LUDWIG-MAXIMILIANS-UNIVERSITÄT AT MÜNCHEN
Department “Institut für Informatik”
Lehr- und Forschungseinheit Medieninformatik
Prof. Dr. Butz

Bachelor’s Thesis

Progressive BVH Refinement in Interactive Ray Tracing

Christian Schmidt
nichtchristianschmidt@gmail.com

Bearbeitungszeitraum:	06.05.2021 bis 23.09.2021
Betreuer:	Changkun Ou
Verantw. Hochschullehrer:	Prof. Dr. Butz

Aufgabenstellung

Progressive BVH Refinement in Interactive Ray Tracing

Problem Statement Path tracing in real time become avaliable on the GPU side in recent years due to the recent advances in image denoising techniques, such as NVIDIA DLSS.

It is interesting to optimize and render everything directly on the CPU in real time for reasonably smaller scenes.

Tasks
- Implement a multi-threaded CPU ray tracer
- Profile and identify the bottleneck of your ray tracer implementation
- Benchmark and compare the performance difference between your CPU ray tracer and an equivalent CUDA ray tracer
- Summarize your findings in a thesis and presenting them to audiences

Requirements
- Have experience or projects using C++ (or Go)
- General knowledge about computer graphics

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

München, September 23, 2021 .....

Acknowledgments

The meshes used to evaluate the path tracer are courtesy of Morgan McGuire's Computer Graphics Archive [McGuire, 2017] and the Standford 3D Scanning Repository. I would like to thank my advisor Chankun Ou for his patience and for all his feedback and input in countless meetings. His contributions in the form of the bench [Ou, 2020a] framework and a Bayesian optimizer [Ou, 2019] helped my work significantly. Finally, I would like to thank my CPU for taking quite the beating in several benchmarking and rendering sessions.

Abstract

Path tracing is a rendering technique that simulates how light travels through a scene, producing physically correct images, albeit at a significant computational cost. Only in recent years has this technology become viable in real-time rendering through powerful denoising techniques and special purpose hardware. This thesis continues this work by first presenting a self-contained overview over the basics of path tracing. A closer look at CPU based interactive path tracing is provided and its performance is evaluated based on a path tracer written from scratch in Go. A particular focus is set on acceleration data structures, which are used to optimize the ray tracing process significantly. Consequently, the state-of-the-art bounding volume hierarchy construction algorithm called progressive hierarchical refinement is presented and applied to the interactive application. Finally, a novel approach for evaluating the build-trace trade-off of that algorithm is proposed and the optimization problem is solved using grid search and Bayesian optimization.

Path Tracing ist eine Technik zur Bildsynthese, welche den Lichttransport durch eine Szene simuliert. Dadurch werden physikalisch plausible Bilder erzeugt, wenn auch auf Kosten eines hohen Rechenaufwands. Erst in den letzten Jahren wurde diese Technologie auch im Real-Time Rendering realisierbar, vor allem durch leistungsstarke Rauschunterdrückungsverfahren und spezielle Hardware. Diese Arbeit knüpft an diese Fortschritte an, indem zunächst ein in sich geschlossener Überblick über die Grundlagen des Path Tracing präsentiert wird. Anschließend wird CPU basiertes interaktives Path Tracing näher betrachtet und dessen Leistung basierend auf einem in Go von Grund auf neu geschriebenen Path Tracer evaluiert. Ein besonderer Fokus liegt auf Datenstrukturen, die der Beschleunigung des Ray Tracing Prozesses dienen. Im Zuge dessen wird der Algorithmus "Progressive Hierarchical Refinement", welcher zur Konstruktion von Bounding Volume Hierarchien dient und dem neusten Stand der Technik entspricht, vorgestellt und in die interaktive Applikation integriert. Zuletzt wird ein neuer Ansatz zur Evaluierung des Konstruktion-Render-Ausgleichs des Algorithmus vorgeschlagen und das Optimierungsproblem durch Rastersuche und Bayessche Optimierung gelöst.

Notation

Here the notation used throughout this thesis is described. Vectors are denoted by bold lowercase letters, e.g. $\mathbf{v}$ and matricies by bold upper case letters, e.g. $\mathbf{M}$. Scalars are lowercase, italicized letters, e.g. c. Points are uppercase, e.g. P. The components of a vector are accessed as

$$\mathbf{v} = \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} = \begin{bmatrix} v_0 \\ v_1 \\ v_2 \end{bmatrix} = \begin{bmatrix} v_x & v_y & v_z \end{bmatrix}^\top$$

where the latter shows the vector transposed, i.e. a column becomes a row. The dot product between to vectors is written as $a \cdot b$.

A matrix $\mathbf{M}$ can be written in the following ways:

$$\mathbf{M} = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{bmatrix} = \begin{bmatrix} \mathbf{m}_0 & \mathbf{m}_1 & \mathbf{m}_2 \end{bmatrix}$$

where $\mathbf{m}_i, i \in \{0,1,2\}$, are the column vectors of the matrix.

Contents

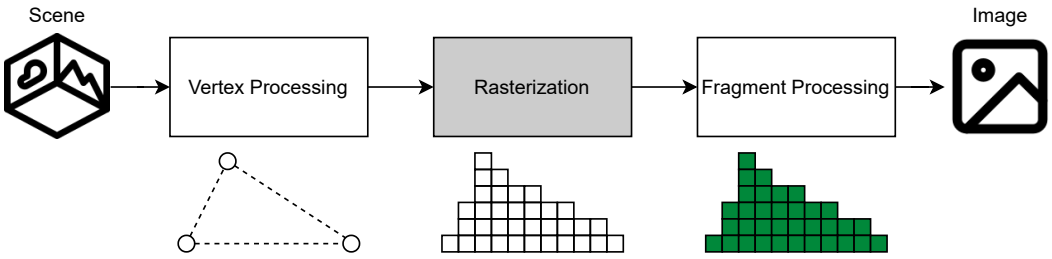


Figure 1.1: Simplified Rasterization Pipeline.

1 Introduction

Photo-realistic image generation has a long tradition in computer graphics and the topic is still far from being exhausted. Among other rendering techniques, physically based rendering plays a key role in modern graphics applications. Ray tracing can be used as such a rendering technique by simulating how light travels through a scene, creating physically plausible images in the process. Consequently, this method has long been a staple in offline rendering, where relatively long rendering times can be tolerated. In real-time applications, however, a rapid rate of images is required to achieve fluent motion. Video games require a frame rate of at least 30 frames per second (FPS), but modern titles tend to target faster rates of 60 FPS and above. Ray tracing is a very computation heavy task and even interactive rates, starting at 6 FPS, can be a daunting task.

As a result, rasterization has dominated real-time rendering and, until fairly recently, has been seen as the sole viable approach. Rasterization takes a divide and conquer approach by dividing a scene into a mesh of basic geometric shapes (e.g. triangles). These are sent to a rendering pipeline (figure 1.1), where each is processed individually. First, vertices are transformed and projected into screen space. The results are then fed to a rasterizer, which performs spatial sampling to determine which pixels are occupied by given primitive. These pixels, often referred to as fragments, are then assigned a color derived from a number of operations. A Z-buffer algorithm is used to determine which fragments are occluded and any visible fragment is written to a frame buffer. This process is highly parallel and can be executed very efficiently on conventional GPUs. One drawback of that parallelism is, that only a single primitive is considered at a time. Effects like shadows, reflections and refractions, however, can only be achieved by taking multiple primitives into account. In practice, complex methods are used to approximate such effects.

By emulating the basic concepts of vision, ray tracing achieves the same effects much simpler and more natural. In reality, objects are perceived after a light ray hits said object and then bounces into the eye. This concept is recreated in ray tracing, only in the reversed way. For each pixel, a ray is cast from the eye point and bounces around the scene until a termination criteria is met. Several billion rays are necessary, to converge towards a sufficient solution of that approach, hence the high computational effort.

In recent years, however, these limitations have been overcome and real-time ray tracing has been enabled and opened up to a consumer market, especially through the use of special-purpose hardware. Such hardware [Nvidia, 2018] is built in a way that accelerates basic ray tracing operations while also facilitating other methods essential to the process, most importantly more advanced denoising techniques. While this technology is undoubtedly a leap in the right direction, performance can still lack at times and additional software optimizations are necessary to support increasingly complex scenes. Additionally, such optimizations could decrease the power consumption of graphic cards by decreasing the number of computations necessary and could be utilized by conventional ray tracing frameworks as well. This thesis continues on previous findings and aims to identify more areas of improvement and then optimize them.

In particular, this thesis provides a self-contained overview over the basics of real-time ray tracing (section 2.1) as well as a broader outline over some related topics, which are important in that context. Acceleration data structures are explored in more detail (section 2.2) and a construction method based on the approach of Progressive Hierarchical Refinement (PHR) [Hendrich et al., 2017] is presented as a state-of-the art algorithm (section 5). The viability of progressive hierarchical refinement in interactive ray tracing is then evaluated based on a CPU path tracer written from scratch in the programming language Go. Key implementation decisions of said path tracer are explained in more detail (section 4), which might be particularly helpful as a starting point for further research. A novel approach for optimizing the build-trace trade-off of PHR is proposed(section 6), evaluated (section 7). Finally, the thesis is concluded by discussing all findings (section 8) and identifying future work (section 9)

2 Preliminaries

This section introduces the basic concepts needed to follow along with the rest of this thesis, while also putting some focus on notable historical work.

2.1 Ray Tracing

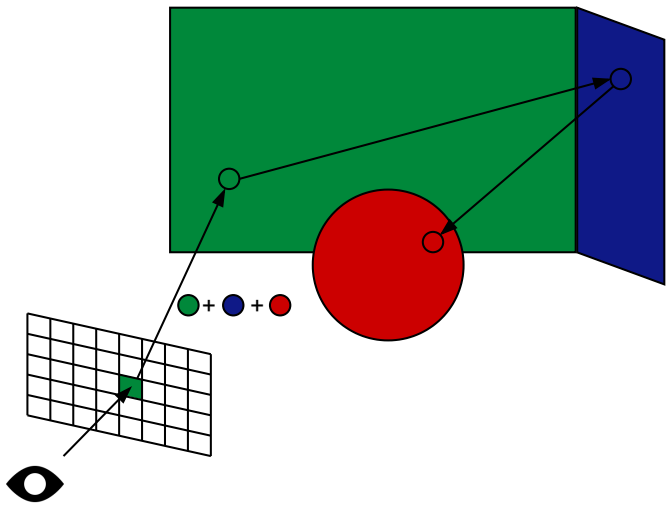
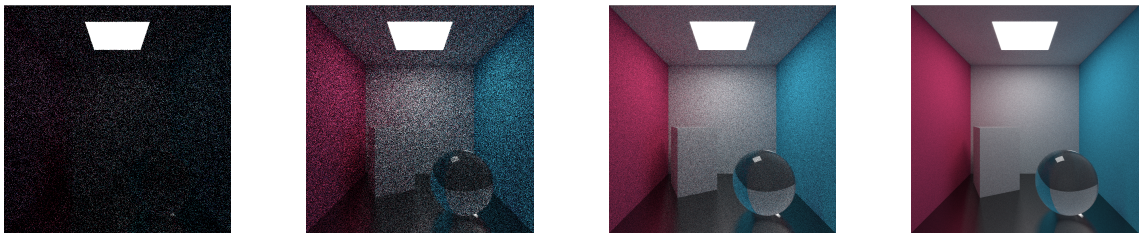


Figure 2.1: Illustration of the path tracing process. A ray is cast into the scene for each pixel in the image plane. When an object is hit, its color is contributed to the pixel and a new ray is spawned.

