



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

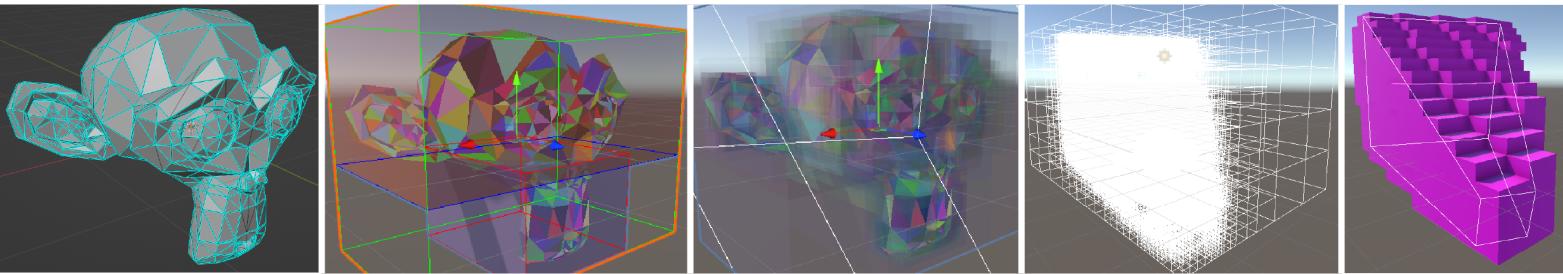
Ray Distribution Aware Heuristics for Bounding Volume Hierarchies Construction

TESI DI LAUREA MAGISTRALE IN
COMPUTER SCIENCE AND ENGINEERING

Author: **Lapo Falcone**

Student ID: 996089
Advisor: Prof. Marco Gribaudo
Academic Year: 2023-24





Abstract

In the last few years, real-time computer graphics have been transitioning from a pipeline based on rasterization to one using ray tracing. Ray tracing makes it possible to accurately simulate the behavior of light rays, enabling developers of graphics content to reproduce high-fidelity scenes without using a plethora of techniques to mimic light transport.

While ray tracing is widely used for off-line rendering, such as for CGI effects in films or animated movies, the same cannot be stated for on-line applications, such as videogames. The main problem with ray tracing is that simulating light transport is computationally expensive, reason why in recent videogames ray tracing is only used on small portions of the scene, or to simulate some effects (such as reflections, shadows, or ambient occlusion).

In order to increase the spread of ray tracing in on-line rendering applications too, research is moving in two macro directions.

The first one is to build GPUs with an architecture more suited for ray tracing, such as the RT cores from Ampere Nvidia GPUs.

The second, but not least important one, is to design software optimizations to make ray tracing cheaper.

One of the problems that is ubiquitous in the ray tracing environment is to detect collisions between a ray and the geometry of the scene to render. Given the huge amount of primitives present in modern graphic applications, it is necessary to use a data structure to accelerate the ray collision retrieval process. The state-of-the-art structure is the Bounding Volume Hierarchy (BVH), which hierarchically organizes primitives, making it possible to skip entire sections of the scene that are spatially far away from the ray that is being traced during BVH traversal.

In this thesis we propose two novel heuristics, working upon the same assumptions, to build higher-quality BVHs, a data structure to make it possible to use them, and a comparative analysis of their performance in different scenarios.

The first heuristic, called **Projected Area Heuristic** (PAH), aims at better estimating the amount of rays that hit each node of the BVH by exploiting some artifacts in the ray distribution in the scene, caused by another optimization used in a previous step of the ray tracing pipeline (namely Monte-Carlo importance sampling).

The second one, **Splitting Plane Facing Heuristic** (SPFH) aims at reducing the overlap among nodes of the BVH, consequently reducing the number of intersection tests needed during the BVH traversal phase.

Keywords: Ray tracing, bounding volume hierarchy, BVH

Abstract in Lingua Italiana

Abstract Italiano

Parole chiave: Ray tracing, bounding volume hierarchy, BVH

Contents

Abstract	ii
Abstract in Lingua Italiana	iii
Contents	iv
Introduction	1
1 Background Theory	9
1.1 Ray Tracing Principles	9
1.1.1 Use Cases	11
1.1.2 Optimizations Overview	11
1.1.3 Backward Ray Tracing	12
1.2 Monte-Carlo and Variance Reduction Techniques	13
1.2.1 Kajiya’s Rendering Equation	13
1.2.2 Monte-Carlo Integration	16
1.2.3 Variance Reduction Techniques	21
1.3 Ray Tracing Acceleration Structures	24
1.3.1 The Need for Acceleration Structures	24
1.3.2 The Bounding Volume Hierarchy	29
1.3.3 BVH Construction	32
2 Projected Area Heuristic	39
2.1 SAH Hypotheses Fall	39
2.2 Parallel Ray Distribution	42
2.3 Point Ray Distribution	43
2.4 The Projected Area Heuristic	44
3 Multiple Influence Areas and the Top Level Structure	47
3.1 Ray-Influence Area Affinity	48

3.2 Local and Global BVHs	49
3.3 Top Level Structure	51
4 Splitting Plane Facing Heuristic	53
5 Implementation	58
5.1 BVH	59
5.1.1 Splitting Plane Search Implementation	61
5.1.2 Traversal	62
5.2 BVH Analyzer	63
5.3 Regions	64
5.3.1 Oriented Bounding Box	65
5.3.2 Axis Aligned Bounding Box	66
5.3.3 AABB for OBB	68
5.3.4 Frustum	68
5.4 Influence Areas	70
5.4.1 Plane Influence Area	71
5.4.2 Point Influence Area	76
5.5 Top Level Structure	80
5.5.1 AABBs for Regions	80
5.5.2 Octree	81
5.6 Additional Structures	88
5.6.1 Test Scene	89
5.6.2 Ray Caster	89
5.6.3 CSV Exporter	90
6 Experimental Results	91
6.1 Test Scenes	92
6.2 Comparison with State of the Art Heuristics	95
6.2.1 PAH versus SAH Precision	95
6.2.2 Projected Area Culling	96
6.2.3 PAH and SPFH versus SAH and LSPH	97
6.2.4 Building Performance	100
6.2.5 Breakeven Hit Percentage	102
6.2.6 SPFH in Isolation	105
6.2.7 PAH in Isolation	106
6.2.8 Siblings Overlapping	108
6.3 Octree versus AABBs for Regions Top Level Structures	112

6.4	Testing Different Properties for PAH and SPFH BVH Construction	116
6.5	Memory Footprint	118
7	Conclusion	122
7.1	Use Cases	122
7.2	Future Developments	123
7.3	Closing Thoughts	124
Bibliography		126
A Collision and Culling Algorithms		131
A.1	Ray-AABB Intersection	131
A.2	Ray-Plane Intersection	133
A.3	Ray-Triangle Intersection	134
A.4	AABB-AABB Intersection	135
A.5	Frustum-AABB Intersection	135
A.5.1	1D Projections Overlapping Test	137
A.6	Point inside AABB Test	138
A.7	Point inside Frustum Test	138
A.8	Point inside 2D Convex Hull Test	139
A.9	2D Convex Hull Culling	140
A.9.1	Vertices inside convex hull	142
A.9.2	Edges intersections	142
A.9.3	Vertices ordering	143
A.10	2D Hull Area Computation	144
B	Sphere Cap Sampling	145
List of Figures		149
List of Tables		154
List of Symbols		155
Ringraziamenti		156



Introduction

What Is Ray Tracing

In the field of computer graphics, we refer to ray tracing as a family of rendering algorithms that simulate light transport, in order to transition from a mathematical representation of a scene to an image on the screen.

The core algorithm of ray tracing is conceptually simple. First, we cast some rays from each light source in the scene. The rays will collide and bounce away from the objects present in the scene, losing part of their energy. At some point a ray will either lose all of its energy and disappear, or hit the eye of an observer at a specific position. The eye of the observer can be thought as a camera, and the position on the camera as the pixel. This means that we can assign a color to the hit pixel, based on the remaining energy of the ray. If more than one ray hits the pixel, the contributions are summed together, if none hits it, the pixel remains black.

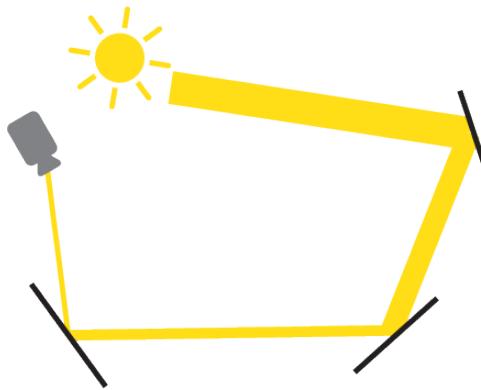


Figure 1: Visualization of the process we described above.

The process described above is a simplification of the behavior of the light in real world, and the way in which our brain can see. A lot of different specializations of the ray tracing algorithm exist. Some of them are designed to simulate light very precisely, such as bidirectional path tracing techniques [23]. Others are designed to be fast.

What all the techniques have in common is the fact that simulating light is an expensive task. The main reason lies in the number of rays that must be cast in order to obtain an accurate rendering of the scene. In the real-world a 100W lightbulb is estimated to produce 10^{20} photons each second [46]. Of course, even in a non real-time scenario, it is computationally impossible to ray trace an image with such an high amount of rays per light. Therefore, one of the most studied research directions in ray tracing is how to reduce the number of photons, while still obtaining an high quality image.

Monte-Carlo and Kajiya Rendering Equation

The tool used to achieve this result is Monte-Carlo integration [46]. With Monte-Carlo methods it is possible to integrate an arbitrary function with a given amount of random samples. In our case, the function to integrate is inside the Kajiya's rendering equation [16].

The Kajiya's rendering equation is, in summary, a recursive integral equation which calculates the amount of light a point in a scene gets from the surrounding environment. Once we know how much light a point receives, we can compute how much of this energy is reflected toward a given direction, by using the BRDF [57].

One common misconception is that a light ray, after hitting a surface, always bounces towards the perfect reflection direction. This is true only for mirror-like surfaces. Instead, in case the ray hits, for example, concrete, then there is the possibility that the ray bounces in any direction. Each possible bouncing direction has a probability of being chosen by the ray, which is described by the BRDF. In the real-world this happens because, at a microscopic level, a surface is made up of many small facets. This model is described by the micro-facet BRDFs [33].

Going back to the Kajiya's rendering equation, as we said, it needs in input the light from the surrounding environment. In order to obtain it, we can cast some probe rays in all directions, which will hit other objects. At this point we can, again, cast some probe rays, and so on, recursively, until a probe ray hits a light source. Now we can reverse the process, by following the probe rays backward, and compute the light hitting a certain point in the scene. The problem lies in the fact that we would need to cast a large amount of probe rays in order to obtain a good approximations of the light incoming to a given point. In other words, we are integrating the light function over the hemisphere oriented toward the surface normal.

Introduction

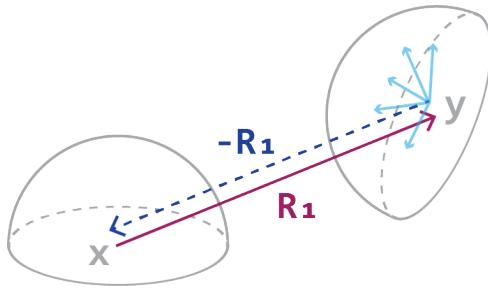


Figure 2: Recursiveness of Kajiya’s rendering equation.

What we described in the previous paragraph is a simplification of the Monte-Carlo method. With Monte-Carlo we have the mathematical proof that we can obtain a good estimate of the light incoming to a point by following this process, which involves the random sampling of the directions where to cast the probe rays.

Monte-Carlo variance reduction techniques, such as importance sampling [32][27], help us to achieve a precise approximation of the light incoming to a point with a low amount of probe rays, which is the goal we set.

