive straddles the split plane, we could sift the primitives using Hoare's partition [Cormen et al., 2009] method. For that purpose, we need to keep two pointers l and r, pointing initially to the start respectively end of the chunk of data allocated for a node. Hoare's partition method increases l in a loop as long as it points to a primitive that has to go left. It then decreases r as long as it points to a primitive belonging to the right. Once l points to a primitive that has to go right and r – to one that has to go left, the method swaps the elements pointed by l and r. The two loops and the swap are repeated as long as l points to a location before r in memory. Hoare's partition runs in $O(N)$.

In the general case, we can not apply Hoare's partition directly, as the primitives may straddle the split plane. We first have to take into account that the combined children's data is larger then that of their parent and we need to increase the storage for the node accordingly. By visiting the children in an order that is opposite to the order of their data in memory (as in Section 4.5.2), we know that the data of the currently processed node will be the last one in the pre-allocated chunk. Thus, we can tail-reallocate the data of the node by simply increasing v_s. The amount to add to the latter is exactly $|S_L|+|S_R|-|S(N)|$ and it is equal to the number of primitives that straddle the split plane.

Having enough storage for both children, we have to modify Hoare's partition to account for the duplicated primitives. Our goal is to re-order the data of the node so that it achieves a $B|L|R|U$ layout in memory (see Figure 4.4). Here B is the data for all primitives that straddle the split plane, L is the data for all primitives that are completely to the left, R – for those to the right, and U is the extra (and uninitialized) memory that we have appended. Once we achieve this order, we can complete the sifting process by copying B onto U and then clipping the bounding boxes of all primitives in B and U to the AABB of the parent (Section 3.3).

We modify Hoare's partition method by introducing a third pointer b, used to account for the duplicated primitives. Initially, we set b to point to the beginning of the node's chunk, l – to where the first left-only primitive will be stored (i.e. $l = b + |S_L| + |S_R| - |S(N)|$), and r – to the end of the node's chunk. As in Hoare's partition, we increment l until we encounter a primitive that has to go in the right child, and we decrement r until we encounter a primitive that has to go in the left child. The difference comes when handling primitives that are shared by both children. If l points to such a primitive, it means that there is at least one primitive in the region B that does not straddle the split plane. Thus in this case, we first increase b as long as it points to a shared primitive. Once it starts pointing to a primitive that is completely to the left or right of the sweep plane, we swap its pointed value with the one pointed to by l. We perform the same steps for r. These

Algorithm 4.2 In-Place Sifting

1: **function** SiftInPlace($M, v_s, C_L, C_R, |S(N)|$)
▷ sifts the primitives in-place
▷ $M \equiv$ the memory used for construction
▷ $v_s \equiv$ how much of M is full (Section 4.5.2)
▷ $C_L, C_R \equiv$ # of primitives in the left/right child.
▷ $|S(N)| \equiv$ # of primitives in the node.
2: $v'_s \leftarrow v_s + C_L + C_R - |S(N)|$
3: **if** $v'_s > |M|$ **then**
4: $M \leftarrow$ ReAllocate $\left(M, \max\left(v'_s, \frac{3}{2}|M|\right)\right)$
5: **end if**
6: $b, b_0 \leftarrow$ AddressOf$(M[0]) + v_s - |S(N)|$
7: $l, l_0 \leftarrow b + C_L + C_R - |S(N)|$
8: $r, r_0 \leftarrow$ AddressOf$(M[v_s - 1])$
9: $u \leftarrow r$
10: $v_s = v'_s$
11: **loop**
12: **while** $l \leq r_0 \ \wedge$ $M[l]$ is not right of the split plane **do**
13: **if** $M[l]$ straddles split plane **then**
14: **while** $b < l_0 \ \wedge$ $M[b]$ straddles split plane **do** $b \leftarrow b + 1$
15: Swap $(M[b], M[l])$
16: **else**
17: $l \leftarrow l + 1$
18: **end if**
19: **end while**
20: **while** $r \geq l_0 \ \wedge$ $M[r]$ is not left of the split plane **do**
21: **if** $M[r]$ straddles split plane **then**
22: **while** $b < l_0 \ \wedge$ $M[b]$ straddles split plane **do** $b \leftarrow b + 1$
23: Swap $(M[b], M[r])$
24: **else**
25: $r \leftarrow r - 1$
26: **end if**
27: **end while**
28: **if** $r \leq l$ **then break**
29: Swap $(M[l], M[r])$
30: **end loop**
31: Copy $M[b_0 \ldots l_0 - 1]$ to $M[r_0 \ldots v_s - 1]$
32: ClipPrimitives$(M[b_0 \ldots l_0 - 1])$
33: ClipPrimitives$(M[r_0 \ldots v_s - 1])$
34: **end function**

steps restore the property that l and r point to primitives that are completely on one side of the split plane. Thus, we can continue with the standard Hoare partition. Algorithm 4.2 summarizes our in-place sifting approach.

To give a good guess on the initial chunk size, we observe that the memory required to build a tree will be approximately $T(N) = N + d\sqrt{N}$, with d being the depth of the tree. This is due to the memory reuse, and because the average number of the primitives straddling the split plane is $O(N)$ (see [Wald and Havran, 2006]). Considering that $d = 1.2 \log_2 N + 2$ is a good estimate for the tree depth (see [Havran, 2000, Section 4.5]), we obtain (for $N > 1200$)

$$T(N) \approx 2 + N + 1.2 \log_2 N \sqrt{N} < 1.4N \tag{4.2}$$

Using the above estimate, we were able to construct all scenes from Section 4.6 without the need of resizing the initial memory chunk. Furthermore, besides the low memory usage, a strong advantage of in-place sifting is its cache friendliness. On one side, the memory is reused, and on the other – the working set for a sub-tree becomes much smaller.

4.5.4 Numerical Stability

An important issue of binning, when combined with in-place sifting, is the numerical stability. For any interval $[a, b]$, the sampling positions when using M bins can be calculated as (see also Algorithm 4.1, line 23)

$$v_i = a + i \, \frac{b - a}{M} \tag{4.3}$$

To calculate the bin into which a point x falls, we can use (see also Algorithm 4.1, line 4)

$$i(x) = \left\lfloor M \, \frac{x - a}{b - a} \right\rfloor \tag{4.4}$$

The issue here is, that there is no guarantee that $x \in \big[v_{i(x)}, v_{i(x)+1}\big)$, due to the rounding of floating point operations. It might happen that a primitive has been counted on both sides during binning, but actually ends up on only one of the sides during sifting. Thus the counts we compute for $|S_L|$ and $|S_R|$ might actually be incorrect, leading to an incorrect allocation size for the chunk during sifting. Even worse, the pointers l, r and b can not be initialized correctly, which effectively makes in-place sifting impossible.

Fortunately, the issue can be solved by changing the rounding mode of the floating point processor. It can be proven (see Section A.1) that if $i(x)$ is calculated in round-down mode and v_i is calculated according to equation (4.3) using round-up mode, then indeed $x \in [v_{i(x)}, v_{i(x)+1})$.

Since changing the rounding mode of the CPU too often would hurt the build performance considerably (as the CPU pipeline needs to be flushed), we use instead both the SSE and the FP87 units of the CPU simultaneously in our implementation. We use round-down for the SSE unit and we instruct the compiler to perform all floating point operations on this unit. To compute the sample positions, we use the FP87 unit in round-up mode, programming the calculations manually in assembly

language. Note that this approach does not work on all architectures (and especially on x64). A solution there is to check the bounds during binning, which however decreases the efficiency of the algorithm.

4.5.5 Parallel and Lazy Construction

Our sampling construction approach also improves two further important aspects of KD-tree construction: parallelism and on-demand (a.k.a. lazy) construction. With our method, they become much more efficient and easier to implement.

In our paper, we present a parallel implementation that uses both data parallelism (through SIMD) and task-based parallelism (through multi-threading). In the first case, we process all three dimensions simultaneously during binning, and we also apply this to the brute-force sampling. To this end, we map each point/vector to the first three components of the SIMD unit. In the second case, we process independent sub-trees with different threads.

To create enough work for the multi-threaded build, we initially start by using single threaded breadth first construction. During that phase we postpone nodes that have fewer than a user specified number of primitives by putting them on a queue. At the end of this pass, we have a large queue of nodes, whose trees can be built independently. At this point we start a user specified amount of threads, each of which fetches nodes from the queue in a loop and fully constructs their sub-trees.

Even though much of the construction time is spent exactly in those small sub-trees that we postpone, as expected our implementation can not achieve linear speed-up because of the initial serial phase. This phase is required however, in order to distribute the tasks evenly among the threads. Thus, we have proposed a way to make it parallel as well.

The idea is to let all threads work cooperatively on the same node. This can be achieved by partitioning the set of corresponding primitives into $S(N) = S_1(N) \oplus S_2(N) \oplus \dots S_K(N)$ and assigning thread i to the set $S_i(N)$. Each thread i will then use binning and compute the counts of primitives to the left/right of each sample location as described in Section 4.3, but only for the primitives in $S_i(N)$. Since the sample locations for all threads are the same, the counts of primitives to the left/right of the samples can be computed by merging the counts collected by the threads. This is also where our algorithm has an advantage over the previous ones: Because of the plane sweep there, processing a node cooperatively by many threads is not as efficient and is much harder to implement.

Once the primitive counts for the samples are known, the split plane position can be computed and the primitives can be sifted. The latter could probably also be done using multiple threads. However, since the nodes at this stage are rather large and their corresponding sets do not fit in the CPU caches, and since sifting in our algorithm requires very few operations per primitive, it is actually bandwidth limited. Thus, on many architectures, it might not make sense to use multiple threads for this purpose.

The second large optimization, that can be applied to kd-tree construction, is on-demand construction. With it, the tree is constructed during traversal and a sub-tree is only constructed if a ray needs to traverse it. On-demand construction

can be applied to the classical construction algorithms too. Using sampling however brings an additional advantage: the events of the nodes do not need to be sorted at all. Thus, whereas the classical algorithms would waste CPU cycles to fully sort the events for a node that will never be traversed (while processing its parent), our sampling approach doesn't. Note that for both the $O(N \log N)$ and the $O(N \log^2 N)$ algorithms, the events of all primitives will be sorted at least one time, namely in the root node.

4.6 Results

In this sections we compare the results of running the $O(N \log N)$ builder, our builder from [Popov et al., 2006] and the builder presented in Section 4.4.2. We use the following test scenes: Shirley6, Hand, Sponza, Bunny, Fairy Forest, Conference, Buddha, and Soda Hall. The results are summarized in table 4.1. The measurements were done on a Lenovo T61P notebook, with a Core 2 Duo T9330 processor, running at 2.5Ghz and with 32K L1 data cache and 6MB L2 cache. Beside the build times, we also give the performance characteristics of the built trees: namely the average traversal cost $\mathcal{C}_{trav}$, and the surface area metric cost $Exp(T)$. For $\mathcal{C}_{trav}$ we shoot 10^6 random rays, originating inside the scene and with uniformly random directions, and we average the cost of all of them. To form the cost of a single ray, we look at the number of nodes it has traversed and the number of intersections it has performed and then we sum the above two, weighed respectively by $\mathcal{C}_T$ and $\mathcal{C}_I$. Finally, we provide in the table the speed-up factor for construction and the drop in quality as percentage for each method.

In this chapter, we try to focus on the core construction algorithm. Thus, we only provide results from a serial implementation of our new algorithm, as we see the task of implementing it in parallel as an orthogonal addition. Results from a parallel implementation of our initial binning algorithm can be found in our paper [Popov et al., 2006], and a much improved implementation is available in [Shevtsov et al., 2007].

In the table we compare the build algorithms on two versions of the geometry of each scene: the original one and a shuffled one. In the second we simply shuffle the order of the triangles. By doing this we try to achieve a worst case scenario for the input geometry so we can show the true strengths and weaknesses of each algorithm.

As it can be seen from the table, the sampling algorithm from our paper does not really increase the traversal cost. In all cases, the increase is below 2% in both the surface area metric cost and the the average ray traversal cost. The increase in construction speed is rather small for the smaller models and the improvement comes primarily because we use radix sorting for the events and incremental SAH calculation. Only once the model is big enough does the strength of sampling really become apparent: for the 1.1M triangle Buddha, we were able to achieve a maximum speedup of 63% and for the 2M triangle Soda Hall – a speedup of 50%.

Our new algorithm from Section 4.4.2 on the other hand shows a significantly improved speed-up even for the small models. It achieves on average a 100% speedup and the decrease in traversal performance is rather small: a maximum of 9% when

measured as surface area cost and a maximum of 7% and average of 3% if measured as the more relevant average traversal cost.

Unfortunately, we were not able to compare our sampling algorithms to the one from [Hunt et al., 2006]. Eventhough their results were measured on a similar Core 2 Duo system and even though we have 4 scenes in common, the trees constructed by both algorithms seem to be quite different. For the FAIRY FOREST model, we can not directly compare neither construction speed, nor surface area cost, as they work on quads directly and thus have to handle only half of the geometry that we do. For the SODA HALL model, their construction is considerably faster, but they produce trees that are four times slower to traverse than ours. In the BUNNY and HAND scenes they achieve better absolute performance and for the first one also better traversal cost. Unfortunately, even though we implemented their algorithm, we were not able to reproduce their results. Furthermore, our implementation of their algorithm was always slower then our new sampling approach algorithm from 4.4.2.

4.7 Summary

In this chapter we showed two novel approaches for fast KD-tree construction based on SAH cost sampling. The first approach was taken from our paper [Popov et al., 2006] and runs in $O(N \log N)$. It uses binning to compute the SAH cost at the sampling positions for the higher tree levels and thus avoids the expensive sorting and book keeping of the classical construction algorithms. For the lower tree levels it switches to classical $O(N \log N)$ construction as all data fits in the CPU caches. The second algorithm uses sampling all the way down to the leafs. To calculate the SAH cost it uses binning up to a certain point and then switches to exact cost calculation by using brute force sampling, and by allocating the samples on the boundaries of the objects. We also introduced several schemes for efficient memory management during construction.

		$O(N \log N)$			Sampling Old					Sampling New				
Scene	N	T_B	Exp	$\mathcal{C}_{trav}$	T_B	Exp	R_S	R_{Q_1}	R_{Q2}	T_B	Exp	R_S	R_{Q_1}	R_{Q2}
Shirley6	804	3.6ms	28.9	16.5	3.3ms	28.9	9%	0.0%	0.0%	1.9ms	28.3	90%	-2.0%	-2.5%
Shirley6$_S$	804	3.8ms	28.9	16.5	3.6ms	28.9	5%	0.0%	0.1%	2.0ms	28.3	91%	-2.0%	-2.5%
Hand	17.1K	0.08s	69.2	28.2	0.08s	69.2	1%	0.0%	0.0%	0.04s	75.5	120%	8.4%	5.8%
Hand$_S$	17.1K	0.08s	69.2	28.2	0.08s	69.2	-2%	0.0%	0.0%	0.04s	75.5	113%	8.4%	5.8%
Sponza	67.5K	0.29s	109.0	28.1	0.28s	110.1	2%	0.9%	0.8%	0.22s	113.9	31%	4.3%	7.3%
Sponza$_S$	67.5K	0.31s	109.0	28.1	0.31s	110.1	1%	0.9%	0.8%	0.24s	113.9	32%	4.3%	7.3%
Bunny	69.5K	0.41s	94.1	32.8	0.37s	95.0	11%	0.9%	0.9%	0.20s	99.4	106%	5.3%	3.8%
Bunny$_S$	69.5K	0.42s	94.1	32.8	0.39s	95.0	9%	0.9%	0.9%	0.21s	99.4	104%	5.3%	3.8%
F.Forest	191.8K	0.54s	61.4	30.7	0.48s	61.8	14%	0.6%	1.2%	0.24s	67.5	124%	9.1%	2.0%
F.Forest$_S$	191.8K	0.56s	61.4	30.7	0.49s	61.8	14%	0.6%	1.2%	0.26s	67.5	117%	9.1%	2.0%
Conference	282.7K	1.13s	78.1	32.1	1.01s	79.2	12%	1.4%	0.9%	0.61s	82.1	87%	4.9%	1.3%
Conference$_S$	282.7K	1.35s	78.1	32.1	1.08s	79.2	25%	1.4%	0.9%	0.67s	82.1	102%	4.9%	1.3%
Buddha	1.1M	4.55s	142.3	44.9	3.63s	143.1	26%	0.5%	0.5%	2.94s	156.5	55%	9.1%	6.1%
Buddha$_S$	1.1M	6.89s	142.3	44.9	4.22s	143.1	63%	0.5%	0.5%	3.14s	156.5	119%	9.1%	6.1%
Soda Hall	2.2M	11.13s	119.8	20.1	8.59s	121.3	30%	1.2%	0.1%	7.36s	122.8	51%	2.4%	0.4%
Soda Hall$_S$	2.2M	16.59s	119.8	20.1	11.04s	121.3	50%	1.2%	0.1%	7.92s	122.8	109%	2.4%	0.4%

Table 4.1: *Comparison of the $O(N \log N)$,* Sampling Old *(from our paper [Popov et al., 2006]), and* Sampling New *(Section 4.4.2) algorithms. The number of triangles is denoted by N, the build time – with T_B, Exp is the surface area cost of the tree, and $\mathcal{C}_{trav}$ is the cost for traversing a random ray. Furthermore, R_S is the speedup, R_{Q_1} – the increase of expected cost (Exp), and R_{Q_2} – the increase of traversal cost. The results in this table are discussed in Section 4.6.*

Chapter 5

Fast Construction of BVHs

Inspired by the good results from the previous chapter, and by the demonstrated performance of BVHs for animated scenes (in [Wald et al., 2007]), we have extended our sampling construction method to BVHs. We present this extension here, following the exposition of our paper [Günther et al., 2007], where it was first introduced. Note that a similar method was independently and in parallel developed in [Wald, 2007].

As with KD-tree construction, building a BVH according to SAH requires a plane sweep and thus it also requires event extraction and event sorting steps (Section 3.4). If extraction and sorting is done at each node, the complexity of the algorithm becomes $O(N \log^2 N)$. If it is done at the beginning, the complexity becomes $O(N \log N)$, but the construction algorithm needs to keep lists of sorted events. Even though not as complex and expensive as with KD-trees, the book keeping of the lists adds complexity to the construction algorithm and thus slows it down. Furthermore, the SAH is evaluated at every centroid, which in practice is unnecessary dense, especially in the top levels of the tree.

5.1 SAH Evaluation Through Binning

Similar to the previous chapter, we can apply sampling to find the optimal location of the split plane (w.r.t. SAH). One key difference however is that since a KD tree is a spatial partitioning structure, the potential bounding boxes of the left and right child of a node are defined solely by the sweep plane, independent of the primitives in them. On the contrary, in BVH construction these bounding boxes are defined only through the primitives. They are the tightest AABBs around the primitives in the left and right subtrees respectively.

Again as before, we place $M + 1$ samples $v_0, v_1, \ldots v_M$ regularly along the candidate split axis d. Since the sweep plane events are the centroids themselves (see Section 3.4.1), we choose v_0 and v_M to be the minimum/maximum along d of the AABB around the centroids (i.e. $v_0 = \mathcal{B}^d_{min}(S_C)$ and $v_M = \mathcal{B}^d_{max}(S_C)$). Here S_C denotes the set of the centroids of all primitives in $S(N)$. In order to compute the SAH cost at position v we need to know the number of primitives to the left and right of v (i.e. $|S_L(v)|$ and $|S_R(v)|$). We also need the two AABBs around the primitives with centroids to the left respectively right of v (i.e. $\mathcal{B}(S_L(v))$ and $\mathcal{B}(S_R(v))$).

As with SAH sampling for KD-trees, we use bins to compute both the counts and the AABBs to the left and right of each sample. In bin B_i we keep the number $C(B_i)$ of primitives whose centroids fall in the range $[v_i, v_{i+1})$. Again, we use the following formula to compute the bin index $i(x)$ from an offset x:

$$i(x) = \left\lfloor M \frac{x - v_0}{v_M - v_0} \right\rfloor$$

We also keep the tightest bounding box $\mathcal{B}(B_i)$ around the primitives that go into B_i. We fill the bins using a single pass over $S(N)$. If the centroid of a primitive p falls inside bin B_i, we add one to $C(B_i)$ and extend $\mathcal{B}(B_i)$ to contain $\mathcal{B}(p)$ by using AABB union.

Once the bins are filled, we need to reconstruct the counts $|S_L(v)|$ and $|S_R(v)|$ as well as the bounding boxes $\mathcal{B}(S_L(v))$ and $\mathcal{B}(S_R(v))$ for each event v. We observe that

$$|S_L(v_i)| = \sum_{j=0}^{i-1} C(B_j)$$

$$\mathcal{B}(S_L(v_i)) = \bigcup_{j=0}^{i-1} \mathcal{B}(B_j)$$

Thus $|S_L(v_i)|$ can be expressed as the i-th element of a prefix sum over $C(B_j)$ and $\mathcal{B}(S_L(v_i))$ – as the i-th element of a prefix union over $\mathcal{B}(B_j)$. We compute $|S_L(v_i)|$ and $\mathcal{B}(S_L(v_i))$ simultaneously by performing the two scans together, in a single pass over the bins. Similarly, we compute $|S_R(v_i)|$ and $\mathcal{B}(S_R(v_i))$ for all events using a single pass over the bins in the opposite direction.

At this point we have enough data to compute the SAH cost at the sampling locations. We do so by visiting each location in a loop. Furthermore, for efficiency reasons we process all three dimensions in parallel, using the SIMD extensions of the CPU.

5.2 Sifting

After determining the optimal split plane, we need to sift the primitives into the children of the node. To reduce memory usage and improve data access locality, we again do in-place primitive sifting. In contrast to KD-trees however, this process is much simpler as primitives can not be duplicated. Furthermore, allocation of additional memory, such as the one to store duplicated primitives in KD-tree sifting, is not needed. We do the sifting using Hoare's partition method [Cormen et al., 2009]. Our goal is to order the memory M holding the primitives corresponding to the node into $M = LR$, where L contains S_L and R contains S_R. To do that, we keep two pointers l and r initially pointing to the first and last primitives of M respectively. Then, we perform the following steps in a loop: First, we move l to the right (increase the pointer) as long as l points to a primitive that must go into S_L. Then, we move r to the left (decrease the pointer) as long as r points to a primitive that must go into S_R. At this point, if l points to a location beyond r we exit the loop. Otherwise,

l points to a primitive that must go into S_R and r points to a primitive that must go into S_L, so we swap the pointed memory locations and continue with the next iteration of the loop.

To reduce the passes over the primitives, we search for the split plane of both children while sifting their primitives from the parent. One issue that arises is that we need to know the bounding boxes around the centroids that go in the left and right children respectively, but we can only find them after the sifting is complete. Thus, we use the AABB around the centroids in the parent and we split this bounding box among the children using the split plane. This will of course result in a AABB that is not tight around the centroids, so some of the first or last bins might remain empty. In turn, this might eventually reduce the quality of the approximation by little, but the algorithm will still produce a correct result.

An alternative solution is to keep track of the bounding box around the centroids that fall in each bin. The centroid AABBs of the children can then be reconstructed using prefix unions over the bins. This however turned out to slow down the construction as more operations are needed to update a bin and the increased memory size of a bin permits less bins to be kept in the L1 caches. Furthermore, the difference of quality of the trees produced with the first approach and those produced with second one was subtle.

A further issue with binning BVH construction is the numerical stability of evaluating the sample positions. Again as in Section 4.5.4, computing v_i using the reverse function of $i(x)$ does not guarantee that a centroid x which goes into bin B_i will keep the property $v_i \leq x^d < v_{i+1}$. Even though less fatal than in KD-tree construction (as we don't rely on it for memory allocations), this instability can sometimes create empty BVH nodes. In our paper, to avoid this issue we also keep in each bin the maximum offset along the split axis of all centroids that fall into the bin. We than use this value as the split plane position. We could have also used the approach from Section 4.5.4, however this would have limited our implementation to 32-bit architectures only, as the FP87 unit is not available on x64 architectures.

5.3 Results and Discussion

In this section, we present the results of running the above described algorithm on an Intel 2.4Ghz Core 2 workstation.

The number of bins is an essential parameter controlling the trade-off between construction speed and traversal performance. More bins lead to more accurate sampling, but they also increase the work needed to calculate the SAH function from the binned data. For the bin data to fit into the 64 kB L1 cache of the CPU, there should be at most 256 bins per dimension. Additionally, binning becomes inefficient if the number of bins is close to the number of to-be-binned primitives. Therefore we adaptively choose the number of bins M per dimension linearly, depending on number of primitives N and a bin-ratio constant R: $M = \frac{N}{R}$ and clamp it to $[M_{min}, M_{max}]$. We have experimented with different parameter sets representing a trade-off between speed and accuracy. We have found that the settings $M_{max} = 128$, $M_{min} = 8$, and $R = 6$ present an optimal trade-off with almost no loss of traversal speed, while the settings $M_{max} = 32$, $M_{min} = 4$, and $R = 16$ favor faster construction.

The results of our builder are summarized in Table 5.1. There, we compare the build times of our algorithm in the two settings described above to an implementation of the exact BVH builder as well as to the KD-tree builders from [Popov et al., 2006] and [Hunt et al., 2006] and the BVH builders described in [Lauterbach et al., 2006] and [Wald et al., 2007]. Our measurements show that we consistently outperform all previous published algorithms, including [Hunt et al., 2006] who use significantly fewer primitives as they do not tessellate quads into triangles. Comparing to the previously published data for a sweep plane BVH builder [Wald et al., 2007], our streamed binning approach is one order of magnitude faster at almost the same BVH quality. Furthermore, using the fast settings, we gain additional 20% of construction performance at almost the same BVH quality.

For their BVH-based ray tracer Lauterbach et al. [Lauterbach et al., 2006] favored construction speed over ray tracing performance to support dynamic scenes. With split-in-the-middle they chose the probably fastest approach to select a partition plane during BVH construction, which unfortunately also decreases ray tracing performance by 50% — 90% [Lauterbach et al., 2006] compared to building the BVH according the SAH . Approximating the SAH with our binning approach achieves faster construction times (also due to faster hardware) while retaining high ray tracing performance.

5.4 Summary

In this chapter we presented a fast and accurate construction algorithm which extends our work from Chapter 4 onto BVHs. Based on cost function approximation through binning, this algorithm consistently outperformed all previous published ones.

Recently our sampling approach was successfully implemented on the GPU [Lauterbach et al., 2009], which addressed the only future work we had left in our paper. Also a much faster BVH construction approach known as LBVH was shown in that paper and later on extended in [Pantaleoni and Luebke, 2010]. Despite their speed however, the trees constructed from those two approaches suffer from a considerably lower traversal performance for scenes with moderate depth complexity.

		KD-tree data		BVH data		Our BVH measurements				
		Opteron 2.6GHz	Core2 2.4GHz	P4 2.8GHz	Opteron 2.6GHz	Core2 2.4GHz				
scene	#tris	Popov06	Hunt06	Lauterbach06	Wald07	Exact SAH	Binning	*Exp*	Fast Binning	*Exp*
Bunny	69,451	513 ms	250 ms	90 ms	—	168 ms	48 ms	99.8%	37 ms	98.9%
Fairy Forest	174,117	1.15 s	0.3 s	—	2.8 s	0.47 s	0.12 s	100.2%	0.10 s	98.8%
Conference	282,641	1.41 s	—	—	5.06 s	0.80 s	0.20 s	99.4%	0.15 s	92.5%
Buddha	1,087,716	—	—	1.7 s	20.8 s	4.38 s	0.84 s	100.0%	0.66 s	98.9%
Soda Hall	2,169,132	—	5.14 s	—	53.2 s	8.78 s	1.59 s	101.6%	1.28 s	103.5%
Power Plant	12,748,510	—	—	—	—	119 s	8.1 s	100.5%	6.6 s	99.4%
Boeing 777	348,216,139	—	—	—	—	5605 s	667 s	98.1%	572 s	94.8%

Table 5.1: *Comparing the performance of KD-tree and BVH construction on similar hardware with different algorithms: [Popov et al., 2006], [Hunt et al., 2006], [Lauterbach et al., 2006], [Wald, 2007]. Due to its huge size the* Boeing 777 *was measured on a 2GHz Opteron with 64GByte RAM, 35 of which were consumed during construction. Note that [Hunt et al., 2006] support quads and thus use considerably fewer primitives for construction. All acceleration structures are built according to SAH with the exception of [Lauterbach et al., 2006], which uses a quick split-in-the-middle and thus suffers from a loss of traversal performance in the order of 50%–90%, compared to a SAH built structure. The reported quality of our proposed binned BVH construction is measured as the ratio of its surface area cost to the surface area cost of a classically built BVH (as in Section 3.4.1). We denote the latter with* Exact SAH.