The basic technique of casting rays through each pixel in a viewing plane out into a three dimensional scene, was first proposed by Appel [Appel, 1968]. Whitted [Whitted, 1980] expanded on that approach by introducing a method that also captures reflections, shadows and refractions. This works by recursively spawning new rays when objects are hit, as described in figure 2.1. Each consecutive hit can contribute to a part of the pixels color. Whether and in what fashion secondary rays are created depends on the material of the encountered object.



(a) 1 spp **(b)** 10 spp **(c)** 100 spp **(d)** 1000 spp

Figure 2.2: Noise at different sample-per-pixel (spp) values. Note that the images were rendered with implicit light sources. Noise is reduced using explicit light rays.

The highest amount of realism can be achieved by a related concept called path tracing [Cook et al., 1984, Kajiya, 1986], which enables the accurate rendering of global illumination and distribution effects. Instead of casting a single ray per pixel, multiple samples are combined to more accurately simulate light transport through a scene. Each ray is cast with a slight offset within the pixel, which inherently solves the problem of aliasing. Secondary rays are scattered in random directions, depending on the surface they hit. Provided a sufficiently large sample size, this allows

for a good approximation of the rendering equation [Kajiya, 1986] and leads to very realistic results. Consequently, path tracing is very common in offline rendering, e.g., in film [Keller et al., 2015] and visual effects, where relatively long render times can be tolerated. Given the limited time to process a frame in real-time, however, only allows to obtain a very limited number of samples. As seen in figure 5.3, these frames tend to suffer from high-frequency noise. Only through recent denoising techniques, further discussed in section 3.1, has this problem been overcome to make path tracing viable in real-time. A more detailed look at path tracing is presented in section 4.

2.2 Acceleration Structures

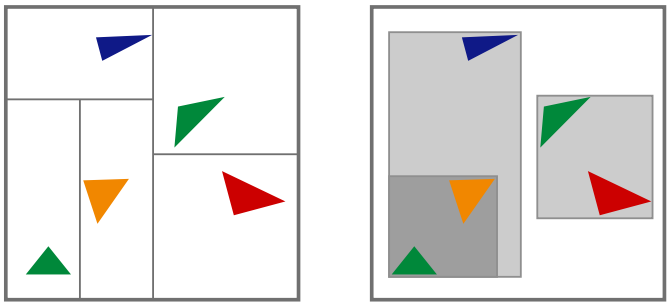


Figure 2.3: Example of space subdivision (left) and object subdivision (right).

The essential and computationally most expensive step in path tracing is identifying the nearest intersection point for each ray. A naive approach would be to test the ray against all scene primitives, which in practice might be several million operations and thus too costly. Acceleration data structures are used to speed up that search process by arranging primitives in a spatial tree structure. Instead of checking all primitives, now the tree can be traversed to find the closest intersection. While doing so, all subtrees that are not hit by the ray can be neglected, reducing the number of total intersection tests drastically. Such data structures can be divided into two categories, space subdivision and object subdivision (figure 2.3).

Space subdivision works by splitting the scene space recursively into smaller subregions, so each leaf in the resulting tree corresponds to a disjoint area in the full scene. This makes traversal of those structures very efficient, as the subregions can be tested in the order any ray passes through them. If a intersection is found, the traversal algorithm can be terminated without needing to check any further nodes. One of the limitations of such approaches is, that an object might lie in multiple subregions at once, i.e. multiple leaves might contain a pointer to the same object. This is problematic, because that way a point lying outside the associated region might be falsely identified as an intersection. The traversal algorithm as described would assume that point as the closest hit, even though there is no guarantee for that. Furthermore, this way the same object might be tested multiple times. Objects can be clipped to solve this problem, however, this introduces a certain computational cost. Alternatively, the traversal algorithm can be extended to also check whether or not a intersection point lies within a node's region.

Object subdivision leads to trees with similar structure, however, each node is also associated with a bounding volume. Bounding volumes are a geometric shapes that enclose all primitives stored in any of the node's children. Axis-aligned bounding boxes (AABBs) are the most popular volumes as they offer a good trade-off between intersection speed and tight fit around geometry. Bounding spheres offer the fastest intersection algorithm, but generally have larger volumes and oriented bounding boxes (OBBs) can provide the closest fit, but lack in intersection speed. When traversing such data structures, each child node needs to be checked as bounding volumes could overlap each other. A possible traversal algorithm is explained more detailed in section 4.4. In

the rest of this section, basic Bounding Volume Hierarchy (BVH) construction algorithms are presented.

A basic top-down algorithm starts at the root of the tree containing all scene primitives. The node is then split into two disjoint sets, which are set as children of the root. This process is repeated recursively for both children until some termination criteria is met and the given node is converted into a leaf. At each step, a bounding volume enclosing all primitives is assigned to the node. Given its exponential nature, splitting primitives into disjoint sets is a very complex problem. According to Popov et al. [Popov et al., 2009], primitives can be split with a complexity of $O(n^6)$, which is not feasible in practice. As a result, primitives are generally split using axis-aligned planes. First, one splitting axis needs to be selected. Primitives are then ordered along that axis, most of the time according to their bounding volume centroid. Finding the split can then be done in three basic ways. A spatial median split cuts the bounding volume in half, object median uses a split with the same number of primitives in both halves and the most common approach splits primitives utilizing a cost model.

The most common cost model is the Surface Area Heuristic (SAH) [Goldsmith and Salmon, 1987, MacDonald and Booth, 1990], which can be used to calculate the cost of a split as follows:

$$SAH(i) = S_L(i)p_L(i) + S_R(i)p_R(i)$$

where $S_L(i)$ and $S_R(i)$ are the surface areas of the bounding boxes of the left and right subsets and $p_L(i)$ and $p_R(i)$ are the number of primitives in the subsets, respectively. Computing a full sweep SAH considering all primitives can be very expensive, thus a very popular method is binning SAH as proposed by Havran et al. [Havran et al., 2006] and Wald et al. [Wald, 2007]. In binning SAH primitives are projected into b equally-spaced bins on which the SAH is then evaluated.

Other cost functions include the occlusion heuristic [Vinkler et al., 2012] based on the assumed visibility of primitives and the ray distribution heuristic [Bittner and Havran, 2009], which takes a sample the of the ray distribution into account. Both cost functions aim to improve the surface area heuristic, but might produce unstable results when used on their own, so it makes sense to combine those probabilities with the ones given by plain SAH.

Bottom-up construction [Walter et al., 2008] starts by considering all primitives in clusters enclosed by bounding volumes. The two closest clusters are then merged to form a new node in the BVH and the process is repeated until only one cluster is left, forming the trees root. This approach is capable of producing trees with a better global cost compared to top-down approaches, which often only consider a local solution to cost functions. However, bottom-up construction in general is more expensive and the top levels might be poorly optimized due to the focus on lower levels.

Incremental construction [Goldsmith and Salmon, 1987] starts with an empty BVH and inserts primitives by traversing the tree and finding an appropriate leaf node. Once the number of primitives in a leaf gets too big, it is split into two new children. In general though, BVHs constructed this way are of lower quality making the approach less interesting. Nonetheless, it might still be useful if only parts of the input are available at the beginning, for example when streaming data.

3 Related Work

3.1 Denoising

Efficient denoising techniques are essential to real-time path tracing, as only a limited number of samples-per-pixel is available for any given frame. While offline methods achieve the best quality, only interactive and real-time approaches are relevant in the context of this thesis. Yan et al. [Yan et al., 2014] proposed a sheared filtering approach that achieves interactive frame rates. Schied et al. [Schied et al., 2017] proposed an approach that combines path tracing output and previous frame data with a noise free G-buffer generated using a rasterization pass to feed a wavelet filter. Mara et al. [Mara et al., 2017] independently proposed a similar ray-tracing/rasterization hybrid method. They used a bilateral filter variant to achieve similar results. Chaitanya et al. [Chaitanya et al., 2017] showed that neural networks can be used for denoising at interactive frame rates by using a convolutional neural network (CNN) to map noisy input images to noise-free output. In this approach, temporal noise was addressed by using recurrent connections in each layer of the CNN. Regression-based noise filtering produces higher quality output at the cost of more expensive computation. Koskela et al. [Koskela et al., 2019] were the first to implement a regression-based reconstruction pipeline that runs in real time.

State-of-the-art denoising approaches are able to produce a denoised, temporally stable sequences of images using only one sample-per-pixel. However, denoising was not the focus of this work and will not be mentioned in the remainder of this thesis. Consequently, the implemented path tracer produces noisy one sample-per-pixel output leaving the choice of denoising technique open, even though applying any denoising technique would be an interesting topic for some future work.

3.2 Space Subdivision

Fuchs et al. [Fuchs et al., 1980] proposed one of the first binary space partitioning trees, also referred to as kd-trees, which is built by recursively splitting the space along a given axis. This cut position is selected in a way that both sides contain a relatively equal number of objects. Glassner [Glassner, 1984] described an approach for generating octrees that, for each recursive step, splits the given subspace at the spatial median along all three axis, resulting in eight new subregions. While trees with higher branching factors generally have a lower depth, binary trees allow for simpler traversal, as only a two-way decision is needed at each step. Kaplan [Kaplan, 1985] expanded on Glassners idea by introducing a very similar implementation utilizing binary trees instead of octrees. Fujimoto et al. [Fujimoto et al., 1986], while also using octrees, achieved a significant speed improvement by using incremental integer arithmetic to optimize the traversal algorithm. Havran and Bittner [Havran and Bittner, 2002] introduced additional traversal improvements utilizing a new termination criteria and a novel approach for clipping primitives. More modern kd-tree construction algorithms [Roccia et al., 2012, cho, 2010, Wu et al., 2011] make use of the Surface Area Heuristic (SAH) [Goldsmith and Salmon, 1987, MacDonald and Booth, 1990] further improving their performance. Li et al. [Li et al., 2017] proposed a construction algorithm based on Morton codes [Morton, 1966] to enable a maximum level of parallelism. Hunt et al. [Hunt et al., 2007] proposed kd-tree construction from a given hierarchy. A similar approach for BVH construction is presented in section 5

3.3 Object Subdivision

Bounding volume hierarchies were first mentioned by James Clark [Clark, 1976a] and also referenced by Turner Whitted [Whitted, 1980]. Meister et al. [Meister et al., 2021] published a report that reviews state-of-the-art BVH methods and discusses best practices.