However, importance sampling, creates artifacts in the distribution of the probe rays in the scene. If, with plain Monte-Carlo, the rays are distributed uniformly in the scene, by using importance sampling their distribution changes, and more rays tend to go toward light sources. This effect is the main hypothesis of the the novel heuristics we propose in this work.

Bounding Volume Hierarchies

Another fundamental task that any ray tracing algorithm must perform, is to find the intersection between a ray and the geometry of the scene. Usually the objects in the scene are described as surfaces made up of a big number of small triangles. Therefore, finding the intersection of a ray with a scene, reduces to finding the intersection between a ray and a large amount of triangles.

One way to do so is by using the brute-force approach: for each ray, we perform the ray-triangle intersection test with each triangle. If we have n rays and m triangles, $m * n$ intersection tests must be carried out.

Considering that we need a lot¹ of rays to achieve realistic images, and that we need a lot of triangles design accurate objects, the brute-force method is too slow.

¹at least one per pixel, at least $1080 \cdot 1920 = 2073600$ on HD displays.

Introduction

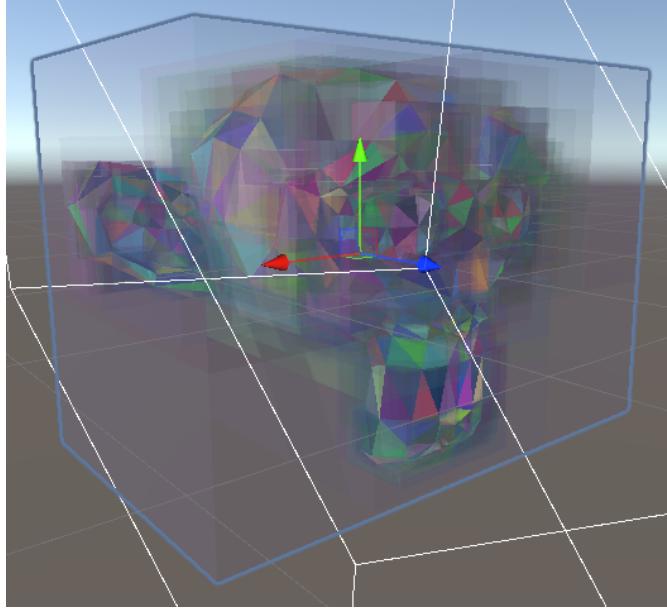


Figure 3: A BVH built with our heuristics over a radial ray distribution, as displayed by our Unity visualizer.

A better way to perform ray-scene intersection is to organize the triangles in a hierarchical fashion. We can first divide the scene into two sections, and delimit each one with a parallelepiped (an Axis-Aligned Bounding Box, or AABB): if the ray doesn't hit one of the two, we can exclude all the triangles present in that part of the scene. Bounding Volume Hierarchies (BVHs) extend this simple idea recursively. First the scene is divided into two parts; then each one of the two parts, is further divided into two sub-parts, and so on. The structure originating from this process is a binary tree. This means that, in order to perform ray-scene intersection, on average only $\log_2 n \cdot m$ intersection tests are needed.

One of the main concerns while building a BVH is how to split a BVH node into two parts so that the final binary tree actually allows to achieve the logarithmic complexity we described, during traversal. The idea is to group triangles that are close together, while minimizing the surface area of the two resulting AABBs. This heuristic is called Surface Area Heuristic (SAH), and is applied at each level of the BVH while it is built. SAH tries to estimate the probability an AABB is hit, and reduce it to the minimum, even though by using a greedy algorithm, due to time constraints.

SAH works under the hypothesis that the ray distribution in the scene is uniform. However, as we described in the previous section, we know that, due to Monte-Carlo importance sampling this is not the case.

By leveraging this finding, we propose two novel heuristics that are aware of some ray

Introduction



distributions in the scene, and use this knowledge to take better decisions while building the BVH.

The ray distributions our heuristics take into consideration are two: parallel ray distribution, and radial ray distribution.

Parallel ray distributions are usually found in proximity of planar area light sources, where, as the name suggests, rays tend to be parallel among them, and perpendicular to the light source. By knowing the presence of this distribution, we can try to estimate better the probability a ray hits an AABB present in that region. Indeed, if all rays are parallel, we know that only some faces of the AABB can be hit, while faces facing away from the camera can never be hit. Knowing this, we state that the probability of an AABB to be hit by a ray is proportional to the area of the polygon obtained by projecting it to a plane perpendicular to the rays. In other words, we use an orthographic projection to project the AABB to the planar area light, and compute the area of the projection.

For what concerns radial ray distributions, they tend to be present near point light sources, such as spotlights. In this case all the rays pass through the point light, forming a radial distribution over a certain solid angle. In this case, we estimate the hit probability by using a perspective projection, with focal point the position of the light source.

We call this heuristic to approximate the probability an AABB is hit Projected Area Heuristic (PAH).

The second novel heuristic we propose, takes into account the same distributions as PAH, but, instead of estimating the hit probability, it tries to minimize the overlapping between sibling nodes.

Let's think about parallel ray distribution. If we try to divide a node in this region into two children with a plane perpendicular to the rays, it is almost certain that the ray will hit both the children. This happens because the two children have a large overlapping area, when projected. If, instead, we cut the node with a plane parallel to the rays, the overlapping will be much smaller in many cases.

We call this method Splitting Plane Facing Heuristic (SPFH), as opposed to the state-of-the-art Longest Splitting Plane Heuristic (LSPH), where a node is always cut perpendicular to its longest side.

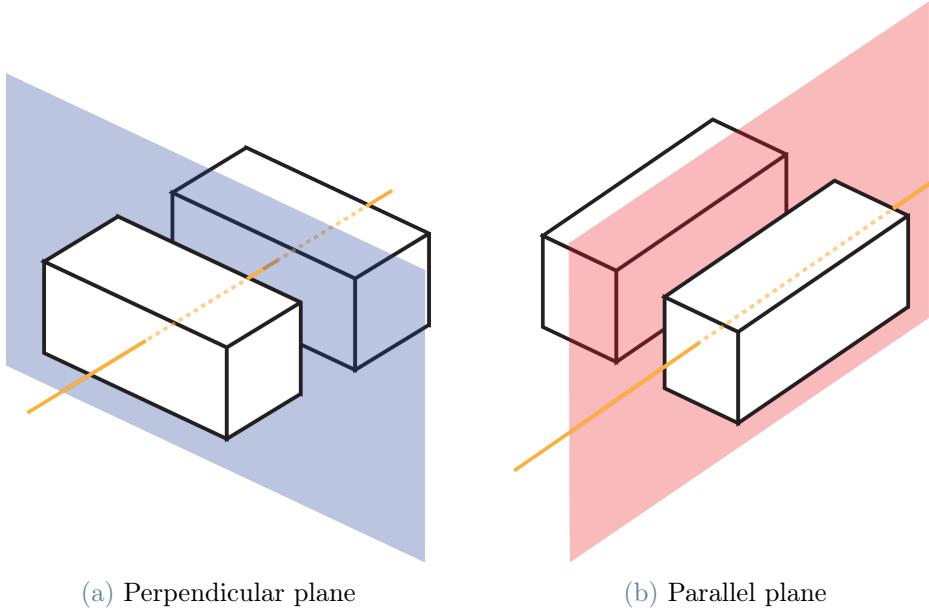


Figure 4: How a node can be split. SPFH would choose the parallel plane

Top Level Structure

If we build a BVH using SAH and LSPH, the information required by the heuristics are uniform in all of the scene, therefore just one BVH can be built. However, PAH and SPFH inherently divide the scene in local regions, where the relevant ray distributions are present. For this reason, we need a top level structure in order to find out whether, given a ray, it is *affine* with a certain ray distribution present in the scene, and can therefore be traced against the associated BVH.

With the term *affine* we refer to the fact that, not only the ray origin must be inside the region where a certain distribution is present, but also its direction must be *similar* to the directions of the rays present in the distribution.

The easier way to find out if a ray origin is in a certain region, is testing if the point is inside each distribution region present in the scene. However, this doesn't scale, therefore we decided to implement an octree structure.

Once we know that a ray origin is inside a certain ray distribution region, it is fast to check whether its direction is similar to the one of the distribution.

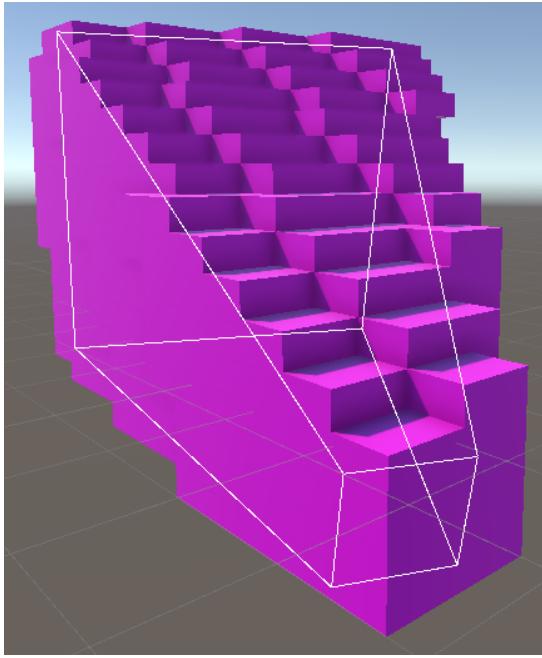


Figure 5: An octree as displayed by our Unity visualizer.

Thesis Outline

- In chapter 1 we will introduce the research context our thesis is part of, and we will explain the theory that is needed to fully understand our work. The focus will be on the parts that are operationally relevant, and we will try to highlight them with concrete examples, along with more rigorous mathematical proofs where needed. In particular we talk about the foundations of ray tracing, the Monte Carlo integration method and a variance reduction technique called importance sampling. Eventually we will explain what a ray tracing acceleration structure is, and why BVHs are the most popular ones.
- In chapters 2, 3 and 4 we will expose our novel heuristics to improve the building of BVHs in some specific scenarios. In these chapters we will mainly explain the theory, and we will only briefly talk about our implementation. The implementation will be diffusely discussed in chapter 5.
 - In chapter 2 we will detail the first novel heuristic we propose: the surface area heuristic (SAH). SAH aims at better estimating the quality of a BVH, in order to be able to make better decisions during the building phase.
 - In chapter 3 we will describe the data structures needed to enable the use of SAH, from a theoretical point of view.

Introduction



- In chapter 4 we will explain the second novel heuristic, called splitting plane facing heuristic (SPFH). SPFH is focused on the selection of the best plane to split the triangles during BVH construction.
- In chapter 5 we will extensively explain the implementation details we omitted from the previous three chapters. We decided to subdivide the thesis like this for two main reasons: do not overload the theory chapters, in order to be able to concisely get to the core concepts; and leave the reader with a section where all the implementation details are concentrated, so that, in case he or she wants to try our framework, this section can be used as a reference.
- In chapter 6 we will show the experimental results related to our heuristics, which we omitted or only briefly mentioned in the previous chapters. We will test our theory with the framework described in the previous chapter, and compare the results with those obtained by using different parameters or scenes, and also with the results obtained by using the state-of-the-art heuristics, namely the Surface Area Heuristic (SAH) and the Longest Splitting Plane Heuristic (LSPH).
- In chapter 7 we will list some possible scenarios where our techniques can be effectively used, based on the results obtained in the previous section. Moreover, we will produce a list of possible future development directions for the novel topics discussed in our thesis. Eventually, we will draw the main take-away points of our work.



1 | Background Theory

In this chapter we will summarize the background knowledge needed to fully comprehend this work.

In section 1.1 we will introduce ray tracing in simple terms, and list some of its advantages, disadvantages and its today's applications.

In section 1.2 we will analyze from a mathematical standpoint how the most used ray tracing algorithms work.

Last, in section 1.3, we will describe the state-of-the-art acceleration structure to make a fundamental ray tracing procedure (ray-scene intersection) faster.