Chapter 6

Construction of High Quality BVHs

Whereas the previous two chapters were focused on fast construction of acceleration structures with minimal loss of quality, in this chapter we will try to build the best possible acceleration structure, disregarding construction times.

We are going to work with BVHs initially, and the final framework will be able to produce both KD-trees and BVHs. Even though several years ago BVHs were thought of as inferior to KD-trees w.r.t. traversal performance [Havran, 2000], Wald et al have shown that they can perform on par to KD-trees if built properly [Wald et al., 2007]. Later research hints that they can even be superior to KD-trees when used for traversal of large coherent packet [Overbeck et al., 2008] or for ray-tracing on GPUs [Aila and Laine, 2009] due to being more shallow than KD-trees.

Our work in this chapter was motivated by the fact that in order to avoid the exponential search time needed to partition the set of primitives in a node, all of the existing BVH construction algorithms used some sort of heuristic. Thus, we started there and developed an approach that can exhaustively explore in polynomial time all ways of partitioning the set of primitives of a node.

By allowing splitting of triangles, we then extended this approach into a construction framework that is able to build *any* practically used binary tree-based acceleration structure. Initially we were not able to improve the quality of the BVHs with this approach. However, we used the framework to study different building strategies and to evaluate their impact on rendering performance. This helped us discover a very important complementary requirement to SAH, which is needed to construct fast trees for ray tracing and is present in all other known construction algorithms. Based on our observations we finally developed a new BVH construction algorithm. The trees produced from the latter enabled ray tracing to perform 2 to 6 times faster compared to the ones constructed by the previous approaches and the speed of construction was comparable to the speed of KD-tree construction. We will present our research in this chapter, following the exposition of our paper [Popov et al., 2009].

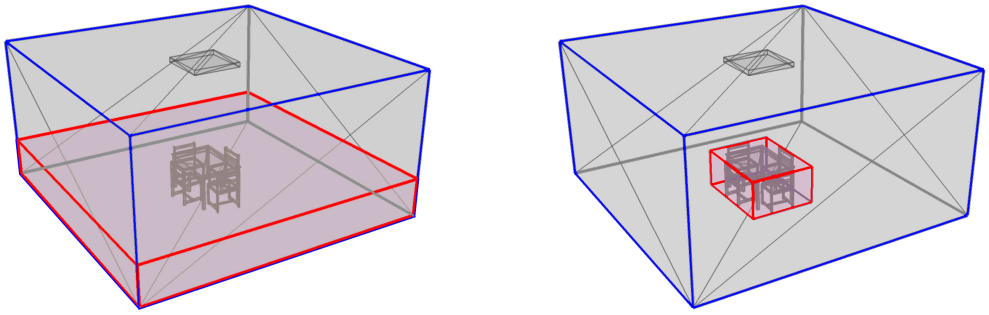


Figure 6.1: *Comparing the nodes created by $\boldsymbol{B}_C$ (left) to the optimal ones (right). The SAH cost of the right solution is much lower, as the ABBB around the red child is tighter and has smaller surface area. The right solution can not be produced by $\boldsymbol{B}_C$ since the children are nested in each other and the centroids of their primitives can not be separated by a plane.*

6.1 Geometric Partitioning

The classical top-down construction algorithm for BVHs from Section 3.4.1 ($\mathbf{B}_C$ from now on), which is also the one that gave best rendering performance prior to this work, uses sweep plane partitioning. It has been adopted from KD trees and approximates primitives by the centroids of their bounding boxes. This approach considers only a fraction of all possible ways to partition the set of primitives corresponding to a node. BVHs allow for simultaneous subdivision along multiple dimensions, including spatial nesting of sibling nodes. Thus, the classical approach can miss partitions with significantly lower cost (Figure 6.1). With a few exceptions, i.e. triangle pre-splitting [Ernst and Greiner, 2007; Dammertz and Keller, 2008] and tree post-processing [Kensler, 2008], there has been little research on how to construct BVHs more optimized for rendering.

As discussed in Section 3.4, the BVH construction algorithm needs to explore all possible ways to partition the set of primitives $S(N)$ of the node N into $S(N) = S_L(N) \oplus S_R(N)$ and it needs to choose the partitioning with minimum cost according to a cost function (usually SAH). Due to the exponential time complexity of this algorithm, exploring all possible ways to partition the set is impossible except in trivial cases.

Our first approach at improving the search for a partitioning was similar to [Ng and Trifonov, 2003]. We used the SAH as a cost function and started by generating a random partitioning $S(N) = S_L \oplus S_R(N)$. Than, we looped over the primitives and tried to move each primitive p into the other set (i.e. if p was in S_L – into S_R and vice versa). If moving the primitive resulted in a lower cost, we kept it in its new set and restarted the iteration over the primitives. Iteration stopped when no primitive could be moved to improve the cost. At this point a local minimum was reached. Thus, to escape it we repeated the randomized partitioning process several times per node. Furthermore, we also generated one partition using a centroid plane sweep. Finally, the partitioning of the node was chosen to be the one with minimum cost from all

generated ones. Generating the centroid plane sweep partitioning as well guaranteed us that the cost of a split will not exceed that of classical construction [Wald and Havran, 2006].

We were not able to measure a difference in the final rendering performance of the trees generated with our approach and the ones produced from the classical one. However, even though the approach was based on random decisions, it was able to improve the SAH cost of the partitions generated from the plane sweep for approximately 2% of the nodes, sometimes considerably. This motivated us to search for a more reliable way to explore the full space of possible partitions for $S(N)$.

6.1.1 From NP Complete to Polynomial

A key observation for the method presented below is that, if a primitive is overlapped by both children of a node, according to the SAH it should go to the child with the smaller probability (i.e. the one with smaller surface area). While this observation is SAH-specific, the algorithm below will work for any cost function that can determine the set into which a primitive should go, based only on the bounding boxes of the left and right child nodes.

According to the observation, if we know the AABBs of the children, we know exactly how to form S_L and S_R. This also gives us the basis for an alternative partitioning algorithm: Instead of testing all $2^{|S(N)|}$ possibilities of partitioning $S(N)$, we can test all possible configurations of child AABBs and form S_L and S_R for each configuration, in order to calculate the cost. Additionally, each configuration should be tested for feasibility, i.e. whether the AABB of each primitive in $S(N)$ is contained in at least one child AABB.

Since we do not allow primitives to be split, the bounding boxes $\mathcal{B}(N_L)$ and $\mathcal{B}(N_R)$ can only start and end at the boundaries of the AABBs of the primitives (Figure 6.2b). We can interpret an AABB as the volume enclosed by the intersection of six half-spaces, with planes perpendicular to the major axes Ox, Oy, and Oz. For the bounding boxes of the child nodes $\mathcal{B}(N_L)$ and $\mathcal{B}(N_R)$, each such plane can be chosen among $2|S(N)|$ candidates. Thus, for a given $S(N)$, there are $O(|S(N)|^6)$ ways to choose $\mathcal{B}(N_L)$ and exactly that much for $\mathcal{B}(N_R)$, which results in $O(|S(N)|^{12})$ ways to choose the bounding boxes of both children of N. From now on, we will refer to a pair of chosen bounding boxes as a configuration.

The number of possible configurations can be further reduced by the observation that each side of the AABB of the parent is shared by at least one of the child AABBs. Thus, we have six degrees of freedom for choosing the twelve half-spaces that form the AABBs of the children.

Once a configuration is formed, in order to form a partitioning of $S(N)$, we need to distribute the primitives into the AABBs of the configuration according to the cost function and we need to test the configuration for feasibility. If $O(Q)$ is the complexity for performing the latter two, finding the best partitioning of a node w.r.t. SAH requires $O(|S(N)|^6 Q)$ time. As the build time is dominated by the time for partitioning the root, the whole build process completes in $O(N^6 Q)$ time for N primitives.

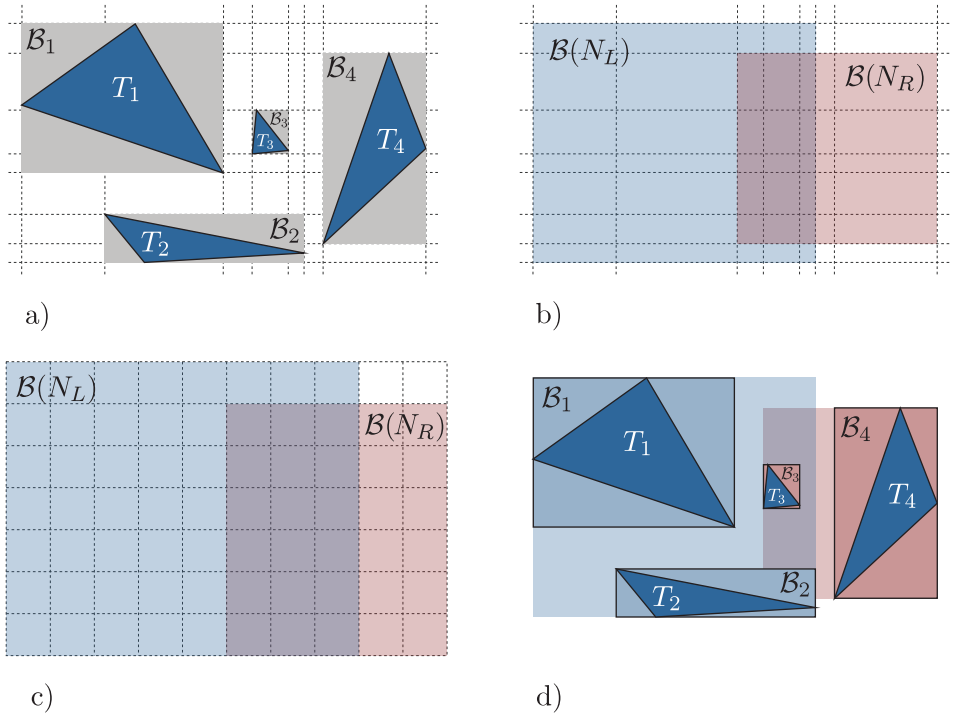


Figure 6.2: *Geometric partitioning. The set of objects, their bounding boxes and their boundaries are shown on a). Candidate AABBs picked at object boundaries in b). Alternatively, candidate AABBs are picked on a grid in c). In d), objects fully contained in both AABBs are put into the one with smaller surface area, and the bounds are tightened around the contained sets.*

6.1.2 A Grid Approximation

Even though we were able to bring down the cost of construction from exponential to polynomial the proposed algorithm is still not practically feasible because of the high degree of the polynomial. We address the issue by introducing an approximation. Instead of picking the half-space planes at the boundaries of the primitives, we pick them on a regular grid that divides the corresponding region of space of the parent node (see Figure 6.2c).

By changing the resolution of the grid we can to control the complexity of the algorithm. If we choose a resolution that results in a grid with V voxels, the time for finding the optimal configuration becomes $O(V^2Q)$. Furthermore, by choosing $V = |S(N)|^{\frac{\alpha}{2}}$, we achieve an adaptive grid resolution and an optimal configuration can be found in $O(|S(N)|^{\alpha}Q)$. For $\alpha \geq 1$, the search time in the root dominates the build process, and thus the whole construction process takes $O(N^{\alpha}Q)$ with N being the number of the primitives in the tree. For practical reasons, we aim at a run time of $O(NQ)$. Thus, we take $\alpha = 1$ and limit the number of grid voxels to $V = K_R\sqrt{|S(N)|}$, where K_R is a constant controlling the grid resolution.

To achieve a voxel shape that is as cubic as possible, we determine the actual resolution $R_X \times R_Y \times R_Z$ of the grid using a binary search on the size of the cube's

Algorithm 6.1 Determine the Optimal Grid Resolution

function ComputeResolution(N, v)
    $R_x = \left\lceil \frac{\mathcal{B}^x_{max}(N) - \mathcal{B}^x_{min}(N)}{v} \right\rceil$
    $R_y = \left\lceil \frac{\mathcal{B}^y_{max}(N) - \mathcal{B}^y_{min}(N)}{v} \right\rceil$
    $R_z = \left\lceil \frac{\mathcal{B}^z_{max}(N) - \mathcal{B}^z_{min}(N)}{v} \right\rceil$
    return (R_x, R_y, R_z)
end function

function ComputeRightBound(N, K)
    $v \leftarrow 1$
    loop
        $(R_x, R_y, R_z) \leftarrow$ ComputeResolution(N, v)
        $c \leftarrow R_x R_y R_z$
        if $c \leq K$ **then break**
        $v \leftarrow 2v$
    end loop
    return v
end function

function ComputeOptimalResolution(N, K, ϵ)
▷ $K \equiv$ the desired voxel count
▷ $N \equiv$ the node
▷ $\epsilon \equiv$ the epsilon (e.g. 10^{-6})
    $l \leftarrow \epsilon$
    $r \leftarrow$ ComputeRightBound(N, K)
    while $r - l > \epsilon$ **do**
        $m \leftarrow (l + r)/2$
        $(R_x, R_y, R_z) \leftarrow$ ComputeResolution(N, m)
        $c \leftarrow R_x R_y R_z$
        if $c < K$ **then** $r \leftarrow m$ **else** $l \leftarrow m$
    end while
    return ComputeResolution(N, l)
end function

side v (see Algorithm 6.1). For a fixed v and a node bounding box $\mathcal{B}(N)$, we calculate the resolution of the grid using

$$R_d = \left\lceil \frac{\mathcal{B}^d_{max}(N) - \mathcal{B}^d_{min}(N)}{v} \right\rceil \quad , \quad d \in \{X, Y, Z\}$$

This way we guarantee that the ratio of the resolution of each side is approximately the same as the ratio of the size of each side of the bounding box: $R_X : R_Y : R_Z \approx \mathcal{B}^X_{size}(N) : \mathcal{B}^Y_{size}(N) : \mathcal{B}^Z_{size}(N)$.

We search for the value of v that will result in the product $R_X R_Y R_Z$ being closest to $K_R\sqrt{|S|}$ but not exceeding it if possible. Since decreasing v increases the product, we set the left boundary of the binary search to some small ϵ (e.g. 10^{-5}). We than find a value of v for which the product becomes below $K_R\sqrt{|S|}$ by starting from $v = 1$ and doubling v as long as the product is above that limit. We use the so found value for the right boundary. Finally, we perform the binary search with the so chosen boundaries and we terminate the search if the distance between the boundaries falls below ϵ.

6.1.3 Cost and Feasibility of a Configuration

Until now we have ignored the complexity of determining the cost of a configuration and whether it is feasible and we have simply denoted it with Q. If we take the naive approach and examine each primitive, we would obtain $Q = N$ and a total build run-time of at least $O(N^2)$ even after using a grid approximation with $\alpha = 1$.

Thus, to accelerate the cost and feasibility calculations in the context of SAH we use an auxiliary BVH over the primitives in $S(N)$. For efficiency reasons, this BVH is built using a centroid-based split in the middle approach. Each node of the auxiliary BVH stores the tight bounds of its children as well as the count of primitives in the sub-tree rooted at the node. Its construction takes $O(N \log N)$, with N being the number of primitives in the node, and does not impact the overall complexity of the algorithm as we will see below.

To test if a configuration is feasible we need to know if each primitive is covered by at least one node. To compute the SAH cost of a configuration we need to know the number of primitives that go into the left respectively right child. We use the auxiliary BVH to compute both of them. We keep two counters C_L and C_R for $|S_L(N)|$ and $|S_R(N)|$ respectively. Starting from the root of the auxiliary BVH, each traversed node N_A is tested against a number of trivial accept and reject tests, as illustrated in figure 6.3. If N_A is trivially rejected, the configuration is declared as infeasible. If N_A passes an accept test, one of the counters ($|C_L|$ or $|C_R|$) is increased with the number of primitives in the sub-tree below N_A. If N_A was neither rejected nor accepted, its children are processed recursively. Once in a leaf, the contained primitives are tested, and either all primitives are accepted (and $|C_L|$ and $|C_R|$ updated accordingly) or the whole configuration is rejected.

Our empirical measurements show that the described algorithm performs a query in $O(\sqrt{N})$ time on average, with N being again the number of primitives in the node. Since the number of configurations in a node is N^α with $\alpha \geq 1$, finding the optimal partitioning of the set of primitives is done in at least $O(N^{\frac{3}{2}})$. Thus, the $O(N \log N)$ complexity for creating the auxiliary BVH in every node can be ignored.

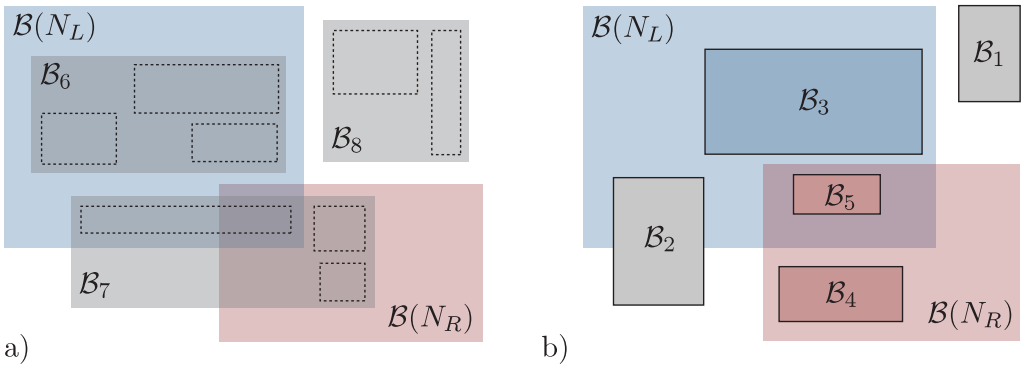


Figure 6.3: *Feasibility tests. In a), the node $\mathcal{B}_6$ from the auxiliary tree will be trivially accepted, the node $\mathcal{B}_8$ will lead to rejecting the whole configuration, and the node $\mathcal{B}_7$ is undecided and its children will be processed recursively. In b), the primitives $\mathcal{B}_3$, $\mathcal{B}_4$, and $\mathcal{B}_5$ will be accepted, while the primitives $\mathcal{B}_1$ and $\mathcal{B}_2$ will lead to the configuration being rejected.*

6.2 A Generic Construction Algorithm

The algorithm from the previous section made it possible to inspect the full search space of possible ways to partition the primitives of a node in a BVH. In this section, we are going to extend it to support sharing of primitives in the leafs of a BVH (we name this an *extended* BVH).

Besides speed, choosing the bounding boxes of the children on a grid gives us another very important advantage: we can choose them *independent* of the actual primitives in the node. This makes no difference in the context of BVH construction. The cost function there is piece-wise constant w.r.t. the boundaries of the AABBs of the children, and thus only changes on the boundary of a primitive. It does however make a difference in the context of an extended BVH as the cost function becomes piece-wise *linear* w.r.t. each side of the child nodes's AABBs.

In KD-tree construction the cost function was one dimensional and we were able to identify its potential minimum points. Here, the cost function is rather six dimensional and we have no easy way of identifying those locations (nor a guarantee on their count). Thus, we use our grid approximation and construct a piece-wise constant approximation of the cost function on it.

The result of modifying our algorithm to handle extended BVHs is a generic construction algorithm that can construct any known and practically usable SAH based binary tree acceleration structure (e.g. KD-trees, BVHs, BKD-tree [Woop et al., 2006], BIHs [Wächter and Keller, 2006], etc) by simply tuning a few parameters.

6.2.1 Primitive Splitting

Up to this point, we have considered a configuration feasible, if each primitive is fully contained in at least one of the child AABBs. We now relax this condition and require each primitive to be fully contained in the union volume of the child AABBs,

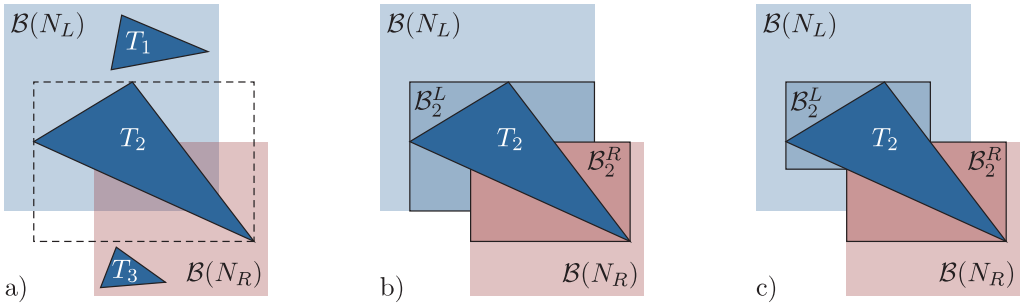


Figure 6.4: *Primitive splitting. Even though $\mathcal{B}(T_2)$ is not completely covered by either $\mathcal{B}(N_L)$ or $\mathcal{B}(N_R)$, the configuration is still considered feasible in a), since T_2 is fully covered by the volume $\mathcal{B}(N_L) \cup \mathcal{B}(N_R)$. One way to split T_2 is to clip it against each child's AABB individually as shown in b). Another is to clip it against the bounding box with smaller surface area and then to form the second bounding box over the part of the polygon that remains outside as shown in c).*

and allow primitives to be split (Figure 6.4). To calculate the SAH cost of a given configuration, we count each split primitive twice – once in the left and once in the right child.

The actual splitting can be done in more than one ways. For triangles, we do it by clipping each triangle with both child AABBs independently. Alternatively, triangles could be clipped against the AABB of the child with smaller probability and the remaining geometry stored in the other child (Figure 6.4). Although this will not affect the SAH cost immediately, it might reduce it in descendant nodes, as their surface area might shrink. However, numerical instabilities prevented us from using this method.

6.2.2 Defining the Search Space

When primitive splitting is allowed, the two bounding boxes of a configuration are not anymore constrained to the bounds of the primitives and can be chosen arbitrarily inside the parent AABB. To explore this continuous space we approximate it by a uniform grid. A very important property of this search approach is that if the grid is fine enough, the search space will examine the configurations used in *all* known construction algorithms, including classic BVH construction and KD tree construction. Thus, in this case the generic algorithm will create the perfect split according to SAH and no other construction algorithm can create a split with lower cost.

We can easily force the generic algorithm produce any SAH binary tree based acceleration structure by modifying the feasibility condition only. To produce a centroid built BVH for example, we need to modify the feasibility condition to test that there exists a plane which separates the centroids of the sets $S_L(N)$ and $S_R(N)$. To produce a KD-tree, we need to modify the condition to only accept configurations that partition the space of the parent. A BKD tree would only accept configurations

in which the sides of the two AABBs along two of the dimensions coincide with the parent's AABB sides.

In practice, we can not afford a fine-enough grid that would capture all known algorithms. Thus, to make sure we don't skip configurations with potentially good costs, we artificially add to the search space the configurations considered by other well known construction algorithms: the configurations from KD tree construction and those from classical (centroid based) BVH construction.

6.2.3 The Algorithm

In this section, we summarize the generic construction algorithm. First, as noted in Section 6.1, besides being able to calculate costs, the cost function needs to be able to tell if a primitive has to go in N_L or N_R based only on $\mathcal{B}(N_L)$ and $\mathcal{B}(N_R)$. Thus we use two functions to abstract the cost function:

$$f_{\mathcal{C}}\big(\mathcal{B}_L, \mathcal{B}_R, S^{BVH}(N)\big) \quad \rightarrow \quad \mathcal{C}$$

$$f_{distr}\big(\mathcal{B}_L, \mathcal{B}_R, S^{BVH}(N)\big) \quad \rightarrow \quad (S_L, S_R)$$

Both functions take as parameters the AABBs of a configuration and the set of all primitives corresponding to the node N, indexed in the auxiliary BVH. The first function returns the cost of the configuration, where as the second one returns the sets of primitives of the left and right nodes respectively for the configuration. A third function

$$f_{feas}\big(\mathcal{B}_L, \mathcal{B}_R, S^{BVH}(N)\big) \rightarrow \{true,\ false\}$$

checks if a configuration is feasible or not. We pass $S^{BVH}(N)$ instead of $S(N)$ to the above functions for efficiency reasons. If a cost function can not make use of the BVH, it can still enumerate all primitives in it efficiently, which would be the equivalent of passing $S(N)$. The function $f_{distr}(.)$ is equivalent to the sifting process in previous construction algorithms.

In the context of SAH $f_{\mathcal{C}}(.)$ uses the auxiliary BVH, as described in Section 6.1.3, to efficiently count the number of primitives that fall into the left and right child. Since the decision whether primitive splitting is allowed is left to the feasibility function, $f_{\mathcal{C}}(.)$ only works with the bounding boxes of the primitives while counting. The cost is calculated from the counts according to SAH (see equation (3.3)). Pseudo code for $f_{\mathcal{C}}(.)$ can be found in Algorithm 6.2.

In our experiments we used several feasibility functions. We used two basic feasibility functions: f_{part} and f_{split}. The first one accepts a configuration if the AABB of each primitive is covered by at least one of the children. The second – if each primitive is covered by the union of the volumes corresponding to the two children. Furthermore we used the f_{kd} feasibility function that does not allow the bounding boxes of a configuration to overlap. Since the feasibility functions are boolean, they can be combined using normal boolean operations. Notice also that the cost, distribution, and feasibility function are sometimes tightly coupled. This becomes apparent if we want to mimic the behaviour of classic BVH construction. In that case, all three functions will need to find a plane that separates the centroids of the primitives going in the left respectively right bounding box. Again, while determining if

Algorithm 6.2 Calculate SAH Cost of Configuration

function GETCOUNTS($N_A, \mathcal{B}_L, \mathcal{B}_R$)
  $\mathcal{B}_c \leftarrow \mathcal{B}_L \cap \mathcal{B}_R$ ▷ common AABB
  $c \leftarrow$ PRIMITIVECOUNT(N_A)
  if $\mathcal{B}(N_A) \subset \mathcal{B}_c$ **then**
    return $\mathcal{A}(\mathcal{B}_L) < \mathcal{A}(\mathcal{B}_R)$? $\{c, 0\}$: $\{0, c\}$
  else if $\mathcal{B}(N_A) \subset \mathcal{B}_L$ **then**
    return $\{c, 0\}$
  else if $\mathcal{B}(N_A) \subset \mathcal{B}_R$ **then**
    return $\{0, c\}$
  else
    $C_L \leftarrow 0, C_R \leftarrow 0$
    if ISLEAF(N_A) **then**
      for all $p \in$ GETPRIMITIVES(N_A) **do**
        if $\mathcal{B}(p) \subset \mathcal{B}_c$ **then**
          if $\mathcal{A}(\mathcal{B}_L) < \mathcal{A}(\mathcal{B}_R)$ **then** $C_L \leftarrow C_L + 1$ **else** $C_R \leftarrow C_R + 1$
        else if $\mathcal{B}(N_A) \subset \mathcal{B}_L$ **then**
          $C_L \leftarrow C_L + 1$
        else if $\mathcal{B}(N_A) \subset \mathcal{B}_R$ **then**
          $C_R \leftarrow C_R + 1$
        else
          $C_L \leftarrow C_L + 1$
          $C_R \leftarrow C_R + 1$
        end if
      end for
    else
      $(C'_L, C'_R) \leftarrow$ GETCOUNTS(LEFTCHILD(N_A), $\mathcal{B}_L, \mathcal{B}_R$)
      $(C''_L, C''_R) \leftarrow$ GETCOUNTS(RIGHTCHILD(N_A), $\mathcal{B}_L, \mathcal{B}_R$)
      $C_L \leftarrow C'_L + C'_R$
      $C_R \leftarrow C''_L + C''_R$
    end if
    return (C_L, C_R)
  end if
end function

function $f_{\mathcal{C}}(\mathcal{B}_L, \mathcal{B}_R, S^{BVH})$
  $(C_L, C_R) \leftarrow$ GETCOUNTS(ROOT(S^{BVH}), $\mathcal{B}_L, \mathcal{B}_R$)
  $cost \leftarrow \mathcal{C}_T + \mathcal{C}_I \dfrac{C_L \mathcal{A}(\mathcal{B}_L) + C_R \mathcal{A}(\mathcal{B}_R)}{\mathcal{A}(\mathcal{B}_L \cup \mathcal{B}_R)}$ ▷ According to SAH (3.3)
  if $T\left(cost, \mathcal{B}_L, \mathcal{B}_R, \text{PRIMITIVECOUNT}\left(\text{ROOT}\left(S^{BVH}\right)\right)\right)$ **then**
    return ∞ ▷ $T(.) \equiv$ termination criteria
  else
    return *cost*
  end if
end function

Algorithm 6.3 Generic Construction Algorithm

```
1: function ConstructTree(B, S(N), S_B(N), O, f_C, f_feas, f_distr)
2:     best ← {cost = ∞}
3:     S^BVH ← ConstructAuxilaryBVH (S_B(N))
4:     for all (B_L, B_R) ∈ O do
5:         if f_feas(B_L, B_R, S^BVH) then
6:             c ← f_C(B_L, B_R, S^BVH)
7:             if c < best.cost then best ← {B_L, B_R, cost = c}
8:         end if
9:     end for
10:    if best.cost = ∞ then
11:        return CreateLeaf(B, S(N))
12:    else
13:        (S_L, S_R, S_B(N_L), S_B(N_R)) ← f_distr(best.B_L, best.B_R, S^BVH)
14:        N_L ← ConstructTree(best.B_L, S_L, S_B(N_L), O, f_feas, f_distr)
15:        N_R ← ConstructTree(best.B_R, S_R, S_B(N_R), O, f_feas, f_distr)
16:        return ConstructNode(B_L, B_R, N_L, N_R)
17:    end if
18: end function
```

a configuration is feasible, we use the auxiliary BVH to reduce the running time of the algorithm.