In the context of interactive and real-time rendering, construction speed is very crucial, especially when dealing with dynamic scenes. However, parallelizing the construction process is not straightforward. One parallel solution is a BVH based on Morton codes, which reduces the construction process to sorting primitives along the Morton curve [Morton, 1966]. Sorting Morton codes with fixed length has a complexity of $O(n)$ and can be parallelized fairly efficiently. Such an approach was first proposed by Lauterbach et al. [Lauterbach et al., 2009] as a top down GPU-based algorithm called Linear Bounding Volume Hierarchy (LBVH). A similar CPU based approach is elaborated further in section 5.1. Pantaleoni and Luebke [Pantaleoni and Luebke, 2010] proposed hierarchical LBVH, which combines LBVH with sweeping SAH in the upper levels of the tree and Garanzha et al. [Garanzha et al., 2011] applied binning SAH using Morton code prefixes as bin indices. Karras [Karras, 2012] improved LBVH by using a special node layout and bottom-up reduction to construct the whole tree in parallel. Apetrei [Apetrei, 2014] further improved the approach by constructing the tree and computing bounding boxes in one go, which was previously done in two seperate steps. Chitalu et al. [Chitalu et al., 2020] combined LBVH with an ostensibly-implicit layout, which is the fastest construction algorithm to date [Meister et al., 2021]. Another improvement was presented by Vinkler et al. [Vinkler et al., 2017] where Morton codes also encode the size of scene primitives. Hou et al. [Hou et al., 2011] proposed another GPU-base parallel algorithm for constructing kd-trees and BVHs by using partial breadth-first search and dumping results to CPU memory in between iterations to control GPU memory.

While space subdivision approaches have previously been regarded as the best acceleration data structure [Havran, 2000], object subdivision has since caught up and overtaken [Vinkler et al., 2015], making it the most popular approach for path tracing. Some of the advantages of bounding volume hierarchies include a predictable memory footprint, robust and efficient query and scalable construction. In addition, bounding volume hierarchies are very beneficial in dynamic scenes [Wald et al., 2007], as they can be re-fit efficiently on scene changes. Because of these advantages, only object subdivision approaches will be considered in the following sections of this thesis.

4 Path Tracer

This section presents a closer look at the general path tracing process and highlights key implementation details of the evaluated CPU path tracer.

4.1 Pipeline

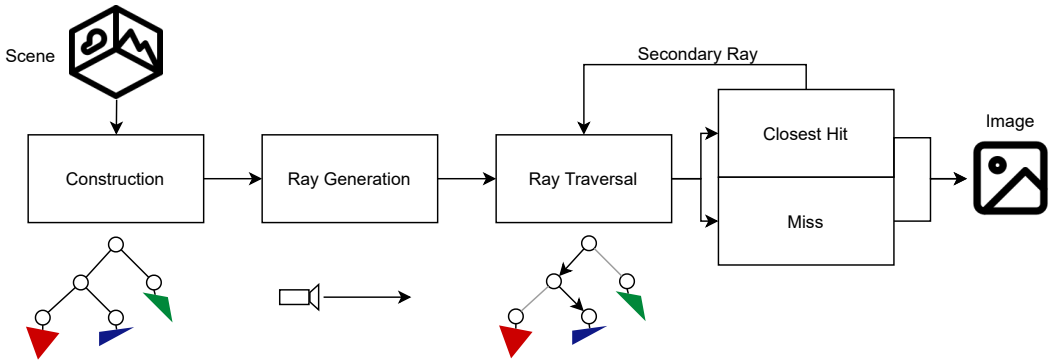


Figure 4.1: Visualization of path tracing process as a pipeline.

The graphics rendering process if often organized into a graphics pipeline [Sugerman et al., 2009], so I generalized the path tracing process in a similar fashion (figure 4.1). In the first stage of that pipeline the acceleration structure is created, which will be discussed further in section 5. Given a static scene, this acceleration structure can be reused for multiple frames. Dynamic scenes require additional care though, as the acceleration structure either needs to be refit or rebuild on scene changes. The implemented uses a hybrid approach further elaborated in section 5.3. The next stages are responsible for ray traversal and intersection testing. First, primary rays are generated from the camera's origin towards each pixel. Then, each ray has to traverse the acceleration structure to find the closest hit. Depending on whether or not this intersection has been found, either the closest hit shader, or the miss shader is called. Each shader can contribute to the color of the pixel, but only the closest hit shader spawns secondary rays. Note that my implementation uses implicit light sources instead of explicitly casting shadow rays. Secondary rays are fed back to the ray traversal stage and after reaching the maximum depth, the closest hit shader exits without creating any additional rays. The steps in these stages are embarrassingly parallel, however, because every ray is independent of the others, SIMD instructions cannot be utilized. Consequently, each pixel is processed concurrently using multithreading.

4.2 Intersection Tests

A ray can be expressed using the parametric form

$$R(t) = O + t\mathbf{d} \tag{1}$$

where point O defines the origin of the ray and vector $\mathbf{d}$ defines the direction along which the ray travels in a straight line. The path tracer supports two types of primitives, spheres and triangles.

4.2.1 Sphere

Spheres are commonly used in ray tracing because of their simple and efficient intersection algorithm [Haines and Akenine-Möller, 2019]. Given a sphere with center C and radius r, all points P at the surface of the sphere can be described by the equation:

$$(P - G) \cdot (P - G) = r^2 \tag{2}$$

By substituting point P in equation 2 by the equation of a ray (equation 1) the intersection points between sphere and ray can be calculated. Simplifying the resulting equation leads to

$$(\mathbf{d}\cdot\mathbf{d})t^2+2(\mathbf{f}\cdot\mathbf{d})t+\mathbf{f}\cdot\mathbf{f}-r^2=at^2+b^t+c=0$$

which is a quadratic function. Consequently, it can be solved using

$$t_{0,1}=\frac{-b\pm\sqrt{b^2-4ac}}{2a}$$

with discriminant $\Delta=b^2-4ac$.

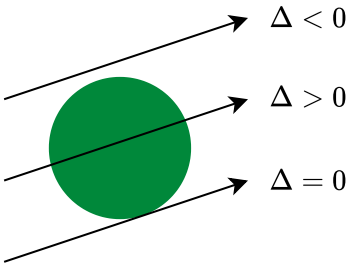


Figure 4.2: Ray-sphere intersection test showing all three cases for discriminant Δ.

As seen in figure 4.2, if $\Delta<0$, the ray misses the sphere. If $\Delta=0$ the ray touches the sphere in one point and otherwise there are two values for t that correspond to different intersection points. Inserting these t-values into equation 1 allows the calculation of intersection points $P_{0,1}=R(t_{1,2})=O+t_{0,1}\mathbf{d}$. The normal n at a given intersection point is simply the vector from the spheres center C to the intersection point P, i.e the normalized normal is calculated using $n=||P-C||$.

4.2.2 Triangle

Triangles have a long tradition in computer graphics, mainly because most geometry can be represented, or at least approximated, using them. Additionally, with 3 vertecies they can never be non-planar. While many ray-triangle intersection algorithms exist, the Möller-Trumbore algorithm [Möller and Trumbore, 1997] is still considered to be relatively fast and is often used as a comparison for other algorithms. Consequently, it is also used in this thesis.

Using barycentric coordinates, a point P on a triangle can be parameterized and expressed through two scalar values:

$$P(u,v)=(1-u-v)V_0+uV_1+vV_2$$

where V_0, V_1 and V_2 are the vertices of given triangle and (u,v) are the barycentric coordinates with $u\geq 0$, $v\geq 0$ and $u+v\leq 1$. Barycentric coordinates do not change when the triangle is transformed, which the Möller-Trumbore algorithm exploits. Geometrically speaking, the algorithm translates the triangle to the origin and transforms it into a unit triangle in y and z, with the ray direction aligned with x. This can be expressed as a system of linear equations:

$$\begin{bmatrix} -\mathbf{d} & (V_1-V_0) & (V_2-V_0)\end{bmatrix}\begin{bmatrix} t\\ u\\ v\end{bmatrix}=O-V_0$$

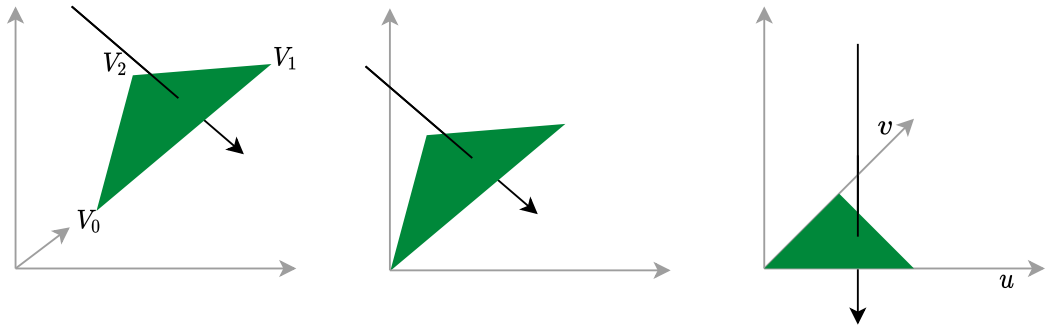


Figure 4.3: Geometric representation of the transformations utilized in the Möller-Trumbore algorithm.

Using Cramer's rule [Brunetti and Caccioppoli, 2014], this system can be solved to obtain the distance t from ray origin to intersection point and the corresponding barycentric coordinates (u, v). Distance t is again plugged into the ray equation to find the world coordinates of intersection point P and the barycentric coordinates are used to interpolate the normal at P, given the vertex normals at V_0, V_1 and V_2.