1.1. Ray Tracing Principles

Ray tracing is a family of rendering algorithms that is used in computer graphics to transition from a mathematical representation of a scene, to an image on the screen.

Conceptually ray tracing is an extremely straightforward technique, that can be summarized in a few steps:

1. Generate a ray of light from a light source;
2. Find out the first object the ray intersects;
3. Compute how much energy is absorbed by the material of the object;
4. Modify the direction of the ray based on the material of the object (for example it may be reflected or refracted);
5. Repeat from 2. until the ray hits the camera or loses all its energy;
6. Color the pixel of the camera hit by the ray based on the energy of the ray.

Ray tracing aims to mimic the real-world behavior of light, and for this reason it can be directly employed to simulate any light effect, starting from the simplest ones, such as perfect reflections and shadows, going towards the most complex ones, such as global

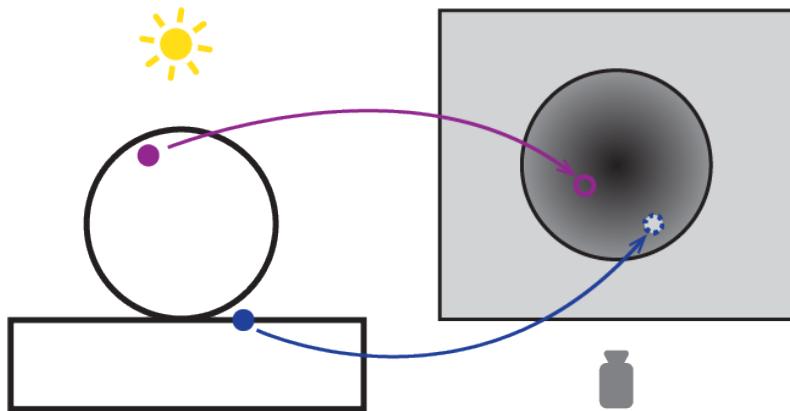


Figure 1.1: The first figure is from the main camera PoV, the second one from the light source PoV. The second figure represents depth: the closer a point is to the light source, the darker. The blue point is in shadow, because the corresponding point in the shadow map is further away than the stored depth.

illumination and caustics. Some of the most accurate ray tracing algorithms can even simulate quantum effects of light [39].

Since ray tracing natively simulates light, it can produce extremely realistic images, without resorting to ad-hoc techniques used to approximate light phenomena in other rendering algorithms, such as the widely spread rasterization pipeline.

To give an intuition of how the ad-hoc methods can be convoluted and produce worse results, we summarize one of the simplest ones used to generate shadows, called shadow mapping [5]. A shadow map is the projection of the scene from the point of view of a point light source, saved in a texture where each pixel stores the distance from the light source to the projected point. When the scene is projected by the main camera, each visible point is transformed into the coordinate system of the shadow map via matrix multiplication (figure 1.1). The point in the new coordinate system is then compared to the point stored in the corresponding pixel of the shadow map. If the point of the shadow map is closer to the light source than the corresponding point projected by the main camera, we deduce that such a point is not visible from the point of view of the light source and, therefore, is in shadow. This specific technique is correct only with point lights, and must be adjusted in case translucent objects are present in the scene.

With ray tracing shadows are natively generated since, if a point is in shadow, no light ray starting from it will hit the camera pixels. Moreover, in principle, it works with any kind of light, not only point lights, and thus produces higher-quality shadows, since no approximations must be made.

1.1.1. Use Cases

Due to the highly realistic images ray tracing algorithms can produce, the technique has found a lot of success in many applications. We can subdivide the use cases into two macro-categories: real-time ray tracing and production ray tracing.

The most prominent use of the second category is in movies. CGI effects and animated films are almost always produced by using ray tracing [20], in particular a very accurate algorithm called bi-directional path tracing [23]. In the first category we can find videogames.

The main difference between real-time and production ray tracing lies in the time constraints for producing a frame. In production ray tracing producing a frame can take a long time, even in the order of magnitude of days. Whereas, in real-time ray tracing, a frame must be produced every 33ms to achieve 30 frames per second (fps), and every 16.5ms to achieve 60 fps, which can be considered today's standard by many PC videogames.

This striking difference makes it so that different techniques must be used depending on the scenario. In real-time ray tracing many approximations must be introduced in order to stay within the frame time budget, whereas in production ray tracing more accurate algorithms can be used, since time constraints are loose. Our work can benefit both categories, since the methods we will introduce can make ray tracing faster, in some situations, without introducing approximations.

1.1.2. Optimizations Overview

Until now we briefly highlighted the strong points of ray tracing, and greatly simplified it. The main issue with ray tracing is that it is computationally expensive to simulate light transport in a convincing way. The reason lies in the rendering equation, and will be explained in section 1.2.

In order to make ray tracing usable in the real world, the industry moved in two directions:

Hardware accelerators GPU vendors, in particular Nvidia, started creating GPUs with specialized cores for ray tracing. These cores have a memory layout that makes it faster to find a ray-triangle intersection. An example of this can be the RT cores from Turing Nvidia GPUs [6].

Software optimizations Software developers introduced new algorithms to improve the performance of a specific part of the ray tracing pipeline. These algorithms can vary

a lot in complexity and results. One of the most common optimizations for real-time ray tracing is to use ray tracing only for some light effects (such as shadows or reflections), while using rasterization for most of the scene.

Software optimizations can, in turn, be subdivided into two big families.

Some optimizations aim at improving the time needed to detect the first intersection of a ray with the scene, without altering the quality of the rendered scene. We will diffusely talk about these optimizations in section 1.3, and this work can be placed into this family.

The other family comprehends optimizations aimed at reducing the number of rays needed to produce a visually acceptable image. This work, while being part of the first family of optimizations, has its foundations in an artifact in the ray distribution in the scene caused by an optimization of this second family. This technique is called importance sampling, and will be discussed in section 1.2.

1.1.3. Backward Ray Tracing

Before going to the next section, it is important to introduce the concept of backward ray tracing¹.

In backward ray tracing light rays don't start from light sources, but from the eye position. Starting from the eye, each ray will then hit a fictitious plane placed in front of the eye (the near plane), similar to what happens in a pinhole camera [59]. The near plane can be subdivided into discrete units displaced in a regular grid, which we will consider the pixels of the final image. Each ray is therefore associated with the pixel it hits, and will contribute to its final color.

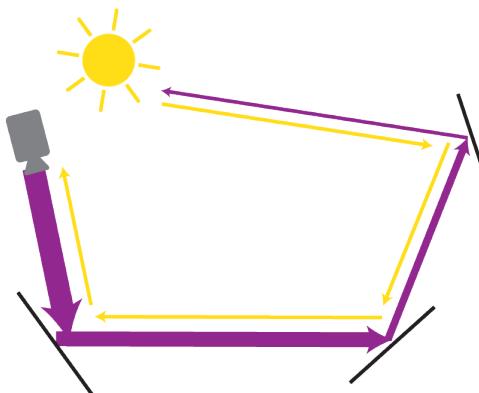


Figure 1.2: How rays are cast in backward ray tracing.

¹In some literature the term *backward ray tracing* can also refer to the opposite family of algorithms, since the first ray tracing methods were indeed backward.

The rays starting from the camera will hit the scene and bounce around. At each bounce a ray loses a fraction of its energy (unless the surface is perfectly reflective), until either it gets completely absorbed, or it hits a light source. In the case a ray hits a light source, we already know how much of its energy will be lost due to intersections with objects, therefore we can immediately compute the color of the pixel associated with the ray, as if we followed that same ray's path backward. If a ray never hits a light source, it will not contribute to the color of its associated pixel, because it never gain energy.

A way of looking at this algorithm is to think we are tracing *importants*, which can be considered the dual concept of *photons* [7]. Thanks to the light reciprocity principle [40], which states that light transmits in the same way in both directions, tracing photons or importants is equivalent.

The advantage of backward ray tracing is that, with a limited budget of rays we can cast, it is more efficient than forward ray tracing. Indeed, in forward ray tracing it is possible a ray never hits the camera, in which case it would be wasted. In backward ray tracing all the rays hit the camera by definition, thus none is wasted.

Of course, if a ray starting from the camera never hits a light source, it could be considered wasted, but some techniques that will be described in section 1.2 make it more likely for a ray to hit a light source in its path.

From this point on, we will consider the scenario where rays are traced backwards.

1.2. Monte-Carlo and Variance Reduction Techniques

In this section we will analyze the mathematical foundations of ray tracing, namely Kajiya's rendering equation. Then we will present a statistical method to resolve the integrals appearing in the rendering equation, called Monte-Carlo. Finally, we will discuss a variance reduction technique, importance sampling, that can be used to obtain better approximations from the Monte-Carlo method, without making it more expensive. We will see how this technique generates artifacts in the ray distribution in the scene, which is one of the hypotheses of our thesis.

1.2.1. Kajiya's Rendering Equation

$$L_o(\bar{x}, \bar{\omega}_o) = L_e(\bar{x}, \bar{\omega}_o) + \int_{\Omega} BRDF(\bar{x}, \bar{\omega}_i, \bar{\omega}_o) \cdot \cos(\bar{n}, \bar{\omega}_i) \cdot L_i(\bar{x}, \bar{\omega}_i) d\bar{\omega}_i$$

This equation, developed by James T. Kajiya in 1986 [16], can be used to calculate the

amount of light *generated* by a point \bar{x} towards a direction $\bar{\omega}_o$. As we will see in the next paragraphs, with the term *generated* we refer to the sum of the light emitted by the point, and the light reflected by it. In order to compute the *generated* light we need to know these variables:

- $L_e(\bar{x}, \bar{\omega}_o)$ The light emitted by the point \bar{x} towards a direction $\bar{\omega}_o$.
- $BRDF$ The bidirectional reflectance distribution function of the material at point \bar{x} .
- \bar{n} The normal to the surface at point \bar{x} .
- $L_i(\bar{x}, \bar{\omega}_i)$ The light incoming to \bar{x} from a generic direction $\bar{\omega}_i$ on the hemisphere Ω orientated toward \bar{n} .

The equation can be divided into two parts:

The first part describes the light emitted $L_e(\bar{x}, \bar{\omega}_o)$, and it is 0 unless the point is a light source. Depending on the light source type, it can assume a uniform value in each direction (point light), a value only in one hemisphere (a plane area light), or a specific value based on the direction, usually controlled by a function (such as in spotlight). In any case the emitted light value is known by the definition of the light source in the scene, therefore can be easily plugged into the equation.

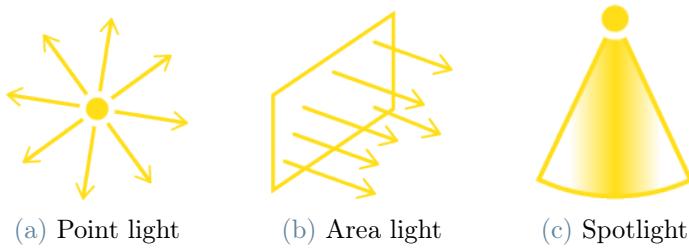
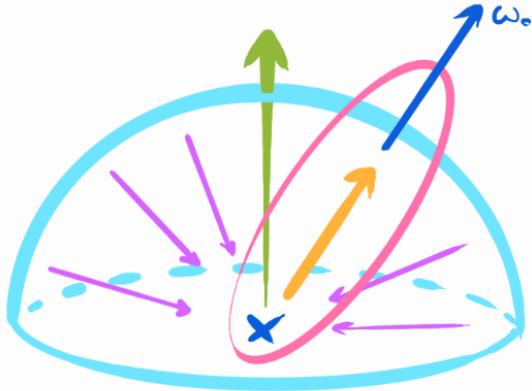


Figure 1.3: Three possible types of light sources.