Another aspect of our algorithm is the search space. To be able to explore different construction algorithms, we have an oracle $\mathcal{O}(N)$ which enumerates all possible configurations for a given node. We used three different basic oracles: $\mathcal{O}_{KD}(N)$, $\mathcal{O}_{CBVH}(N)$, and $\mathcal{O}_{Grid}(N)$, corresponding to the search spaces of a KD tree builder, a centroid based plane sweep BVH builder, and to the regular grid search space defined in Section 6.2.2 respectively. Furthermore, since an oracle practically returns a set of configurations, oracles can be combined by using an union over their sets. We used one compound oracle in our work: $\mathcal{O}_{Gen} = \mathcal{O}_{Grid}(N) \cup \mathcal{O}_{KD}(N) \cup \mathcal{O}_{CBVH}(N)$. It considers all configurations taken by BVH and KD-tree construction and then augments this search space with a regular grid one, whose resolution is only limited by the available computational power and the time we are willing to spend in construction. This guarantees that $\mathcal{O}_{Gen}$ will always visit at least one configuration which is as at least as good as the best configuration visited by any previous construction algorithm.

Our generic algorithm has four parameters: the search space oracle and the three functions $f_{\mathcal{C}}$, f_{distr}, and f_{feas} described above. The algorithm itself is rather simple (see Algorithm 6.3). It is based on standard recursive top-down construction. For each processed node, it iterates over all configurations offered by the oracle, and selects the cheapest feasible one (using f_{feas} and $f_{\mathcal{C}}$). It then forms the child sets and bounding boxes through f_{distr} and recurses. To accelerate the computation of the cost and feasibility functions, it constructs an auxiliary BVH in advance for each node.

	Early ray termination	Centroid sweep	Grid w/ recursive cost evaluation
Surface Area Cost Exp	-	142.6	124.0
Traversal Steps	yes	62.5	81.0
Intersection Steps	yes	15.2	8.0
Ray Cost	yes	85.3	93.0
Traversal Steps	no	122.3	137.4
Intersection Steps	no	48.7	26.7
Ray Cost	no	195.3	177.5

Table 6.1: *Performance of* Sponza *with recursive cost evaluation vs. classic construction. The surface area cost does not correlate to the rendering performance (ray cost) if the rays terminate.*

6.3 Patching the SAH

As discussed in the results Section 6.4 below, we implemented the generic construction algorithm and tested it on a variety of scenes with a variety of settings, measuring the surface area cost $Exp(T)$ and the traversal characteristics of the produced trees.

To our greatest surprise, not only did the algorithm not improve the traversal quality of the produced trees, it actually *degraded* it. Since, our initial goal was to build better BVHs, we used in our experiments the instance $\mathbf{B}_{GP}$ of the generic algorithm with the following settings: $\mathcal{O} = \mathcal{O}_{Grid} \cup \mathcal{O}_{CBVH}$, $f_{feas} = f^{part}$, and for f_{distr} and $f_{\mathcal{C}}$ we used functions that would put each shared primitive according to the SAH in the child with smaller surface area (Section 6.1). With the so chosen oracle, at every node our algorithm did a partitioning that was at least as good as the one from a centroid based plane sweep construction (a.k.a. $\mathbf{B}_{CBVH}$). For some of the scenes, it was actually able to find a partition with lower SAH cost for up to 20% of the nodes. However, the produced trees had a *higher* surface area cost (see Table 6.2). Puzzled by the behaviour of the algorithm, we also ran experiments with the more general $\mathbf{B}_{GPS}$ instance of the generic algorithm, which uses $f_{feas} = f^{split}$, $\mathcal{O} = \mathcal{O}_{Gen}$, and functions f_{distr} and $f_{\mathcal{C}}$ that clip each primitive to the bounding box of the child and take the AABB of the result. Since, this instance of the builder used a finer oracle, it was able to find even cheaper partitionings of $S(N)$ for some nodes. The traversal characteristics of the produced trees were however again worse.

The above results showed an inconsistency with the common belief that SAH cost and surface area cost correlate. Even more, as visible from the table, the better search resolution we used and the more we improved the SAH at every node, the worse trees we obtained from our algorithm. Thus the produced trees exposed actually an *inverse* correlation. Of course the SAH is only a heuristic, but until now it had functioned well for all previously known construction algorithms. This lead us to suspect that the previous algorithms had some unidentified invariant complementary to the SAH that our algorithm didn't.

Before searching for this property, we decided to first try to improve the cost function. The SAH represents an upper limit for the optimal surface area cost which

is computed by one level of look-ahead. To improve on SAH, we modified the cost function to actually build centroid based SAH BVHs for both S_L and S_R and then to take the surface area cost of the built trees as their cost (instead of just using the number of primitives). This way we gave each configuration a much tighter upper bound for the best possible surface area cost. As we anticipated this new cost function did improve the surface area cost of the trees produced. Some of the results, namely for the SPONZA scene, are shown in Table 6.1.

Using the new cost function however, we observed an even more interesting fact: traversal performance and surface area cost of the built trees did *not* correlate any more. Similar results have been also reported in [Ng and Trifonov, 2003; Kensler, 2008]. To find the reason of this discrepancy, we modified the traversal algorithm that we used for testing to conform to the assumptions of the surface area cost model: namely to enumerate all intersections instead of only the first one. With this modification, we managed finally to achieve the desired correlation of surface area cost and traversal performance.

Up to this point, in contrast to the common belief, our experiments had confirmed that: 1) minimizing SAH locally can have a noticeably adverse effects on the cost of the whole tree, and 2) minimizing the surface area cost does not by itself guarantee better rendering performance. On the other hand, with the exception of [Ng and Trifonov, 2003; Kensler, 2008], all other SAH based construction algorithms do not seem to have the above two problems. Looking into them, we identified one major difference. Since they were all based on a plane-sweep, they actually tried to enforce as good as possible space separation of the child nodes. In our algorithm on the other hand it was perfectly fine to nest children in each other, as long as the SAH cost was lower.

Considering the above observations, we formed our main hypothesis: to achieve good rendering performance of the produced trees, a construction algorithm should not only aim at minimizing the SAH cost, but also at separating the children of a node in space. Our intuition is that by doing so, we increase the chances of a ray to terminate early, which works around the unrealistic assumption in the cost model that rays will miss all geometry and will never terminate. In contrast, when two child nodes are nested in each other, even if the ray finds an intersection in one of them, it still has to process the other one as there might be a closer intersection in it.

To verify our hypothesis, we modified the $\mathbf{B}_{GPS}$ builder to enforce space partitioning by setting the feasibility function to $f_{kd} \wedge f_{split}$. As the results in the next section show, we were finally able to achieve our goal – i.e. to construct better trees.

6.3.1 Overlap-Aware SAH

Enforcing space partitioning through f_{feas} is one way of guaranteeing space partitioning but it is unfortunately too restrictive. Similar to centroid plane sweep based construction, we would like to allow nodes to overlap to some degree. Furthermore, even though $\mathbf{B}_{GP}$ produced worse trees in general, it did produce a better BVH for the SHIRLEY6 scene. Due to the geometric symmetry in this scene, the classic algorithm [Wald et al., 2007] produced two siblings contained in each other, resulting in significant overlap (Figure 6.1). The geometric partitioner again found two nodes

nested in each other, but the inner node had much smaller surface area and thus probability of being hit. Since the partitioning in question was at the root of the BVH, the traversal performance of the tree built from $\mathbf{B}_{GP}$ was around 10% higher than the one from $\mathbf{B}_{CBVH}$.

To control the overlap, we propose to modify the cost function. To detect if and to what extent two bounding boxes of a configuration overlap, we look at the ratio of the volume of their intersection to the volume of their union. If the two boxes do not overlap, this ratio will be 0. If they coincide with each other, the ratio will be 1. In all other cases, the ratio will be a value between 0 and 1.

The new cost function is derived from (3.3). If we denote the volume of the bounding box $\mathcal{B}$ with $V(\mathcal{B})$, the cost becomes:

$$\underset{ESAH}{Exp} = C_T + C_I \left(C_O \frac{V(\mathcal{B}_L \cap \mathcal{B}_R)}{V(\mathcal{B}_L \cup \mathcal{B}_R)} + 1 \right) \frac{\mathcal{A}(\mathcal{B}_L)|S_L| + \mathcal{A}(\mathcal{B}_R)|S_R|}{\mathcal{A}(\mathcal{B}_L \cup \mathcal{B}_R)} \tag{6.1}$$

Here, the parameter C_O controls the overlap. Setting C_O to 0, would make $\underset{ESAH}{Exp}$ equivalent to $\underset{SAH}{Exp}$. Setting it to $+\infty$ (or a very large number), would be equivalent to using classical SAH as the cost function and to using f_{kd} as the feasibility function (in the context of the generic construction algorithm). We will show the effect of values different from 0 and ∞ on various scenes later on in Section 6.4.

Notice that if we use extended SAH as the cost function $f_{\mathcal{C}}$ in the generic algorithm, we also need to account for that in f_{distr}. Fortunately, we can use the same sifting function as we use with SAH. If a primitive is shared between the AABBs of a configuration, $f_{\mathcal{C}}$ for $\underset{ESAH}{Exp}$ will count it in the bounding box with smaller surface area, which matches the behaviour of $f_{\mathcal{C}}$ for $\underset{SAH}{Exp}$.

6.4 Results and Discussion

We implemented the above presented generic algorithm in C++. Since even an $O(N^{\frac{3}{2}})$ complexity can result in very large construction times for reasonably large scenes and large search grid resolutions, we used OpenMPI to make it parallel over a cluster of computers. We exploited parallelism in both the configuration search process, by distributing the search for large grids, as well as over the the nodes, by distributing the construction of sub-trees with relatively few primitives. Since the target of our research was to create the "perfect" tree, we have not measured exact build times, which in some cases were very high: for some scenes and grid resolutions they were as high as 10 hours on over 100 CPU cores.

Our measurements are summarized in Table 6.2 and Table 6.3. We used the following scenes to perform the tests, with respective triangle counts given in brackets: Bunny (69.5K), Sponza(67.5K), Fairy Forest(191.8K), Conference(282.7K), Venice(1M), and Soda Hall(2.2M).

We used four types of builders: a classical centroid based builder $\mathbf{B}_{CBVH}$ for reference and three builders based on the generic algorithm. The first one, $\mathbf{B}_{GP}$ uses $\mathcal{O}_{Grid} \cup \mathcal{O}_{CBVH}$ as an oracle, SAH for f_{distr} and $f_{\mathcal{C}}$, and f_{part} for feasibility. This builder corresponds to the geometric partitioning algorithm described in Section 6.1

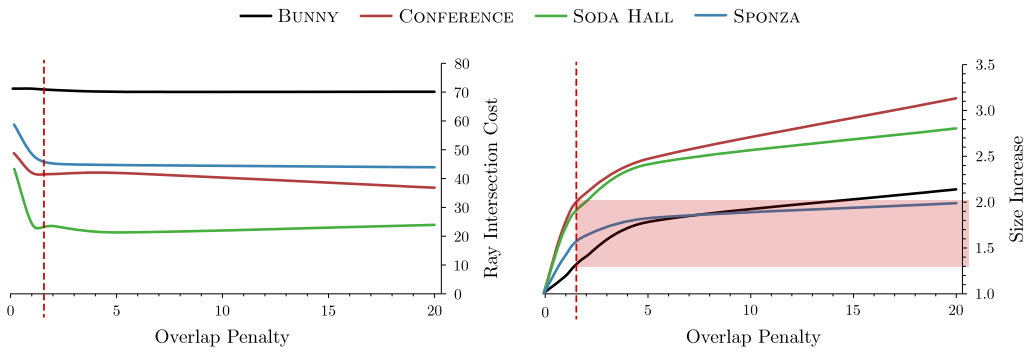


Figure 6.5: *The effect of C_O in $\mathbf{B}_{Gen}$ on the tree quality and on the required storage size. Setting C_O to 1.5 (the vertical red line) results in an almost optimal quality, where as the size increase over $\mathbf{B}_{CBVH}$ remains in the range of 1.25× to 2× (the red rectangle on the right).*

and is the one that can find the cheapest way to partition $S(N)$ w.r.t. SAH (from all possible $2^{|S(N)|}$ ways to do so). The second builder $\mathbf{B}_{Gen}$ is the most general of all builders. It uses the extended SAH from Section 6.3.1 for the cost and distribution functions and f_{split} for feasibility. Its oracle is $\mathcal{O}_{Gen}$. Finally we also look at $\mathbf{B}_{GenKD}$ which uses the oracle $\mathcal{O}_{KD}$, a feasibility function $f_{kd} \wedge f_{split}$ and the normal SAH for cost calculation and sifting. The trees produced from this last builder roughly correspond to KD trees, however no empty space nodes are produced, as the bounding boxes of the nodes are always tight.

In both tables we were interested in the traversal characteristics of the constructed trees. Beside the surface area cost $Exp(T)$, we also measured the cost $\mathcal{C}_1^{rand}$ of traversing a randomly chosen ray throught the tree. We did so by averaging over one million rays for each tree. The cost of a ray is computed by counting the number of traversed tree nodes and the number of intersected primitives and adding the counts, weighted by the traversal cost C_T and the intersection cost C_I from (3.1). The same costs are used during tree construction for the automatic termination criteria. For all trees we have chosen the commonly accepted values of $C_I = 1.5$ and $C_T = 1$. Furthermore, we measured the real traversal performance in million rays per second (MRays/s). We did so for completely random rays (denoted as FPS_1^{rand}) as well as for packets of rays. We measured traversal speed FPS_4 for primary rays using plain 4-wide SIMD packet traversal, range packet traversal FPS_{256}^{range} as discussed in [Wald et al., 2007; Overbeck et al., 2008], and partition packet traversal FPS_{256}^{part} as discussed in [Overbeck et al., 2008]. All measurements were done on a single core of a Core2 Duo processor, with 6MB of L2 cache and running at 2.5GHz.

Table 6.2 compares the traversal characteristics of $\mathbf{B}_{CBVH}$ and $\mathbf{B}_{GP}$. For the small and mid-sized scenes we used two grid resolution multipliers: $K_R = 2^6$ and $K_R = 2^{12}$, where as for the large ones, due to excessive construction times, we used only $K_R = 2^6$. Note that even with 1 primitive, the grid will have at least 8 cells, resulting in a fine grid with resolution of $2 \times 2 \times 2$ (for a primitive with uniformly sized AABB). As discussed in Section 6.3, even though $\mathbf{B}_{GP}$ finds cheaper ways to

partition $S(N)$ for many of the nodes, the performance of the built trees is in contrast inferior to those built by $\mathbf{B}_{CBVH}$.

Table 6.2 compares the traversal characteristics of $\mathbf{B}_{CBVH}$, $\mathbf{B}_{Gen}$, $\mathbf{B}_{GenKD}$, and finally $\mathbf{B}_{KD}$, which is a KD tree builder whose tree has been converted into a BVH, its nodes refitted to the contained geometry, and empty leafs deleted. For $\mathbf{B}_{Gen}$ we use a high overlap value in this set of experiments ($C_O = 10^{16}$). As can be noted from the table, $\mathbf{B}_{Gen}$ always produces the best possible trees. In some cases, it increases the performance of the produced trees by up to 50%. Closely behind is the $\mathbf{B}_{GenKD}$ algorithm. Based on the latter, we will develop in Section 6.5 a new much faster construction algorithm, that produces identical trees.

One apparent disadvantage of the trees constructed by the builders that allow primitive sharing is their relatively high storage requirements when compared to $\mathbf{B}_{CBVH}$. The primary factor that influences the tree size is the overlap factor. Thus, we ran $\mathbf{B}_{Gen}$ with different values of C_O. Figure 6.5 summarizes our measurements. Apparently, C_O has most effect on the tree quality when its value is in the range $[0, 2]$. Values larger than 2 improve the quality only little, but the tree size increases considerably. Thus, based on our experiments we have found a value of $C_O = 1.5$ to be optimal. With this value, the BVH quality is almost optimal, while its size increases at most two times for the tested scenes.

Scene	Build type	Exp	$\mathcal{C}_1^{rand}$	Trav.$_1^{rand}$	Int.$_1^{rand}$	FPS$_1^{rand}$	FPS$_4$	FPS$_{256}^{range}$	FPS$_{256}^{part}$
Bunny	$\mathbf{B}_{CBVH}$	66.6	32.5	24.7	5.2	0.7	4.4	13.0	11.0
	$\mathbf{B}_{GP}$ $(K_R = 2^6)$	68.1	33.7	26.6	4.7	0.7	4.2	12.8	10.4
Sponza	$\mathbf{B}_{CBVH}$	142.6	68.3	46.7	14.4	0.4	1.4	8.1	4.8
	$\mathbf{B}_{GP}$ $(K_R = 2^6)$	167.1	85.5	54.6	20.6	0.3	0.9	7.7	3.6
	$\mathbf{B}_{GP}$ $(K_R = 2^{12})$	150.0	75.8	51.0	16.5	0.4	0.8	7.2	3.4
Fairy Forest	$\mathbf{B}_{CBVH}$	58.5	33.5	23.6	6.6	1.0	1.9	5.9	5.3
	$\mathbf{B}_{GP}$ $(K_R = 2^6)$	68.9	46.9	29.5	11.6	0.7	1.0	3.4	3.6
	$\mathbf{B}_{GP}$ $(K_R = 2^{12})$	80.7	44.3	27.8	11.0	0.7	1.0	3.3	3.5
Conference	$\mathbf{B}_{CBVH}$	86.2	51.3	35.1	10.8	0.6	1.9	6.5	5.8
	$\mathbf{B}_{GP}$ $(K_R = 2^6)$	88.0	57.9	40.8	11.4	0.6	1.8	4.8	5.5
	$\mathbf{B}_{GP}$ $(K_R = 2^{12})$	85.2	54.8	40.8	9.3	0.6	1.9	4.6	5.3
Venice	$\mathbf{B}_{CBVH}$	95.0	38.3	29.7	5.7	0.8	1.7	1.9	3.3
	$\mathbf{B}_{GP}$ $(K_R = 2^6)$	108.8	61.8	51.1	7.1	0.5	1.0	1.9	2.8
Soda Hall	$\mathbf{B}_{CBVH}$	166.0	43.1	36.5	4.4	0.8	2.0	6.6	5.3
	$\mathbf{B}_{GP}$ $(K_R = 2^6)$	167.5	56.8	46.7	6.7	0.5	1.5	4.4	4.1

Table 6.2: *Performance comparison of several BVHs produced by the geometric partitioner $\boldsymbol{B}_{GP}$ to a BVH built by the classic centroid sweep construction $\boldsymbol{B}_{CBVH}$. We have used two different grid resolutions for the BVHs produced by $\boldsymbol{B}_{GP}$: $K_R = 2^6$ and $K_R = 2^{12}$.The traversal algorithms we have used for determining the quality of the built BVHs are described in Section 6.4. It can be seen, that although $\boldsymbol{B}_{GP}$ considers partitions taken from $\boldsymbol{B}_{CBVH}$ at each step and chooses the one with lower SAH cost, the resulting trees have higher expected cost and lower ray tracing performance. Statistics have been gathered on a single core of an Intel Core2 Duo processor with image resolution of 1024^2.*

Scene	Builder	Size	Exp	$\mathcal{C}_1^{rand}$	Trav.$_1^{rand}$	Int.$_1^{rand}$	FPS$_1^{rand}$	FPS$_4$	FPS$_{256}^{range}$	FPS$_{256}^{part}$
Bunny	$\mathbf{B}_{CBVH}$	1.0 Mb	66.6	32.6	24.7	5.2	0.7	4.4	13.0	11.0
	$\mathbf{B}_{Gen}$	3.1 Mb	67.2	31.6	26.0	3.7	0.8	4.1	7.3	9.3
	$\mathbf{B}_{GenKD}$	3.1 Mb	69.0	32.3	26.4	3.9	0.8	4.0	7.5	9.3
	$\mathbf{B}_{KD}$	6.6 Mb	95.3	43.7	39.2	3.0	0.7	3.0	6.0	7.0
Sponza	$\mathbf{B}_{CBVH}$	1.0 Mb	142.6	68.3	46.7	14.4	0.4	1.4	8.1	4.8
	$\mathbf{B}_{Gen}$	2.6 Mb	121.2	43.2	32.8	6.9	0.8	2.0	7.6	5.9
	$\mathbf{B}_{GenKD}$	2.8 Mb	119.3	43.0	33.5	6.3	0.8	2.0	7.4	5.8
	$\mathbf{B}_{KD}$	8.0 Mb	120.7	43.1	38.0	3.4	0.8	1.8	7.3	5.2
Fairy Forest	$\mathbf{B}_{CBVH}$	2.5 Mb	58.5	34.0	23.6	6.9	1.0	1.9	5.9	5.1
	$\mathbf{B}_{Gen}$	12.2 Mb	69.9	36.4	25.1	7.5	1.0	1.9	3.8	5.1
	$\mathbf{B}_{GenKD}$	11.6 Mb	70.0	36.3	25.2	7.4	1.0	1.9	3.7	5.1
	$\mathbf{B}_{KD}$	15.4 Mb	73.2	37.6	30.8	4.5	0.9	1.7	3.5	4.3
Conference	$\mathbf{B}_{CBVH}$	4.1 Mb	86.2	51.3	35.1	10.8	0.6	1.9	6.5	5.8
	$\mathbf{B}_{Gen}$	4.4 Mb	74.5	36.2	29.9	4.2	1.0	2.1	4.6	5.8
	$\mathbf{B}_{GenKD}$	4.2 Mb	74.0	37.2	30.9	4.2	1.0	2.1	4.2	5.7
	$\mathbf{B}_{KD}$	30.0 Mb	77.0	39.1	33.1	4.0	0.9	2.0	5.8	5.6
Venice	$\mathbf{B}_{CBVH}$	17.7 Mb	95.0	38.3	29.7	5.7	0.8	1.7	2.0	3.3
	$\mathbf{B}_{Gen}$	86.7 Mb	84.0	29.2	25.0	2.8	1.1	2.1	1.8	3.7
	$\mathbf{B}_{GenKD}$	66.8 Mb	86.7	29.6	25.4	2.8	1.2	2.2	1.9	3.9
	$\mathbf{B}_{KD}$	141.0 Mb	102.7	35.5	32.3	2.1	1.0	1.9	1.9	3.4
Soda Hall	$\mathbf{B}_{CBVH}$	32.4 Mb	166.0	43.1	36.5	4.4	0.8	2.0	6.6	5.3
	$\mathbf{B}_{Gen}$	152.0 Mb	121.3	23.6	19.5	2.7	1.2	3.2	10.8	8.7
	$\mathbf{B}_{GenKD}$	80.0 Mb	126.2	24.6	20.1	3.0	1.2	3.1	10.7	8.7
	$\mathbf{B}_{KD}$	252.0 Mb	136.0	29.3	27.6	1.1	1.0	2.8	11.0	7.7

Table 6.3: *Comparison of different configurations of our generic BVH construction algorithm. The construction and traversal algorithms are described in Section 6.4. Performance can be dramatically improved for small packets and incoherent rays, when space subdivision is enforced. This can also increase tree size which can in turn reduce coherence for large ray packets. All traversal algorithms benefit from space subdivision on scenes with high geometric and/or depth complexity. Statistics have been gathered on a single core of an Intel Core2 Duo processor with image resolution of* 1024^2.

6.5 A Spatial Construction Algorithm

The results in Table 6.3 show that $\mathbf{B}_{GenKD}$ achieves a quality very close to this of $\mathbf{B}_{Gen}$, which in turn is the algorithm that produces the best trees so far. On the other hand $\mathbf{B}_{GenKD}$ has an important advantage over the latter: due to the oracle we used in it, the number of configurations that need to be examined in a node is relatively small: exactly $2|S(N)|$. Based on this observation, we have designed a construction algorithm that produces the same trees as $\mathbf{B}_{GenKD}$ but in a much faster way with a total run-time of $O(N \log^2 N)$. Furthermore, this run-time can further be reduced to $O(N \log N)$ by using the approach from [Wald and Havran, 2006]. And finally the algorithm can further benefit from the approximate construction methods described in Chapter 4.

The algorithm is a hybrid between KD-tree construction and BVH construction. It inherits the event positions and the primitive splitting from KD-tree construction. That is, it keeps for each node, beside the set of primitives $S(N)$ corresponding to the node, the set $S_{\mathcal{B}} = \{\mathcal{B}(p \cap \mathcal{B}(N)) \mid p \in S(N)\}$. From BVHs, it inherits the ability to have tight bounding boxes around the geometry contained in the children. To this end, the status of the sweep plane consists of the counts of primitives to the left and right of the sweep plane, as well as the tightest respective AABB around them. As with BVHs, to efficiently perform the sweep in $O(N)$, the algorithm does two passes in both directions to compute the tightest AABBs to the left/right of all events. The result of this construction process is an extended BVH.

We implemented the above described spatial split builder (a.k.a. $\mathbf{B}_{SS}$) as a proof of concept. As expected, it performed slightly faster than traditional $O(N \log^2 N)$ KD-tree construction and much faster than $\mathbf{B}_{GenKD}$. Interestingly it also produced slightly better trees than $\mathbf{B}_{GenKD}$ (and not identical as expected). Investigating the issue, we found out that this was due to $\mathbf{B}_{SS}$ being more numerically stable then $\mathbf{B}_{GenKD}$. Since the results of $\mathbf{B}_{SS}$ were almost identical to $\mathbf{B}_{GenKD}$, we don't present them separately here.

One disadvantage of the new algorithm is that it produces trees with large storage requirements as discussed in the previous section. Unfortunately, it is not possible to use the overlap penalty in this case to control the tree size. Thus, we used another solution: Since the increase of storage came primarily because the trees were getting too deep, we tried to create more shallow trees by tuning the termination threshold T_T of the automatic termination criteria (Section 3.2.2): we stopped construction once $r_Q(N) > T_T$. We were able to find a value for T_T for each scene, such that the increase of size was under two times and the quality of the tree was near the one from $\mathbf{B}_{GenKD}$, but unfortunately those values were scene dependent. Table 6.4 shows the results of running our algorithm with different values for T_T on the Sponza scene.

Table 6.4 also compares our spatial split algorithm to existing previous work, namely early split clipping [Ernst and Greiner, 2007] and the edge volume heuristic [Dammertz and Keller, 2008], on two variants of Sponza scene: the original one and one rotated around two major axes. It can be seen from the table that on the original version the pre-split algorithms can not improve the produced BVHs, whereas $\mathbf{B}_{SS}$ improves the rendering performance by more than 50%. Furthermore, no choice of parameters for the pre-splitting methods can increase performance by more than a

Scene	Type	Size	$Exp(T)$	Cost_{ray}	FPS
SPONZA	$\mathbf{B}_{CBVH}$	1.0 MB	142	85	0.60
original	$\mathbf{B}_{EVH}$ $(t = 14)$	1.0 MB	142	85	0.60
	$\mathbf{B}_{EVH}$ $(t = 16)$	1.0 MB	142	85	0.60
	$\mathbf{B}_{ESC}$ (80)	1.1 MB	185	103	0.50
	$\mathbf{B}_{ESC}$ (200)	1.1 MB	167	96	0.55
	$\mathbf{B}_{Gen}$	2.6 MB	121	63	0.95
	$\mathbf{B}_{SS}$	2.8 MB	119	63	0.96
SPONZA	$\mathbf{B}_{CBVH}$	1.0 MB	144	396	0.11
rotated	$\mathbf{B}_{EVH}$ $(t = 14)$	1.3 MB	148	190	0.25
	$\mathbf{B}_{EVH}$ $(t = 16)$	1.6 MB	154	170	0.28
	$\mathbf{B}_{ESC}$ (80)	1.3 MB	155	179	0.29
	$\mathbf{B}_{ESC}$ (200)	1.2 MB	147	196	0.26
	$\mathbf{B}_{Gen}$	11.4 MB	86	100	0.59
	$\mathbf{B}_{SS}$	1.5 MB	134	199	0.24
	$\mathbf{B}_{SS}$	2.6 MB	120	175	0.27
	$\mathbf{B}_{SS}$	4.3 MB	101	134	0.36
	$\mathbf{B}_{SS}$	7.7 MB	89	105	0.54
	$\mathbf{B}_{SS}$	10.3 MB	87	100	0.57

Table 6.4: *Comparison of our construction algorithms to pre-split methods for single primary rays on two variations of* SPONZA. *The pre-split methods are denoted as $\boldsymbol{B}_{EVH}$ for the method from [Dammertz and Keller, 2008] and as $\boldsymbol{B}_{ESC}$ for the method from [Ernst and Greiner, 2007]. The parameter in the brackets is the threshold t from [Dammertz and Keller, 2008] in the first case and the surface area limit SA_{max} from [Ernst and Greiner, 2007] in the second. Our algorithms achieve larger speed-up on both scenes, and pre-splitting methods only help for non-axis-aligned geometry. The size and quality of SS-BVH can be controlled by the termination criterion.*

factor of 3 over $\mathbf{B}_{CBVH}$. Depending on the termination criterion, our spatial split builder can achieve up to 6 times speed-up, and for the same storage requirements, the tree performance is comparable to that of the pre-splitting methods.