4.2.3 Axis-aligned bounding box

Finally, bounding volume hierarchy traversal depends on a ray-bounding box intersection test. Axis-aligned bounding boxes are represented by the bounds B^{min} and B^{max}, which define two planes for each dimension. The intersection distances t between a given ray $R(t) = O + t\mathbf{d}$ and a dimension k can be computed fairly simple:

$$t_k^{min} = \frac{B_k^{min} - O_k}{\mathbf{d}_k}$$

$$t_k^{max} = \frac{B_k^{max} - O_k}{\mathbf{d}_k}$$

Whether or not a ray intersects a box, can be determined by a few simple value comparisons, as seen in figure 4.4. The algorithm is optimized further by pre-computing sign and inverse direction for all rays and adding an early exit once it is clear the ray missed.

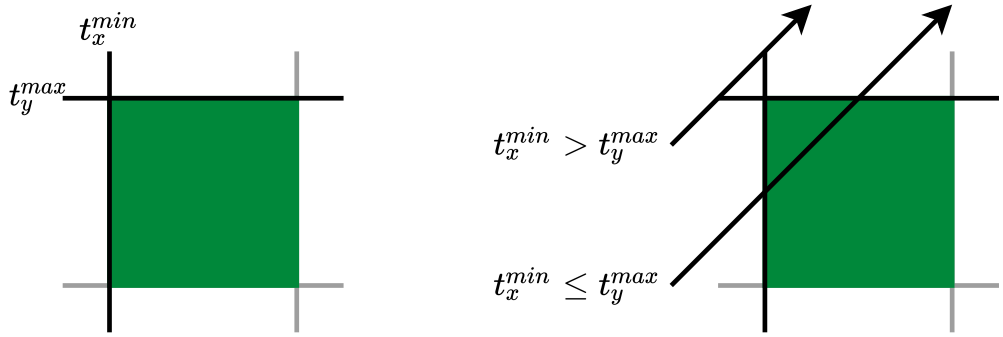


Figure 4.4: 2D example of ray-AABB intersection. The top ray misses, while the bottom one hits the box.

4.3 Materials

Each primitive has an associated material. A material has an albedo specifying how reflective it is, i.e. how much color is contributed to the traced path, and an amount of emitted light. The scatter function differs between materials and is used to generate secondary rays. Note that it is also possible for materials to not scatter at all, which is used for light sources.

Diffuse materials scatter rays in random directions. To achieve true Lambertian reflectance [Weik, 2001], random points are picked on the surface of a unit sphere and added to the normal at intersection point. This results in a distribution of $cos(\Phi)$, where Φ is the angle from the normal.

A smooth reflective material [de Greve, 2004] does not scatter rays in a random direction, so the resulting rays point purely in the reflection direction $\mathbf{d}_r$ (figure 4.5):

$$\mathbf{d}_r = \mathbf{d} - 2\mathbf{n}(\mathbf{d} \cdot \mathbf{n})$$

where $\mathbf{n}$ is the normalized normal at intersection point and $\mathbf{d}$ is the direction of the incoming ray. For diffuse reflection a random vector is added to the reflected ray, similar to the above mentioned diffuse scattering.

Refractive materials [de Greve, 2004] utilize Snell's law to represent dielectrics like glass objects. Snell's law can be written as

$$sin\theta' = \frac{\eta}{\eta'} sin\theta$$

where θ and θ' are angles from the normal and η and η' are refractive indices. The refracted direction $\mathbf{d}_t$, as shown in figure 4.5, can be split up into a parallel and a perpendicular part:

$$\mathbf{d}_t = \mathbf{d}_\perp + \mathbf{d}_\parallel$$

Solving for $\mathbf{d}'_\parallel$ and $\mathbf{d}'_\perp$ leads to:

$$\mathbf{d}'_\perp = \frac{\eta}{\eta'}(\mathbf{d} + (-\mathbf{d} \cdot \mathbf{n})\mathbf{n})$$

$$\mathbf{d}'_\parallel = -\sqrt{1 - |\mathbf{d}'_\perp|^2}\mathbf{n}$$

Adding both parts together, leads to the refracted direction $\mathbf{d}_t$

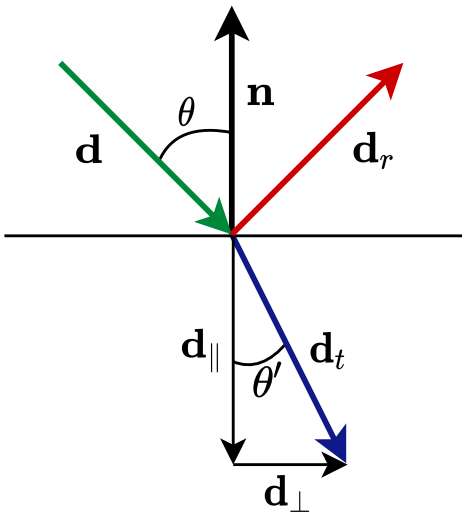


Figure 4.5: Figure showing how a ray's direction (green) is reflected (red) and refracted (blue).

4.4 Traversal

To find the nearest intersection, bounding volume hierarchies are traversed in a top-down manner. Usually, this is done using a stack [Meister et al., 2021] to store nodes that might contain an intersection. First, the root is pushed to the stack. While the stack is not empty, nodes are popped and checked for an intersection with their bounding box. In case the bounding volume is hit, either the nodes children are pushed onto the stack in the case of an interior node, or all primitives are tested when dealing with a leaf node. If no intersection with the bounding box is found or the distance to the intersection is bigger than previously found intersections, then the node can be discarded. Once the stack is empty, the closest intersection found is returned.

However, this method proved to be less efficient than using a recursive procedure as described in algorithm 1. Given that the implementation is written for the CPU, this implicitly uses the CPU stack, which results in an equivalent execution as described above, albeit without explicitly managing a stack data structure.

Algorithm 1: Pseudocode of recursive BVH traversal

```
// Start traversal at root
Traverse(root)

Function Traverse(node):
    if node.aabb not intersected then
        return
    if node is leaf then
        foreach primitive do
            test for ray-primitive intersection
    else
        foreach child do
            Traverse(child)
```

5 Progressive Hierarchical Refinement

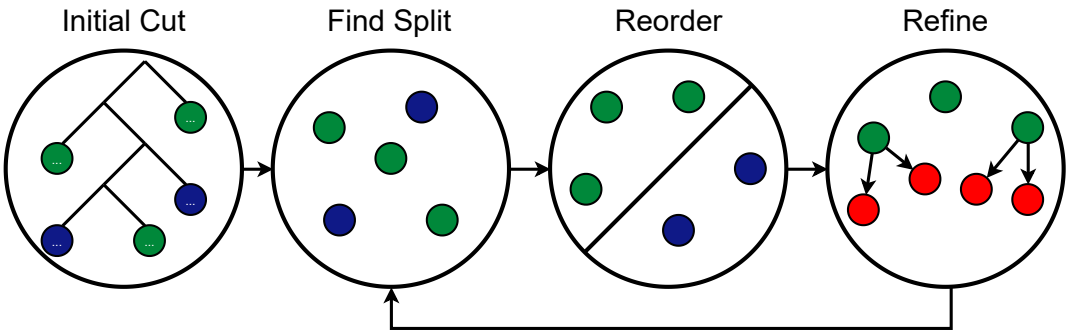


Figure 5.1: Illustration of the progressive hierarchical refinement algorithm.

This section further explains the algorithm used to construct bounding volume hierarchies. Bounding volume hierarchies are constructed using progressive hierarchical refinement (PHR) as proposed by Hendrich et al. [Hendrich et al., 2017]. As previously established, applying full sweep SAH to all scene primitives is magnitudes too slow. PHR tackles this problem by first constructing an auxiliary BVH, which then serves as a hierarchy to find much smaller sets of nodes on which full sweep SAH can be applied fairly inexpensively. The two resulting cuts are then refined, meaning that some nodes within those cuts are replaced by their children, if their bounding box surface area is above a certain threshold. Afterwards, the algorithm is applied recursively to the refined cuts until the full BVH is constructed. An illustration of that process can be seen in figure 5.1.

5.1 Auxiliary Bounding Volume Hierarchy

As the auxiliary BVH is only needed as a description of the scene's hierarchy, construction speed is the main priority. Multiple fast builders have been tested in the original paper [Hendrich et al., 2017], but linear bounding volume hierarchies (LBVH) turned out to be the best choice. LBVH was first proposed by Lauterbach et al. [Lauterbach et al., 2009] as a top-down algorithm that assigns Morton codes to all primitives and then builds the tree as a binary radix tree. The algorithm itself has since been improved multiple times [Karras, 2012, Apetrei, 2014, Chitalu et al., 2020] making it one of the fastest approaches to date [Meister et al., 2021]. However, these approaches exploit the massive parallelism GPUs can provide by building the BVH in a bottom up fashion, which makes less sense on the limited amount of cores CPUs provide. Consequently, the approach used here is closer to the top-down algorithm proposed by Lauterbach et al. [Lauterbach et al., 2009] with a few adjustments.

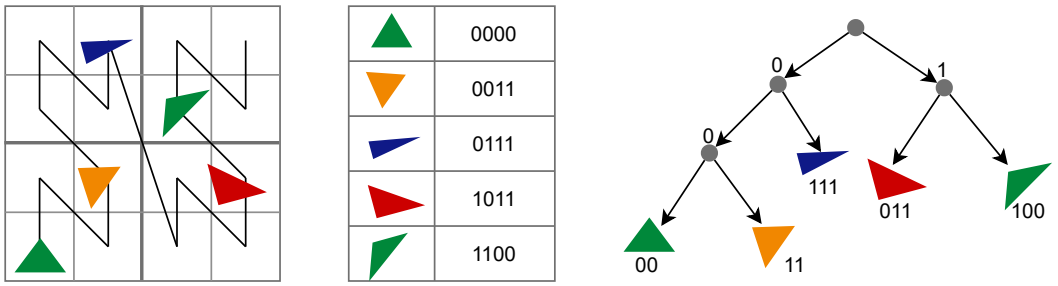


Figure 5.2: Left shows 2D Morton curve using 2 bits per dimension. The table in the center shows the corresponding Morton codes and the right shows the resulting tree structure, equivalent to a binary radix tree.

Construction of the auxiliary BVH starts off by sorting all primitives along a Morton curve [Morton, 1966]. This space filling curve subdivides the scene space into a uniform grid, resulting in Morton codes of fixed length (figure 5.2). Computation of Morton codes is done fairly efficiently by interleaving successive bits of the primitives' quantized bounding box centroids.