The second part of the equation presents an integral, and represents the light reflected by the material of the point towards the direction $\bar{\omega}_o$. In simple words, this second term tells us that, in order to compute the reflected light, we have to know, first, how much light is incoming to the point from all the directions in the hemisphere Ω . And second, the properties of the material, summarized in the *BRDF* [57]. The *BRDF* tells us how much of the incoming light is reflected towards the direction $\bar{\omega}_o$.



$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_{\Omega} f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i) (\omega_i \cdot \mathbf{n}) d\omega_i$$

Figure 1.4: A visual representation of the Kajiya rendering equation from [4].

The term $\cos(\bar{n}, \bar{\omega}_i)$ is called geometry term, and reflects a core property of light, independent of the *BRDF*. Indeed, based on the angle a beam of light intersects a surface, the beam of light will enlighten a smaller or bigger area of the surface. The smallest surface is illuminated when the direction of the beam of light and the normal to the surface are parallel. Since the energy carried by the beam of light is constant, we can deduce that, the bigger the enlightened surface, the lower the energy per area unit. In particular, if we let L be the energy of the beam and α the angle between the beam direction and the normal, then the energy received per area unit equals to $L \cdot \cos(\alpha)$.

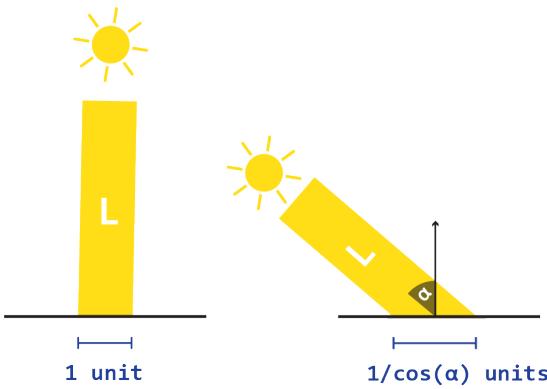


Figure 1.5: The geometry term.

When we compute the integral term of the rendering equation, the BRDF is known by definition, and the geometry term is a simple cosine. On the other hand, we almost never have an analytical form of the L_i term, and it is indeed this specific part of the equation what really makes ray tracing expensive.

In order to compute the L_i term for one specific direction $\bar{\omega}_i$, we can cast a probe ray R_1 from point \bar{x} toward $\bar{\omega}_i$. The probe ray will hit another point \bar{y} . Now, since we want to compute how much light R_1 is carrying towards \bar{x} , we have to resolve the rendering equation at point \bar{y} and with outgoing direction $-\bar{\omega}_i$, namely $L_o(\bar{y}, -\bar{\omega}_i)$. In other words, the light incoming to one point from one direction, equals the light outgoing from a second point towards the same direction, with inverted sign. This recursive pattern is usually ended after a certain number of bounces (when we can consider that the ray is carrying an infinitesimal amount of energy), or when a probe ray hits a light source (where the generated light is only influenced by the emitted light term of the rendering equation).

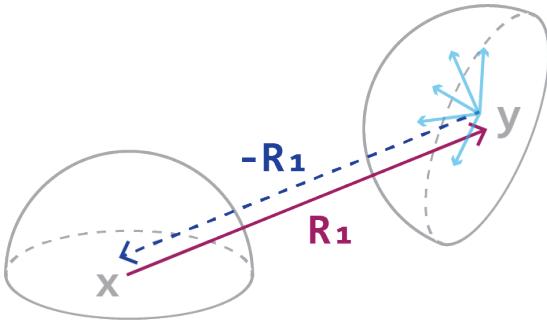


Figure 1.6: Recursiveness of the integral term of the Kajiya rendering equation.

The process we've just described only returns the amount of light incoming to \bar{x} from an arbitrary direction $\bar{\omega}_i$. But, as we can see, in the rendering equation an integral over the hemisphere Ω is present, therefore, in order to have a mathematically correct result, we would need to cast an infinite amount of probe rays. This is not feasible, and leads us to the Monte-Carlo method.

1.2.2. Monte-Carlo Integration

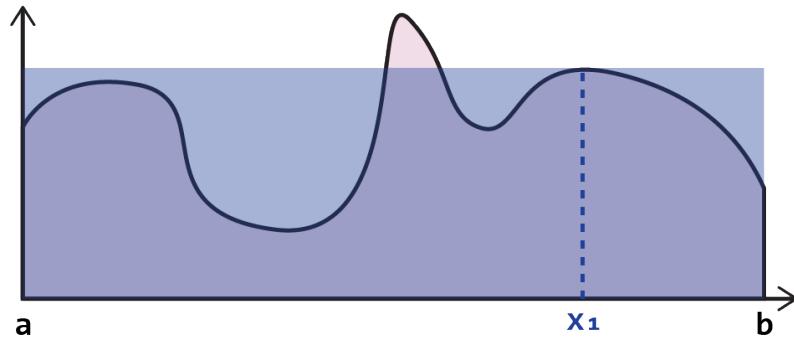
In this section we will provide an intuition of the Monte-Carlo integration method, and its basic mathematical foundations necessary to fully understand this work.

Monte-Carlo integration [46] is a method by which it is possible to compute the numerical integral of a function by using random samples. Given a one-dimensional positive integrable function f , we can interpret its definite integral in the $[a, b]$ interval as the area between the curve and the x-axis. Now, if we select a random point k_0 uniformly in the $[a, b]$ interval, we can compute the area A_r of the rectangle with side \bar{ab} and height $f(k_0)$ as $\bar{ab} \cdot f(k_0)$. We can interpret it as a very rough approximation of the area under the function, which, by definition, is equal to the definite integral value. If we repeat this process N times and compute the average of the various A_r s, we'll usually get a better

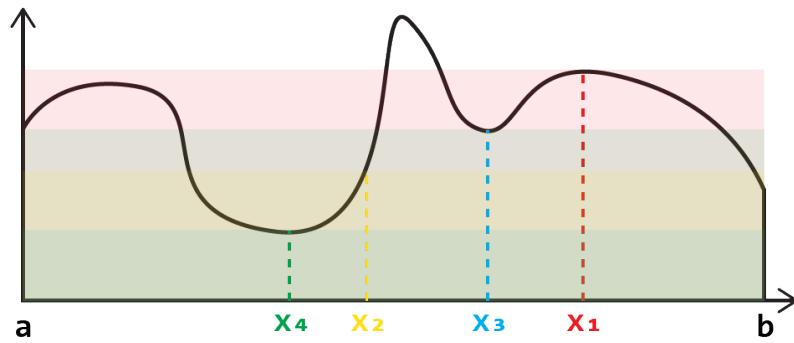
1 | Background Theory

17

estimate of the area under the function, because sometimes we will underestimate its value, and sometimes we will overestimate it.



(a) Area approximation with one sample.



(b) Area approximation with 4 samples

Figure 1.7: Monte-Carlo area approximation. When we have many samples, sometimes the area will be overestimated, some other times underestimated.

We can formalize this process with this formula, where $X_i \sim \frac{1}{b-a}$ is a uniform random variable in the $[a, b]$ interval.

$$\langle F^N \rangle = \frac{1}{N} \cdot (b - a) \cdot \sum_{i=0}^{N-1} f(X_i)$$

$\langle F^N \rangle$ is referred to as the basic Monte-Carlo estimator [48] [35]. The Monte-Carlo estimator, being a linear combination of random variables, is a random variable itself. Monte-Carlo theory states that the expected value of the Monte-Carlo estimator equals the definite integral of f . Below we prove this statement:

1 | Background Theory



$$\begin{aligned}
E[\langle F^N \rangle] &= E[(b-a) \cdot \frac{1}{N} \sum_{i=0}^{N-1} f(X_i)] \\
&= (b-a) \cdot \frac{1}{N} \cdot \sum_{i=0}^{N-1} E[f(X_i)] \\
&= (b-a) \cdot \frac{1}{N} \cdot \sum_{i=0}^{N-1} \int_a^b f(x) \cdot \frac{1}{b-a} dx \\
&= \frac{1}{N} \cdot \sum_{i=0}^{N-1} \int_a^b f(x) dx \\
&= \int_a^b f(x) dx
\end{aligned}$$

In order to go from line 2 to line 3, it is important to remember the law of the unconscious statistician (LOTUS) [43], where p is the probability density function of the random variable X :

$$E[f(X)] = \int_{\Omega} f(x) \cdot p(x) dx$$

The LOTUS is easier to understand in its discrete case, where $E[f(x)] = \sum f(x) \cdot p(x)$. This can, for example, be visualized to compute the expected value of a fair die throw: each side has $\frac{1}{6}$ chances of appearing, therefore the expected value equals: $\frac{1}{6} \cdot 1 + \frac{1}{6} \cdot 2 + \frac{1}{6} \cdot 3 + \frac{1}{6} \cdot 4 + \frac{1}{6} \cdot 5 + \frac{1}{6} \cdot 6 = 3.5$. If the die was unfair and, for example, 6 has a probability to appear of $\frac{1}{2}$, and the remaining 5 sides have a probability of $\frac{1}{10}$ then the expected value would be skewed towards high numbers: $\frac{1}{10} \cdot 1 + \frac{1}{10} \cdot 2 + \frac{1}{10} \cdot 3 + \frac{1}{10} \cdot 4 + \frac{1}{10} \cdot 5 + \frac{1}{2} \cdot 6 = 4.5$.

Monte-Carlo integration can be used even if the random variable X_i follows a non-uniform distribution. Let the Probability Density Function (PDF) of X_i be p , then the Monte-Carlo estimator can be written as [48] [35]:

$$\langle F^N \rangle = \frac{1}{N} \cdot \sum_{i=0}^{N-1} \frac{f(X_i)}{p(X_i)}$$

We can prove the formula with an analogous proof to the one above (this time the integration domain is Ω):

1 | Background Theory

$$\begin{aligned}
E[\langle F^N \rangle] &= E\left[\frac{1}{N} \cdot \sum_{i=0}^{N-1} \frac{f(X_i)}{p(X_i)}\right] \\
&= \frac{1}{N} \cdot \sum_{i=0}^{N-1} E\left[\frac{f(X_i)}{p(X_i)}\right] \\
&= \frac{1}{N} \cdot \sum_{i=0}^{N-1} \int_{\Omega} \frac{f(x)}{p(x)} \cdot p(x) dx \\
&= \frac{1}{N} \cdot \sum_{i=0}^{N-1} \int_{\Omega} f(x) dx \\
&= \int_{\Omega} f(x) dx
\end{aligned}$$

Monte-Carlo integration enjoys some important properties that make it suitable to solve many problems concerning integrals, among which the rendering equation.

First, the Monte-Carlo estimator $\langle F^N \rangle$ is consistent, meaning that, as N tends to infinity, the estimator converges to a value.

Moreover, it is also unbiased, therefore the value it converges to is the value of the definite integral of the function Monte-Carlo is estimating [36].

Compared to other integration techniques, such as Riemann sum, Monte-Carlo doesn't suffer from the *curse of dimensionality* [?]. *Curse of dimensionality* means that the complexity of the algorithm to compute the integral grows exponentially as the number of its dimensions increases: $\mathcal{O}(k^d)$. This is particularly relevant in our case, since we are working in a 3-dimensional domain.

We can now calculate the variance and standard deviation of the Monte-Carlo estimator $\sigma[\langle F^N \rangle]$ in order to show the convergence rate of this technique. Here we use the random variable Y as $\frac{f(X)}{p(X)}$:

1 | Background Theory



$$\begin{aligned}
\sigma[\langle F^N \rangle] &= \sqrt{V[\langle F^N \rangle]} \\
&= \sqrt{V\left[\frac{1}{N} \cdot \sum_{i=0}^{N-1} Y_i\right]} \\
&= \sqrt{\frac{1}{N^2} \cdot V\left[\sum_{i=0}^{N-1} Y_i\right]} \\
&= \sqrt{\frac{1}{N^2} \cdot \sum_{i=1}^{N-1} V[Y_i]} \\
&= \sqrt{\frac{1}{N^2} \cdot N \cdot V[Y]} \\
&= \sqrt{\frac{1}{N} \cdot V[Y]} \\
&= \frac{1}{\sqrt{N}} \cdot \sigma[Y]
\end{aligned}$$

In the above proof [47] we assumed that the random variables Y_i are independent to go from line 3 to 4; we also used the result $V[a \cdot X] = a^2 \cdot V[X]$ to go from line 2 to 3.