6.6 Summary

In this chapter we tried to develop a construction algorithm that would create the "perfect" acceleration structure. To do this, we developed a theoretical framework that unifies all previously known construction algorithms for SAH based binary tree acceleration structures. Furthermore, this framework can explore a much larger space of possible ways to distribute the primitives of a node into its children and it can find the true optimal split w.r.t. to SAH, which sometimes has a much lower cost then the ones found by traditional KD-tree and BVH construction.

Based on results obtained from the implementation of this framework, we were also able to identify a hidden invariant of previous construction algorithms that actually allowed the surface area cost model, despite being based on false assumptions, to

correlate with rendering performance. Furthermore, we were able for the first time to consistently produce trees which had an inverse correlation of surface area cost and rendering performance.

Based on experiments with the generic algorithm, we have concluded that the optimal partition strategy for most scenes is to perform space subdivision. Finally, we have developed a simple and robust SAH-based BVH construction algorithm that creates trees with close to optimal rendering performance.

Throughout the chapter we point out many directions for future work. We believe that the most relevant open problem remains the development of a better cost model for tree construction that can account for early ray termination. This way, explicit enforcement of space subdivision would no longer be needed, and more optimal trees could be obtained w.r.t. size and ray tracing performance.

Another possible direction of research is to improve the spatial split building algorithm to produce smaller trees by extending its search space and taking into account the overlap factor as well. This issue has been addressed partly in [Stich et al., 2009].

Part II

GPU Ray Tracing

Chapter 7

GPU Ray Tracing Background

The second part of the thesis is dedicated to our contribution to interactive GPU ray tracing. We present it in Chapter 8 and Chapter 9, but before that we introduce the necessary background on GPU ray tracing in this chapter. We look into traversal of acceleration structures in greater detail, as efficient traversal has been the major show stopper for GPU ray tracing prior to our work. We then discuss the relevant GPU architectures, their application to general purpose problem solving, and their limitations. Finally, we present the related work on GPU ray tracing.

7.1 Acceleration Structure Traversal

Given a tree based acceleration structure and a ray R, the job of a traversal algorithm is to identify the sequence of the leafs intersected by R. A good traversal algorithm should be efficient, robust, and simple to implement.

For KD-trees the first ray traversal algorithm was developed by Kaplan [1985]. He also was the the first to use a BSP tree to accelerate ray tracing. The algorithm did a repetitive computation of a point-location query along the ray path within the KD-tree and was later called the *sequential* ray traversal algorithm. Later, Jansen [1986] introduced a *recursive* ray traversal algorithm which significantly differed from the sequential algorithm in the way it identified the nodes of the KD-tree to be visited. Instead of starting at the root each time, this algorithm recursively descended in the children of a node along the ray path. Thus, it visited each node at most once per ray. The algorithm was later on improved in terms of efficiency and robustness in [Havran, 2000]. Finally, MacDonald and Booth [1989] described a ray traversal algorithm that uses neighbor-links (a.k.a. ropes) on the faces of the leafs to eliminate recursion completely. This work was later improved by Havran and Bittner [1998a].

In this section, we look into both sequential and recursive KD-tree traversal. The first one is the base of previous work on GPU ray tracing, while the second one is the de facto standard way to traverse a KD-tree on the CPU. We also discuss packet KD-tree traversal, as it is similar to our work presented in Chapter 8. We leave the discussion of rope traversal for that chapter as well, so we can look at it after introducing the programming model limitations of the GPU and after discussing the previous work in GPU ray tracing. Finally, to give the necessary background for Chapter 9, we discuss single ray and packet BVH traversal.

Algorithm 7.1 Sequential KD-tree Traversal

1: **function** INTERSECT(R) ▷ ray $R = (R_O, R_D)$
2: $(t_e, t_x) \leftarrow$ INTERSECTAABB $(R, \mathcal{B}$(tree root)) ▷ [Kay and Kajiya, 1986]
3: $t_e \leftarrow \max(0, t_e)$
4: **if** $t_x < t_e$ **then return** no intersection
5: $P_E \leftarrow R_O + t_e R_D$, $P'_X \leftarrow R_O + t_x R_D$
6: $P = P_E$
▷ Step 1 - point location query
7: **while** $P \neq P'_X$ **do**
8: $N \leftarrow$ tree root
9: $P_X \leftarrow P'_X$
10: **while** N is not leaf **do**
11: $(v, d) \leftarrow$ SPLITPLANE(N)
12: $P_P \leftarrow R_O + R_D \dfrac{v - R_O^d}{R_D^d}$
13: **if** $P_P \in \mathcal{B}(P, P_X)$ **then** $P_X \leftarrow P_P$
14: **if** $P^d < v$ **then**
15: $N \leftarrow N_L$
16: **else**
17: $N \leftarrow N_R$
18: **end if**
19: **end while**
▷ Step 2 - intersect with contained geometry
20: $C \leftarrow (t = \infty)$ ▷ Best found intersection, t denotes the distance
21: **for all** $p \in$ GETPRIMITIVES(N) **do**
22: $t_i \leftarrow$ GETINTERSECTIONDISTANCE(R, p)
23: **if** $t_i < C.t$ **then** $C \leftarrow (t_i, p)$
24: **end for**
25: $P_i \leftarrow R_O + R_D\ C.t$
26: **if** $P_i \in \mathcal{B}(P_E, P_X)$ **then return** C ▷ Early ray termination
▷ Step 3 - exit leaf
27: $P \leftarrow P_X$
28: **end while**
29: **return** no intersection
30: **end function**

7.1.1 Sequential Traversal of KD-trees

One way to regard traversal is as motion, where a point P moves continuously along the ray, until it either hits a surface or exits the acceleration structure. This is the principle behind sequential traversal (Algorithm 7.1), which repeats three distinct steps in a loop. In the first step, it uses a point location query to find the leaf that contains P. In the second, it intersects the ray with the geometry contained in that leaf. If an intersection is found, the traversal loop is terminated. Finally, in the third step, P is advanced along the ray, just past the exit point of the ray from the leaf. The traversal loop continues as long as P is inside the AABB of the KD-tree. Sequential traversal assumes that P is initially contained in the tree. In case it is not, it advances it along the ray to the entry point of the ray in the root's AABB.

To perform a point location query, the sequential algorithm starts at the root of the tree and descends down (line 8). Since a split plane Π partitions the space of its node among the two children, sequential traversal looks at the relation between P and Π. If P is contained in the positive half-space w.r.t. Π, the algorithm descends to the right node, otherwise – to the left one. The search terminates, once a leaf is reached. Since the leafs of a KD-tree form a partitioning of the space of the root, each point from the scene is contained in exactly one leaf, and it is the one found by the described search process.

To determine the exit point from a leaf, the sequential traversal algorithm needs to intersect the ray with the AABB of the leaf. One option would be to store this AABB together with the leaf, however this would increase the storage requirements of the tree. A better approach is to compute the intersection implicitly. To do so, the algorithm keeps a point P_X while descending down in the point location step, such that at any point in time, P_X is the exit point of the ray from the currently processed node. Initially, P_X is computed by intersecting the ray with the AABB of the root (line 9). At each node, it is updated (line 13) by computing the intersection point of the ray with the current split plane, and assigning it to P_X if it lies between P and P_X (i.e. if the intersection point is contained in the AABB defined by P and P_X).

The obvious disadvantage of sequential traversal is that it performs many redundant steps during the point-location search. Even though the paths from the root to successively visited leafs share usually much of the nodes, the point-location search still starts from the root and visits each shared node multiple times. Furthermore, except for the leafs, the exit point calculations are thrown away instead of being reused for the next point location query.

7.1.2 Early Ray Termination

The sequential algorithm above terminates traversal as soon as it encounters an intersection contained in the current leaf. This leads to a correct result in this case, since the leafs of a KD-tree form a space partitioning. Thus, due to the fact that traversal enumerates the leafs along the ray's direction, any intersection found in a leaf beyond will be further away from the ray's origin and can be safely ignored. This technique is known as *early ray termination*.

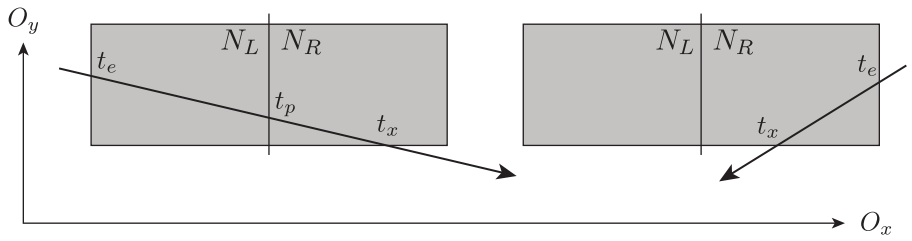


Figure 7.1: *Ray distance intervals. In the left picture, the ray intersects the split plane and its movement is constrained by the interval* $[t_e, t_p]$ *in* N_L *and by* $[t_p, t_x]$ *in* N_R*. On the right, the ray only intersects* N_L *and its movement is constrained by* $[t_e, t_x]$ *for both by* N_L *from* N*. Furthermore, the traversal order can be determined by looking at the sign of the ray's direction along* Ox*. It is left-to-right for the left picture and right-to-left for the right one.*

As we will see below, early ray termination can be applied to most cases of BSP traversal, due to the space partitioning nature of the tree. Furthermore, a form of early ray termination can also be applied to BVH traversal, even though BVHs do not partition space (see Section 7.1.5). In this case however, it is less efficient, as there might be leafs that are still unvisited but can contribute with a closer intersection.

For binary visibility queries (i.e. the $V(X, Y)$ function), early ray termination can be relaxed to accept any intersection contained in the line segment $X\bar{Y}$. Thus, traversal can terminate even sooner, which further improves the performance (especially for BVHs).

7.1.3 Recursive Traversal of KD-trees

The disadvantages of sequential traversal are avoided by the recursive ray traversal algorithm (see [Jansen, 1986]) through an interval based approach. Recursive traversal represents the motion of P as function of the distance t along the ray: $P = R_O + tR_D$ with R_O being the ray's origin and R_D – its direction. For every node N intersected by the ray, the movement of P inside N is constrained by the interval $[t_e, t_x]$, with t_e and t_x being the signed distances to the entry and exit points (P_E and P_X) respectively. If the ray intersects the split plane of N, this interval is partitioned into $[t_e, t_p] \oplus [t_p, t_x]$ by the node's children (see Figure 7.1), with t_p being the signed distance to the intersection point P_P of the ray with the split plane. If the ray does not intersect the split plane, it only intersects one of the children of N and the motion of P is constrained by $[t_e, t_x]$ in this child.

Based on the above observation, the traversal algorithm (Algorithm 7.2) recursively partitions the ray into segments and visits them in order (w.r.t. the signed distance along the ray). At each point in time, it keeps the currently visited segment, specified as a distance interval $[t_e, t_x]$, and the current node N. Initially N is chosen to be the tree's root and the interval is computed by intersecting the ray with $\mathcal{B}(N)$. If the ray origin is contained in the AABB of the root, the entry distance is set to 0.

Algorithm 7.2 Distance Based Recursive KD-tree Traversal

```
1: function INTERSECT(R)                                   ▷ ray R = (R_O, R_D)
2:     travStack ← new stack for storing traversal data
3:     N ← tree root
                                                ▷ Using slabs [Kay and Kajiya, 1986]
4:     (t_e, t_x) ← INTERSECTAABB(R, B(N))
5:     t_e ← max(0, t_e)
6:     if t_x < t_e then return no intersection
7:     loop
8:         if N is leaf then
9:             C ← (t = ∞)                  ▷ closest intersection, t denotes distance
10:            for all p ∈ GETPRIMITIVES(N) do
11:                t_i ← GETINTERSECTIONDISTANCE(R, p)
12:                if t_i < C.t then C ← (t_i, p)
13:            end for
14:            if 0 < C.t < t_x then return C                          ▷ Early exit
15:            if EMPTY(travStack) then return no intersection
16:            (N, t_e, t_x) ← POP(travStack)
17:        else
18:            (v, d) ← SPLITPLANE(N)
19:            N_near ← N_L, N_far ← N_R
20:            if R_D^d < 0 then SWAP(N_near, N_far)
21:            t_p ← (v − R_O^d) / R_D^d
22:            if t_x < t_p then
23:                N ← N_near
24:            else if t_e > t_p then
25:                N ← N_far
26:            else
27:                PUSH(travStack, (N_far, t_p, t_x))
28:                N ← N_near
29:                t_x ← t_p
30:            end if
31:        end if
32:    end loop
33: end function
```

At each visited node N, the algorithm orders the children as "near" and "far" w.r.t. to the ray's direction (line 20; also see Figure 7.1). It does so by looking at the sign of the ray's direction along the axis perpendicular to the split plane. It then computes the distance t_p to the intersection of the ray with the node's split plane. If $t_p < t_e$ then the near child is not intersected by the ray and thus the algorithm searches for an intersection in the far child only (line 25). Similarly, if $t_p > t_x$ the algorithm searches for intersection in the near child only. If $t_p \in [t_e, t_x]$ then both children are visited and the algorithm recursively searches for an intersection in both of them. It visits first the "near" interval $[t_e, t_p]$, descending into the near child (line 28) and upon return from the recursion, if no intersections has been found, it visits $[t_p, t_x]$, descending into the far child (line 16).

Since the traversal algorithm is tail recursive, it is usually implemented in iterative manner and the tail calls are eliminated. This is also the way we present it in Algorithm 7.2. The iterative implementation also allows for an efficient early exit once the closest intersection has been found (line 14), as no stack needs to be unwind.

The above algorithm becomes numerically unstable if any component of the ray direction approaches 0. Furthermore, if the ray is parallel to any split plane, it doesn't work at all. An improved variant of recursive traversal has been presented in [Havran, 2000, Section 5.4.2]. Instead of working with distances along the ray, this algorithm represents the ray intervals using points (Algorithm 7.3). For each visited segment of the ray, it keeps the entry and exit points P_E and P_X. When processing a node, the algorithm looks at the location of the split plane w.r.t. P_X and P_E. Assuming the latter is at offset v from the origin and perpendicular to the axis d, if $v < P_E^d$ then the ray only intersects the right child of the current node (line 24). Similarly, if $v > P_X^d$, the ray only intersects the left child (line 22). In both cases the algorithm descends in the intersected child, keeping the current ray segment unchanged (i.e. P_E and P_X). Finally, if $P_E^d \leq v \leq P_X^d$, the ray intersects both children, and they are both traversed recursively (line 25). Again, the order of the children is determined by the direction of the ray.

Besides numerical stability, the improved recursive traversal has one further advantage. In the distance based variant, the signed distance to the plane is always computed, even if the ray only ever traverses one of the children. When working with points however, this case is detected in advance, saving in this way the computations required for the ray/plane intersection.

7.1.4 Packet Traversal of KD-trees

As discussed in Section 2.4, packet techniques attempt to emulate multi-threading using the SIMD units of a processor. In this section, we discuss how to "packetize" recursive KD-tree traversal (see also [Wald et al., 2001]).

Packet traversal (Algorithm 7.4) is mostly the same as the point variant of recursive traversal, but processes all rays in the packet against each visited node. Thus, beside the segments of all rays in the packet (defined through P_E and P_X for each ray) and the currently traversed node, the algorithm also keeps an *active* ray mask. The latter is a boolean mask, specifying which rays from the packet intersect the currently processed node. In its original formulation (in [Wald et al., 2001]), the algorithm

Algorithm 7.3 Point Based Recursive KD-tree Traversal

```
1: function INTERSECT(R)                                   ▷ ray R = (R_O, R_D)
                                                ▷ Using slabs [Kay and Kajiya, 1986]
2:     (t_e, t_x) ← INTERSECTAABB(R, B(N))
3:     t_e ← max(0, t_e)
4:     P_E ← R_O + t_e R_D, P_X ← R_O + t_x R_D
5:     travStack ← new stack for storing traversal data
6:     N ← tree root
7:     if t_x < t_e then return no intersection
8:     loop
9:         if N is leaf then
10:            C ← (t = ∞)                  ▷ closest intersection, t denotes distance
11:            for all p ∈ GETPRIMITIVES(N) do
12:                t_i ← GETINTERSECTIONDISTANCE(R, p)
13:                if 0 < t_i < C.t then C ← (t_i, p)
14:            end for
15:            P_i ← R_O + R_D C.t
16:            if P_i ∈ B(P_E, P_X) then return C                    ▷ Early exit
17:            if EMPTY(travStack) then return no intersection
18:            (N, P_E, P_X) ← POP(travStack)
19:        else
20:            (v, d) ← SPLITPLANE(N)
21:            if v > P_X^d then
22:                N ← N_L
23:            else if v < P_E^d then
24:                N ← N_R
25:            else
26:                N_near ← N_L, N_far ← N_R
27:                if R_D^d < 0 then SWAP(N_near, N_far)
28:                P_P ← R_O + R_D (v − R_O^d) / R_D^d
29:                PUSH(travStack, (N_far, P_P, P_X))
30:                N ← N_near
31:                P_X ← P_P
32:            end if
33:        end if
34:    end loop
35: end function
```

Algorithm 7.4 Packet Traversal for KD-trees

function PACKETINTERSECT($\vec{R}$) ▷ ray packet $\vec{R} = (\vec{R_O}, \vec{R_D})$
  $N \leftarrow$ tree root
  $(\vec{t_e}, \vec{t_x}) \leftarrow$ PACKETINTERSECTAABB $\left(\vec{R}, \mathcal{B}(N)\right)$
  $\vec{t_e} \leftarrow \max((0,0,\ldots), \vec{t_e})$
  $\vec{P_E} \leftarrow \vec{R_O} + \vec{t_e}\vec{R_D}$, $\vec{P_X} \leftarrow \vec{R_O} + \vec{t_x}\vec{R_D}$
  travStack $\leftarrow$ new stack for storing traversal data
  $\vec{C} \leftarrow (\vec{t} = (\infty, \infty, \ldots))$ ▷ closest intersection, t denotes distance
  $\vec{m_t} \leftarrow \vec{t_x} < \vec{t_e}$ ▷ terminated ray mask
  $\vec{m_a} \leftarrow \neg\vec{m_t}$ ▷ active ray mask
  if ALL($\vec{m_t}$) **then return** C
  loop
    if N is leaf **then**
      for all $p \in$ GETPRIMITIVES(N) **do**
        $\vec{t_i} \leftarrow$ PACKETGETINTERSECTIONDISTANCE($\vec{R}, p$)
        $\vec{m_u} \leftarrow (0,0,\ldots) < \vec{t_i} < \vec{C}.t$ ▷ $\vec{m_u} \equiv$ update mask
        $\vec{C}.t \leftarrow$ BLEND($\vec{m_u}, (\vec{t_i}, (p,p,\ldots)), \vec{C}.t$)
      end for
      $\vec{P_i} \leftarrow \vec{R_O} + \vec{R_D}\ \vec{C}.t$
      $\vec{m_t} \leftarrow \vec{m_t} \vee \vec{P_i} \in \mathcal{B}(\vec{P_E}, \vec{P_X})$
      if ALL($\vec{m_t}$) $\vee$ EMPTY(travStack) **then**
        return C ▷ nothing more to traverse
      else
        $(N, \vec{P_E}, \vec{P_X}, \vec{m_a}) \leftarrow$ POP(travStack)
        $\vec{m_a} \leftarrow \vec{m_a} \wedge \neg\vec{m_t}$ ▷ account for terminated rays
      end if
    else
      $(v, d) \leftarrow$ SPLITPLANE(N)
      if ALL $\left((v,v,\ldots) > \vec{P_X^d} \wedge \vec{m_a}\right)$ **then**
        $N \leftarrow N_L$
      else if ALL $\left((v,v,\ldots) < \vec{P_E^d} \wedge \vec{m_a}\right)$ **then**
        $N \leftarrow N_R$
      else
        $N_{near} \leftarrow N_L$, $N_{far} \leftarrow N_R$
        if ANY $\left(\vec{R_D^d} < 0 \wedge \vec{m_a}\right)$ **then** SWAP(N_{near}, N_{far})
        $\vec{P_P} \leftarrow \vec{R_O} + \vec{R_D}\dfrac{(v,v,\ldots) - \vec{R_O^d}}{\vec{R_D^d}}$
        $\vec{m_f} \leftarrow m_a \wedge (\vec{P_X^d} > (v,v,\ldots))$ ▷ far child active ray mask
        PUSH(travStack, $(N_{far}, \vec{P_P}, \vec{P_X}, \vec{m_f})$)
        $\vec{m_a} \leftarrow \vec{m_a} \wedge (\vec{P_E^d} < (v,v,\ldots))$
        $\vec{P_X} \leftarrow \vec{P_P}$, $N \leftarrow N_{near}$
      end if
    end if
  end loop
end function

assumes that either the direction vectors of all rays in the packet belong to the same octant, or all rays have a common origin. This assumption is necessary to guarantee that the children of a node will always be processed in the same order by all rays in the packet.

A node is processed as follows. Using SIMD operations, P_X of all rays in the packet is tested against the split plane. If $P_X^d < v$ for all rays, then the packet descends into the left child, keeping the ray segments and the active mask unchanged (line 29). Similarly, if $P_E^d > v$ for all rays, the algorithm descends into the right child. If neither of these is true, then at least one ray of the packet has to traverse both nodes. Thus, the algorithm first descends recursively in the near child with all rays, updating the active mask and P_X for the rays that require it (line 39). After returning from the recursion, if a ray in the packet that has to traverse the far child has still found no intersection, the algorithm descends into the right child (line 23). Which child is near and which is far is determined by looking at the direction sign of any active ray from the packet (line 34).

It is easy to see that packet traversal works correctly. Each ray in the packet will visit at least the nodes that it would visit during single-ray recursive traversal. A little less obvious is the benefit of using packets. Packet traversal relies on the fact that SIMD operations execute approximately as fast as scalar operations on current processor architectures. Thus, if the rays in the packet are coherent (e.g. primary or shadow rays), then they will most probably visit the same tree nodes and traversal will be accelerated up to SIMD-width times. In case the packets are fully incoherent, traversal will still be as fast as single ray traversal.

Another option to emulate multi-threaded traversal using SIMD would be to allow the rays to traverse the tree independently. This option requires however that the hardware provides scatter and gather instructions, which is not the case even on the latest CPU architectures. Furthermore, it is not yet clear whether allowing the rays to be independent will not hurt other efficiency aspects, such as caching and branch prediction.

7.1.5 BVH Traversal

The intuition behind BVH traversal (Algorithm 7.5) is simple: if a ray does not intersect a node, it will not intersect any of its children. Thus, the simplest way to traverse a BVH is to test if the ray intersects the AABB of the current node, and in case it does – to descend into both children. A more efficient approach is to also account for early ray termination, by traversing the BVH in a similar fashion to a KD-tree. In this case, there are two major differences that have to be accounted for. First, since the nodes of a BVH can overlap, the segments that they divide the ray into can overlap as well. Thus, traversal can not terminate as soon as it finds an intersection in the current leaf. It can however still terminate early, by remembering the closest found intersection distance, and by only processing subtrees whose entry distance is smaller than the remembered one (line 8).

The second difference is that each node of a BVH knows its AABB explicitly, which allows to compute the entry/exit distances directly from the node. This fact can be used in packet traversal [Wald et al., 2007], to reduce the storage requirements for

Algorithm 7.5 BVH Traversal

```
1: function INTERSECT(R)                                   ▷ ray R = (R_O, R_D)
2:     travStack ← new stack for storing traversal data
3:     C ← (t = ∞)                          ▷ closest intersection, t denotes distance
4:     N ← tree root
5:     (t_e, t_x) ← INTERSECTAABB(R, B(N))
6:     t_e ← max(0, t_e)
7:     loop
8:         if t_x < t_e ∨ C.t < t_e then
                                                  ▷ the node is either not intersected
                                          ▷ or it is behind the closest found primitive
9:             if EMPTY(travStack) then break
10:            (N, t_e, t_x) ← POP(travStack)
11:            continue
12:        end if
13:        if N is leaf then
14:            for all p ∈ GETPRIMITIVES(N) do
15:                t_i ← GETINTERSECTIONDISTANCE(R, p)
16:                if 0 ≤ t_i < C.t then  C ← (t_i, p)
17:            end for
18:            if EMPTY(travStack) then break
19:            (N, t_e, t_x) ← POP(travStack)
20:            continue
21:        else
                                                  ▷ Using slabs [Kay and Kajiya, 1986]
22:            (t_e, t_x) ← INTERSECTAABB(R, B(N_L))
23:            (t_e^f, t_x^f) ← INTERSECTAABB(R, B(N_R))
24:            N ← N_L, N_far ← N_R
25:            if t_e < t_e^n then
26:                SWAP(t_e, t_e^f), SWAP(t_x, t_x^f), SWAP(N, N_far)
27:            end if
28:            PUSH(travStack, (N_far, t_e^f, t_x^f))
29:        end if
30:    end loop
31: end function
```

the stack. Since there is no need to store the distances, and since the active ray mask can also be computed from the AABB of the node, packet traversal only needs to store a singe node on the stack for the entrire packet (the far one). Furthermore, computations can also be re-organized, so that the entry distance of a node is computed only once. Alternatively, the direction of the split axis could be stored in the node [Mahovsky, 2005, Section 4.5]. In this case, the order of the children can be determined from the ray directions only (in most cases), similar to KD tree traversal.

7.2 Graphics Processing Units

Besides traversal, an equally important ingredient of GPU ray tracing is the understanding of the GPU architecture and its applications to general purpose problem solving. This is the focus of our discussion in this section.

Because of the high demands of graphics applications, GPUs were designed to have much higher raw computation power and bandwidth than CPUs. However, this was achieved at the expense of flexibility, and harnessing this power for general purpose problem solving is hard even with the latest generation of GPUs. Historically, GPUs have emerged as dedicated rasterization hardware (see also Section 2.2.4). They offered a fixed function illumination algorithm with fixed function shading, that was controlled by user specified parameters. Due to the increasing demand on rendering quality and flexibility, the GPU hardware has evolved since then, first allowing small programs to run and transform the vertices of the geometry and later on allowing programmable shading. Current GPUs can be programmed to execute arbitrary programs, and thus can be viewed as powerful general purpose co-processors with graphics capabilities.

The first attempts to execute general purpose code on a GPU were made on fixed function devices. Through clever combinations of blending and texturing logic, the GPU was treated as a vector machine [Trendall and Stewart, 2000] and was used for calculating interactive caustics. Adding programmable shading to the GPU lead to the next step in the general purpose GPU programming (GPGPU), as it broadened the problems that could be solved there. It also inspired a new wave of GPGPU research aiming to adapt the solutions to existing computationally intensive problems to the GPU. In turn, this has lead to the next generation of fully programmable hardware, as well as to mature programming environments, such as CUDA [NVIDIA Corporation, 2007], Micrsoft's Direct Compute, and OpenCL.

For the purposes of this thesis, we will discuss two successive GPU architectures in detail: GPUs based on Shader Model 3 and those based on the Tesla architecture from NVIDIA. For completeness, we will also discuss briefly newer NVIDIA architectures (i.e. Fermi) and architectures from other vendors (i.e. ATI and Intel).