Morton codes are assigned in parallel by processing $[n/t]$ primitives per thread, with n being the number of scene primitives and t being the number of threads. Afterwards, the primitives are sorted according to their Morton code using a parallel bucket sort implementation. In each thread, $k = 2^{12}$ empty buckets are created and filled with $[n/t]$ primitives. By using individual buckets for each thread, no further synchronization necessary for the bucketing. However, an atomic counter is used to keep track of the total number of primitives in each bucket across all threads, which is then used to find the intervals in the original array each bucket occupies. After bucketing is finished, all non-empty buckets with the same index are merged and directly written to the mentioned interval in the input array, before being sorted in place using Go's built-in sort function. This step is also done in parallel by utilizing the worker pattern to send buckets with the same index to each thread until all buckets have been processed.

The corresponding BVH can be constructed by recursively splitting the set of primitives at the highest bit within the current interval. This is done using a Go channel and entries representing a node in the finished tree. Each thread fetches such a node and finds the split in the corresponding array by applying linear search. If the node does not become a leaf, the resulting cuts are sent to the channel to be processed by idle threads.

Finally, the bounding boxes of the tree are updated in parallel by starting at the tree's leaves and traversing towards the root. Whenever a thread visits a node, the bounding box is updated using the child or primitive bounding boxes. Then the parents atomic counter is incremented and if all children are set, the thread also processes the parent. Otherwise, the thread fetches an unprocessed leaf from a queue. The same procedure is executed when refitting the LBVH on scene changes.

5.2 Algorithm

The main progressive hierarchical refinement algorithm starts by identifying a set of nodes that separate root and leaves of the auxiliary LBVH. Nodes are selected in parallel using a priority queue. A thread pops an entry from that queue and compares its bounding volume surface area to a given threshold. If the surface area is below that threshold or the cut has reached its maximum size, the node is added to the initial cut. Otherwise, its children are inserted into the priority queue to be processed by another thread. The resulting cut is several magnitudes below the full primitive count and can be processed fairly inexpensively using full sweep SAH.

Cuts are split by evaluating an adapted version of the surface area heuristic for all three axis and choosing the split with the lowest cost. The cut becomes a leaf, if the cost of not splitting at all is the lowest. Each axis is evaluated by sorting nodes along given axis according to their bounding box centroid. The SAH cost for a split at the index i in the sorted set is given as

$$C(i) = S_L(i)n_L(i) + S_R(i)n_R(i)$$

where $S_L(i)$ and $S_R(i)$ are the bounding boxes surface areas of the left and right subsets and $n_L(i)$ and $n_R(i)$ are the number of nodes in the corresponding subtrees. Note that this expression is very similar to the SAH cost presented in section 2.2. However, instead of using the number of primitives in the left and right cuts, the number of nodes is used. According to Hendrich et al. [Hendrich et al., 2017], this improves the performance by a few percent, as the number of nodes better reflects the complexity of the given subtree.

The SAH evaluation algorithm first computes all right costs $S_R(i)n_R(i)$ by incrementally extending the split bounding box and storing the partial costs, which is more efficient than computing the full bounding box at each step. The full cost is then calculated in the same fashion, extending the bounding box from the left to solve $S_L(i)n_L(i)$.

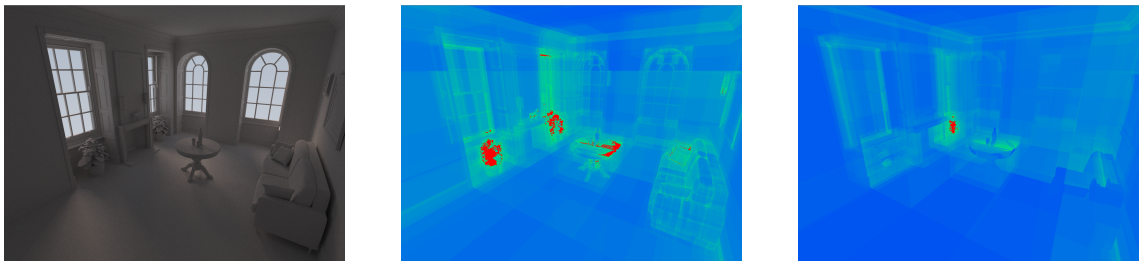
Splitting the cut reduces the number of nodes in each new cut and doing so multiple times would lead to a cut size of one. To keep cuts at a larger size for longer and to better utilize the hierarchical information the auxiliary BVH can provide, cuts are refined after splitting. Keeping cuts at a constant size, as proposed by Hunt et al. [Hunt et al., 2007], would become rather expensive towards the bottom of the tree due to the growing number of cuts. As a solution, Hendrich et al. [Hendrich et al., 2017] proposed an adaptive refinement approach based on the current depth in the tree. This approach makes cuts shrink towards the bottom of the tree, which balances the computational cost between different levels of the tree. The BVH quality is not worsened significantly by doing so, as the impact of SAH gets lower further down the tree.

Refinement works by comparing node bounding box surface areas to an adaptive threshold. Nodes with a surface area below this threshold are kept within the cut. Otherwise, the node is replaced by its children. This adaptive threshold is given as

$$t_d = S/2^{\alpha d+\delta}$$

with S being the surface are of the scene bounding box and d the current depth in the tree. The parameter α describes how quickly cuts shrink towards the bottom and δ determines the size of the initial cut for $d = 0$. The setting of these parameters determines the build-trace trade-off of the algorithm and is elaborated further in section Hyperparameters.

Construction of the BVH uses a similar setup as previously mentioned for the LBVH generation. A thread pool of t threads pops entries from a channel, consisting of a cut and parent node index. The cut is split using the described method, the resulting cuts are refined and then fed back into the channel if the node did not become a leaf. Additionally, a higher branching factor can be specified to build a multi-BVH [Wald et al., 2008]. In that case, a thread keeps splitting the biggest resulting cut until enough children have been generated or no more splits exist. The effect of multi-BVHs on the rendering performance is evaluated in section 7.1.



(a) Render **(b)** LBVH **(c)** PHR-HQ

Figure 5.3: Visualization of the number of traversal steps for primary rays using LBVH and PHR (red color corresponds to 100 traversal steps per ray).

5.3 Integration into Interactive Path Tracing

An approach for integrating progressive hierarchical refinement into interactive applications was mentioned in the original paper [Hendrich et al., 2017], but its validation remained an open topic. The idea was to only build the auxiliary BVH once in the beginning and then refit it very efficiently between frames. Refitting is done by updating bounding boxes of all BVH nodes. The hierarchy of the structure stays unchanged during this procedure. As described in the end of section 5.1, this is done using a parallel recursive procedure that traverses the tree in a bottom-up fashion and merges child bounding boxes using an atomic counter for synchronization. Doing so is fairly efficient in comparison to a full BVH rebuild, however, refitting generally leads to some extend of BVH quality loss, especially for significant scene changes. Applying PHR to the refitted tree counteracts this loss by building a new BVH.

6 Hyperparameters

This section proposes an equation to evaluate the build-trace trade-off of the PHR algorithm. Two possible solutions for the optimization problem are them presented in the form of grid search and Bayesian optimization. As established in the previous section, PHR depends on the parameters α and δ describing initial cut size and shrink rate. These parameters can be set to adjust the trade-off between build time and trace performance. However, the optimal parameters can differ between scenes and view points, especially when dealing with dynamic scenes and refitted auxiliary trees. Frame size also plays a role as higher resolutions profit more from well optimized bounding volume hierarchies while the hit taken by the extended build time is not as significant given the overall higher computational effort.

6.1 Evaluation

Ultimately, the overall frame time including BVH build time and render duration needs to be minimized to unlock the full potential of progressive hierarchical refinement. Stopping the actual execution times is neither efficient nor reliable enough, so an efficient metric that makes the trade-off quantifiable is needed.

A number of metrics have been proposed to estimate the quality of a given bounding volume hierarchy. A well known cost model based on the surface area heuristic, taken from [Meister et al., 2021], is given by the recurrence equation

$$c(N) = \begin{cases} c_T + \sum_{N_c} P(N_c|N)c(N_c) & \text{if } N \text{ is interior node,} \\ c_I|N| & \text{otherwise} \end{cases}$$

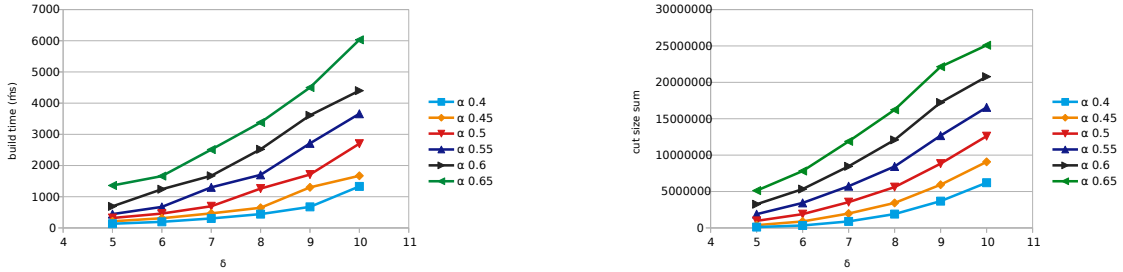
where $C(N)$ is the cost of the subtree with root N, N_c is a child of node N, $P(N_c|N)$ is the conditional probability of traversing node N_c when N is hit and $|N|$ is the number of primitives in the subtree with root N.

Constants c_T and c_I express the average cost of a traversal step and ray-primitive intersection, respectively. Utilizing the micro-benchmarking capabilities of Go revealed that a triangle intersection takes approximately 7 nanoseconds on average, while a traversal step takes around 12 nanoseconds. Consequently, these constants are set to $c_T = 2$ and $c_I = 1$, roughly representing the ratios between those values. The conditional probabilities of traversing a node are expressed using the surface area heuristic [Goldsmith and Salmon, 1987, MacDonald and Booth, 1990]

$$P(N_c|N)^{SAH} = \frac{SA(N_c)}{SA(N)}$$

where $SA(N)$ and $SA(N_c)$ are the bounding box surface areas of nodes N and N_c, respectively.

Evaluating the build complexity of PHR given a set of parameters and an arbitrary scene is not as trivial, as there is no clear correlation between the parameter values and associated build times. For scenes with medium to high complexity, the tree size of the resulting BVH could be used as a metric with an average correlation coefficient of around 0.98 between tree size and build time. This correlation does not hold true for simpler scenes though, as trees stop growing once no favorable splits can be found anymore. A better metric is the total sum of all cut sizes, as seen in figure 6.1, which can be obtained through a minimal adjustment of the progressive hierarchical refinement algorithm. The surface area heuristic evaluation is the most expensive part of PHR and thus the total number of evaluated nodes directly correlates with the execution duration. Using an atomic counter, the performance penalty is very marginally to non existing. Furthermore, an alternative implementation can be used for the optimization process. The correlation between this number and the BVH construction time averages to 0.99 across all tested scenes.