It is possible to observe that the convergence rate is inversely proportional to the square root of the number of samples \sqrt{N} . This means that to reduce the error by a factor of 2, we would need to increase the samples by a factor of 4. This result is not ideal, and we will analyze other ways to reduce variance without increasing the number of samples in the next section.

Going back to the case of the rendering equation, it is now clear that we can leverage the mathematical theory behind the Monte-Carlo technique to solve it. In this case the integration domain is the hemisphere, and the probe rays cast are the random samples. As the number of probe rays grows, the function describing the incoming light is estimated with ever greater accuracy. However, since the rate of convergence is quadratic, to double the estimation accuracy 4 times more probe rays are needed. The error in the estimate of the incoming light translates into noise in the final rendered image.

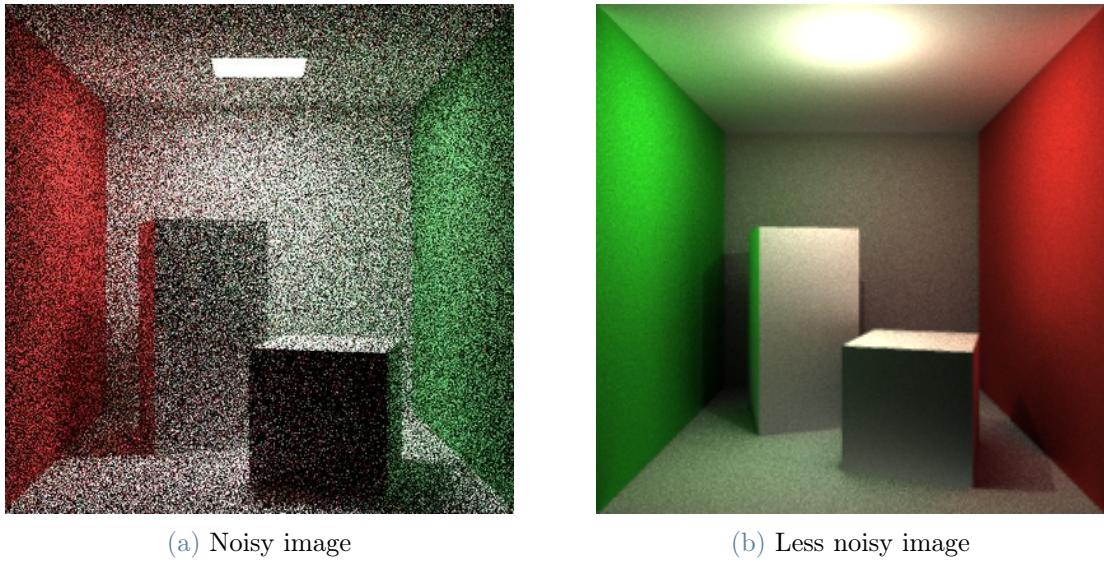


Figure 1.8: Noise in a ray traced image.

Actually, in many ray tracing techniques, just one probe ray is cast, but the estimated light entering a point is temporally accumulated. This works because Monte-Carlo integration is unbiased.

1.2.3. Variance Reduction Techniques

In this section we will describe importance sampling, a variance reduction technique applicable to Monte-Carlo method to reduce the error of the estimate, without increasing the number of samples. This technique is particularly relevant in ray tracing, since we have a very limited number of probe rays we can cast. Eventually, we will show how importance sampling can be applied to the ray tracing context, and how its usage generates artifacts in the ray distribution in the rendered scene.

Variance reduction techniques, as the name suggests, are methods used in the Monte-Carlo context to reduce the variance of the estimator without increasing the computational effort.

One of the variance reduction techniques employed in ray tracing, and that is relevant to our work, is called Importance Sampling (IS). Let's imagine that we want to integrate a constant function. In this case, the positions where we place our samples are irrelevant: each time we run the Monte-Carlo simulation, the estimator will return the same result, therefore the variance is always 0.

The intuitive idea of importance sampling is to try to run the Monte-Carlo simulation

always on a constant function. From a theoretical point of view, this is extremely easy: we simply need to divide the integrand f by a function proportional to it. Given the general Monte-Carlo estimator, this can be achieved by choosing a sampling PDF p proportional to f :

$$\langle F^N \rangle = \frac{1}{N} \cdot \sum_{i=0}^{N-1} \frac{f(X_i)}{p(X_i)} \longrightarrow \frac{1}{N} \cdot \sum_{i=0}^{N-1} \frac{f(X_i)}{k \cdot f(X_i)}$$

From an operative point of view, with importance sampling it is more likely to get a sample from a portion where the integrand has a high value, but, at the same time, this sample will have a lower weight in the final estimate (because it is divided by a higher value). Whereas, if we get a sample from a portion where the integrand has low values, which is rare, its weight will be bigger. This means that the estimate comes from a finer-grained sampling of the portions where the integrand carries most of its information.

Even though from a mathematical point of view this is simple, from a concrete standpoint the technique can be difficult to apply. The main reason is that in many cases, such as in ray tracing, we don't have an analytical form of the integrand function, therefore finding a function proportional to it is problematic. Moreover, a bad choice of the PDF, can lead to an increase in the variance of the estimator, even compared to the uniform PDF.

Going back to the ray tracing context, the integrand function has this equation:

$$\int_{\Omega} BRDF(\bar{x}, \bar{\omega}_i, \bar{\omega}_o) \cdot \cos(\bar{n}, \bar{\omega}_i) \cdot L_i(\bar{x}, \bar{\omega}_i) d\bar{\omega}_i$$

It is possible to note that the function is made up of 3 terms: the $BRDF$, the geometry term and the incoming light term. We can therefore use a probability function proportional to one of the terms to reduce the variance:

Cosine sampling The sampling PDF is proportional to the geometry term of the rendering equation.

BRDF sampling The sampling PDF is proportional to the BRDF. BRDFs can assume complex analytical forms, therefore sampling the BRDF is not always easy.

Light sampling The sampling PDF should be proportional to the L_i term of the rendering equation.

Light sampling is particularly problematic because we almost never have an analytical form of the L_i function. The most used form of light sampling is called next event

estimation (NEE), and is achieved by casting rays towards direct light sources. The way in which the light source to sample is chosen can be completely random or require complex strategies, such as in the case described in this article in the book *Ray Tracing Gems*: [27].

The main issue with next event estimation is that a direct light can be occluded by an object, or, conversely, a strong light could come from an indirect source, such as a reflection. For this reason, another technique, called path guiding [50], has been developed to consider mainly indirect lighting in the sampling PDF. Path guiding can be computationally expensive, therefore is often used in non-real-time scenarios, even though approximations have been proposed even for real-time ray tracing, such as [10].

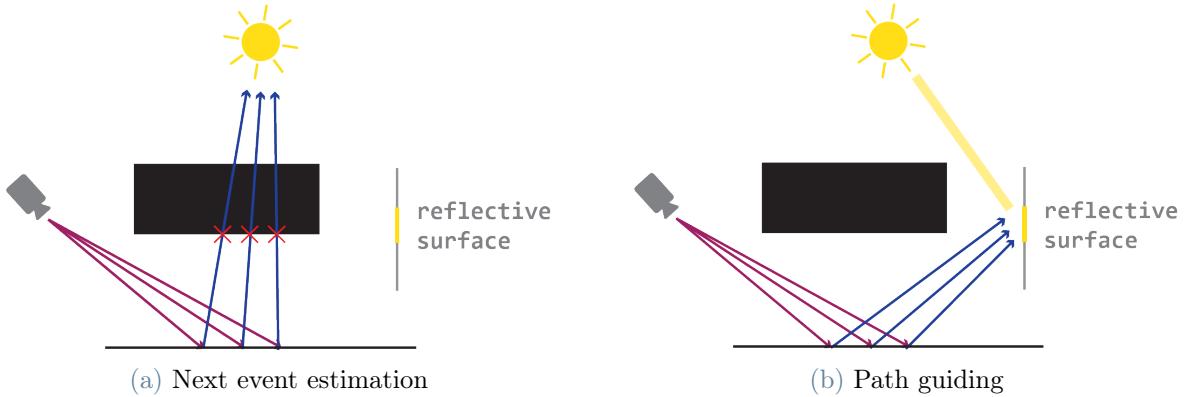


Figure 1.9: With NEE light is directly cast toward direct light sources. With path guiding, indirect illumination is taken into account in order to build a better sampling PDF, at a higher cost.

Using BRDF sampling or light sampling in isolation as variance reduction PDFs can give better or worse results mainly based on the rendered section of the scene. For example, in a well illuminated scene with a mirror, using light sampling gives bad results. This happens because the BRDF of the mirror is null everywhere except for a spike in the perfect reflection direction. This means that its weight in the integrand is much more important than the weight of the L_i term, which is almost constant being the scene well illuminated. If we use light sampling we will cast rays almost uniformly, and many of them will lose all of their energy as soon as they hit the mirror and reflect to a non-perfect-reflection direction.

On the contrary, in a scene with rough materials (such as concrete) and with few distant light sources, using light sampling would be beneficial, since many of the rays would be cast toward a light, instead of toward the void.

To remedy the problem of choosing the right importance sampling technique based on the portion of the scene, a new technique has been developed, called Multiple Importance Sampling (MIS) [45].

The key take away from this section is that with importance sampling we use a non-uniform PDF to sample probe rays from. This creates artifacts in the ray distribution in the scene. In particular, if next event estimation is used, a big part of the rays will tend to go toward light sources. In chapter 2 we will study how these artifacts in the ray distribution can be exploited to design a faster technique to traverse a bounding volume hierarchy, which is the main subject of the next section 1.3.

1.3. Ray Tracing Acceleration Structures

In this section we will describe the first family of software optimizations we talked about in section 1.1.2. These optimizations aim at reducing the time needed to find the intersections between a ray and the geometry of the scene. Usually this is achieved by organizing the scene geometry in a spatial data structure, or by wisely choosing the order in which rays are cast, in order to improve the spatial coherency of the data accesses on the GPU [51]. After a brief description of older acceleration structures, we will focus on the state-of-the-art acceleration data structure, called Bounding Volume Hierarchy (BVH), and present how it is constructed. We will eventually show the assumptions on which the construction algorithm is based, whose refutation will be the main subject of the novel approach we propose in the next chapter 2.

1.3.1. The Need for Acceleration Structures

As we've seen throughout this chapter, in any of the algorithms of the ray tracing family, casting rays is the core of the procedure. Rays are primarily cast from the camera toward the scene, and subsequently, based on the specific algorithm used, they hit objects and bounce off of them. Until now, however, we have overlooked how, in a concrete scenario, the algorithm can detect what is the object hit by the ray.

Before delving into the algorithms that can be used to carry out this task, it is important to understand how an object is described in the world of computer graphics. In the history of computer graphics a lot of techniques to represent an object have been developed. Some examples are volumetric rendering with voxels [19], signed distance fields [9] and implicit functions that work best with the ray marching algorithm [15], or point clouds. By the way, the most common method to represent 3D objects are polygonal meshes.

A polygonal mesh describes an object by defining its surface via points, edges and faces (usually triangles). Meshes's success derived from the fact that they were one of the first representations proposed in the world of computer graphics. This led to a big part of the research converging to this specific representation form. For example, GPUs are specialized for the rendering of triangular meshes via rasterization, even though in the last few years their architecture has been made more flexible.