7.2.1 Shader Model 3

Probably the most important step in the GPU evolution w.r.t. ray tracing was the introduction of hardware that supports DirectX9's Shader Model 3 (SM3). It added support for true branching and looping in the pixel shaders, which allowed for a large class of algorithms with non-trivial branching logic to be executed in a single

kernel call. In turn, this considerably reduced the bandwidth requirements of such programs, as well as the overhead due to multiple kernel launches.

The GPUs from that generation have several limitations. As with any GPU, in order to deal with the enormous computational demands, they require high amounts of independent work, which they execute in parallel. Furthermore, they are implemented as very wide SIMD processors, with a SIMD width reaching as high as 1024 for some GPUs. Thus, GPGPU algorithms have to expose very coherent branching decisions across the independent units of work. Finally, GPUs have no caches and hide memory access latencies through hardware multi-threading. In order for the latter to work well, GPGPU programs have to typically perform many arithmetic instructions for every word transferred from memory to the chip. A coherent memory access pattern, at least inside the a SIMD unit of pixels is also advisable, in order to minimize the high-latency memory requests to the main GPU memory from the memory controller.

With SM3 hardware, the unit of work is a pixel shader. A GPGPU program is executed by binding the program to the pixel shader's stage, binding the input data to textures and drawing a screen aligned quad. This launches a 2D array of independent program instances (one for every pixel in the quad). After the draw operation completes, the render targets contain the output data, which in turn could be used again as input data to a next draw call by binding it to a texture.

For many algorithms a severe limitation of SM3 hardware is that shaders can not make any global changes (i.e. write to memory). All values they compute are discarded, except those that remain stored in a special set of registers upon program termination. In turn, those registers get eventually written to the render targets. Shaders in SM3 can be regarded as pure register programs, and since indirect register addressing is also not allowed in SM3, many common data structures, such as a stack or queue for example, can not be implemented efficiently inside a shader.

With respect to Whitted style ray tracing, the major SM3 limitation is the inability to implement a stack. Thus, standard recursive traversal of KD-trees and BVHs can not be implemented directly. Apart from that, Whitted style ray tracing fits well to the other requirements and limitations of SM3. It is inherently parallel, so there is enough independent work to feed the GPU. The rays it traces are to a high degree coherent, and thus it exposes high coherency in the branching decisions and memory accesses among neighbouring rays. Finally, even though ray traversal is considered data intensive, our experiments show that there are enough arithmetic operations to cover the latency of memory reads (see Section 8.4.3).

7.2.2 The Tesla Architecture

The Tesla architecture [NVIDIA Corporation, 2007] lifted most of the limitations of previous GPUs. Programs running on it can read and write freely from and to GPU memory. They can synchronize and communicate among each other. The SIMD width was reduced substantially to only 32. Finally, Tesla made the transition from a VLIW architecture (see [Fisher, 1983]) to a more RISC like one, enabling finer compiler optimizations for non-graphics workloads in this way.

The Tesla GPU is mixture of a SIMD and MIMD machine: it consists of multiple independent SIMD cores (a.k.a. SMs). The unit of work is a thread and each thread executes an instance of the same program. Threads are grouped in chunks (a.k.a. warps) of width 32, which are always executed in SIMD fashion. The chunks themselves are grouped into blocks. Computations on the GPU are started from the host processor, by specifying the program (a.k.a. kernel) that will execute, the number of threads in a block and the number of blocks.

Each SM can process several blocks in parallel, using time slicing. This allows the SM to schedule the chunks in an order, that can cover memory and instruction latencies. As there might be more blocks specified in the launch configuration than the GPU can handle simultaneously, the GPU waits until an SM has a free block slot, and assigns the next block to the SM.

The rationale of grouping threads into blocks is to enable efficient inter-block communication. For this purpose, the GPU has a very high bandwidth and low latency on-chip memory, known as the shared memory. The threads of a block can communicate with each other using this memory. Additionally, the threads within a block can synchronize their execution with a block wide barrier. Finally, since the threads in a chunk are processed in SIMD fashion, any data read or written to the shared or global memory from one of the threads, will be immediately visible in the next instruction to all other threads of the chunk. Thus, the execution of the chunk can be viewed as a CRCW PRAM machine [Fortune and Wyllie, 1978] with 32 processors.

The off-chip memory of Tesla GPUs is logically partitioned into four regions. The first region, local memory, is itself equally partitioned among all running threads. It is used to store thread private data. The second one, global memory, is shared among all threads. The third region, texture space, holds textures and the last one holds constant memory, which can not be written to during execution.

The different regions of memory are accessed using different instructions with different trade-off. Global memory is the most general type of memory and is accessible by all threads, but requests to global memory have a very high latency (around 600 processor cycles). Thus, having a high ratio of arithmetic to memory instructions and a coherent access pattern is very important. On a Tesla GPU, if sequential threads of a chunk access sequential and suitably aligned addresses (a.k.a. coalesced access), then the memory controller optimizes the access and issues a large and therefore efficient and fast request to the off-chip memory. In all other cases, the controller issues 32 requests, resulting in a memory operation that is $32\times$ more expensive. On newer class NVIDIA's GPUs (starting with the GT200 architecture), the memory controller tries to group the accesses into as few as possible requests, but a coherent memory access pattern is still very important.

Requests to local memory on the other hand are almost always coalesced. Typically, all threads of a chunk access the same address in their part of the local memory, which leads to sequential addresses in physical memory and to a coalesced access. The design purpose of local memory is to have memory space for spilling registers. The advantage of using it over manually implementing the same in global memory, is that local memory is allocated and managed automatically, and less instructions are needed to calculate the address. The obvious disadvantage of using local memory is that it is thread private.

Texture memory and reading from textures is suitable for reading and filtering data organized in a 2D or 3D array. Furthermore, texture reads are cached, though the latter is used for decreasing bandwidth and not latency (as with the CPU). The disadvantage of textures is that they are read-only.

Finally, constant memory is best suited for passing uniform parameters to the kernels. It is also cached, and this time for latency, but it is read-only and has a very limited size (64 KBytes). If the requested data is in the caches, reading from constant memory is practically free.

The on-chip memory consists of a register file and shared memory. Upon starting a kernel, the register file is partitioned to all threads that will run on a particular SM and each thread receives its own set. This way, the SM does not need to save and restore the state of a thread when switching chunks, since the states of all threads are disjoint. Similarly, shared memory is partitioned among the blocks running on the same SM.

The number of running threads on a SM is determined by three factors: the number of registers requested by the kernel, the size of the shared memory requested by each block and the number of threads in a block. The above three parameters are known at kernel launch time and determine the *occupancy* of the GPU. The latter is the ratio of the number of threads that a GPU can run simultaneously with these parameters to the maximum number of threads that a GPU can run. A large ratio helps the GPU to better cover the instruction and memory latencies. A kernel that requires a lot of registers or a lot of shared memory will reduce the occupancy, which usually leads to under utilization of the GPU and lower performance. An explanation of how to best choose the block size, given the other two parameters, as well as a more in-depth description of the Tesla architecture is available in [NVIDIA Corporation, 2007]. The first generation of Tesla hardware (a.k.a. the G80) has a 32Kb register file and 16KB of shared memory per core. Each SM can run up to 768 threads with hyper-threading. Thus, 100% occupancy can only be reached if each thread does not use more than 10 registers and 5 words of shared memory.

7.2.3 Fermi and Beyond

Above, we only talk about DX9 GPUs and the Tesla architecture from NVIDIA. While there are more recent NVIDIA architectures, as well as architectures from other manufactures (i.e. Intel and AMD), we decided to omit them for reasons stated below.

At the time of writing, GPUs produced by Intel target the low-cost market segment. Their computational performance and bandwidth are more than an order of magnitude lower than current discrete high-end solutions from NVIDIA and AMD. In fact, Intel's GPUs share the memory controller with the CPU currently, so their bandwidth to memory is not higher than the one of the CPU. Furthermore, the theoretical peak performance of Intel GPUs is approximately the same as the CPU's one (counting all cores). Thus, it is questionable if there is any benefit of using current Intel GPUs for ray tracing. There are two further GPU architectures from Intel that are potentially interesting for ray tracing, namely Larabee [Seiler et al., 2008]

and MIC. Unfortunately, the first one has been cancelled, while processors from the second are not yet available on the market.

From the perspective of computational power and memory bandwidth, GPUs from AMD and NVIDIA are more or less equal. However, until recently they differed significantly in their architecture. While NVIDIA's architectures (starting from Tesla) are more RISC like, those from AMD were still based on VLIW. As a consequence, AMD GPUs were less suited for programs that involve significant data dependencies [Zhang et al., 2011] and ray tracing is one of them. This situation might have changed with the very latest generation of AMD GPUs, which was released recently and has also transitioned to a RISC like architecture. There is still however not enough data to confirm that.

Finally, there are newer architectures from NVIDIA as well, but they are similar to Tesla and we see the changes they introduce as rather incremental. Fermi [NVIDIA Corporation, 2009], the generation after Tesla adds many performance and functionality enhancements, including a two level cache hierarchy, new mechanisms for efficient inter-thread communications, and many performance enhancements for integer and double precision operations.

7.2.4 CUDA

While the G80 can be seen as the first GPU designed to support general purpose computing natively, the real breakthrough in the programming model of the GPU can be attributed to NVIDIA's CUDA toolkit [NVIDIA Corporation, 2007]. The latter allows a GPU to be programmed in C++ and in a manner very similar to multi-threaded CPU programming.

With CUDA, the programmer specifies the kernel using serial code. A GPU program is then executed by creating multiple GPU threads, each of which runs the same kernel. To simplify these tasks, CUDA provides a unified C++ compiler that allows mixing of CPU (a.k.a. host) and GPU (a.k.a. device) code in the same source file as well as calls to GPU kernels directly from the host code. It is the task of the compiler to separate the host and device code and to generate the necessary glue code.

CUDA introduces several new keywords to C++, which allow the compiler to distinguish between host and device code and between the different types of memory used on the GPU. GPU specific run-time operations, such as memory allocation, device-to-host and host-to-device memory copies, and texture creation, are handled on the host through a provided run-time library. CUDA also offers a run-time library for the GPU, which mainly includes mathematical and synchronization routines. Most of the latter are implemented as intrinsics and are directly converted to their corresponding GPU instructions by the compiler.

Following the success of CUDA, there have been a number of other frameworks for native GPU programming, including OpenCL, Microsoft's DirectCompute, and lately OpenACC. Their main goal is to address portability between different GPU architectures. With the first two frameworks, the GPU is programmed in a very similar fashion to graphics programming: the programmer writes compute shaders in a domain specific language and manages the GPU memory through buffer-like

objects. This has the advantage of a familiar programming model. In fact, DirectCompute is part of DirectX and OpenCL has an interface very similar to OpenGL. There are however some disadvantages to this model: The domain specific language is less powerful than C++ and does not expose less common instructions (e.g. for video encoding), which can be very useful for specific tasks; The programmer is burdened with more glue code and more state management; Code for data structures and functions has to be duplicated for the GPU and the CPU.

The third framework (OpenACC) takes an approach which is similar to OpenMP. It provides a C++ compiler and allows portions of the code to be annotated as parallel using pragmas. It is the task of the compiler then to extract the GPU code and to create the glue code. In many cases using OpenACC is even easier then CUDA, but the latter is still required in some scenarios, for finer control over the parallelism and resource management. Unfortunately, OpenACC is currently only supported on NVIDIA GPUs and on multi-core CPUs, so it can not yet be considered as portable.

7.3 GPU Ray Tracing

Having discussed the common traversal algorithms and the current GPU architectures, we now proceed to GPU ray tracing. In this section we introduce the previous work, and we will show our contribution in chapters 8 and 9.

The first step toward GPU ray tracing was made in 2002 with the Ray Engine [Carr et al., 2002]. At that time, the programmability of the GPUs was severely limited and thus traversing an acceleration structure on the GPU was impractical, and in most cases even impossible (i.e. for hierarchical structures). Thus, the Ray Engine implemented only the ray-triangle intersection on the GPU while streaming geometry from the CPU. This division of labor resulted in high communication costs, which greatly limited performance.

In the same year Purcell et al. [2002] showed through hardware simulation that it is indeed possible to overcome this limitation by moving essentially all computations of ray tracing onto the GPU. The GPU was treated as a stream processor and each of the different tasks, primary ray generation, acceleration structure traversal, triangle intersection, shading, and secondary ray generation, was implemented as a separate streaming kernel. Due to the difficulty of implementing efficient kd-trees, they chose a simple regular grid as their acceleration structure. A concrete implementation of their GPU ray tracer was later shown in [Purcell, 2004], running on a new generation of hardware. The implementation was able to achieve a performance of roughly 125k rays/s for non-trivial scenes. Its main bottlenecks were the suboptimal acceleration structure as well as the high bandwidth requirements. This basic approach to ray tracing on the GPU was the base for several other implementations, including [Christen, 2005; Karlsson, 2004].

Usage of a better acceleration structure was shown in [Foley and Sugerman, 2005]. They implemented two algorithms: sequential traversal and *KD-Backtrack*. The first one is described in Section 7.1.1 and for the second one they store in each node a pointer to its parent. This way their algorithm does not have to restart at the root each time, but can rather walk up the tree to locate the next node with unprocessed children. The back tracking technique was able to improve the absolute performance

by roughly a factor of 2 (compared to sequential traversal), reaching speeds as high as 350K rays/second. As with previous GPU ray-tracers, both algorithms remained heavily bandwidth-limited. Another attempt of using KD-trees for GPU ray tracing was shown in [Ernst et al., 2004], but the paper does not show any performance results for non-trivial scenes.

Carr et al. implemented a limited ray tracer on the GPU, based on geometry images [Carr et al., 2006]. Their implementation supports only a single triangle mesh without sharp edges, which is difficult to create from most models. For an acceleration structure, they use a bounding volume hierarchy with a predefined structure. Due to the limited model support comparing performance is difficult, but for reasonable model sizes it was not much higher than the above approaches.

The high computation power of the GPU was also utilized to implement ray casting of piecewise quadratic surfaces [Stoll et al., 2006] and NURBS [Pabst et al., 2006]. However, these papers do not use an acceleration structure.

7.4 Summary

Compared to the CPU ray tracers at that time, and especially to the packet traversal ones, the above implementations performed consistently slower (by more than an order of a magnitude). This was despite the fact that GPUs were an order of a magnitude faster than CPUs, measured in floating point operations per second. Thus, all research so far was showing that GPUs were not suited for interactive ray tracing.

With the introduction of shader model 3 GPUs, true branching and looping constructs became possible. This allowed to substantially reduce the bandwidth requirements of many multi-pass algorithms by reformulating them in a single kernel. Unfortunately, recursive traversal was not one of them, as shader model 3 did not provide the means necessary to implement a stack.

The above problem was addressed and solved in two independent publications: our own [Popov et al., 2007] and [Horn et al., 2007]. They both showed for the first time that GPU ray tracing is indeed feasible and can be competitive and even superior to its CPU counterpart. The first publication is part of the research presented in this thesis. It relies on rope traversal and we present it in detail in Chapter 8. The second one is an extension of sequential traversal, which it augments with a short stack containing the last few nodes of the tree path.

In Chapter 8 we also present our research on stackless packet traversal on the GPU for KD-trees. This work is targeted at the Tesla architecture and uses shared memory to keep the packet synchronized. Finally in Chapter 9 we present our work on GPU packet traversal for BVHs. Its results are notable for showing for the first time that GPU ray tracing is not restricted to small models only. We demonstrated that by rendering the Power Plant model, consisting of 12 million primitives, at interactive frame rates. Also, at the time of publication, our BVH ray tracer was the fastest GPU ray tracer available.

Chapter 8

Stackless KD-Tree Traversal

Prior to the work in this chapter, limitations in the programming and memory models of GPUs have kept the performance of GPU ray tracers well below that of their CPU counterparts. Due to the main bottleneck, namely visibility queries, GPU ray tracers were not able to process more than a few hundred thousand rays per second. Here, we present two new algorithms for single ray and packet traversal, that completely eliminate the need for maintaining a stack during KD-tree traversal and that reduce the number of traversal steps per ray. While CPUs benefit moderately from the stackless approach, it improves GPU performance significantly.

The presentation in this chapter follows our paper [Popov et al., 2007]. Even though the algorithm itself is less relevant in practice today, due to much better ones being available (see Chapter 9 and [Aila and Laine, 2009]), the work is important since it was one of the first to show that GPU ray tracing is really competitive to its CPU counterpart. Furthermore, the single ray rope traversal algorithm presented here is still the preferred way to achieve interactive frame rates for ray tracing on DirectX 9 class hardware. According to several unpublished experiments, it seems to still be the most effective way to perform ray tracing on GPUs based on VLIW AMD architectures.

We present three main contributions in this chapter: (1) We review and adapt an efficient, stackless ray traversal algorithm for kd-trees [Havran et al., 1998a]. (2) Based on this algorithm we present a *novel, stackless packet traversal algorithm* that supports arbitrary ray bundles and can handle complex ray configurations efficiently. (3) We present a *GPU implementation* of these algorithms that achieves higher performance than comparable CPU ray tracers. The GPU implementation uses the Compute Unified Device Architecture (CUDA) framework [NVIDIA Corporation, 2007] to compile and run directly on the latest generation of GPUs.

8.1 Related Work

At the time we performed the research in this chapter, KD-trees built according to SAH were considered as the best known acceleration structure for ray tracing of static scenes on the CPU. The preferred traversal algorithm in practical applications was the point version of recursive traversal, presented in Section 7.1.3. As already discussed, in order to increase memory coherence and save per-ray computations,

this algorithm was extended to support tracing of entire ray packets using the SIMD features of modern CPUs and to even larger groups of rays with the help of frustum traversal. Because of the success on the CPU, several attempts exist to implement KD-trees on the GPU. Having introduced them partially in the previous chapter, we are going to discuss them here in more detail, in order to analyze their weak points.

Ernst et al. [2004] showed an implementation of a (parallel) stack for kd-tree traversal on the GPU, using several kernels executed in a multi-pass fashion. The traversal procedure was broken in many small kernels, most of which ended at the points where traversal would do a memory write in the normal algorithm. Each kernel was executed on all rays and stencil buffers were used to mask kernels for rays that are currently not active (i.e. have terminated, or want to take a different branch). In addition, occlusion queries were used to detect if a branch had to be visited at all. Thus, the algorithm could be viewed indeed as CPU traversal, whose operations are performed using very wide-SIMD instructions, executed on the GPU. The frequent kernel switches introduced a high overhead, the bandwidth requirements for storing intermediate results were huge, and there was a large penalty introduced by the divergence of the taken branches across all rays. Thus, the resulting frame rates were much too low for interactive ray tracing even for small scenes. Additionally, the parallel stack consumed large amounts of memory.

One year later, Foley and Sugerman [2005] presented two implementations of stackless kd-tree traversal algorithms for the GPU, namely the sequential traversal algorithm (called *KD-restart* in the paper) and *KD-backtrack*. Both algorithms clearly outperformed earlier regular grids on the GPU (i.e. [Purcell et al., 2002]). However, despite the high GPU compute power and despite the efficient acceleration structure, they were still not able to outperform the CPU implementations. They achieved a peak performance of around 300k rays/s for reasonably complex scenes (i.e. BART Robots and BART Kitchen) on high-end hardware, which was several times smaller than what CPU ray-tracers could achieve on a single core at that time [Wald et al., 2003]. Besides bandwidth limitations, another reason for their relative low performance was the high number of redundant traversal steps.

Independently, and in parallel to the work presented in this chapter, Horn et al. [2007] developed an interactive GPU ray tracer that achieved similar high performance to ours. They extended the kd-restart algorithm, adding a short stack in order to avoid some but not all of the redundant traversal steps.

For bounding volume hierarchies Thrane and Simonsen [2005] demonstrated stackless BVH traversal on the GPU. They outperformed both regular grids and the kd-restart and kd-backtrack variants for kd-trees. They used traversal with no early termination, which allowed them to use one traversal order for all visited nodes for all rays. To avoid the stack, they simulated a stack "pop" operation by pre-processing the BVH, adding an "escape link" to each node. Their performance was still inferior to CPU ray tracing and was limited mainly by the absence of early termination.

8.2 Efficient Stackless KD-Tree Traversal

As demonstrated above, implementing an efficient ray tracer on the GPU that takes full advantage of its raw processing power is challenging. In particular, an efficient

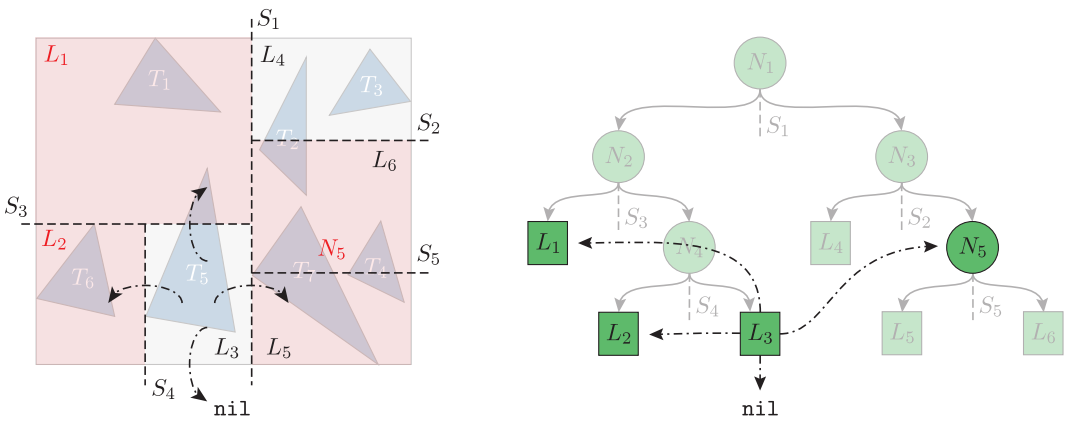


Figure 8.1: *Ropes of a KD-tree in 2D. The ropes assigned to the left and top sides of the bounding box of the leaf L_3 point to the leafs L_1 and L_2 respectively. Since the right side of the bounding box is adjacent to both L_5 and L_6, its rope points to the node N_5, which contains them both. The bottom side of the bounding box has no neighbours and thus its rope points to* `nil`.

implementation needs to be based on a *stackless design*. While the latest GPUs would allow for a stack to be implemented in a fast but small on-chip memory (a.k.a. shared memory on NVIDIA's Tesla architecture), the memory requirements of such an implementation would most likely prohibit good parallelism. A viable stackless algorithm should outperform existing algorithms, and should be simple and small enough to comfortably be implemented in a single GPU kernel in order to avoid the bandwidth and switching overhead of multi-pass implementations. Additionally, register usage should be minimized such that optimal parallelism can be achieved on the latest GPUs (see [NVIDIA Corporation, 2007]).

We base our GPU ray tracing solution on a seemingly little noticed previous approach to stackless traversal of spatial subdivision trees, which uses the concept of neighbor cell links [Samet, 1984, 1989; MacDonald and Booth, 1989], or *ropes* [Havran et al., 1998a]. In the following, we first discuss the kd-tree with ropes as the basis for a single ray traversal algorithm. Later, we present our new extension that efficiently supports the stackless traversal of kd-trees with packets of rays.

8.2.1 Single Ray Stackless KD-Tree Traversal

The main goal of any traversal algorithm is the efficient front-to-back enumeration of all leaf nodes pierced by the ray. From that point of view, any traversal of inner nodes of the tree (also called "down traversal") by a recursive algorithm can be considered overhead that is only necessary to locate the leafs themselves.

The recursive down traversal step for a KD-tree is only necessary because of the irregular structure of the latter. In a regular grid for example, all neighbouring voxels are always known and thus grids are usually traversed using a simple digital differential analyzer. Something similar can be accomplished for KD-trees using ropes.

Ropes are pointers stored on the six sides of the bounding box of a leaf. A rope R for the side s of the bounding box of a leaf L, points to a node from the tree that contains all leafs adjacent to L and shares the plane of s (see Figure 8.1). Faces that have no adjacent leafs lie on the border of the scene and their ropes point to a special `nil` node.

Ropes can be used to eliminate the need of a stack for single ray traversal (see Algorithm 8.1). To this end, motion has to be again represented through the movement of a point. As with sequential traversal, the first step is to locate the leaf that contains the current position of the point. This is done through a point location query, which is a form of binary search and does not require stack. If the ray does not intersect anything in the leaf, traversal has to continue by advancing the point and locating a suitable start node for the next point location query (a.k.a. the exit step). This is where the stack comes into play with recursive traversal, and for sequential traversal the query always starts at the root. To locate a deep entry node without the use of a stack, rope traversal first determines the exit point and the exit side, by intersecting the ray with the AABB of the leaf (line 15). It then follows the rope and starts the next point location query at the node pointed by it.

With a suitable choice of ropes, the sequence of traversed nodes will match that of recursive traversal (see Section 8.2.2). Furthermore, we show how to optimize the ropes for even more efficient traversal than the recursive one (w.r.t. traversal steps). This idea is discussed in[Havran, 2000, Section 5.3.3] and it is further extended by placing multiple exit links on each side of the AABB, organized in a two dimensional KD-tree. With respect to the GPU however, multiple exit links would be less efficient. Even though they further reduce the total count of traversal steps per ray, their implementation is more complex and increases the register pressure of the kernels. It also introduces further points of divergence.

8.2.2 Rope Construction

The only requirement for a rope is to point to a node, that contains all leafs adjacent to the rope's side. This node can be chosen arbitrary as long as it fulfils the requirement. One option is to set all ropes, excluding those on the boundaries of the scene, to point to the root. In this case, rope traversal will perform the same steps as sequential traversal. When constructing a KD-tree with ropes, we will be interested however if we can reproduce the traversal sequence of recursive KD-tree traversal, and whether we can do better than that.

Rope construction relies on a simple fact: During construction, each node inherits five of the six planes of its parent's AABB. Thus, if we assume that each node can have ropes attached to its AABB, five of the six ropes can be inherited from the parent as well. This will not break the above adjacency requirement for the ropes, as the corresponding five sides of the AABB are smaller or equal to the parent's ones. The sixth side lies on the split plane in the parent. Thus, a safe choice for the rope there is the node's sibling. This is the base of the rope construction algorithm (Algorithm 8.2), which works as a post processing step after construction. It starts initially with the AABB of the root, initializing the six ropes to `nil`, and visits the tree in depth-first manner. It computes the AABBs and the nodes as described above and only stores them in the leafs.