(a) Build Time **(b)** Build Cost

Figure 6.1: Both the build time and corresponding build cost for the San Miguel scene. This corresponds to a correlation of 0.994.

Let the build cost b be defined as

$$b(\alpha,\delta) = \sum_{cut \in PHR'(\alpha,\delta)} |cut|$$

where $b(\alpha,\delta)$ is cost of a PHR execution given the parameters α and δ, $PHR'(\alpha,\delta)$ is a execution of the progressive hierarchical refinement algorithm resulting in a set of all cuts processed and $|cut|$ is the length of given cut.

I propose an equation to evaluate the combined cost of a frame rendered using the PHR algorithm, which also factors in the build-trace trade-off. Given a pair of parameters α and δ, this cost is defines as

$$e(\alpha,\delta) = ||b(\alpha,\omega)|| + \omega^2 ||c(PHR(\alpha,\omega))||$$

with $||b(\alpha,\delta)||$ being the normalized build cost and $||c(PHR(\alpha,\omega))||$ the normalized SAH cost of the resulting BVH. The coefficient ω functions as a weight determining to what extend trace performance should be favored over build duration. For $\omega = 1$, both trace and build performance are weighed equally, $\omega > 1$ favors higher trace speed and $\omega < 1$ encourages faster build times. These trade-off depends on scene complexity and frame size, so ω is defined as:

$$\omega = (\frac{|prim|}{\omega_p})^2 * (\frac{w*h}{\omega_r})$$

where $|prim|$ is the total number of primitives in the scene and w and h are width and height of the rendered frame, respectively. ω_p and ω_r are constants depending on the performance of the used path tracer and the targeted frame rate, so they stay constant across all scenes. ω_p expresses a number of primitives and ω_r a frame size at which the trade-off should be equal, i.e. $\omega = 1$. Once those numbers are exceeded, BVH quality is favored increasingly.

Combining the equation with the previously mentioned metrics and given a search space over the parameters α and δ (e.g. $\alpha \in A = [0.4, 0.6], \delta \in D = [5, 10]$) leads to the expression:

$$e(\alpha,\delta) = \frac{b(\alpha,\delta)}{\max_{\alpha\in A,\delta\in D} b(\alpha,\delta)} + \omega^2 \frac{c(PHR(\alpha,\delta))}{\max_{\alpha\in A,\delta\in D} c(PHR(\alpha,\delta))}$$

where $PHR(\alpha,\delta)$ is a execution of the PHR algorithm using the parameters α and δ resulting in the root node of a bounding volume hierarchy.

6.2 Grid Search

The brute force approach grid search is the obvious first choice to solve this optimization problem. A number of possible values is chosen for each parameter (e.g. $\alpha \in \{0.45, 0.5, 0.55\}$ $\delta \in \{5, 6, 7, 8\}$) and all possible combinations are tested and evaluated. By postponing the evaluation of $e(\alpha, \delta)$ until all individual cost evaluations $b(\alpha, \delta)$ and $c(PRH(\alpha, \delta))$ are available makes their maxima directly accessible, which is another advantage of grid search.

Usually, grid search would be too costly for such a time critical task. In this case however, only two parameters need to be optimized and the search space is relatively small making grid search a potentially viable approach.

6.3 Bayesian Optimization

A more efficient approach compared to grid search is Bayesian optimization [Pelikan et al., 1999], which explores the search space by taking previous observations into account. This is done by placing a prior probability distribution over the objective function $e(\alpha, \delta)$. First, parameters α and δ are chosen by random in the given search space. Following their evaluation, the prior is updated to form a posterior distribution. Based on this posterior distribution, an acquisition function is constructed to select the next point worth exploring.

In particular, the Bayesian optimizer [Ou, 2019] utilizes a Gaussian process to define the prior/posterior distribution, and expected improvement is used as exploration strategy.

One issue compared to the aforementioned grid search approach is, that the maximum BVH and build costs are not directly available when evaluating $e(\alpha, \delta)$. This is solved by running PHR once with the lowest possible parameters in the search space to obtain the maximum BVH cost and once with the highest possible parameters to obtain the maximum build cost. These numbers correspond to the respective maxima very reliably and there evaluations can also be factored into the prior probability distribution.

7 Results

Both the BVH builder and path tracer were implemented in Go and only utilize the CPU. Code is optimized moderately without exploiting any SIMD instructions. A series of tests was conducted to compare the build times, ray tracing performance and resulting time per frame rendered between different hyperparameter configurations. LBVH was used as reference, PHR-Fast and PHR-HQ used the parameters proposed by Hendrich et al. [Hendrich et al., 2017], namely $\alpha = 0.5, \delta = 6$ and $\alpha = 0.55, \delta = 9$, respectively. PHR-Grid uses parameters based on the proposed grid search approach over the search space $\alpha \in \{0.4, 0.45, 0.5.0.55\}, \delta \in \{6, 7, 8, 9\}$ and PHR-BO uses parameters resulting from a Bayesian optimization over the equivalent interval $\alpha \in [0.4, 0.55], \delta \in [6, 9]$. The Bayesian optimization itself is executed utilizing the bo framework [Ou, 2019] and is based on a Gaussian process and expected improvement as exploration strategy.

To make results more reliable, all numbers were averaged over ten executions using the bench framework [Ou, 2020a]. Furthermore, the CPU, an AMD Ryzen 2600 eight core processor with 3.4 GHz, was locked to 90 percent capacity to prevent irregularities due to overheating or other high performance fluctuations. Note that the deviation percentages are left out for clarity in the tables presented in this thesis, but the full results are available in the attached files.

Render times are also averaged over three representative views for each scene. To keep times in an interactive window, only the relatively small resolutions 256x256 and 512x512 were tested with one sample per pixel.

7.1 Multi-Bounding Volume Hierarchies

As mentioned in section 5.2, the PHR algorithm allows the construction of bounding volume hierarchies with higher branching factors, which is especially useful for SIMD path tracers. Even though the evaluated path tracer does not utilize any SIMD instructions, I compared the build and trace performance of different multi-BVHs. As expected, the performance difference between branching factors was insignificant in most cases. 4-ary BVHs had slightly faster trace times, while 16-ary BVH construction was slightly slower. The following tests were all performed on 2-ary BVHs.

7.2 Frame Performance

The main part of the experiment was about comparing the resulting frame times of all configurations. Table 7.1 shows build time, SAH cost, average render time over the compared view points and the resulting frame times. Note that the PHR build times do not include construction of the auxiliary bounding volume hierarchy, as those would be reused over several frames.

First of all, the numbers clearly show the impact different PHR parameters can have on the build and trace time of the algorithm. PHR-Fast was indeed fairly fast, but the achieved trace speed is even below the baseline, LBVH, in some cases. PHR-HQ had the fastest trace speed in most cases, but the high build duration often leads to higher frame times. A noticeable difference can be seen between the different scene types. PHR performed comparably worse in single object scenes like Bunny, Dragon and Happy Buddha, often not exceeding the render performance of LBVH by much. Sibenik's and Sponza's render times on the other hand, were improved more significantly by applying PHR. Finally, PHR-Grid and PHR-BO were able to improve frame times significantly in a number of cases. However, Bayesian optimization delivered rather inconsistent results and no approach was able to find the optimal parameters in every case. This is probably a result of overfitting the evaluation function and their parameters to certain scenes. Nonetheless, both PHR-Grid and PHR-BO are able to improve frame times compared to PHR-Fast and PHR-HQ. Figure 7.1 shows the average relative performance compared to LBVH in percent. While

PHR-Fast only improves frame times by 33% on average, both optimization approaches surpass an average improvement of 50%.

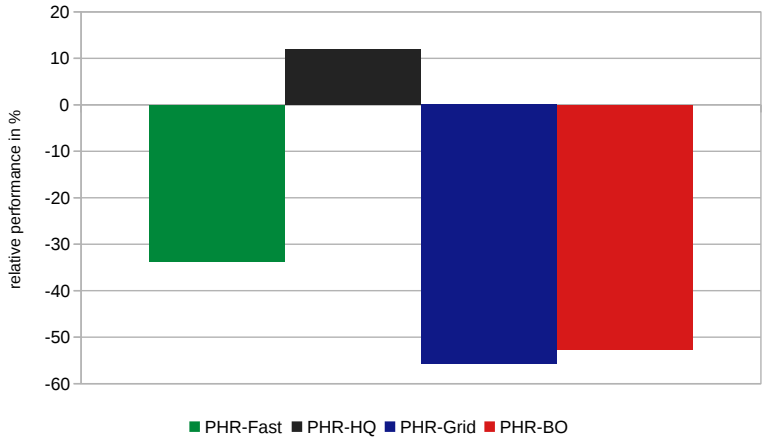


Figure 7.1: Relative difference to LBVH in percent.

7.3 Optimization Performance

Figure 7.2 shows optimization times divided through average frame time, or in other words, how many frames could potentially be rendered instead of executing the optimization. Considering that the average performance increase of both methods amounted around 50%, i.e. can potentially half the rendering time, optimizations start to become viable at half a frames duration and below. So even though the search space used in grid search is relatively low, the achieved times never reached what would be feasible in interactive applications. Bayesian optimization performed considerably better, but only reached competitive times in a few cases. Note that reaching this time does not equal a performance increase but just a hypothetical chance that frame times are increased.

This shows that the presented optimization approaches still lack in performance and are not yet viable. PHR needs to be executed to evaluate the cost function, which is especially costly for parameters that result in high build times. This could be improved by limiting the search spacer further or making it dynamic and related to the scene's complexity. An interrupt after a maximum execution time might also be a solution. Bayesian optimization uses a costly run with maxed out parameters to determine the maximum build cost. This could be solved by reusing max values from previous optimizations. This topic is discussed further in section 8.3.

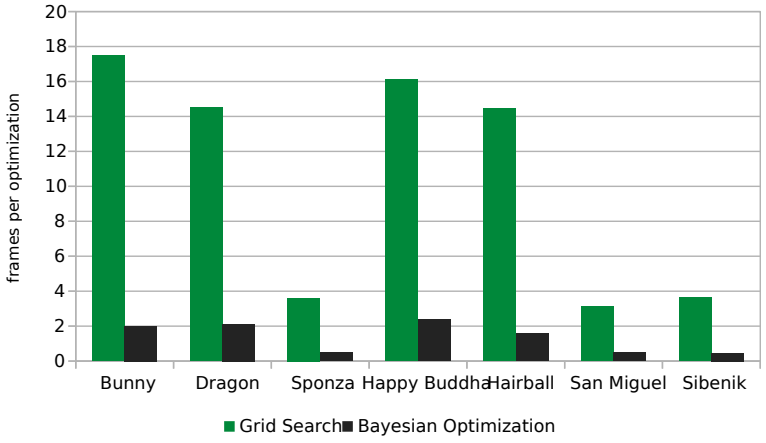
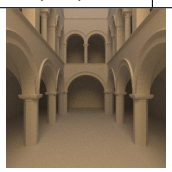
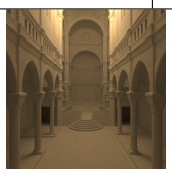


Figure 7.2: Number of potential frames during optimization.