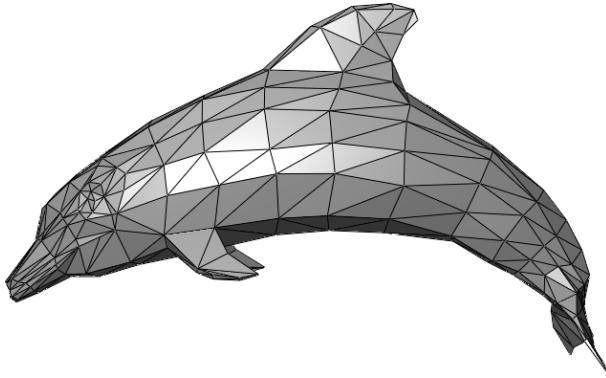


Figure 1.10: A triangular mesh.

Meshes are also handy to create for graphic artists, and can be easily modeled to represent many kinds of objects, even though they are not suited to represent liquids or gases, such as clouds. Meshes, since they describe only the outside surface of an object, can prove difficult to use in applications where the interior is important, such as medical visualizations or games where objects can be dynamically destroyed. In these contexts it is usually preferred to employ voxels. Moreover meshes are a discrete representation, meaning that, once created, they cannot be used to visualize the details of the object at an arbitrary resolution, contrary to implicit functions. However, techniques to change the resolution based on the dynamic situation have been developed, such as LODing, tessellation [8] or more recent methods such as Nanite [17].

Regardless of their advantages and disadvantages, triangular meshes are by far the most used representation, therefore in this work we decided to focus on them in the scenario where the rendering technique is an algorithm of the ray tracing family.

In order to find the intersection between a ray and a list of meshes making up a scene, we need to test the intersections between the ray and all the triangles making up the meshes, and keep the closest one².

²In some scenarios, such as when we just want to check for occlusion, it is not necessary to find the closest intersection, but just to find an intersection. In this case some optimizations can be developed, but the core concepts remain the same.

The naive way of detecting the intersection is the brute-force solution of iterating over all of the triangles and storing the closest intersection found so far. The algorithm we used to detect the intersection between a ray and a triangle in our implementation can be found in appendix A.3. Despite working, this method cannot be practically used either in real-time scenarios, nor in non-real-time ones. The problem resides in the fact that there are too many triangles to test against. Even in a non-real-time case this approach would be too expensive. Indeed, even if the time budget for a frame is big, meshes created for non-real-time purposes usually have a way higher triangle resolution, and also way more rays to trace. From a complexity point of view, tracing a scene with n triangles and m rays in this way has complexity $\mathcal{O}(m \cdot n)$.

This is the reason why acceleration structures have been developed. Most of the acceleration structures organize the triangles in a hierarchical fashion, so that it is possible to exclude a big chunk of them if a ray doesn't hit certain parts of the scene. One very simple example of acceleration structure would be to divide the scene into 2 parts and keep track of what triangles reside on each one. Now, if a ray doesn't hit one of the 2 parts, it is possible to exclude all of the triangles contained in that region, without having to test them one by one.

Acceleration structures can be divided into 2 categories:

Space partitioning The space is recursively subdivided into disjoint regions. Objects (triangles) can potentially appear in more than one region, if they are overlapping.

Object partitioning Objects are subdivided into disjoint sets, enclosed into spatial regions. The regions can potentially be overlapping.

Historically, initially the first family of acceleration structures has been used, but todays state-of-the-art acceleration structure is the BVH, part of the second family. We will now very briefly list some of the most famous acceleration structures historically used for ray tracing, and, in the next section, we'll diffusely talk about the BVH.

The first space partitioning acceleration structure developed is the uniform grid. In this structure the 3-dimensional space of the scene is subdivided into static fixed-sized cells in a regular pattern. Then, each triangle is assigned to all the cells it at least partially covers. This can be at least a single cell, to, potentially all the cells of the scene. When a ray is traced, first, a cell it intersects is found, and then, all the triangles inside this cell are tested against the ray. This is repeated for all the cells the ray hits in its path. If an intersection P is found, then, all the cells that are at a further distance from the ray origin than P can be discarded without testing individual triangles.

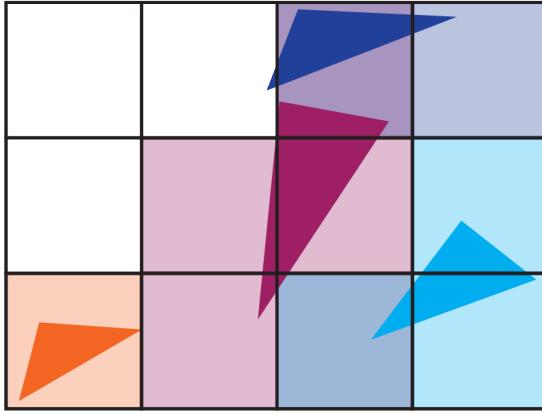


Figure 1.11: The uniform grid in 2D.

The uniform grid, while being an improvement over the brute-force approach, still lacks adaptability. Based on how the scene is composed, it is possible for some cells to be empty and others overcrowded. Moreover, as in any space partitioning data structure, it is possible that a triangle is tested more than once if it overlaps more than one cell. This could be avoided by using some intersection caching techniques, such as mailboxing [21].

In order to solve the adaptability problem of the uniform grid, octrees started being used. An octree is another 3D spatial data structure, and it is a tree where each node has 8 children. To subdivide a node into 8 children, it is divided by 3 planes, one perpendicular to each dimension, passing from the center of the parent node. This process is recursively executed until a specified amount of objects (triangles) remains inside a single node, or until the depth of the tree gets past a threshold. Since the octree is a complete tree, and the size and center of each child can be automatically computed starting from the size of the root and the level³, it is possible to store the octree in an implicit array structure, making it more efficient both from an access pattern and space point of view.

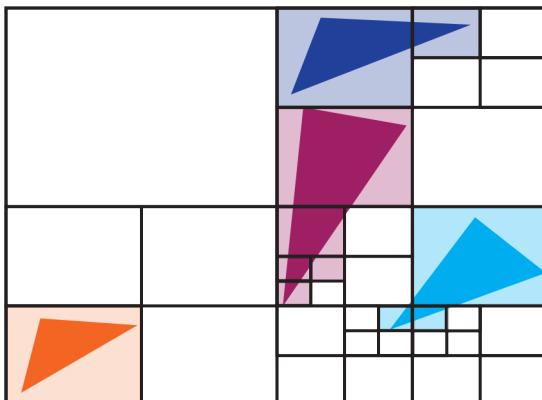


Figure 1.12: A 2D octree (quadtree).

³ $size_{children} = size_{root}/2^{level}$ given that the root has level 0.

1 | Background Theory

It is important to note how octrees adapt better to the scene. Indeed, compared to fixed grids, they alleviate the *teapot in the stadium* problem, namely the issue that appears when there is a scene with different densities of objects. For example, precisely what happens when we have a small but high-resolution object (a teapot), inside a big and mostly empty space (a stadium). With octrees this problem is handled by having few big and empty cells in the regions where there is a low density of triangles, and many small cells where there are high-resolution objects. In this way less memory is wasted, and even traversal is improved. Indeed, if a ray passes through empty space, just few cells must be checked, instead of a bigger amount as in the case of a uniform structure like the fixed grid. Conversely, in case a ray goes through a dense region, fewer triangle tests will be carried out, because the cells the ray intersects are smaller and not overcrowded. Another advantage of octrees is that they are a tree data structure. This implies that if a node is not hit by a ray, all its children can be immediately discarded.

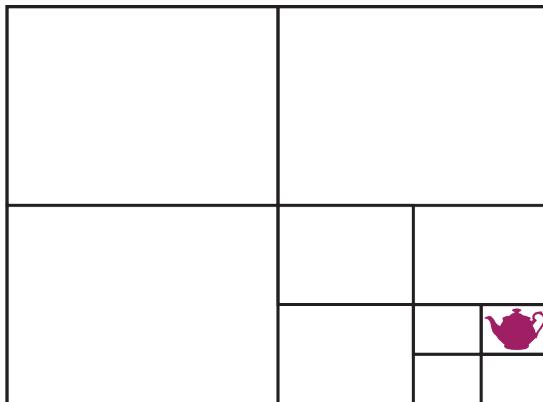


Figure 1.13: How an octree adapts to a *teapot in the stadium* kind of scene.

Another structure that is a generalization of octrees and delivers similar performance is the kd-tree. In 3D kd-trees each node is split into 2 children by a plane. At each level the plane is perpendicular to one of the dimensions, in a round-robin fashion (for example, at level 0 it is normal to x , at level 1 to y , at level 2 to z and so on). Differently from octrees, the plane doesn't necessarily have to pass through the center of the parent node, making their construction even more flexible.

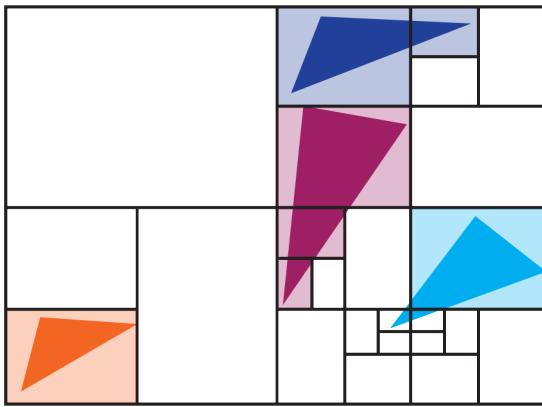


Figure 1.14: A kd-tree in 2D.

These are the most prominent spatial data structures used for ray tracing. However, nowadays an object partition structure is used, called bounding volume hierarchy, which we will analyze in the next section.

1.3.2. The Bounding Volume Hierarchy

A BVH [52] is a binary tree⁴ where each node wraps some of the triangles in the scene in a bounding volume. Given a node A wrapping the triangles in the set T_A , then the two children of A , called B and C , wrap the triangles in the sets T_B and T_C such that $T_B \cup T_C = T_A$. Thanks to this structure, if a ray doesn't intersect a bounding volume, we deduce that it will not intersect any of the triangles contained in the bounding volume too, making it possible to discard them without having to perform any additional intersection test.

Assuming the best-case scenario where a ray always hits only one of the 2 children of a given node during traversal, if the BVH is balanced and has one triangle per leaf, then the complexity of finding an intersection between a ray and the scene is $\mathcal{O}(\log_2(n))$, where n is the number of triangles. This is not always the case, since it often happens that a ray hits both the children of a given node. One of the heuristics that we propose in this thesis in chapter ??, aims at reducing the number of this expensive situation, by building the BVH in a smart way.

The bounding volume in which a BVH encloses triangles should have a form that is cheap to intersect with a ray, and also cheap to build given a set of triangles.

A choice can be a sphere, which is easy to test against a ray. Moreover, building the tightest enclosing sphere given a set of triangles is as simple as finding the 2 triangles

⁴Technically it can be a tree with any breadth, but binary trees are the most common ones.



furthest away in each direction, computing the middle point and setting as radius the distance between the furthest triangle and the middle point. The negative aspect of using bounding spheres lies in the fact that a sphere extends equally in all directions in 3D space. This means that if the triangles are displaced in a way where they extend more in one direction than in other ones, the bounding sphere will present a lot of slack space, and won't enclose the triangles tightly. This is not ideal, because in the traversal phase a non-tight bounding volume generates a lot of false positives, namely situations where the bounding volume is hit but no triangle is intersected.

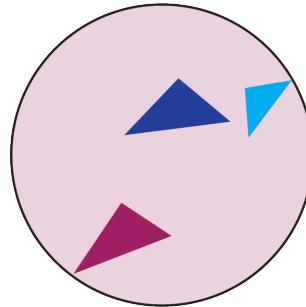


Figure 1.15: Bounding circle (2D bounding sphere) can present large slack spaces.

At the opposite side of the spectrum there are polygonal bounding volumes. In this case they are able to tightly enclose the triangles, but building them is not easy, and intersecting them can prove as expensive as intersecting the set of triangles itself.