Algorithm 8.1 Single Ray Stackless KD-Tree Traversal

1: **function** INTERSECT(R) ▷ ray $R = (R_O, R_D)$
2: $N \leftarrow$ the root node ▷ $N \equiv$ current traversed node
3: $(t_e, t_x) \leftarrow$ INTERSECTAABB $(R, \mathcal{B}(N))$ ▷ Entry/exit distances
▷ Computed according to [Kay and Kajiya, 1986]
4: $t_e \leftarrow \max(0, t_e)$

5: $C \leftarrow (t = \infty)$ ▷ Best found intersection, t denotes the distance
6: **while** $t_e < C.t$ **do**
▷ Down traversal (point location query)
7: $P \leftarrow R_O + t_e R_D$
8: **while** $\neg$ ISLEAF(N) **do**
9: $N \leftarrow \begin{cases} \text{LEFTCHILD}(N) & \text{, if } P \text{ is on left of split plane} \\ \text{RIGHTCHILD}(N) & \text{, otherwise} \end{cases}$
10: **end while**

▷ At a leaf. Check for intersection with contained triangles
11: **for all** $p \in$ GETPRIMITIVES(N) **do**
12: $t_i \leftarrow$ GETINTERSECTIONDISTANCE(R, p)
13: **if** $0 < t_i < C.t$ **then** $C \leftarrow (t_i, p)$
14: **end for**
▷ The exit step. Use a modified slabs method
15: $p_e \leftarrow$ PERCOMPONENTIF $(R_D < (0,0,0),\ \mathcal{B}_{min}(N),\ \mathcal{B}_{max}(N))$
16: $\lambda_{exit} \leftarrow \frac{p_e - R_O}{R_D}$ ▷ Per-component vector operations
17: $s_{exit} \leftarrow \arg\min_{s \in \{1..3\}}(\lambda_{exit}[s])$
18: $t_e \leftarrow \lambda_{exit}[s_{exit}]$
19: $N \leftarrow$ ROPEVALUE($p_e[s_{exit}]$)
20: **end while**

21: **return** C
22: **end function**

Algorithm 8.2 Rope Construction and Optimization

```
1: function OptimizeRope(R, s, B)
                                  ▷ R ≡ the rope, passed by reference
                  ▷ s ≡ the side of R – 1..3 for B_min, the rest for B_max
2:     while R does not point to a leaf leaf do
3:         N' ← node pointed to by R
4:         (v, d) ← split plane of N' (offset and axis)
5:         if d = s ∨ d + 3 = s then                ▷ Split plane is parallel to s
6:             if s ≤ 3 then R ← N'_R else R ← N'_L
7:         else if B^d_min > v then                 ▷ The split plane is above B
8:             R ← N'_R
9:         else if B^d_max < v then                 ▷ The split plane is below B
10:            R ← N'_L
11:        else
12:            break
13:        end if
14:    end while
15: end function

16: procedure ProcessNode(N, R, B)
                                  ▷ N ≡ current node, R ≡ ropes of N
17:    if N is leaf then
18:        Store RS and B with N
19:        N_bounding-box ← AABB
20:    else
21:        if Rope optimization required then
22:            for s ∈ {1..6} do OptimizeRope(R[s], s, B)
23:        end if
24:        (v, d) ← split plane of N (offset and axis)
25:        R_L ← R, R_R ← R,
26:        B_L ← B, B_R ← B,
27:        R_L[d + 3] ← N_R, B_L[d + 3] ← v
28:        R_R[d] ← N_L, B_R[d] ← v

29:        ProcessNode(N_L, R_L, B_L)
30:        ProcessNode(N_R, R_R, B_R)
31:    end if
32: end procedure

33: procedure CreateRopes(T)
34:    N ← GetRootNode(T)
35:    ProcessNode(N, {nil, ...nil}, B(N))
                          (6)
36: end procedure
```

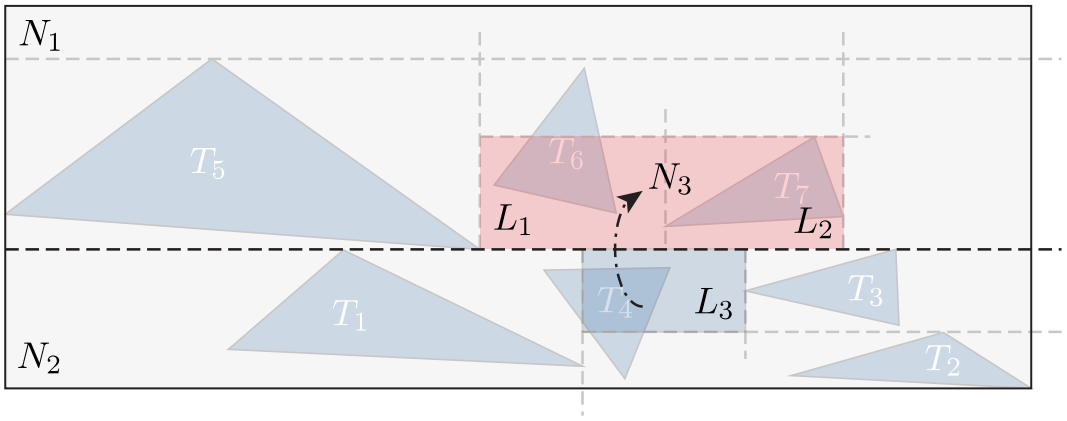


Figure 8.2: *Rope optimization. Recursive traversal will continue with N_1 after leaving L_3 trough the top side. Instead, rope traversal can link the top side of L_3 to N_3, and can thus continue from a deeper node after exiting L_3, resulting in less traversal steps.*

It is easy to see that rope traversal with the above construction algorithm will visit the exactly same nodes as the recursive one. More interesting is that the ropes can be "optimized", so that traversal performs even better than that. The ropes can be created in such a way, that they point to the *deepest* possible entry point into the tree (see Figure 8.2).

To this end, we modify the rope creation (see Algorithm 8.2, function OPTIMIZEROPE). After assigning the ropes to the AABBs of a node N as described previously, we try to push each of them as much as possible down the tree. For a rope R pointing to a node N', we look whether it is possible to reroute R to one of the children of N'. For this, it is sufficient to look at the split plane of N'. If the split plane is parallel to the side s where R is attached, we can safely reroute R (line 5). To decide whether to take the left or right child, we look whether s is at the minimum or maximum of the AABB. In case the split plane is perpendicular to s, we can still push down R as long as the split plane does not intersect s. In this case, if the AABB of N is below the split plane we push R to the right child (line 7), and otherwise (i.e. when it is above) – to the left one.

Both the standard way of constructing ropes (which we will refer to as "unoptimized") and the "optimized" one run in $O(N)$ w.r.t. the number of nodes in the tree.

8.2.3 Stackless Traversal for SIMD Packets of Rays

As we have seen in the previous chapter, packet traversal is a very successful technique on the CPU. Thus, taking into account that the GPU is in essence a multi-core SIMD machine, we have developed a new packet rope traversal algorithm for it. Our algorithm borrows ideas from [Wald et al., 2001] and [Reshetov, 2006]. The purpose of using packets on the GPU is to reduce off-chip bandwidth, avoid memory fetch latencies, and eliminate incoherent branches [Houston, 2006]. The algorithm achieves the above goals by exploiting ray coherence whenever it is present.

Our packet traversal algorithm is an extension of its single ray counterpart. It operates on packets of 32 rays (the SIMD width of the GPU) and processes all rays in the packet against one node at a time. It works similar to the packet traversal described in Section 7.1.4, in the sense that it traverses the same sequence of nodes (if all ray directions are in the same quadrant). As we will discuss below, our packet algorithm requires unoptimized ropes to work efficiently.

In order to implement the traversal algorithm, each ray of the packet maintains a separate state. For each ray R, this state consists of the currently traversed node N_C and the corresponding entry point P_E. Our stackless packet traversal algorithm operates on one node N_T at a time by processing all rays of the packet against it. However, only rays that are currently in N_T (i.e. where $N_T = N_C$) participate in the computations. We name such rays *active rays*.

As with single ray rope traversal, our algorithm works in two stages: the *down traversal* or point location stage and the *exit* stage. The first one descends down the tree, starting from N_C until it reaches a leaf. The decision of which child to take for a particular value of N_C (i.e. N_L or N_R) is taken according to the rules below. These rules are designed to maximize traversal efficiency. The rules are also illustrated on Figure 8.3.

1. If the entry points of all active rays are contained in only child A (i.e. they lie on the same side of the split plane), we descend into that child. In this case, all rays will either only traverse A or they will first traverse A and then the sibling node B (Figure 8.3, cases 10 through 15). Thus, by descending into A we guarantee that it will be visited only once with the packet.

2. If the directions of all active rays, which have their entry point in child node B, do not point toward the other child A with respect to the splitting axis, we choose descend into A. In this case, all rays with entry point in A will either want to only traverse A or they will want to traverse first A and then B (figure 8.3, cases 1, 4, 8, and 9). Those, with entry point in B will only traverse B, so traversing first A with the packet and then B, will guarantee that each child will be visited only once.

3. In all other cases, one of the children needs to be traversed twice, whereas the other – only once. Thus, similar to [Reshetov, 2006], we descend into the child containing more entry points.

Down traversal stops once a leaf is reached. At this point the algorithm intersects all rays with the contained geometry. Any ray from the packet that terminates in this leaf is marked as permanently inactive, by setting its N_C to the special outside node `nil`. The mark is permanent, since a packet can not visit the `nil` node, unless all rays have terminated (see below). As in the single ray case, we determine the exit point and exit node for each active ray by intersecting it with the axis-aligned bounding box of the leaf and by following the rope of the exit face. This defines the new entry points P_E and the new current nodes N_C for all active rays.

To continue traversal, we now have to exit the leaf (perform an *exit step*) with the whole packet. In general, the active rays will not all leave through the same face, which makes the choice of the next N_T hard. Obviously, we need to choose

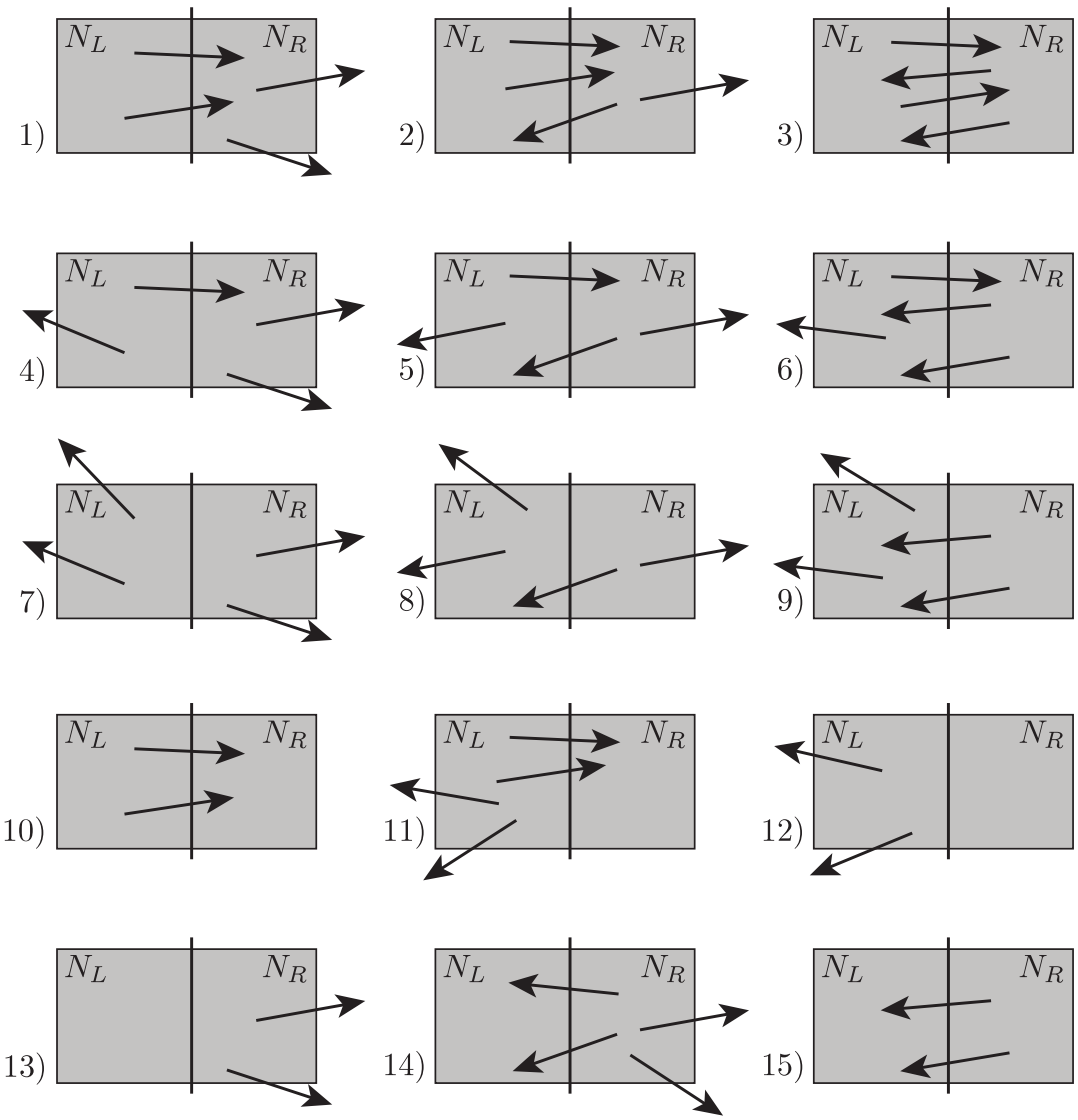


Figure 8.3: *All cases of entry points and directions of the packet with respect to the child nodes. In Rows 1-3, both children of the current node contain entry points of rays. In row 1, all rays with entry points in N_L want to traverse first N_L and then N_R. In row 2, some of those rays want to traverse first N_L then N_R and the others – only N_L. In row 3 all such rays want to traverse only N_L. For rows 1-3, column 1 contains the cases where all rays with entry points in N_R want to traverse only N_R, column 2 – the cases where some want to traverse only N_R, while others want to traverse both N_R and N_L, and column 3 – those in which all rays with entry points in N_R want to traverse first N_R and then N_L. In cases 1 and 4, the packet traversal can save work if it first descends into N_L and then N_R, where as in cases 8 and 9 – N_R and then N_L. In cases 2, 3, 5, 6, and 7, the traversal order is not important. Rows 4 and 5 contain the cases when the entry points of all rays fall within the same node. Thus in row 4, the traversal algorithm has to descend first in N_L, while in row 5 – in N_R.*

from the set S of current nodes for the non-terminated rays of the packet (i.e. $S = \{N_C \mid N_C \neq \texttt{nil}\}$). If all rays have terminated S will be empty and we can terminate the traversal for the whole packet.

We can choose any node from S and still obtain correct traversal behavior. However, we want to guarantee efficiency, at least in the coherent case (i.e. when the directions of all rays in the packet belong to the same quadrant). Thus, we want to regroup the rays of the packet so that no node is processed twice. To this end, we assume that the rays are indeed coherent. We observe that according to the above rules, the rays can only regroup at a node N_{far}, if the packet has finished processing the subtree of its sibling N_{near}. This has two consequences: we need to always choose the deepest node in S for the next T_C, and we need to use *unoptimized* ropes. The latter is required in order to detect when the subtree N_{near} has been fully processed. With unoptimized ropes, this check is performed by simply looking at the deepest node in S. If that node is N_{far} or higher, processing of N_{near} has finished. With optimized ropes, the rays that finish working in the sub-tree of N_{near} might skip N_{far} and jump directly somewhere deep in its sub-tree. Note also, that no two distinct elements of S can be at the same depth (which can be proven by induction). This guarantees that the deepest node is only one.

For efficiency reasons, we do not keep the depth of the nodes explicitly in our algorithm. We rather rely on a property of the construction algorithm. Since the tree is constructed in depth first order, the memory occupied by a node precedes any memory of its descendants. Thus, nodes deeper in the tree are at higher memory addresses than their ancestors. To choose the deepest node from S, we simply choose the node with largest address. Also, to simplify the detection of packet termination, we give the special node `nil` an address in memory that is smaller than the address of the root of the tree. Thus, checking if S is empty becomes equivalent to checking if N_T is `nil`.

Our packet traversal algorithm was designed for NVIDIA's Tesla architecture. Thus, we use the PRAM programming model for a concurrent read concurrent write (CRCW) machine [Fortune and Wyllie, 1978] to describe it in pseudo-code (Algorithm 8.3). As already discussed (see Section 7.2.2), this model is very close to the actual hardware implementation of the latest GPUs. In the algorithm, we make use of standard PRAM reduction techniques to perform parallel OR, parallel SUM, and parallel maximum over the SIMD unit of the GPU. A parallel OR returns the disjunction of a given condition over all processors of the PRAM machine and runs in $O(1)$. The other two reduction operations return the sum respectively the maximum of a given value over the processors. They are known as scan operations and run in $O(\log P)$, with P being the number of processors.

8.3 Implementation

We implemented two variants of the ray tracing algorithm: One based on single ray rope traversal and one based on packet rope traversal. Both variants were implemented on top of CUDA [NVIDIA Corporation, 2007] and as a proof of concept we implemented the single ray variant on top of DirectX 9. We also implemented Whitted style ray tracing on top of each traversal method. In the CUDA versions,

Algorithm 8.3 PRAM Stackless Packet Traversal for KD-Trees

1: **function** FINDINTERSECTION(R) ▷ ray $R = (R_O, R_D)$
2: $N_C \leftarrow$ the root node of the KD-tree ▷ $N_C \equiv$ current node for ray
3: $(t_e, t_x) \leftarrow R \cap \mathcal{B}(N_C)$ ▷ Entry/exit distance for ray
4: $t_e \leftarrow \max(0, t_e)$
5: $C \leftarrow (t = \infty)$ ▷ Best found intersection, t denotes the distance
6: **loop**
7: **if** $t_x \geq t_e$ **then** $N_C \leftarrow$ `nil` ▷ Ray has terminated

▷ $N_T \equiv$ node processed by packet
▷ Choose deepest N_C from all rays. Part of exit stage
8: $N_T \leftarrow$ PRAMMAX(ADDRESSINMEM(N_C))
9: **if** $N_T =$ `nil` **then break** ▷ No more active rays
10: $P_E \leftarrow R_O + t_e R_D$

11: **while** $\neg$ISLEAF(N_T) **do** ▷ The down traversal stage
12: $(v, d) \leftarrow$ SPLITPLANE(N_T) ▷ (offset, axis)
13: $(N_L, N_R) \leftarrow$ GETCHILDREN(N_T)
14: $active \leftarrow N_C = N_T$
15: **if** $active$ **then** ▷ Ray is active. Advance N_C
16: **if** $P_E^d < v$ **then** $N_C \leftarrow N_L$ **else** $N_C \leftarrow N_R$
17: **end if**
18: $b1 \leftarrow$ PRAMOR $\left(active \wedge\ P_E^d < v\ \wedge R_D^d > 0\right)$ ▷ Rule 1
19: $b2 \leftarrow \neg$PRAMOR $\left(active \wedge P_E^d \geq v\right)$ ▷ Rule 2
20: $v3 \leftarrow \begin{cases} -1 & , active \wedge P_E^d < v \\ 1 & , active \wedge P_E^d \geq v \\ 0 & , \neg active \end{cases}$
21: $b3 \leftarrow$ PRAMSUM($v3$) < 0 ▷ Rule 3
22: **if** $b1 \vee b2 \vee b3$ **then** $N_T \leftarrow N_L$ **else** $N_T \leftarrow N_R$
23: **end while**
▷ The exit step
24: **if** $N_C = N_T$ **then**
25: Intersect R with N_C's geometry; update C_T ▷ Alg. 8.1, ln. 11
26: Update t_e and N_C as with single ray ▷ Alg. 8.1, ln. 15
27: **end if**
28: **end loop**
29: **return** C_T
30: **end function**

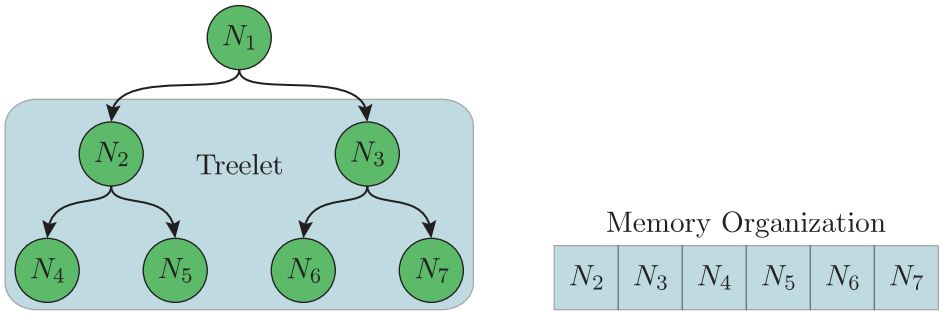


Figure 8.4: *Organizing the KD-tree nodes in treelets in memory allows for higher memory coherence and increases the cache hit ratio.*

the entire ray tracing routine was implemented in a single kernel, whereas the DX9 version was implemented using the multi-pass approach with the following kernels: primary ray generation, ray-scene intersection, shading, and secondary ray generation. The implementation of the single ray traversal follows Algorithm 8.1 literally.

For our packet traversal algorithm we map one ray to one thread and we exploit the SIMD structure provided by the GPU by mapping one packet to one SIMD unit. Horizontal operations, such as choosing the next node to traverse by the packet are carried through PRAM reduction operations over the high-bandwidth low-latency shared memory of the GPU.

At the time we wrote our paper presented in this chapter, GPUs had very limited memory caching support through L2 texture caches, and this support was not exposed in CUDA. Thus, to speed-up our traversal algorithm, we implemented a mechanism for read-ahead to shared memory, whose purpose was to reduce the number of round trips to off-chip memory required to bring in the data of the nodes. The mechanism works by reading a block of data simultaneously, using all threads of a SIMD unit and reading consecutive addresses in consecutive threads (base address + thread index in warp). This way, the data of a whole node can be brought into shared memory with a single request only. Furthermore, the mechanism also achieves read-ahead, since it reads more data then the node actually requires. Since the tree is constructed in DFS manner, the data for few of the nodes on the path starting from the current node and going only left is brought in as well. This increases the chances that the data required for the next traversal or intersection steps is already in shared memory when requested and no further memory request will be required.

We also reorganized the storage of the tree to benefit further from the read-ahead. First, we store the geometry data of the leaf together with its AABB and its ropes, to increase the chance of having the data in shared memory at the beginning of an exit step. Second, we store the internal nodes in treelets similar to [Havran, 1997]. A treelet is a sub-tree of fixed depth with nodes stored together in memory (see Figure 8.4). Thus, because of the read-ahead optimization, accessing the root of a treelet during down traversal, will also bring in the nodes that will be accessed in the next few down traversal steps, saving bandwidth and round-trips to off-chip memory. Even though we change the memory layout of the tree in this way, the argument that deeper nodes will have larger addresses than their ancestors still hold.

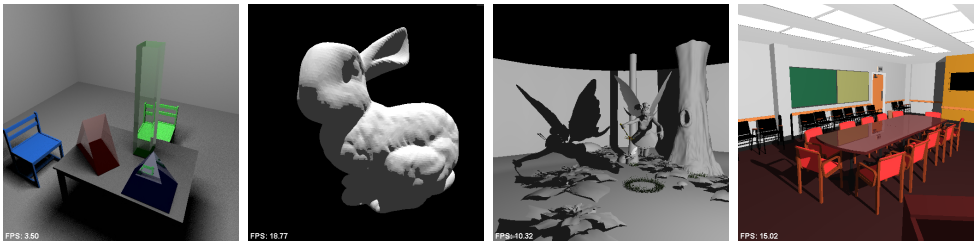


Figure 8.5: *The test scenes and their respective view points:* SHIRLEY6, BUNNY, FAIRY FOREST, *ans* CONFERENCE. *The frame rates for rendering the above images on a GeForce 8800GTX, at a resolution of* 512×512 *with full illumination and materials were 3.5, 18.7, 10.3, and 15 respectively.*

8.4 Results and Discussion

To evaluate the proposed stackless traversal algorithms we implemented them both on the GPU and also on the CPU as a reference. For the measurements presented in this section we used an AMD 2.6GHz Opteron workstation for the CPU implementation and for the GPU one we used another workstation equipped with a NVIDIA GeForce 8800 GTX graphics card. Both the CPU and GPU were considered high-end hardware at the time this research was performed.

We tested our implementations using a variety of scenes, ranging from simple to reasonably complex: SHIRLEY6, BUNNY, FAIRY FOREST, and CONFERENCE. The scenes and the viewpoints for the tests can be seen on Figure 8.5. More statistical data for the scenes is available in Table 8.2.

8.4.1 Memory Requirements

One apparent disadvantage of stackless traversal is the increased storage requirements for the ropes and the bounding boxes of the leafs. However, assuming a compact representation with 8 bytes per node [Wald, 2004], the KD-tree with ropes can not be more than a factor of 4 larger. To show this, we take the ratio of the size S_N of a kd-tree without ropes to the size S_R of a kd tree with ropes:

$$1 \leq \frac{S_R}{S_N} = \frac{48N + 8(2N-1) + 4\sum r_i}{8(2N-1) + 4\sum r_i} < 4 \quad \text{for } \sum r_i > 2$$

were N is the number of leafs and r_i is the number of triangles referenced by leaf i. We assume that we need 4 bytes per reference. The first term in S_R is the overhead of the rope storage.

In practice we encountered a ratio of about 3 (Table 8.2). Although this factor seems high in relation to the kd-tree *alone*, if all the other data is regarded, such as precomputed data for fast triangle intersections, vertex attributes and textures, the factor becomes substantially lower. Thus, the memory overhead of storing the ropes is often reasonable in comparison to the overall memory requirements.

8.4.2 Traversal Steps

Table 8.3 summarizes the average number of traversal steps needed to intersect a ray with the scene for several algorithms: recursive traversal, sequential traversal, single ray rope traversal with optimized ropes and finally packet rope traversal. We also compare to sequential traversal as it is the traversal algorithm used in the previously fastest GPU ray tracer [Foley and Sugerman, 2005].

The measurements clearly show that single ray rope traversal saves up to 5/6 of all down traversal steps compared to sequential traversal. Furthermore, the down traversal of kd-restart/kd-backtrack from [Foley and Sugerman, 2005] is more expensive as both algorithms need to intersect the ray with the split plane, in contrast to the simple point location query, performed by our stackless traversal algorithm.

Ray traversal can be further optimized by using the fact that single ray rope traversal does not require a stack to keep track of what needs to be visited next. Instead the state of a ray only consists of its current node and its entry point. Thus, if the traversal knows a deeper initial entry node it can start directly there.

This fact can be exploited for rays with common origin (as in the case of a pinhole camera) to drastically reduce the number of down traversal steps. Instead of starting from the root for every ray, we can first find the leaf that contains the common origin and then start there with each ray. Combined with optimized ropes, this saves up to 2/3 of the down traversal steps in practice, compared to recursive traversal (see Table 8.3). Furthermore, a similar trick can be applied to secondary rays: We know the leaf where traversal should start, since the origin of the ray is actually the result of a ray/scene intersection.

Similar approaches have also been taken in [Reshetov et al., 2005] and later on in [Benthin, 2006; Wald et al., 2006a, 2007]. In particular, the *entry point search* of [Reshetov et al., 2005] is close to the above optimization. However it does not start at a leaf in the general case and it thus amortizes the down traversal cost over a smaller number of rays.

Packet Traversal

Compared to single ray, packets perform more traversal steps (see Table 8.3). On the other hand, packet traversal is characterized by very coherent memory access and by coherent branch decisions. Thus, on the GPU it outperforms single ray traversal for most scenes (Table 8.1). Its main disadvantage when implemented on a GPU becomes the large packet size, dictated by the SIMD size of the GPU.

On the CPU, stackless packet traversal is slower than single ray traversal. One reason is the relatively large size of the CPU cache and the implicit read-ahead. Thus, a coherent memory access pattern is not as important as on the GPU. Another reason is the code divergence within a GPU warp with single ray traversal. This is not an issue on the CPU, which is MIMD, but can seriously reduce performance on the GPU. Also, using SIMD for the packets on the CPU cannot improve performance a lot, since the single ray implementation already uses SIMD instructions where appropriate. Thus, the overhead introduced by packets becomes an issue.

	OpenRT	CPU		GPU			
Scene	F_f^P	F_s^P	F_p^P	F_s^P	F_p^P	F_s^S	F_p^S
Shirley6	6.6	3.80	3.49	10.6	36.0	4.8	12.7
Bunny	—	2.16	1.71	8.9	12.7	4.9	5.9
FairyForest	3.6	1.57	1.27	5.0	10.6	2.5	4.0
Conference	3.9	2.14	1.78	6.1	16.7	2.7	6.7

Table 8.1: *Absolute performance for single ray and packet stackless traversal on the CPU and the GPU. Performance is given in frames per second at resolution* 1024×1024*, including shading. A subscript "s" denotes single ray stackless traversal, "p" denotes packet stackless traversal, and "f" denotes frustum stack based traversal. A superscript "P" denotes primary rays only, while "S" denotes primary and secondary rays. For the latter we use one point light source for all scenes.* Conference *was ray traced with a reflective table, taking approximately* $1/6$ *of the screen. For comparison we also list the OpenRT performance data with* 4×4 *rays/packet and frustum culling from [Wald et al., 2007].*

8.4.3 Absolute Performance

The absolute performance of our GPU ray tracer in frames per second is summarized in Table 8.1. For the most complex of the tested scenes, namely Conference, we achieve nearly 17 FPS for primary rays with eye-light shading. At the time the research was performed, this was a large improvement over the maximum of 400 thousand rays per second achieved by previous work. Furthermore, our algorithm was not bound by bandwidth like previous ones. To reach to this conclusion, we performed tests by under and overclocking separately the GPU and its memory. We noticed that the performance of our algorithm scales with the GPU clock but not with the memory one.

Our results were also an improvement over CPU ray tracing. Comparing to OpenRT [Wald, 2004] running on a single core of a high-end Opteron CPU, we were able to achieve a speedup of 3-6× for most scenes. At that time OpenRT was one of the industry standard CPU based ray tracers. Our GPU ray tracer had similar performance to a CPU ray tracer running on four CPU cores. Note however that in contrast to the wide availability of GPUs, machines with four CPU cores at the time of writing were uncommon, expensive and primarily used as high end workstations or servers. Interestingly, the situation has not changed with time. With the latest generation of hardware, today's GPU ray tracers are still almost an order of magnitude faster than their GPU counterparts.

In general CPU based ray tracers have been shown to achieve higher performance by using more advanced traversal methods, most notably frustum culling techniques [Reshetov et al., 2005; Benthin, 2006; Wald et al., 2006a, 2007]. However, these frustum methods are usually not flexible, since all rays of a packet have to share a common origin and their directions need to have the same signs. Furthermore, even if secondary rays are supported, tracing them with frustums will be much slower than for primary rays [Boulos et al., 2007]. Our implementation kept an almost constant ray throughput independent of the ray generation, which was not the case with the

CPU implementation (not shown in the table). Secondary rays were almost twice slower in OperRT, due to the fact that OpenRT used frustum culling for primary rays (as showed in [Wald et al., 2007]), which it could not use for secondary rays.