Table 7.1: Performance comparison of a representative selection of tested configuration at 256x256 resolution.

	build time (ms)	render time (ms)	frame time ms	build time (ms)	render time (ms)	frame time ms
	 Bunny #triangles 144k			 Sponza #triangles 66k		
LBVH	152.0	21.9	173.9	70.4	227.0	297.5
PHR-Fast	70.5	20.1	90.6	27.6	184	211.6
PHR-HQ	273.0	20.7	293.7	94.7	182	276.7
PHR-Grid	14.4	20.8	35.2	17.2	207	224.2
PHR-BO	14.2	20.7	34.97	21.7	200	221.7
	 Dragon #triangles 817k			 Sibenik #triangles 75k		
LBVH	900.0	28.1	928.2	72.1	226.1	298.1
PHR-Fast	150.0	32.5	182.5	32.5	205	237.5
PHR-HQ	1090.0	29.0	1119.0	98.4	187	285.4
PHR-Grid	31.7	63.5	95.2	19.1	236	255.1
PHR-BO	30.8	66.1	96.9	28.1	220	248.1
	 Happy Buddha #triangles 1087k			 San Miguel #triangles 5617k		
LBVH	1080	25	1105	6010.0	816.0	6826.6
PHR-Fast	190	27.6	217.6	432.0	7830.0	8262.0
PHR-HQ	1250	23.7	1273.7	2600.0	812.3	3412.3
PHR-Grid	40.5	57.5	98	1300.0	1856.6	3156.6
PHR-BO	42.1	58	100.1	815.0	3203.3	4018.3

8 Discussion

8.1 CPU vs GPU

In general, CPUs are designed to execute serial instructions on an intermediate amount of data very fast, while GPUs are optimized to process instructions in parallel using little memory but maximizing throughput. Consequently, CPUs tend to utilize a bigger coherent cache than GPUs do and consist of few powerful cores that use techniques like pipelining, branch prediction and out-of-order execution to operate on data as efficiently as possible. GPUs, on the other hand, are built from many weaker cores that favor throughput over efficiency in individual cores, which makes graphic cards excel with highly coherent tasks. Conventional GPUs are optimized to run the graphics pipeline, with hardware units constructed specifically for tasks in the rasterization process.

While path tracing is an embarrassingly parallel problem, the coherency of GPUs cannot be utilized to a full extend, as individual rays can often be very incoherent. Through the random nature of path tracing, rays might be scattered in all kinds of directions hitting different primitives in the process. In order to find ray intersections, it is also advantageous to have the whole acceleration structure in memory. With increasing scene complexity, this cannot always be provided by GPUs, given the limited amount of on board memory. Data transfer between CPU and GPU is costly and should be avoided when possible.

Given those disadvantages, one goal of this thesis was to evaluate the performance of CPU path tracing. While interactive frame rates were achieved given reasonably small scenes and resolutions, the numbers did suggest that the presented CPU path tracer has not reached sufficient performance to be used as the sole rendering engine behind interactive, let alone real-time applications. This might be changed in the future though, as the tested implementation was only optimized moderately and still provides a lot of potential for further performance gains, especially through the use of SIMD instructions, better data layouts and further path tracing optimization methods. Moreover, the programming language used to implement this path tracer has a few shortcomings in terms of performance, which will be discussed in the next section (Programming Language).

Nevertheless, a better use case would be in the context of a hybrid path tracer where equivalent CPU and GPU implementations are combined. CPUs are mostly unused in real-time rendering, so the available capacity could be used to render frame subregions and relief the GPU that way. The BVH construction algorithm also showed promising results, so the CPU could be used mainly for the generation of acceleration structures. Note that these approaches are only hypothetical, but they provide interesting topics for future research nonetheless.

8.2 Programming Language

The project was implemented in Go, an open source programming language originally aimed at networking applications. Consequently, the language provides great support for writing multi-threaded applications. This is facilitated through a lightweight thread management system in the form of so called goroutines [Deshpande and Weiss, 2012]. These are not managed by the operating system but through Go's runtime scheduler, so they only exist in a virtual environment. Goroutines can be created very cheaply, as there initial stack only consists of a few kilo bytes. On blocking calls the scheduler automatically moves other goroutines to the blocked operating system thread. This keeps concurrent programming very efficient and simple, which was one of the main reasons for using Go. The language also offers great out of the box support for micro-benchmarking and benchmarking in general, which came in handy as well.

However, while Go allows to write code somewhat close to the system, it is still a rather high level language. It uses a garbage collector that, even though very well optimized, is a substantial

performance sink. Go's compiler also favors compile time over execution performance, so the performance is not comparable to faster programming languages.

As a result, some optimizations were necessary to overcome some of Go's shortcomings. For example, many parts of the project are written with a zero allocation approach to avoid the need for expensive garbage collection. Structures are reused were possible and C style pointer parameter returns are used to avoid allocating return values. Slices, go's implementation of extendable arrays, are allocated with an appropriate capacity when available and reused when feasible. Even though Go supports pointers, arguments are often used as values which might sound counter intuitive at first. This is based on an issue mentioned by Changkun Ou [Ou, 2020b] and has to do with different memory access patterns. Pointers need to be dereferenced, while the Go compiler is able to inline value calls to avoid moving values between registers.

Despite Go being a relatively fast programming language, the focus on code readability over code performance is noticeable. Implementing the project in a faster language like C, C++ or Rust, might lead to better results, so translating the implementation into one of those languages and comparing their performance might be an interesting topic for future work.

8.3 Execution of the Optimization Step

While the grid search approach delivered more reliable results than Bayesian optimization, it is too costly to be used in interactive applications. Bayesian optimization had more competitive execution speeds, but its results were not optimal and fairly inconsistent. Whether adding the optimization time on top of the frame time actually improves the overall rendering performance depends on how ineffective the previous parameters were. Optimizing very malformed parameters might be worth it, but otherwise the optimization process adds useless computation complexity. However, there is also a lot of optimization potential left open, so this could be changed in the future.

Furthermore, the optimization time could be decreased by only executing it after a certain amount of time or frames has elapsed. This makes sense, as only significant scene changes amount to a favorable parameter optimization. In practice, many parts of dynamic scenes stay static over large periods of time and only parts of the scene are actually changed. Consequently, running the optimization on every frame might not lead to any parameter changes anyways. However, efficiently executing parameter optimization in dynamic scenes remains an open topic.

8.4 Ray Distribution Problem

The surface area heuristic used to split cuts in PHR and evaluate the cost function proposed in section 6.1 assumes that ray origins and ray directions are uniformly distributed outside the scene's bounding box. This is a quite unrealistic assumption, especially in the context of real-time rendering where the camera often moves through a scene. One SAH correction proposed by Bittner and Havran [Bittner and Havran, 2009] is the Ray Distribution Heuristic based on sampling of the ray distribution:

$$P(N_c|N)^{RDH} = \frac{R(N_c)}{R(N)}$$

where $R(N)$ is the number of rays hitting the bounding box of node N. Even though using RDH on its own might lead to unstable results due to under or oversampling, it would be an interesting topic to combine RDH with both the splitting process and the hyperparameter evaluation. Especially the latter could profit from such an evaluation, as the optimal PHR parameters differ between view points for each scene. This is not considered by the evaluation proposed in section 6.1, as it is based on the SAH and suffers from the ray distribution problem.

9 Conclusion and Future Work

Given the extensive nature of the topic, a lot of work is left open. Regarding the path tracer, incorporating any sort of denoising is a necessary step to achieving usable results. The performance is not quite at a sufficient point, so finding additional optimizations is another open topic. As mentioned before, Go is not an ideal language for such a performance critical task, so translating the project into a better fit language would be interesting as well. Importance sampling is another technique that was not considered in this work, but might improve the results by a decent amount.

The implementation progressive hierarchical refinement could be improved further by optimizing construction of auxiliary BVHs, as this part still lacks in performance. The full sweep SAH evaluation at the core of the algorithm is relatively expensive, so replacing it by binning SAH might be advantageous for interactive and real-time applications. Evaluating the influence of other cost functions like the RDH to both the split function and the parameter cost evaluation might lead to interesting results. SIMD instructions were not used in any part of this project, so applying those where appropriate could improve the path tracer significantly.

Path tracing is an essential part in computer graphics and recent advancements in real-time path tracing pushed its popularity even further. This thesis presented an overview over the basics of path tracing and how an interactive path tracer might be implemented based on an example written for CPUs in the programming language Go. The evaluation of this path tracer showed that interactive frame rates can be achieved for reasonably small scenes and resolutions, but competitive usage requires additional work.

Furthermore, the state-of-the-art algorithm called progressive hierarchical refinement was presented and extended by a novel optimization technique for the used hyperparameters. This method was tested extensively and its potential performance increase was validated.

Acronyms

AABB Axis-aligned Bounding Box. 4

BVH Bounding Volume Hierarchy. 5

LBVH Linear Bounding Volume Hierarchy. 7, 13

OBB Oriented Bounding Box. 4

PHR Progressive Hierarchical Refinement. 2

RDH Ray Distribution Heuristic. 23

SAH Surface Area Heuristic. 5

SIMD Single Instruction, Multiple Data. 8

Bibliography

References

[cho, 2010] (2010). Parallel sah k-d tree construction. In Laine, S., Hunt, W., and Doggett, M., editors, *High-Performance Graphics 2010 - ACM SIGGRAPH / Eurographics Symposium Proceedings, HPG 2010*, High-Performance Graphics - ACM SIGGRAPH / Eurographics Symposium Proceedings, HPG, pages 77–86. Association for Computing Machinery.

[Apetrei, 2014] Apetrei, C. (2014). Fast and Simple Agglomerative LBVH Construction. In Borgo, R. and Tang, W., editors, *Computer Graphics and Visual Computing (CGVC)*. The Eurographics Association.

[Appel, 1968] Appel, A. (1968). Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, page 37â45, New York, NY, USA. Association for Computing Machinery.

[Bittner and Havran, 2009] Bittner, J. and Havran, V. (2009). Rdh: Ray distribution heuristics for construction of spatial data structures.