Another option is using Oriented Bounding Boxes (OBB). Oriented bounding boxes in 3D are rectangle parallelepipeds. This means that they can have a different extension in 3 different arbitrary directions (each one perpendicular to the remaining two), making them better at tightly enclosing any distribution of triangles compared to spheres. However, computing a tightly enclosing OBB starting from the set of triangles involves using the principal component analysis [53], which can be too expensive in real-time scenarios. Moreover, intersecting a ray against an OBB involves a rotation transformation, which, again, can be too slow, especially during the traversal phase. In general OBBs are too slow in many scenarios, but, given their ability to tightly enclose triangles, can be used along with AABBS [55].

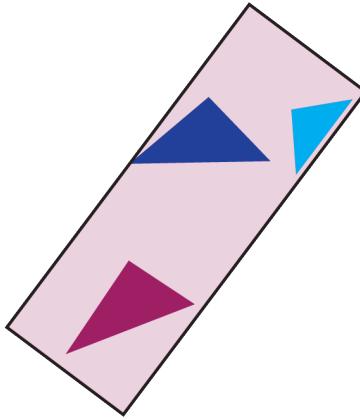


Figure 1.16: A 2D OBB.

Axis-Aligned Bounding Boxes (AABB) are the most used bounding volumes. AABBs are OBBs that have no rotation, and are always aligned with the cartesian axes. This means that AABBs can indeed have a different extension in 3 directions, but the directions are limited to the ones of the cartesian axes. These limitations make them worse at tightly enclosing triangles than OBBs, but at the same time make it way easier to build them and intersect them with a ray. To build an AABB starting from a set of triangles, it suffices to iterate over all the triangles and keep track of the minimum and maximum point in all 3 dimensions. An AABB is then fully described by its minimum and maximum point. The algorithm to test whether a ray intersects an AABB is fast and can be implemented in a GPU-friendly branchless way, as described in appendix A.1.

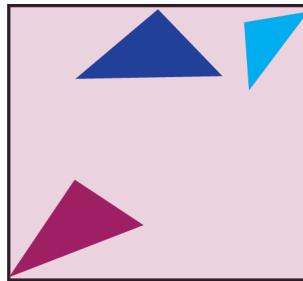


Figure 1.17: A 2D AABB.

AABBs are the most used bounding volumes both in real-time and non-real-time ray tracing. For this reason, from now on, when we refer to a BVH we will always consider the case of a BVH based on AABBs.

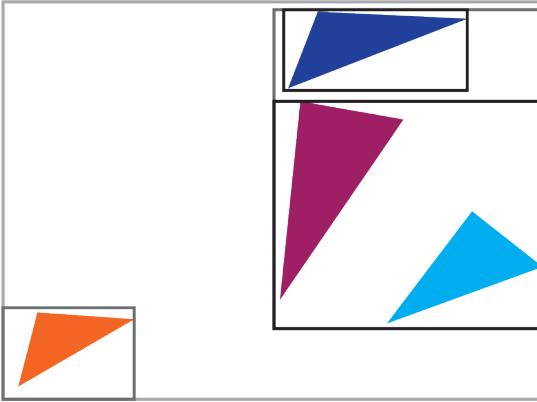


Figure 1.18: A 2-dimensional BVH.

Despite it is not clear that BVHs' raw performance is better than kd-trees', especially on large scenes [49], they became the state-of-the-art acceleration structure for many practical reasons.

First of all, being an object partitioning data structure, they can solve the *teapot in a stadium* problem even better than kd-trees. This is because it is possible to completely cut off empty space, as it is possible to observe in figure 1.18.

Moreover, and this is especially relevant in a dynamic real-time scenario such as a videogame, BVHs are easier to update when the geometry of the scene changes [22].

BVHs construction algorithm is also simpler than kd-trees', allowing a faster build-time, which can be crucial in highly dynamic real-time applications. And they also have a smaller memory footprint compared to kd-trees, as mentioned in [44] and [54].

1.3.3. BVH Construction

Building the optimal BVH according to some metric is an NP-complete problem. In order to do so, the algorithm would have to build all the possible BVHs given a set of triangles, compute the metric and keep the best one. The number of possible different binary trees grows factorially with the number of leaves. Given n leaves, the number of possible binary trees is the $n - 1$ Catalan number [38]: $C_n = \frac{(2n)!}{(n+1)! \cdot n!}$. The number of possible BVHs is even bigger, because, since a leaf can contain more than one triangle, it is not a given that the number of leaves corresponds to the number of triangles.

Given that building the optimal BVH is not feasible even in a non-real-time scenario, research started studying methods to build a BVH with an acceptable quality faster. This resulted in the use of greedy algorithms and heuristics.

The base algorithm for building a BVH can be summarized in these steps:

Algorithm 1.1 Summarized BVH construction algorithm.

```

1: function BUILDBVH(triangles)
2:   leftNode  $\leftarrow$  []
3:   rightNode  $\leftarrow$  []
4:   cost  $\leftarrow \infty$ 
5:   loop cost  $<$  threshold or all possible splits attempted
6:     [leftNodeTmp, rightNodeTmp]  $\leftarrow$  SplitTriangles(triangles, howToSplit)
7:     costTmp  $\leftarrow$  ComputeCost(leftNodeTmp, rightNodeTmp)
8:     if costTmp  $<$  cost then
9:       leftNode  $\leftarrow$  leftNodeTmp
10:      rightNode  $\leftarrow$  rightNodeTmp
11:      cost  $\leftarrow$  costTmp
12:    if not StopCriterion(cost, leftNode, rightNode) then
13:      BuildBvh(leftNode.triangles)
14:      BuildBvh(rightNode.triangles)
  
```

The proposed algorithm is a greedy one. This means that it tries to minimize the cost function locally, at each level of the BVH, but when a decision is taken, it is not possible to change it. Making the local optimal choice doesn't imply that the global minimum of the cost function will be found, but at least it makes the problem tractable.

The algorithm can be divided into 3 steps: triangles splitting, cost computation and stopping criterion.

Stopping Criterion

The stopping criterion phase decides whether the *BuildBvh* function should make the recursive call and split the new children nodes. The stopping criterion can be based on different metrics, among which:

- Children cost
- Total triangles in children
- BVH level

Triangles Splitting

Given the set of triangles of the parent node, in this phase the algorithm tries to split them into two subsets and build the enclosing bounding volumes.

In order to split the triangles into two subsets, a plane is chosen, and based on which side of the plane the barycenter of each triangle is, the triangle is assigned to one of the two subsets. This means that it is possible that a triangle has some vertices in the opposite half-space compared to the one it has been assigned to. This is one of the reasons why two children can have some of their volumes overlapping.

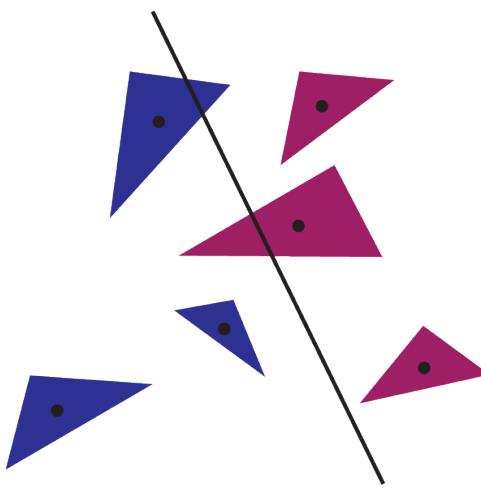


Figure 1.19: Triangles splitting via arbitrary plane.

As shown by [1] too much overlapping between children nodes leads to a decrease in the quality of the final BVH. For this reason, in chapter ?? we propose a novel technique trying to take into account the overlapping during this phase of BVH construction.

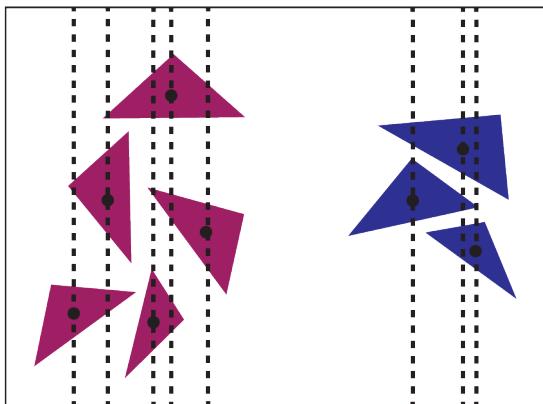
In order to choose which plane to use to split the triangles, some optimizations can be employed. In BVHs based on AABBs it is useful to try to split the triangles only in the 3 cartesian directions. Usually just one of the directions is chosen, in order to further optimize the process. The way in which the direction is chosen can vary based on the methods used. An often employed technique is to cut along the longest dimension of the parent AABB. Another one is to cut in a round-robin fashion, similar to what is done in kd-trees. In chapter ?? we propose a novel method, as mentioned above.

Another advantage of using only the 3 cartesian directions is that, in order to detect the half-space a point falls into, it is sufficient to compare one of its coordinates with the coordinates of a point the plane is passing through. This is an optimization compared to computing the cross-product, like in the case where the plane has an arbitrary normal

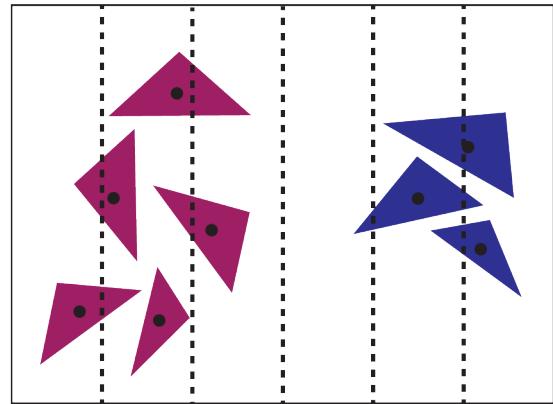
direction.

After choosing a splitting direction, all possible cuts in that direction should be tested. Given an amount on n triangles to be split, trying all the possible cuts in one direction means trying $n - 1$ subdivisions, where the splitting plane is placed each time on the barycenter of a triangle. Since n can be a big number, especially while running the algorithm on the first levels of the BVH, where nodes are big, a technique called binning can be introduced.

With binning, instead of trying every possible cut, a predefined number of cuts is attempted. The splitting plane is translated each time by a fixed length (a bin) along the chosen cutting direction from the position of the previous cut. By using binning it is possible to miss some of the possible cuts, especially in a region dense of triangles. However, the most relevant cuts are often positioned where there is a big gap in one direction between two consecutive triangles. This is because it is possible to leverage the big gap to cut off as much space as possible. Whereas, if the gap is small, it is likely that the resulting children nodes' AABBs overlap. For this reason binning is an efficient approximation, especially in the higher levels of the BVH.



(a) Triangles splitting without binning



(b) Triangles splitting with binning

Figure 1.20: With binning it is possible to lose some cuts, but the most relevant ones are always found, for example the cut dividing the blue and purple triangles.

After generating the two subsets of triangles, the enclosing AABBs must be built. The process to build an AABB from a set of triangles is straight-forward, and is summarized by this algorithm:

Algorithm 1.2 AABB building from triangles set.

```

1: function BUILDAABB(triangles)
2:   min  $\leftarrow (\infty, \infty, \infty)$ 
3:   max  $\leftarrow (-\infty, -\infty, -\infty)$ 
4:   for all t  $\in$  triangles do            $\triangleright t.v_k$  is the kth vertex of the triangle t
5:     min.x  $\leftarrow \text{Min}(\text{min.x}, t.v_0.x, t.v_1, t.v_2.x)$ 
6:     max.x  $\leftarrow \text{Max}(\text{max.x}, t.v_0.x, t.v_1, t.v_2.x)$ 
7:     min.y  $\leftarrow \text{Min}(\text{min.y}, t.v_0.y, t.v_1, t.v_2.y)$ 
8:     max.y  $\leftarrow \text{Max}(\text{max.y}, t.v_0.y, t.v_1, t.v_2.y)$ 
9:     min.z  $\leftarrow \text{Min}(\text{min.z}, t.v_0.z, t.v_1, t.v_2.z)$ 
10:    max.z  $\leftarrow \text{Max}(\text{max.z}, t.v_0.z, t.v_1, t.v_2.z)$ 
11:   return Aabb(min, max)
  
```

Cost Computation - Surface Area Heuristic

In the previous section we have described how a standard BVH construction algorithm can split the triangles of a node into two subsets, and compute the AABB for each one of them. In this section we will analyze how the algorithm decides whether a split is better than another one.