We also compare our performance to the work of Horn et al [Horn et al., 2007], a competing GPU ray tracer which was independently developed in parallel to ours. Their ray tracer achieves 15.2M primary rays/s in the CONFERENCE scene. Using the same view our GPU implementation traces 22M rays/s. However, note that the difference in speed might be due to the different hardware we have used.

8.5 Summary

Probably the biggest contribution of the work presented in this chapter was to show that in fact real-time GPU ray tracing is possible and feasible. Furthermore, ropes are probably still the best way to do fast ray tracing on hardware that can not support a stack (i.e. DX9 class hardware).

The packet traversal algorithm shown here has been superseded by our shared stack BVH traversal (see Chapter 9). Later on, research has shown that unlike on CPUs, packets on the GPU make less sense, because of the large SIMD width, and that the fastest traversal is indeed a single ray one [Aila and Laine, 2009].

Our implementation for this paper was limited in many ways. The occupancy was low due to the small register file of the Tesla architecture, texture caches were not available, and the CUDA compiler was still in a pre-release stage, and thus it was unstable and did not optimize code well. Furthermore, there was no profiler available, and the coalescing rules of the GPU were not documented. Thus, as it turned out later, our algorithm was in practice doing uncoalesced reads.

In the light of the above, it would be interesting to revisit single rope traversal on the GPU. The optimized traversal sequence that it offers, together with the ability to start traversal at a leaf might offer better performance than even the one in [Aila and Laine, 2009].

Another interesting direction for future work is the development of packet traversal algorithms that exploit ideas of frustum test and interval arithmetic to amortize traversal decisions over many rays and thus to further improve performance for certain sets of rays (e.g. primary and shadow).

Scene	Tris	Leafs	Empty Leafs	References	Size	Size w/ Ropes	Rope Overhead
Shirley6	0.8 K	3 K	1 K	1.86	82.4 KB	266 KB	3.23
Bunny	69 K	349 K	183 K	2.53	6.9 MB	23 MB	3.30
Fairy Forest	174 K	721 K	382 K	3.01	14.9 MB	48 MB	3.22
Conference	282 K	1.24 M	515 K	3.13	27.8 MB	85 MB	3.06

Table 8.2: *Used test scenes together with statistical data of the SAH KD-tree. The average number of references to triangles per non-empty leaf is denoted as "references'. Ropes increase the KD-tree size approximately 3 times.*

	Recursive Trav.		Sequential Trav.		Single Ray Stackless Trav.				Packet Stackless	
Scene	N_{down}	N_{pops}	N_{down}	$N_{restart}$	N_{down}	N_{down}^{ENS}	N_{down}^{OPT}	N_{exits}	N_{down}	N_{exits}
Shirley6	17.5	3.64	30.9	3.64	17.5	11.5	6.23	3.64	12.0	3.64
Bunny	24.3	4.82	81.0	4.94	24.5	24.5	20.9	4.94	31.0	7.57
FairyForest	38.9	7.97	105	7.97	39.0	33.0	25.3	7.97	39.9	10.6
Conference	33.2	6.64	82.9	6.64	33.2	22.2	13.0	6.64	25.6	7.71

Table 8.3: *Number of steps for the different kd-tree traversal algorithms for primary rays. The down traversal steps are denoted with N_{down}, whereas the up-traversal steps that rely on a stack pop, restart, or leaf exit are denoted with N_{pops}, $N_{restart}$, and N_{exits} respectively. The down traversal steps for traversal that starts at a leaf (due to initial entry node search) are further marked with "ENS", and those for optimized ropes – with "OPT". All numbers are given as per ray average.*

Chapter 9

Shared Stack BVH Traversal

The work in this chapter follows our paper [Günther et al., 2007], where we present a new BVH packet traversal algorithm designed especially for the GPU. By allowing the rays to share the same traversal stack, our algorithm makes it possible to recursively traverse a BVH while keeping the stack in on-chip memory.

9.1 Background

As discussed in Section 7.3, the traversal of hierarchical acceleration structures has been historically difficult on the GPU, because of its recursive nature. To efficiently support recursion, traversal algorithms require a stack, but the limitations of DirectX9 and older hardware have made its implementation nearly impossible. With the introduction of the Tesla architecture from NVIDIA, it became possible to support random writes to GPU memory. However, due to the high latency of off-chip memory and the absence of memory caches, an efficient stack implementation was sill regarded as infeasible.

Before the method presented here, there have been multiple attempts at GPU ray tracing (see Sections 7.3 and 8.1). Among them only our work from the previous chapter and the work of Horn et al. [2007] were able to really achieve interactive frame rates. However, both of them were limited to medium-sized static scenes, mostly due to the use of KD-trees as an acceleration structure. This is because, on one hand KD-trees are slow to build and do not allow dynamic changes and refitting like BVHs, and on the other – their storage requirement is high due to the primitive duplication in the leafs. Additionally, using ropes as we did in the previous chapter further increases this requirements.

Recently, BVHs have become the acceleration structure of choice for interactive ray tracing. Built the right way, their performance is similar to the one of a KD-tree [Wald et al., 2007]. Furthermore, in contrast to a KD-tree, their size is determined only by the number of primitives as there is no primitive duplication in the leafs. Finally, they are much better suited for accelerating the ray tracing of animated scenes [Wald et al., 2007].

This is why we focus on GPU traversal of BVHs in this chapter. As already discussed, there have been two previous attempts for that: [Carr et al., 2006] and [Thrane and Simonsen, 2005]. The first work was based on geometry images and

was thus limited to scenes consisting of a single closed mesh only. The second work did not have scene limitations, but its performance was limited. This was mainly because it did not support ordered BVH traversal and early ray termination.

9.2 The Traversal Algorithm

In our approach, we use BVHs built according to SAH and in contrast to [Thrane and Simonsen, 2005] we support ordered, view dependent traversal.

To avoid the per-ray stack, previous GPU ray tracing implementations augmented the spatial indexing data structure in a way such that they could directly traverse from one node to another along the ray direction (see Chapter 8 and [Thrane and Simonsen, 2005]). Alternatively, they needed to restart traversal after each visited leaf [Foley and Sugerman, 2005]. This resulted in either a large spatial indexing structure or in sub-optimal traversal.

We solve the above problems by taking a different approach. Instead of fully removing the stack, we reduce its size to the point where it can fit in the shared memory of a GPU. We achieve this by using a BVH as an acceleration structure and by tracing the rays together in a packet.

As discussed in Section 7.1.5, BVHs are traversed in recursive manner, starting from the root and visiting only the subtrees of nodes that are intersected by the ray. The order of visiting the nodes is not important in the sense that any order will lead to the correct result. However, the most efficient traversal is obtained by ordering the nodes along the ray, according to the ray's entry distance. As discussed, this is achieved by looking locally at the entry distance to the two children while processing a node. Alternatively, the axis of maximum separation could be stored [Mahovsky, 2005] in the node, which saves computations during traversal but can sometimes lead to incorrect order for the children.

The key observation that enables our algorithm to work is that BVH traversal does not need to store the entry or exit distances or points in the traversal stack. They can be computed on the fly, since the AABB is always stored with the node. Thus, the per-ray traversal stack only holds pointers to the far nodes that still need to be traversed. Applied to packet traversal, this means again that the stack needs to hold only the far nodes that have to be visited by the packet in the future. Thus, if we use 32 bit pointers to the nodes (which is the case, because of the limited GPU memory), the amortized storage requirement of a ray becomes exactly 1 bit per node.

Our algorithm (see Algorithm 9.1) maps one ray to one thread and thus one packet to one SIMD width warp. It traverses the tree synchronously with the whole packet, working on one node at a time and processing the whole packet against it. If the node is a leaf, it intersects the rays in the packet with the contained geometry. Each thread keeps the distance to the closest intersected primitive found so far in a variable. If a closer primitive is found in the leaf, this variable is updated.

If the currently visited node is not a leaf, the algorithm intersects all rays of the packet with both children to determine the entry/exit distances. Each ray determines for itself which of the two children it intersects and in case it intersects both, in which order it wants to visit them (line 18). We use the slabs test [Kay and Kajiya, 1986] to determine if a ray intersects a node. In case it does, we also compare the entry

Algorithm 9.1 Shared Stack BVH Traversal

```
1: function ParallelRead(D)
                                  ▷ Reads D in parallel, assuming that |D| < 32
2:     i_t ← index of thread in warp
3:     T ← shared memory of size |D|
4:     if i_t < |D| then T[i_t] ← D[i_t]
5:     return T
6: end function

7: function Intersect(R)                                   ▷ ray R = (R_O, R_D)
8:     N ← pointer to the BVH root                  ▷ Currently processed node
9:     C ← (t = ∞)          ▷ Closest intersection, t denotes distance
10:    S ← the traversal stack stored in shared memory
11:    loop
12:        if N is leaf then
13:            Intersect R with contained geometry, updating C if necessary
14:            if StackIsEmpty(S) then break
15:            N ← PopFromStack(S)        ▷ Only thread 0 updates the top of S
16:        else
                                  ▷ N_L and N_R are stored in shared memory
17:            (N_L, N_R) ← ParallelRead(GetChildPointers(N))
18:            (λ1, λ2) ← R ∩ B(N_L)                  ▷ using [Kay and Kajiya, 1986]
19:            (μ1, μ2) ← R ∩ B(N_R)
20:            b1 ← (λ1 < λ2) ∧ (λ1 < C.t) ∧ (λ2 ≥ 0)          ▷ Ray visits N_L
21:            b2 ← (μ1 < μ2) ∧ (μ1 < C.t) ∧ (μ2 ≥ 0)          ▷ Ray visits N_R
22:            B1 ← PramOr(b1)                                  ▷ Packet visits N_L
23:            B2 ← PramOr(b2)                                  ▷ Packet visits N_R
24:            if B1 ∧ B2 then
25:                v ← { −1 , b1 ∧ (λ1 < μ1 ∨ ¬b2)
                       {  0 , ¬b1 ∧ ¬b2                        ▷ v ≡ ray's vote
                       {  1 , otherwise
26:                (N_N, N_F) ← (N_L, N_R)
27:                if PramSum(v) ≤ 0 then Swap(N_N, N_F)
28:                if first thread in warp then PushStack(S, N_F)
29:                N ← N_N
30:            else if B1 then
31:                N ← N_L
32:            else if B2 then
33:                N ← N_R
34:            else
35:                if StackIsEmpty(S) then break
36:                N ← PopFromStack(S)    ▷ Only thread 0 updates the top of S
37:            end if
38:        end if
39:    end loop
40: end function
```

distance of the ray to the closest so far found intersection distance. A ray only visits a node if the latter distance is larger than the first. Each ray that visits both children determines their order by comparing the respective entry distances.

Once each ray knows which children it wants to visit and in what order, the algorithm has to decide how to continue with the whole packet. If none of the rays wants to visit any of the two children, the algorithm takes the next node from the stack. In case some rays want to visit one child and none wants to visit the other one, this child becomes the next traversed node. Otherwise, if both nodes have to visited and all rays agree on a traversal order, the far node is put on the stack and the near one becomes the next visited node. Finally, if none of the previous conditions are met, then both nodes have to be traversed and the order of traversal is irrelevant (traversal is correct with any order, but potentially slower).

After the algorithm processes a leaf or if it processes a node where none of the rays wants to visit any of the two children, the next node to be traversed is taken from the stack. If the stack is empty at that point the algorithm terminates.

9.2.1 Voting

To order the children of a node, based on the traversal preferences of packet's rays, we use a voting process and PRAM reduction (line 25). Each thread votes with -1 if it wants to visit the left node first or only the left node, with 1 if it wants to visit the right one first (or only the right one) and with 0 if it does not want to visit any of the nodes. Then, the packet takes the sum of all votes and visits the left one first, if this sum is smaller than 1 and the right one otherwise. The sum is computed using a PRAM prefix sum in $O(\log N)$ and takes exactly 5 steps for the 32-wide SIMD unit of the Tesla architecture.

It is easy to see that the above presented voting scheme always takes the right decision when both nodes have to be traversed. In case all rays that visit the current node (a.k.a. *active rays*) want to intersect both its children and they want to do it in the same order, the voting scheme will choose the near node correctly and the traversal will be able to make use of the entry distance in the far node for early ray termination. In case there is an incoherent decision, i.e. one ray wants to traverse left-to-right while another wants to traverse right-to-left, both children have to be visited anyway, so the order is not important. In case some of the active rays want to traverse only the child N_1 and the rest want to traverse both, but with N_1 first, the voting will again choose N_1 as near, enabling the traversal to skip the sibling of N_1 using early termination. In case some active rays want to traverse again only N_1 and the rest want to traverse both nodes but with N_1 as a far child, both children must be visited anyway, so their order does not matter. Finally, if some nodes want to traverse only the left child and some want to traverse only the right one, the traversal order is again not important.

The above voting scheme is used to determine the traversal order only in case both children have to be traversed. To determine which of the two children need to be actually traversed, our algorithm again uses voting implemented through a PRAM OR operation, running in $O(1)$. First, each ray votes with one if it wants to traverse

	KD-Tree w/ Ropes		Shared Stack BVH	
scene	primary	+2ndary	primary	+shadow
Fairy Forest	10.6	4.0	13.2 (14.6)	4.8
Conference	16.7	6.7	16 (19)	6.1
Soda Hall	—	—	13.6 (16.2)	5.7
Power Plant	—	—	6.4	2.9

Table 9.1: *Absolute ray tracing performance of our BVH-based GPU ray tracer. We measure the frames per second (FPS) on a NVIDIA Geforce GTX 8800 GPU at resolution 1024×1024. We compare it to the fastest previous GPU ray tracer (Chapter 8). Primary rays are eye-light shaded. Additionally we report performance numbers when illuminating with a single point light and tracing shadow rays. The numbers in brackets denote the FPS when using a precomputed triangle projection test [Wald et al., 2001].*

the left node. Then, in a second vote, it votes with one it it wants to traverse the right node.

9.3 Results and Discussion

We implemented the above algorithm inside a Whitted style ray-tracer. We used NVIDIA's CUDA [NVIDIA Corporation, 2007] and implemented the whole ray tracing pipeline in a single kernel. Even though we used the same CUDA compiler version as in the previous chapter, which was still in beta and did not aid us too much in reducing the register count, we were able to reach 63% occupancy of the GPU for primary rays with eye light shading and 38% with full Phong shading with shadows and multiple light sources. We did not tune our code additionally to reduce the register count.

To intersect the packet with a triangle, we use the general packet intersection algorithm presented in [Kensler and Shirley, 2006]. Our algorithm carries out all ray independent computations of the intersection using the first six threads of a warp (i.e. 6-wide SIMD). We work with the raw geometry directly and the only precomputed data structure that our ray tracer relies on is the BVH. Even though this decreases rendering speed by approximately 20%, compared to using a fast projected intersection test [Wald et al., 2001], it allows us to update the geometry and ray trace deformable scenes. Furthermore it allows us to store larger scenes in the size-limited GPU memory.

Table 9.1 presents the absolute ray tracing performance (excluding BVH construction time) of our GPU ray tracer for the scenes and viewpoints presented in Figure 9.1. For all our tests we have used a workstation equipped with a NVIDIA Geforce 8800GTX GPU. The table also compares the performance to the ray tracer from the previous chapter, running on the same hardware. Although kd-trees are usually more efficient for ray tracing than BVHs (see [Havran, 2000]) on the CPU, the BVH ray tracer actually achieves comparable and even slightly faster frame rates on the GPU. One obvious reason is that our parallel BVH traversal algorithm has

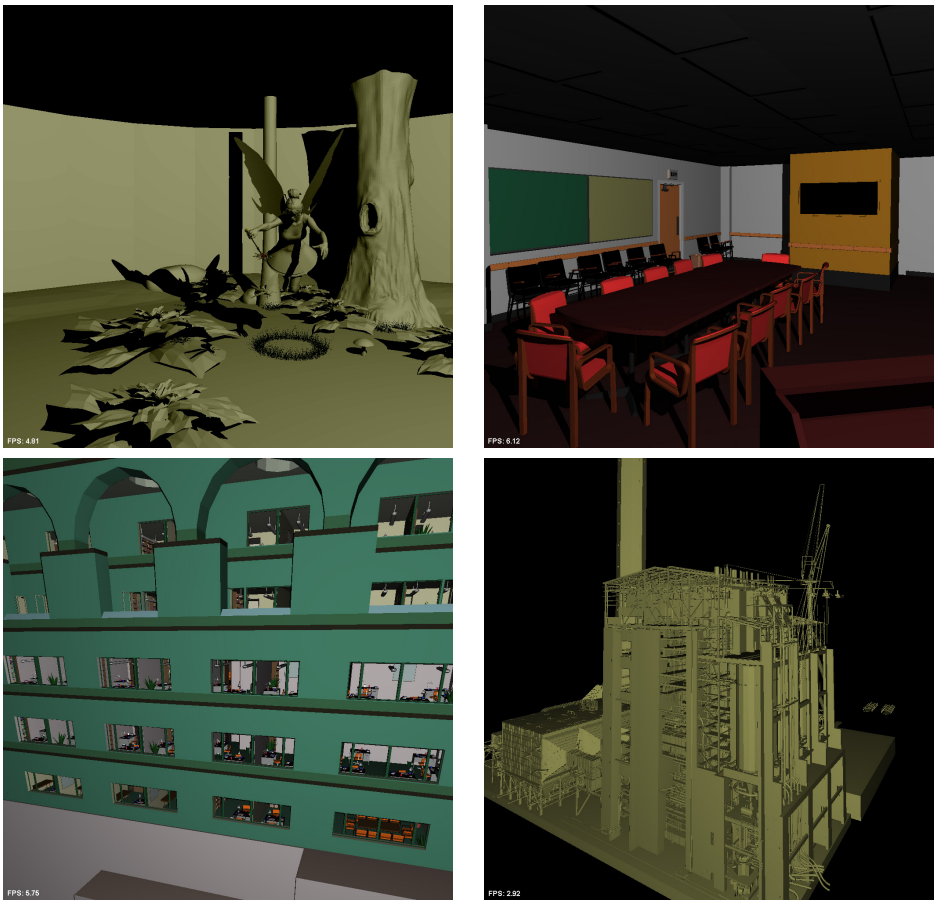


Figure 9.1: *From left to right and top to bottom:* Fairy Forest, Conference, Soda Hall, *and* Power Plant, *rendered on a GeForce 8800GTX at* 1024×1024, *with one light source and respective frame rates of 4.8, 6.1, 5.7, and 2.9 FPS.*

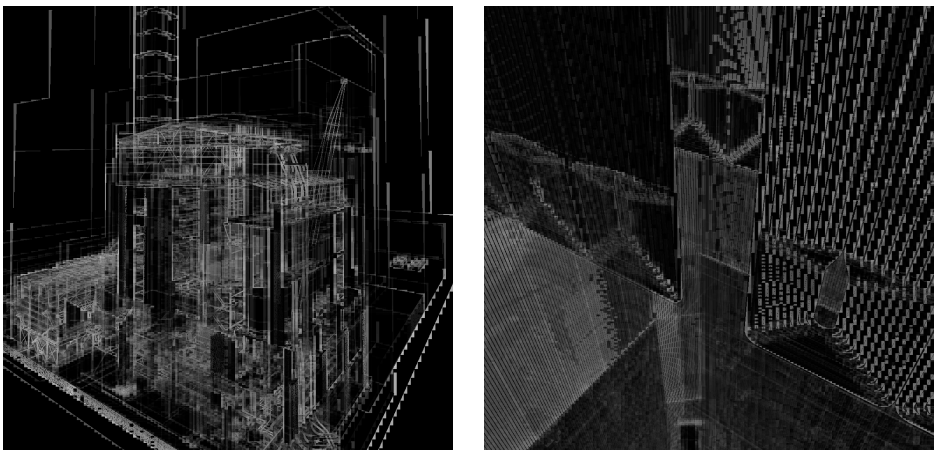


Figure 9.2: *Visualization of the SIMD utilization during traversal of the complex* Power Plant *scene. Brightness indicates the percentage of inactive traversal steps.*

a less complex implementation and uses less live registers and thus we get a higher GPU utilization of 63% compared to the 33% of Chapter 8 for primary rays. Another possible reason is that BVHs, being shallower than KD-trees, introduce less divergence and are thus more packet and GPU friendly.

Naturally, the efficiency of our packet traversal algorithm depends on the coherence of the traversal decisions of the rays in a packet. In Figure 9.2 we display the ratio of inactive traversal steps of a ray to the number of all traversal steps of its packet. The incoherent traversal decisions are clearly visible on the object boundaries. For the two shown views of the complex Power Plant scene the average SIMD utilization is still about 88% and 85%, respectively.

9.4 Summary

In this chapter we presented a packet traversal algorithm for BVHs that amortizes the storage requirements of the traversal stack over all rays and is thus very well adapted to GPU implementations. Besides showing the fastest GPU ray tracer at the time this research was done, the main contributions of the work presented in this chapter and in our paper [Günther et al., 2007] were: 1) showing for the first time that BVHs are actually not only viable for GPU ray tracing but they are actually the fastest alternative and 2) considerably pushing the limit of the scene size that GPU ray tracing can handle (by almost two orders of magnitude). Currently, BVHs are still considered as the best choice of acceleration structure for GPU ray tracing on modern hardware.

As of today there are more efficient methods for traversing BVHs on the GPU (e.g. [Aila and Laine, 2009]) and we discuss them in Chapter 10. Nevertheless, the preferred acceleration structure for GPU ray tracing remains the BVH.

Part III

Conclusion

Chapter 10

Conclusion

In this thesis, we tried to achieve two interconnected goals: develop new fast construction algorithms that would allow dynamic scenes to be ray traced at interactive speeds; and bring interactive ray-tracing to the commodity computer. To this end, we developed two new fast construction algorithms for KD-trees and BVHs (Chapter 4 and Chapter 5); we studied and improved the construction of acceleration structures w.r.t. traversal performance (Chapter 6); and we developed several new traversal algorithms for KD-trees and BVHs, specifically tailored to the GPU (Chapter 8 and Chapter 9). In this chapter, we will discuss shortly to what extent we were able to fulfill our goals with the presented work, how relevant the latter is as of today (i.e. several years later), and how it has evolved in follow-up work.

Fast Acceleration Structure Construction

Our extended algorithm from Chapter 4 remains in our opinion the fastest way to construct highly optimized KD-trees on the CPU.

In our implementation we were not able to achieve interactive speeds except in the case of small models. This issue is addressed in a follow-up implementation, presented in [Shevtsov et al., 2007], which is able to build a KD tree even for large models (i.e. Soda Hall with 2 million polygons) several times per second. This is achieved by a carefully optimized parallel implementation and a parallel initial decomposition.

Ideas from our KD-tree construction algorithm have also been used for parallel construction of SAH KD-trees on the GPU in [Danilewski et al., 2010]. There, the authors develop several alternative binning construction techniques. For each node they select one of them, based on the number of primitives in the node. This way, they perform efficient load balancing and achieve optimal GPU usage.

With regard to our fast BVH construction algorithm, there are two related papers that we would like to mention, namely [Wald, 2007; Lauterbach et al., 2009]. The first one, which was developed independently and in parallel to ours, explores the same problem and proposes a very similar solution. The second shows among others a GPU construction algorithm based on our work.

Performance-wise, we think that our algorithm and also [Wald, 2007], are still the fastest ways to build a SAH based BVH on the CPU. Even though there are two faster BVH construction algorithms, namely the LBVH [Lauterbach et al., 2009] and HLBVH [Pantaleoni and Luebke, 2010], they are not based on SAH, but rather

on a form of split in the middle. In the first one, this results in almost an order of magnitude slower traversal on all non-scanned models. The second one is an extension to the first that allows for deeper LBVHs to be built. Unfortunately, due to the way the authors present the results and due to the chosen scenes, it is not known how their traversal performance compares to a SAH built BVH.

Construction of Optimized Acceleration Structures

Interestingly, the initial goal of our work in Chapter 6 was to develop an algorithm which can construct BVHs in a non-greedy way. We expected to improve their traversal performance in this way. As described in that chapter however, even though we achieved this goal, the results did not meet our expectations. This motivated the further research presented there, which lead to a new acceleration structure, its construction algorithm, and an interesting conclusions about the SAH cost function.

Independent and in parallel to our work, Stich et al [2009] developed an algorithm which constructs trees with very similar performance to ours and in a similar fashion to our spatial construction algorithm (Section 6.5). Their solution is currently built into the Optix ray tracing framework [Parker et al., 2010] and the BVHs it produces are regarded as the most optimal acceleration structures for GPU traversal.

GPU Ray Tracing

With regard to GPU ray tracing, we think that our work from chapters 8 and 9 achieves its goals, bringing interactive ray tracing to commodity desktop computers. In the recent years our work has been largely superseded by [Aila and Laine, 2009]. The latter introduces several single-ray traversal algorithms that considerably outperform our packet based ones, even for primary rays. As a result, it further increases the performance gap between GPU and CPU ray tracing. Even though there is no consensus currently how large this gap is, in our opinion ray tracing based rendering runs approximately an order of magnitude faster on the GPU than on all cores together of a modern CPU. We base this statement on recent unpublished experiments of ours, for which we use recursive Whitted style ray tracing combined with instant radiosity and direct Monte-Carlo illumination sampling. The hardware we test on is a NVIDIA Quadro 6000 GPU and an Intel Core i7 2600K CPU, running at 4.3Ghz.

Final Words

At present, real-time ray tracing on commodity computers has become a fact. Even more, it has been recognized by many big companies involved in graphics, and some of them already provide commercially developed libraries for it (e.g. NVIDIA's Optix and Intel's Embree). However, ray tracing is still not fast enough. Even the fastest ray tracers today provide a budget of only a few rays-per-pixel-per-frame for real-time rendering. And while this might be enough for some scenarios of Whitted style ray tracing, it is definitely not enough for more advanced rendering algorithms (i.e. for global illumination).

In our opinion, interactive rendering based on ray tracing will continue to evolve. On one hand, the underlying implementation will get continually faster, through the

introduction of faster and more efficient hardware and through better ray-tracing algorithms. This low-level research will be performed mainly by the hardware companies. On the other hand, researchers and companies that use this technology will be the driving force of new rendering algorithms, that push the limit of what can be done with the currently available ray-budget.

In our opinion, it is this second direction that is currently exciting to explore. Besides for physically correct rendering, interactive ray tracing can be used as a tool for efficient sampling. This opens the door to a whole new and unexplored area of interactive rendering algorithms, that render approximate solutions but are based on physically correct principals. These algorithms will be able to correctly approximate "difficult" light transport effects, previously impossible with rasterization, and they will considerably further enhance the realism of interactive applications.

Appendix A

Proofs

A.1 Numerical Stability of In-Place Sifting

As mentioned in Section 4.5.4, there is no guarantee that a point falling in bin i will really be between the samples v_i and v_{i+1} (i.e. $x \in \left[v_{i(x)}, v_{i(x)+1}\right)$), since calculations are performed with limited precision. We prove here, that calculating the bin number with the floating point unit set to round-down and calculating the sample position in round-up mode will yield the desired results.

A floating point unit (FPU) of a CPU works on a discrete set of *representable* real numbers $\mathcal{F} = f_0 < f_1 < \ldots < f_N$. Usually, $f_0 = -\infty$, $f_N = \infty$, and all other numbers are rational. The FPU has several *rounding modes*, and for each of them it uses a set of operators over $\mathcal{F}$. We are interested here in the round-down and round-up modes and their respective sets of operators $\hat{\circ}$ and $\check{\circ}$ with $\circ \in \{+, -, *, /, \lfloor\ \rfloor, \lceil\ \rceil\}$. The result of $a \check{\circ} b$ for a binary operator, or respectively $\circ a$ for a unary one, is the highest representable number that is less or equal to $a \circ b$ respectively $\circ a$. The same is true for the round up version $\hat{\circ}$ of the operator, where the lowest representable number that is greater or equal to the exact result is taken instead.

For our proof, we will need the four lemmas below. Since the first two are easy to prove, we will only state them here.