[Brunetti and Caccioppoli, 2014] Brunetti, M. and Caccioppoli, R. (2014). Old and new proofs of cramer's rule. *Applied mathematical sciences*, 8:6689–6697.

[Chaitanya et al., 2017] Chaitanya, C. R. A., Kaplanyan, A. S., Schied, C., Salvi, M., Lefohn, A., Nowrouzezahrai, D., and Aila, T. (2017). Interactive Reconstruction of Monte Carlo Image Sequences using a Recurrent Denoising Autoencoder. *ACM Transactions on Graphics*, 36(4):12.

[Chitalu et al., 2020] Chitalu, F., Dubach, C., and Komura, T. (2020). Binary ostensibly-implicit trees for fast collision detection. *Computer Graphics Forum*, 39:509–521.

[Clark, 1976a] Clark, J. H. (1976a). Hierarchical geometric models for visible surface algorithms. 19(10):547â554.

[Clark, 1976b] Clark, J. H. (1976b). Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554.

[Clark, 1976c] Clark, J. H. (1976c). Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554.

[Cook et al., 1984] Cook, R. L., Porter, T., and Carpenter, L. (1984). Distributed ray tracing. *ACM SIGGRAPH Computer Graphics*, 18(3):137–145.

[de Greve, 2004] de Greve, B. (2004). Reflections and refractions in ray tracing.

[Deshpande and Weiss, 2012] Deshpande, N. and Weiss, N. (2012). Analysis of the go runtime scheduler.

[Fuchs et al., 1980] Fuchs, H., Kedem, Z. M., and Naylor, B. F. (1980). On visible surface generation by a priori tree structures. 14(3):124â133.

[Fujimoto et al., 1986] Fujimoto, A., Tanaka, T., and Iwata, K. (1986). ARTS: Accelerated Ray-Tracing System. *IEEE Computer Graphics and Applications*, 6(4):16–26. Conference Name: IEEE Computer Graphics and Applications.

[Garanzha et al., 2011] Garanzha, K., Pantaleoni, J., and McAllister, D. (2011). Simpler and faster hlbvh with work queues. pages 59–64.

[Glassner, 1984] Glassner, A. S. (1984). Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–24. Conference Name: IEEE Computer Graphics and Applications.

[Goldsmith and Salmon, 1987] Goldsmith, J. and Salmon, J. (1987). Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20.

[Haines and Akenine-Möller, 2019] Haines, E. and Akenine-Möller, T., editors (2019). *Ray Tracing Gems*. Apress. http://raytracinggems.com.

[Havran, 2000] Havran, V. (2000). *Heuristic Ray Shooting Algorithms*. PhD thesis.

[Havran and Bittner, 2002] Havran, V. and Bittner, J. (2002). On improving kd-trees for ray shooting. In *In Proc. of WSCG 2002 Conference*, pages 209–217.

[Havran et al., 2006] Havran, V., Herzog, R., and Seidel, H.-P. (2006). On the fast construction of spatial data structures for ray tracing. *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing, IEEE, 71-80 (2006)*.

[Hendrich et al., 2017] Hendrich, J., Meister, D., and Bittner, J. (2017). Parallel BVH Construction using Progressive Hierarchical Refinement. *Computer Graphics Forum*, 36:487–494.

[Hou et al., 2011] Hou, Q., Sun, X., Zhou, K., Lauterbach, C., and Manocha, D. (2011). Memory-scalable gpu spatial hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics*, 17(4):466–474.

[Hunt et al., 2007] Hunt, W., Mark, W. R., and Fussell, D. (2007). Fast and lazy build of acceleration structures from scene hierarchies. In *2007 IEEE Symposium on Interactive Ray Tracing*, pages 47–54.

[Kajiya, 1986] Kajiya, J. T. (1986). THE RENDERING EQUATION. 20(4):8.

[Kalantari et al., 2015] Kalantari, N. K., Bako, S., and Sen, P. (2015). A Machine Learning Approach for Filtering Monte Carlo Noise. *ACM Transactions on Graphics (TOG) (Proceedings of SIGGRAPH 2015)*, 34(4).

[Kaplan, 1985] Kaplan, M. R. (1985). The use of spatial coherence in ray tracing. In *ACM SIGGRAPH*, volume 85, pages 22–26.

[Karras, 2012] Karras, T. (2012). Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, page 33â37, Goslar, DEU. Eurographics Association.

[Keller et al., 2015] Keller, A., Fascione, L., Fajardo, M., Georgiev, I., Christensen, P., Hanika, J., Eisenacher, C., and Nichols, G. (2015). The path tracing revolution in the movie industry. In *ACM SIGGRAPH 2015 Courses*, SIGGRAPH '15, New York, NY, USA. Association for Computing Machinery.

[Koskela et al., 2019] Koskela, M., Immonen, K., Mäkitalo, M., Foi, A., Viitanen, T., Jääskeläinen, P., Kultala, H., and Takala, J. (2019). Blockwise multi-order feature regression for real-time path-tracing reconstruction. *ACM Trans. Graph.*, 38(5).

[Lauterbach et al., 2009] Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., and Manocha, D. (2009). Fast bvh construction on gpus. *Computer Graphics Forum*, 28(2):375–384.

[Li et al., 2017] Li, Z., Deng, Y., and Gu, M. (2017). Path compression kd-trees with multi-layer parallel construction a case study on ray tracing. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '17, New York, NY, USA. Association for Computing Machinery.

[MacDonald and Booth, 1990] MacDonald, J. D. and Booth, K. S. (1990). Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6(3):153–166.

[Mara et al., 2017] Mara, M., McGuire, M., Bitterli, B., and Jarosz, W. (2017). An efficient denoising algorithm for global illumination. In *Proceedings of High Performance Graphics*, New York, NY, USA. ACM.

[McGuire, 2017] McGuire, M. (2017). Computer graphics archive. `https://casual-effects.com/data`.

[Meister et al., 2021] Meister, D., Ogaki, S., Benthin, C., Doyle, M., Guthe, M., and Bittner, J. (2021). A survey on bounding volume hierarchies for ray tracing. *Computer Graphics Forum*, 40:683–712.

[Möller and Trumbore, 1997] Möller, T. and Trumbore, B. (1997). Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools*, 2(1):21–28.

[Morton, 1966] Morton, G. (1966). A computer oriented geodetic data base and a new technique in file sequencing.

[Nvidia, 2018] Nvidia (2018). Nvidia turing gpu architecture. White Paper WP-09183-001$_v$01.

[Ou, 2019] Ou, C. (2019). bo. https://github.com/changkun/bo. Open sourced under MIT license on GitHub, 2019. Last access 2021-01-09, master branch, commit hash 9260b3ccfc857748c9996e7bf373fe8730eea168.

[Ou, 2020a] Ou, C. (2020a). bench. https://github.com/golang-design/bench. Open sourced under GPL-3.0 license on GitHub, 2020. Last access 2021-01-09, master branch, commit hash ee8e327ca74084d890dd11aad7856b232e95a308.

[Ou, 2020b] Ou, C. (2020b). Pointers might not be ideal as arguments. Last access 2021-05-09.

[Pantaleoni and Luebke, 2010] 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, page 87â95, Goslar, DEU. Eurographics Association.

[Pelikan et al., 1999] Pelikan, M., Goldberg, D. E., and Cantú-Paz, E. (1999). Boa: The bayesian optimization algorithm. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 1*, GECCO'99, page 525â532, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

[Popov et al., 2009] Popov, S., Georgiev, I., Dimov, R., and Slusallek, P. (2009). Object partitioning considered harmful: Space subdivision for bvhs. pages 15–22.

[Roccia et al., 2012] Roccia, J.-P., Paulin, M., and Coustet, C. (2012). Hybrid CPU/GPU KD-Tree Construction for Versatile Ray Tracing. In Andujar, C. and Puppo, E., editors, *Eurographics 2012 - Short Papers*. The Eurographics Association.

[Schied et al., 2017] Schied, C., Kaplanyan, A., Wyman, C., Patney, A., Chaitanya, C. R. A., Burgess, J., Liu, S., Dachsbacher, C., Lefohn, A., and Salvi, M. (2017). Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination. *Los Angeles*, page 12.

[Sugerman et al., 2009] Sugerman, J., Fatahalian, K., Boulos, S., Akeley, K., and Hanrahan, P. (2009). Gramps: A programming model for graphics pipelines. *ACM Transactions on Graphics*, 28:1–11.

[Timrb, 2008] Timrb (2008). Ray trace diagram. https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.png. Published under the CC BY-SA 3.0 license.

[Vinkler et al., 2017] Vinkler, M., Bittner, J., and Havran, V. (2017). Extended morton codes for high performance bounding volume hierarchy construction. In *Proceedings of High Performance Graphics*, HPG '17, New York, NY, USA. Association for Computing Machinery.

[Vinkler et al., 2015] Vinkler, M., Havran, V., and Bittner, J. (2015). Performance comparison of bounding volume hierarchies and kd-trees for gpu ray tracing. *Computer Graphics Forum*, 35:n/a–n/a.

[Vinkler et al., 2012] Vinkler, M., Havran, V., and Sochor, J. (2012). Visibility driven bvh build up algorithm for ray tracing. *Computers Graphics*, 36.

[Wald, 2007] Wald, I. (2007). On fast construction of sah based bounding volume hierarchies. pages 33 – 40.

[Wald et al., 2008] Wald, I., Benthin, C., and Boulos, S. (2008). Getting rid of packets - efficient simd single-ray traversal using multi-branching bvhs -. In *2008 IEEE Symposium on Interactive Ray Tracing*, pages 49–57.

[Wald et al., 2007] Wald, I., Boulos, S., and Shirley, P. (2007). Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Trans. Graph.*, 26.

[Walter et al., 2008] Walter, B., Bala, K., Kulkarni, M., and Pingali, K. (2008). Fast agglomerative clustering for rendering. *2008 IEEE Symposium on Interactive Ray Tracing*, pages 81–86.

[Weik, 2001] Weik, M. H. (2001). *Lambert's cosine law*, pages 868–868. Springer US, Boston, MA.

[Whitted, 1980] Whitted, T. (1980). An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349.

[Wu et al., 2011] Wu, Z., Zhao, F., and Liu, X. (2011). Sah kd-tree construction on gpu. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, page 71â78, New York, NY, USA. Association for Computing Machinery.

[Yan et al., 2014] Yan, L.-Q., Uday Mehta, S., Ramamoorthi, R., and Durand, F. (2014). Fast 4d sheared filtering for interactive rendering of distribution effects. Technical Report UCB/EECS-2014-174, EECS Department, University of California, Berkeley.