Lemma A.1.1. *Either $a \check{\circ} b < a \circ b < a \hat{\circ} b$ or $a \check{\circ} b = a \circ b = a \hat{\circ} b$. In any case, it holds that $a \check{\circ} b \leq a \circ b \leq a \hat{\circ} b$. Furthermore, $a \check{\circ} b$ and $a \hat{\circ} b$ are two consecutive representable numbers, and there is no representable number between them.*

Lemma A.1.2. *If $a_1 \leq a_2$ then $a_1 \check{\circ} b \leq a_2 \check{\circ} b$ and $a_1 \hat{\circ} b \leq a_2 \hat{\circ} b$ for any $\circ \in \{+, -, *, /\}$. Furthermore $\check{\circ} a_1 \leq \check{\circ} a_2$ and $\hat{\circ} a_1 \leq \hat{\circ} a_2$ for the unary operator $\circ \in \{\lfloor\ \rfloor, \lceil\ \rceil\}$.*

Lemma A.1.3. *Let $f(a, b_N, \ldots, b_1)$ be a function that can be defined in the following*

manner:

$$f(a, b_1, \ldots, b_M) = f_N(a, b_1, \ldots, b_M)$$

$$f_i(a, b_1, \ldots, b_i) = \begin{cases} f_{i-1}(a, b_1, \ldots, b_{i-1}) \circ_{\mathrm{i}} b_i & \text{, for binary operators} \\ \circ_{\mathrm{i}} f_{i-1}(a, b_1, \ldots, b_i) & \text{, for unary operators} \end{cases}$$

$$f_0(a) = a$$

with $\circ_i \in \{+, -, *, /, \lfloor\ \rfloor, \lceil\ \rceil\}$. *In this case, if* $a_1 \leq a_2$ *then*

$$\check{\mathrm{f}}(a_1, b_1, \ldots, b_M) \leq \check{\mathrm{f}}(a_2, b_1, \ldots, b_M)$$

$$\hat{\mathrm{f}}(a_1, b_1, \ldots, b_M) \leq \hat{\mathrm{f}}(a_2, b_1, \ldots, b_M)$$

In the above, we have denoted with $\check{\mathrm{f}}(.)$ *and* $\hat{\mathrm{f}}(.)$ *the functions obtained by replacing each operator in* $f(.)$ *with its repsective round-up and round-down equivalent*

Proof. The proof follows from Lemma A.1.2 by induction. □

Lemma A.1.4. *Let* $\circ$ *be a binary operator from the set* $\{+, -, *, /\}$ *and let* $\diamond$ *be its inverse (i.e.* $(a \circ b) \diamond b = a$*). It holds that for any* a *and* b *from* $\mathcal{F}$

$$(a \,\check{\circ}\, b) \,\hat{\diamond}\, b \leq a \leq (a \,\hat{\circ}\, b) \,\check{\diamond}\, b \tag{A.1}$$

Proof. We will first prove that $(a \,\check{\circ}\, b) \,\hat{\diamond}\, b \leq a$. From Lemma A.1.1 we obtain that either

$$f_1 = (a \,\check{\circ}\, b) \,\check{\diamond}\, b < \underbrace{(a \,\check{\circ}\, b) \diamond b}_{f_2} < (a \,\check{\circ}\, b) \,\hat{\diamond}\, b = f_3 \tag{A.2}$$

or

$$f_1 = (a \,\check{\circ}\, b) \,\check{\diamond}\, b = \underbrace{(a \,\check{\circ}\, b) \diamond b}_{f_2} = (a \,\check{\circ}\, b) \,\hat{\diamond}\, b = f_3 \tag{A.3}$$

From Lemma A.1.1, it holds that $a \,\check{\circ}\, b \leq a \circ b$ and thus $f_2 = (a \,\check{\circ}\, b) \diamond b \leq (a \circ b) \diamond b = a$. It is easy to see, that if (A.3) holds then $f_3 = f_2 \leq a$. In the case that (A.2) holds, either $f_1 < f_2 \leq a < f_3$, or $f_1 < f_2 < f_3 \leq a$. According to Lemma A.1.1, there is no representable number between f_1 and f_3. However a is representable which makes the first inequality impossible. Thus, it holds that $f_3 \leq a$, which proves that $(a \,\check{\circ}\, b) \,\hat{\diamond}\, b \leq a$ in all cases. The second part of (A.1), namely that $a \leq (a \,\hat{\circ}\, b) \,\check{\diamond}\, b$ can be proven analogically, by applying Lemma A.1.1 to the $\diamond$ operator in $(a \,\hat{\circ}\, b) \diamond b$. □

We will now prove our original claim. For this purpose we revisit the functions $v(i)$ and $i(x)$ and slightly modify them:

$$S = b \,\check{-}\, a$$

$$j(x) = (M * (x - a)) \,/ S$$

$$i(x) = \lfloor j(x) \rfloor$$

$$v(i) = ((i * S)/M) - a$$

The modification consists in always calculating S in the same rounding mode. What we want to prove is that

$$\hat{v}\left(\check{i}(x)\right) \leq x < \hat{v}\left(\check{i}(x)+1\right) \tag{A.4}$$

We assume here that both integers $\check{i}(x)$ and $\check{i}(x)+1$ are representable, which is the case in our algorithm, since the number of bins is small and we use 32-bit floating point numbers [IEEE, 2008].

To prove (A.4), we look at the sequence $F = \{f_0 < f_1 < \ldots < f_N\}$ of all representable numbers, such that $\check{i}(f_k) = i_0$ for an arbitrary fixed i_0.

We first observe that F contains all representable numbers in the range $[f_0, f_N]$. To prove this, let f be such that $f_0 < f < f_N$. Applying Lemma A.1.3 to $\check{i}(x)$, we obtain that $\check{i}(f_0) \leq \check{i}(f) \leq \check{i}(f_N)$, which can only be possible if $i(f) = i_0$.

With this observation, proving A.4 becomes equivalent to proving

$$f_{-1} < \hat{v}\left(\check{i}(f_0)\right) \leq f_0 \tag{A.5}$$

with f_{-1} being the previous representable number before f_0.

The right side of (A.5) follows from A.1.4 and A.1.2.

$$\hat{v}\left(\check{i}(f_0)\right) = \hat{v}\left(\left\lfloor \check{j}(f_0) \right\rfloor\right) \leq \hat{v}\left(\check{j}(f_0)\right) = \tag{A.6}$$

$$= (((((f_0 \mathbin{\check{-}} a) \mathbin{\check{*}} M) \underbrace{\mathbin{\check{/}} S) \mathbin{\hat{*}} S)}_{A.1.4} \mathbin{\hat{/}} M) \mathbin{\hat{+}} a \leq \tag{A.7}$$

$$\leq (((f_0 \mathbin{\check{-}} a) \underbrace{\mathbin{\check{*}} M) \mathbin{\hat{/}} M)}_{A.1.4} \mathbin{\hat{+}} a \leq \tag{A.8}$$

$$\leq (f_0 \underbrace{\mathbin{\check{-}} a) \mathbin{\hat{+}} a}_{A.1.4} \leq f_0 \tag{A.9}$$

To prove the left side of (A.5), we assume the contrary, i.e. $f_{-1} \geq \hat{v}\left(\check{i}(f_0)\right)$. From Lemma A.1.3, $\check{j}(f_{-1}) \geq \check{j}\left(\hat{v}\left(\check{i}(f_0)\right)\right)$ and from A.1.4, $\check{j}\left(\hat{v}\left(\check{i}(f_0)\right)\right) \geq \check{i}(f_0)$. Thus, $\check{j}(f_{-1}) \geq \check{i}(f_0)$. From A.1.3, $\check{j}(f_0) \geq \check{j}(f_{-1})$. Thus $\check{j}(f_0) \geq \check{j}(f_{-1}) \geq \check{i}(f_0)$, which implies that $\check{i}(f_{-1}) = \check{i}(f_0)$, as $i(.)$ is the floor of $f(.)$. This is however not possible, due to the way we defined f_0 and f_{-1}. We have reached a contradiction, which proves that the left side of (A.5) is correct.

Appendix B

Common Notation

$\mathcal{A}(R)$	The surface are of the convex shape or region of space R.
AABB	Axis Aligned Bounding Box.
$\mathcal{B},\ \mathcal{B}(p),\ \mathcal{B}(N)$	An AABB, the tightest AABB around a primitive p, or the AABB corresponding to a node N.
$\mathcal{B}(P_1, P_2)$	The AABB defined by the diagonal P_1P_2.
$\mathcal{B}^d_{min}, \mathcal{B}^d_{max}$	The minimum respectively maximum of the AABB along the axis d. Can also appear as $\mathcal{B}^d_{min}(.)$ and $\mathcal{B}^d_{max}(.)$.
$\mathcal{B}^d_{size}$	The size of an AABB along the dimension d. Can also appear as $\mathcal{B}^d_{size}(.)$.
$C(p)$	The centroid of the primitive p.
$cost(v)$	The objective function to be minimized by a sweep plane algorithm.
$\mathcal{C}_T, \mathcal{C}_I$	The traversal and intersection costs respectively.
d	A dimension ($d \in \{x, y, z\}$ or $d \in \{0, 1, 2\}$).
$Exp(T), Exp(N)$	The expected cost of the tree T or node N for ray tracing according to the surface area cost model.
$\underset{SAH}{Exp(N)}$	The expected SAH cost of a node N.
$h(x, \omega)$	The ray tracing operator. Returns the first surface point intersected by the ray with origin x and direction ω.
N	A node from a tree based acceleration structure.
N_L, N_R	The left respectively right child of the node N in a binary tree acceleration structure.
p	A primitive.
P, P_E, P_P, P_X	In traversal algorithms: the current position along the ray as a point, the entry point, the exit point, and the intersection point of the ray with the split plane.
$R(N)$	The region of space corresponding to the node N.
R, R_O, R_D	In traversal: a ray, its origin, and its direction.
S	The set of all primitives in the scene
$S(N)$	The set of primitives in the sub-tree with root N. Typically used during construction.
S_L, S_R	Equivalent to $S(N_L)$ and $S(N_R)$ respectively.

$S_{\mathcal{B}}, S_{\mathcal{B}}(N)$	A set of AABBs, obtained by clipping primitives, or alternatively their AABBs, with the AABB of a node. More formally $S_{\mathcal{B}}(N) = \{\mathcal{B}(p \cap \mathcal{B}(N)) \mid p \in S(N)\}$ in the first case, and $S_{\mathcal{B}}(N) = \{\mathcal{B}(p) \cap \mathcal{B}(N) \mid p \in S(N)\}$ – in the second.
t_e, t_x	In traversal: the entry and exit distances of the ray w.r.t. some node.
v_{start}, v_{end}	The start and end events of a sweep plane algorithm.
v_i	An event from the sweep plane algorithm.
$V(x, y)$	The binary visibility operator. Has a value of 1 iff x and y are mutually visible and 0 otherwise.
x^d	The value of the scalar component of the point x along the dimension d.

Bibliography

Aila, T. and Laine, S. (2009). Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 145–149, New York, NY, USA. ACM.

Akenine-Möller, T., Haines, E., and Hoffman, N. (2008). *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA.

Apodaca, A. and Mantle, M. (1990). RenderMan: pursuing the future of graphics. *Computer Graphics and Applications, IEEE*, 10(4):44 –49.

Benthin, C. (2006). *Realtime Ray Tracing on Current CPU Architectures*. PhD thesis, Saarland University.

Bigler, J., Stephens, A., and Parker, S. (2006). Design for parallel interactive ray tracing systems. In *Interactive Ray Tracing 2006, IEEE Symposium*, pages 187 –196.

Blender. `http://www.blender.org/`.

Blinn, J. F. (1977). Models of light reflection for computer synthesized pictures. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '77, pages 192–198, New York, NY, USA. ACM.

Boulos, S., Edwards, D., Lacewell, J. D., Kniss, J., Kautz, J., Shirley, P., and Wald, I. (2007). Packet-Based Whitted and Distribution Ray Tracing. In *Proceedings of Graphics Interface 2007*.

Carr, N. A., Hall, J. D., and Hart, J. C. (2002). The ray engine. In *Proceedings of Graphics Hardware*, pages 37–46. Eurographics Association.

Carr, N. A., Hoberock, J., Crane, K., and Hart, J. C. (2006). Fast GPU ray tracing of dynamic meshes using geometry images. In *Proceedings of Graphics Interface*. A.K. Peters.

Catmull, E. E. (1974). *A subdivision algorithm for computer display of curved surfaces*. PhD thesis, University of Utah. AAI7504786.

Christen, M. (2005). Ray Tracing auf GPU. Master's thesis, Fachhochschule beider Basel.

Clark, J. H. (1976). Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, 19:547–554.

Cleary, J. G. and Wyvill, G. (1988). Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer*, 4:65–83. 10.1007/BF01905559.

Cook, R. L. and Torrance, K. E. (1982). A reflectance model for computer graphics. *ACM Trans. Graph.*, 1:7–24.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms.* The MIT Press.

Crow, F. C. (1977). Shadow algorithms for computer graphics. *SIGGRAPH Comput. Graph.*, 11:242–248.

Dammertz, H., Hanika, J., and Keller, A. (2008). Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. In *Computer Graphics Forum (Proc. 19th Eurographics Symposium on Rendering)*, pages 1225–1234.

Dammertz, H. and Keller, A. (2008). Edge Volume Heuristic – Robust Triangle Subdivision for Improved BVH Performance. In *IEEE/Eurographics Symposium on Interactive Ray Tracing*.

Danilewski, P., Popov, S., and Slusallek, P. (2010). Binned SAH Kd-Tree Construction on a GPU. Technical report, Saarland University, Computer Graphics Lab.

de Berg, M., Cheong, O., van Kreveld, M., and Overmars, M. (2008). *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition.

Donnelly, W. and Lauritzen, A. (2006). Variance shadow maps. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, I3D '06, pages 161–165, New York, NY, USA. ACM.

Dutre, P., Bala, K., Bekaert, P., and Shirley, P. (2006). *Advanced Global Illumination*. AK Peters Ltd.

Erikson, C. M. (2000). *Hierarchical levels of detail to accelerate the rendering of large static and dynamic polygonal environments*. PhD thesis, The University of North Carolina at Chapel Hill. AAI9968589.

Ernst, M. and Greiner, G. (2007). Early Split Clipping for Bounding Volume Hierarchies. *Symposium on Interactive Ray Tracing*, 0:73–78.

Ernst, M. and Greiner, G. (2008). Multi bounding volume hierarchies. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pages 35 –40.

Ernst, M., Vogelgsang, C., and Greiner, G. (2004). Stack implementation on programmable graphics hardware. In *VMV*, pages 255–262.

Fisher, J. A. (1983). Very long instruction word architectures and the eli-512. In *Proceedings of the 10th annual international symposium on Computer architecture*, ISCA '83, pages 140–150, New York, NY, USA. ACM.

Foley, T. and Sugerman, J. (2005). Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '05, pages 15–22, New York, NY, USA. ACM.

Fortune, S. and Wyllie, J. (1978). Parallelism in random access machines. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118. ACM Press.

Fujimoto, A., Tanaka, T., and Iwata, K. (1986). ARTS: Accelerated Ray-Tracing System. *IEEE Computer Graphics and Applications*, 6:16–26.

Glassner, A. S. (1984). Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22.

Goldsmith, J. and Salmon, J. (1987). Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7:14–20.

Goral, C. M., Torrance, K. E., Greenberg, D. P., and Battaile, B. (1984). Modeling the interaction of light between diffuse surfaces. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 213–222, New York, NY, USA. ACM.

Günther, J., Friedrich, H., Seidel, H.-P., and Slusallek, P. (2006a). Interactive ray tracing of skinned animations. *The Visual Computer*, 22(9):785–792. (Proceedings of Pacific Graphics).

Günther, J., Friedrich, H., Wald, I., Seidel, H.-P., and Slusallek, P. (2006b). Ray tracing animated scenes using motion decomposition. *Computer Graphics Forum*, 25(3):517–525. (Proceedings of Eurographics).

Günther, J., Popov, S., Seidel, H.-P., and Slusallek, P. (2007). Realtime ray tracing on GPU with BVH-based packet traversal. In *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*, pages 113–118.

Hanrahan, P. and Lawson, J. (1990). A language for shading and lighting calculations. In *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '90, pages 289–298, New York, NY, USA. ACM.

Havran, V. (1997). Cache sensitive representation for the BSP tree. In *Compugraphics'97*, pages 369–376. GRASP – Graphics Science Promotions & Publications.

Havran, V. (2000). *Heuristic Ray Shooting Algorithms*. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

Havran, V., Bittner, J., and Žára, J. (1998a). Ray tracing with rope trees. In Szirmay-Kalos, L., editor, *14th Spring Conference on Computer Graphics*, pages 130–140.

Havran, V., Kopal, T., Bittner, J., and Žára, J. (1998b). Fast robust BSP tree traversal algorithm for ray tracing. *J. Graph. Tools*, 2:15–23.

Helmholtz, H. V. (1867). *Handbuch der Physiologischen Optik*. Leopold Voss, Leipzig.

Horn, D. R., Sugerman, J., Houston, M., and Hanrahan, P. (2007). Interactive k-d tree GPU raytracing. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, I3D '07, pages 167–174, New York, NY, USA. ACM.

Houston, M. (2006). Performance analysis and architecture insights. In *SUPERCOMPUTING 2006 Tutorial on GPGPU, Course Notes*. `http://www.gpgpu.org/sc2006/slides/10.houston-understanding.pdf`.

Hunt, W., Mark, W. R., and Stoll, G. (2006). Fast kd-tree construction with an adaptive error-bounded heuristic. In *2006 IEEE Symposium on Interactive Ray Tracing*. IEEE.

IEEE (2008). IEEE standard for Floating-Point arithmetic. *IEEE Std 754-2008*, pages 1–58.

Jansen, F. W. (1986). Data structures for ray tracing. In *Proceedings of a workshop (Eurographics Seminars on Data structures for raster graphics*, pages 57–73, New York, NY, USA. Springer-Verlag New York, Inc.

Jensen, H. W. (1996). Global illumination using photon maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, pages 21–30, London, UK. Springer-Verlag.

Jensen, H. W. (2001). *Realistic image synthesis using photon mapping*. A. K. Peters, Ltd., Natick, MA, USA.

Jensen, H. W., Marschner, S. R., Levoy, M., and Hanrahan, P. (2001). A practical model for subsurface light transport. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 511–518, New York, NY, USA. ACM.

Kajiya, J. T. (1986). The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150.

Kalojanov, J., Billeter, M., and Slusallek, P. (2011). Two-Level Grids for Ray Tracing on GPUs. In Min Chen, O. D., editor, *EG 2011 - Full Papers*, pages 307–314, Llandudno, UK. Eurographics Association.

Kalojanov, J. and Slusallek, P. (2009). A parallel algorithm for construction of uniform grids. In *HPG '09: Proceedings of the 1st ACM conference on High Performance Graphics*, pages 23–28, New York, NY, USA. ACM.

Kaplan, M. R. (1985). Space-tracing: A constant time ray-tracer. *Computer Graphics*, 19(3):149–158. (Proceedings of SIGGRAPH 85 Tutorial on Ray Tracing).

Karlsson, F. (2004). Ray tracing fully implemented on programmable graphics hardware. Master's thesis, Chalmers University of Technology.

Kay, T. L. and Kajiya, J. T. (1986). Ray tracing complex scenes. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '86, pages 269–278, New York, NY, USA. ACM.

Keller, A. (1997). Instant radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 49–56, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

Kensler, A. (2008). Tree Rotations for Improving Bounding Volume Hierarchies. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pages 73–76.

Kensler, A. and Shirley, P. (2006). Optimizing ray-triangle intersection via automated search. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 33–38.

Lafortune, E. P. and Willems, Y. D. (1993). Bi-directional path tracing. In *Proceedings of the Third International Conference on Computer Graphics and Visualization Techniques*, pages 145–153.

Lafortune, E. P. F., Foo, S.-C., Torrance, K. E., and Greenberg, D. P. (1997). Non-linear approximation of reflectance functions. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97, pages 117–126, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co.

Lauterbach, C., eui Yoon, S., and Manocha, D. (2006). RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In *In Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 39–45.

Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., and Manocha, D. (2009). Fast BVH Construction on GPUs. *Computer Graphics Forum*, 28(2):375–384.

Lext, J. and Akenine-möller, T. (2001). Towards Rapid Reconstruction for Animated Ray Tracing. In *EUROGRAPHICS 2001/Short Presentations*.

MacDonald, J. D. and Booth, K. S. (1989). Heuristics for ray tracing using space subdivision. In *Graphics Interface Proceedings 1989*, pages 152–163, Wellesley, MA, USA. A.K. Peters, Ltd.

Mahovsky, J. (2005). *Ray Tracing with Reduced-Precision Bounding Volume Hierarchies*. PhD thesis, The University of Calgary.

Musser, D. R. (1997). Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993.

Muuss, M. J. (1995). Towards real-time ray-tracing of combinatorial solid geometric models. In *Proceedings of BRL-CAD Symposium*.

Ng, K. and Trifonov, B. (2003). Automatic bounding volume hierarchy generation using stochastic search methods. In *CPSC532D Mini-Workshop "Stochastic Search Algorithms"*.

NVIDIA Corporation (2007). *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide.*

NVIDIA Corporation (2009). *Fermi Compute Architecture Whitepaper.*

Overbeck, R., Ramamoorthi, R., and Mark, W. R. (2008). Large Ray Packets for Real-time Whitted Ray Tracing. In *IEEE/EG Symposium on Interactive Ray Tracing (IRT)*, pages 41—-48.

Pabst, H.-F., Springer, J. P., Schollmeyer, A., Lenhardt, R., Lessig, C., and Froehlich, B. (2006). Ray casting of trimmed NURBS surfaces on the GPU. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 151–160.

Pantaleoni, J. and Luebke, D. (2010). HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 87–95, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

Parker, S., Martin, W., Sloan, P.-P. J., Shirley, P., Smits, B., and Hansen, C. (1999). Interactive ray tracing. In *In Symposium on interactive 3D graphics*, pages 119–126.

Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M. (2010). OptiX: a general purpose ray tracing engine. In *ACM SIGGRAPH 2010 papers*, SIGGRAPH '10, pages 66:1–66:13, New York, NY, USA. ACM.

Pharr, M. and Humphreys, G. (2004). *Physically Based Rendering: From Theory to Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Phong, B. T. (1975). Illumination for computer generated pictures. *Commun. ACM*, 18:311–317.

Pixar (1989). The renderman interface.

Popov, S., Georgiev, I., Dimov, R., and Slusallek, P. (2009). Object partitioning considered harmful: space subdivision for BVHs. In *HPG '09: Proceedings of the Conference on High Performance Graphics 2009*, pages 15–22, New York, NY, USA. ACM.

Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P. (2006). Experiences with streaming construction of SAH KD-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 89–94.

Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P. (2007). Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum*, 26(3). (Proceedings of Eurographics), to appear.

Purcell, T. J. (2004). *Ray Tracing on a Stream Processor.* PhD thesis, Stanford University.

Purcell, T. J., Buck, I., Mark, W. R., and Hanrahan, P. (2002). Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, 21(3):703–712.

Reinhard, E., Smits, B. E., and Hansen, C. (2000). Dynamic acceleration structures for interactive ray tracing. In *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*, pages 299–306, London, UK. Springer-Verlag.

Reshetov, A. (2006). Omnidirectional ray tracing traversal algorithm for kd-trees. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 57–60.

Reshetov, A., Soupikov, A., and Hurley, J. (2005). Multi-level ray tracing algorithm. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1176–1185, New York, NY, USA. ACM.

Ritschel, T., Grosch, T., Kim, M. H., Seidel, H.-P., Dachsbacher, C., and Kautz, J. (2008). Imperfect Shadow Maps for Efficient Computation of Indirect Illumination. *ACM Trans. Graph. (Proc. of SIGGRAPH ASIA 2008)*, 27(5).

Samet, H. (1984). The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260.

Samet, H. (1989). Implementing ray tracing with octrees and neighbor finding. *Computers and Graphics*, 13(4):445–60.

Santaló, L. A. (1976). Integral Geometry and Geometric Probability. In Rota, G. C., editor, *Encyclopedia of Mathematics and its Applications*, volume 1. Addison-Wesley.

Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., and Hanrahan, P. (2008). Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):18:1–18:15.

Shevtsov, M., Soupikov, A., and Kapustin, A. (2007). Highly Parallel Fast KD-tree Construction for Interactive Ray Tracing of Dynamic Scenes. *Comput. Graph. Forum*, 26(3):395–404.

Stich, M., Friedrich, H., and Dietrich, A. (2009). Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 7–13, New York, NY, USA. ACM.

Stoll, C., Gumhold, S., and Seidel, H.-P. (2006). Incremental raycasting of piecewise quadratic surfaces on the GPU. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 141–150.

Subramanian, K. R. and Fussell, D. S. (1991). Automatic termination criteria for ray tracing hierarchies. In *Graphics Interface '91*, pages 93–100.

Sutherland, I. E. and Hodgman, G. W. (1974). Reentrant polygon clipping. *Commun. ACM*, 17(1):32–42.

Teller, S. J. (1992). *Visibility computations in densely occluded polyhedral environments*. PhD thesis, Berkeley, CA, USA. UMI Order No. GAX93-30757.

Thrane, N. and Simonsen, L. O. (2005). A comparison of acceleration structures for gpu assisted ray tracing. Master's thesis, University of Aarhus.

Trendall, C. and Stewart, A. J. (2000). General calculations using graphics hardware, with applications to interactive caustics. In *In Eurographics Workshop on Rendering*, pages 287–298. Springer.

Tsakok, J. A. (2009). Faster incoherent rays: Multi-BVH ray stream tracing. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 151–158, New York, NY, USA. ACM.

Veach, E. (1998). *Robust monte carlo methods for light transport simulation*. PhD thesis, Stanford University, Stanford, CA, USA. Adviser-Guibas, Leonidas J.

Wächter, C. and Keller, A. (2006). Instant ray tracing: The bounding interval hierarchy. In *In Proceedings of the Eurographics Symposium on Rendering*, pages 139–149.

Wald, I. (2004). *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University.

Wald, I. (2007). On fast Construction of SAH based Bounding Volume Hierarchies. In *Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing*.

Wald, I., Benthin, C., and Boulos, S. (2008). Getting Rid of Packets – Efficient SIMD Single-Ray Traversal using Multi-Branching BVHs. In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2008*.

Wald, I., Benthin, C., and Slusallek, P. (2003). Distributed interactive ray tracing of dynamic scenes. In *PVG '03: Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, page 11, Washington, DC, USA. IEEE Computer Society.

Wald, I., Boulos, S., and Shirley, P. (2007). Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1).

Wald, I. and Havran, V. (2006). On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 61–69.

Wald, I., Ize, T., Kensler, A., Knoll, A., and Parker, S. G. (2006a). Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics*, 25(3):485–493. (Proceedings of ACM SIGGRAPH).

Wald, I., Ize, T., Kensler, A., Knoll, A., and Parker, S. G. (2006b). Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics*, pages 485–493. (Proceedings of ACM SIGGRAPH 2006).

Wald, I., Slusallek, P., Benthin, C., and Wagner, M. (2001). Interactive rendering with coherent ray tracing. In *Computer Graphics Forum*, pages 153–164.

Walter, B., Bala, K., Kulkarni, M., and Pingali, K. (2008). Fast agglomerative clustering for rendering. In *Proceedings of the 2008 IEEE Symposium on Interactive Ray Tracing*, pages 81 –86.

Walter, B., Drettakis, G., and Parker, S. (1999). Interactive rendering using the render cache. In Lischinski, D. and Larson, G., editors, *Rendering techniques '99 (Proceedings of the 10th Eurographics Workshop on Rendering)*, volume 10, pages 235–246, New York, NY. Springer-Verlag/Wien.

Ward, G. and Simmons, M. (1999). The holodeck ray cache: an interactive rendering system for global illumination in nondiffuse environments. *ACM Trans. Graph.*, 18:361–368.

Ward, G. J. (1992). Measuring and modeling anisotropic reflection. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '92, pages 265–272, New York, NY, USA. ACM.

Watt, A. (1993). *3d Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

Whitted, T. (1980). An improved illumination model for shaded display. *Commun. ACM*, 23:343–349.

Williams, L. (1978). Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12:270–274.

Woop, S., Marmitt, G., and Slusallek, P. (2006). B-KD trees for hardware accelerated ray tracing of dynamic scenes. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 67–77, New York, NY, USA. ACM.

Woop, S., Schmittler, J., and Slusallek, P. (2005). RPU: A programmable ray processing unit for realtime ray tracing. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2005)*, 24(3):434–444.

Zhang, Y., Peng, L., Li, B., Peir, J.-K., and Chen, J. (2011). Architecture comparisons between Nvidia and ATI GPUs: Computation parallelism and data communications. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 205 –215.

Zhou, K., Hou, Q., Wang, R., and Guo, B. (2008). Real-time KD-tree construction on graphics hardware. In *ACM SIGGRAPH Asia 2008 papers*, SIGGRAPH Asia '08, pages 126:1–126:11, New York, NY, USA. ACM.

List of Figures

List of Tables

List of Algorithms