To choose the best split, a BVH construction algorithm sorts the splits proposed by the splitting triangles phase, by assigning to each one a value through a cost function.

The aim of a cost function is to accurately predict how *good* the final BVH will be. The concept of *goodness* or *quality* of a BVH is directly related to how fast an arbitrary ray is able to traverse it to find the first intersection with a triangle. It is possible to evaluate the cost function for every node as if it was a leaf node, and this is useful while building the BVH with the greedy algorithm. It is also possible to compute the cost function of the overall BVH, which is used to evaluate the quality of the BVH built.

The most used cost function is called Surface Area Heuristic (SAH) [26], and is based on a very simple idea: the smaller the surface area of a node's AABB, the less likely is for this node to be hit by a ray. If a node is hit less times by a ray, it means that fewer checks are needed to traverse the BVH, therefore its quality, as we defined it above, is higher.

Given an AABB, it is trivial to compute its surface area as:

$$A = (max_x - min_x) \cdot (max_y - min_y) + (max_x - min_x) \cdot (max_z - min_z) + (max_y - min_y) \cdot (max_z - min_z)$$

1 | Background Theory

Now, we would like to have a cost function that is agnostic to the absolute size of the scene. For this reason, instead of using directly the surface area to compute the cost metric, we transform it into the probability that a ray hits a specific node.

To do so SAH makes some reasonable assumptions on some properties of the rays in the scene:

- A ray always hits the AABB enclosing the whole scene;
- All rays have origin outside the AABB enclosing the whole scene, and their positions are uniformly distributed;
- Rays never intersect any primitive, and are infinitely long;
- Rays are uniformly distributed in their direction space.

With these assumptions in place, which we'll analyze better in section 2.1, we can define the probability a ray hits a node K as:

$$p(\text{ray hits } K) = \frac{\text{Area}_K}{\text{Area}_{\text{root}}}$$

Since all rays hit the root AABB, we can interpret it as the certain event. And since the rays' directions are uniformly distributed, the surface area can be mathematically interpreted as a measure of how likely it is for a ray to hit an AABB.

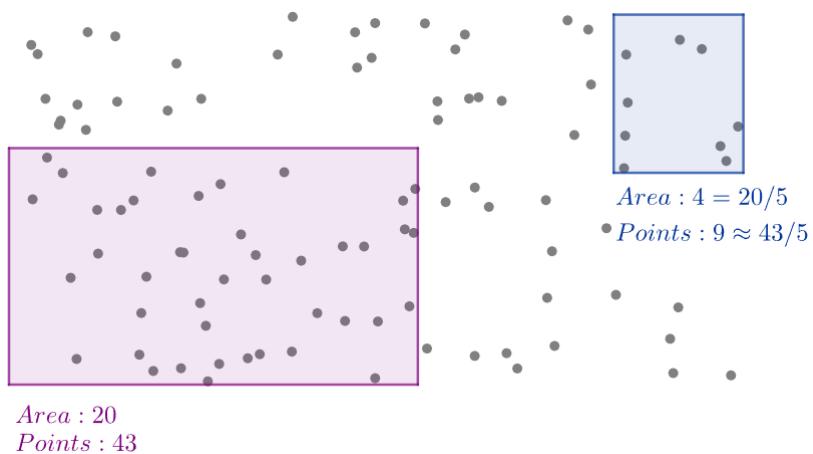


Figure 1.21: Visual relationship between AABB area and hit probability in 2D, under the assumption rays are uniformly distributed.

Finally, in order to compute the cost function, we have to also take into account the cost of the intersection test between a ray and a triangle, and a ray and an AABB. In literature a cost of 1.2 is assigned to the first test, and a cost of 1 to the second one.

1 | Background Theory



We now have all the elements to define the SAH cost function for a node K :

$$Cost_{SAH}(K) = \frac{Area_K}{Area_{root}} \cdot \#triangles_K \cdot 1.2$$

The $\#triangles_K \cdot 1.2$ part tells us that when this node is hit by a ray, a ray-intersection test per triangle must be carried out. The term $\frac{Area_K}{Area_{root}}$ weights the cost of computing the ray-intersection tests with the probability that this node is actually hit. This means that, even if a node has a lot of triangles in it, if these triangles are all packed in a small region, the cost of the node will not be too high, because it is unlikely for a ray to hit the node in the first place.

It is important to note that this formula returns the cost of a node in isolation. But the nodes we are dealing with are part of a BVH. In other words, when an internal node of a BVH is hit, this doesn't trigger a ray-triangle test for all of its triangles (as suggested by the formula), but only recursively triggers ray-AABB tests against its two children. Only when this formula is used on leaf nodes, the returned cost actually reflects the real cost of a ray intersecting the node.

Therefore why are we suggesting to use this formula? The answer is that we are building the BVH with a greedy algorithm. When we have to evaluate a node, we have no way of knowing how its children will be, therefore the only way is to estimate its cost as if it didn't have any children at all, as if it was a leaf node.

However it is also useful to have a way to evaluate the cost of a BVH a-posteriori. In this case, based on the node type, we can have two situations:

Leaf node The cost of a leaf node can be computed with the same formula we have analyzed, for the reasons we have stated above.

Internal node When an internal node is hit, the next step of the traversal is to perform a ray-AABB test with each one of the two children. Therefore the cost is:

$$Cost_{SAH}(K_{intern}) = \frac{Area_K}{Area_{root}} \cdot 2 \cdot 1.$$

By merging the formulae to compute the cost of leaf and internal nodes, the cost function of a BVH assumes this form:

$$Cost_{SAH}(BVH) = \sum_{L \in \text{leaf nodes}} \frac{Area_L}{Area_{root}} \cdot \#triangles_L \cdot 1.2 + \sum_{I \in \text{internal nodes}} \frac{Area_I}{Area_{root}} \cdot 2 \cdot 1$$

2 | Projected Area Heuristic

In chapter 1 we described how ray tracing works. In particular, in section 1.2 we have seen how importance sampling generates artifacts in the ray distribution of the scene. In section 1.3 we described how it is possible to build an acceleration structure to speed up the ray-scene intersection process, and we have shown how the surface area heuristic is the state-of-the-art method to build high-quality BVHs. In this chapter, we will show that some of the hypotheses of the SAH are not satisfied in a real-world scenario, due to the use of importance sampling. We will eventually propose a new heuristic, called Projected Area Heuristic (PAH), and show the situations where it can be used to build BVHs. In this chapter we will describe the problem and the solutions from a theoretical point of view; all the implementation details will be thoroughly discussed in chapter 5.

2.1. SAH Hypotheses Fall

In section 1.3 we have seen that the surface area heuristic, used to build high-quality BVHs with the described greedy algorithm, works under these hypotheses:

- A ray always hits the AABB enclosing the whole scene;
- All rays have origin outside the AABB enclosing the whole scene, and their positions are uniformly distributed;
- Rays never intersect any primitive, and are infinitely long;
- Rays are uniformly distributed in their direction space.

In a standard situation, the first hypothesis can be considered true for all the rays. It is still possible, in some scenarios, that a part of the rays doesn't hit the scene. Examples can be found in the discussion on the test cases for our thesis in chapter 6, however, in most scenarios, all the rays hit the scene, because casting a ray away from the scene would be wasteful.

For what concerns the second hypothesis, the situation is completely different. Indeed, in many applications the camera is inside the scene, therefore the rays' origin is inside

2| Projected Area Heuristic



the scene too. But even in a situation where the camera is placed outside the scene, still the majority of rays would have origin inside. This happens because in many algorithms of the ray tracing family, the primary rays (rays cast from the camera position) are in a much smaller number than the secondary rays. With secondary rays we refer to what we called in the previous chapter *probe rays*, namely those rays originating after a ray-triangle intersection. As we have seen, generating probe rays to evaluate the rendering equation is a recursive process, and in order to obtain an image with an acceptable amount of noise, a lot of probe rays must be cast.

Studies about this issue with the surface area heuristic have been carried out. For example, in this article [11], Fabianowski et al. divide the scene into discrete regions, and try to evaluate the hit probabilities from the center of each one.

The third and fourth hypotheses have both to do with how the rays are distributed in the 3D space of the scene. It is easy to see how the third hypothesis doesn't hold true in a real-world scenario. Of course primitives will block rays, thus potentially creating regions of the scene where few or no rays are present (imagine the region of space inside an opaque box, or even more in general, the space inside the boundaries of meshes).

For what concerns the last hypothesis, we have to remember how Monte-Carlo integration works. In theory, after a ray hits an object, there is an equal probability it bounces toward any direction in the hemisphere. This behavior, considering that the other hypotheses hold true, would create a uniform distribution of rays in the scene. However, as we have seen in section 1.2, importance sampling makes it so that the probe rays distribution is not uniform. In particular, if direct light sampling is used, a lot of probe rays will tend to go toward light sources. This artifact in the ray distribution shows how not even the last SAH hypothesis holds true.

2| Projected Area Heuristic

41

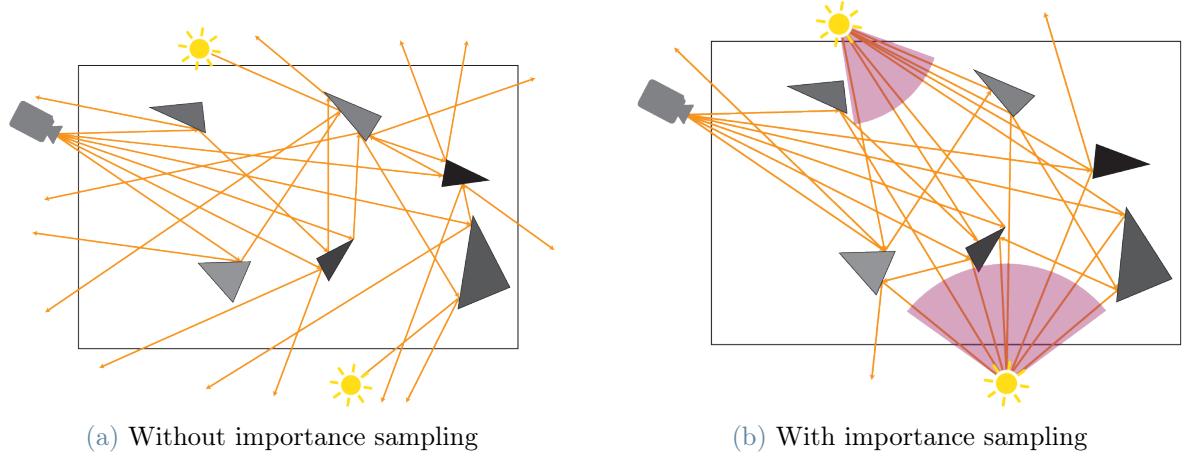


Figure 2.1: With importance sampling rays are not distributed uniformly in the scene, but some artifacts naturally arise.

Since rays are not distributed uniformly in the scene, it means that we cannot interpret the surface area of the AABB as related to the probability a ray hits it (figure 1.21). For example, if an almost flat, but long AABB is present in a region where rays are parallel to the longest side of the AABB, the probability one of the ray hits it is very low. This wouldn't be reflected by the surface area of the AABB, which would assume a big value because of its long side. This can be visualized in the image below (2.2).

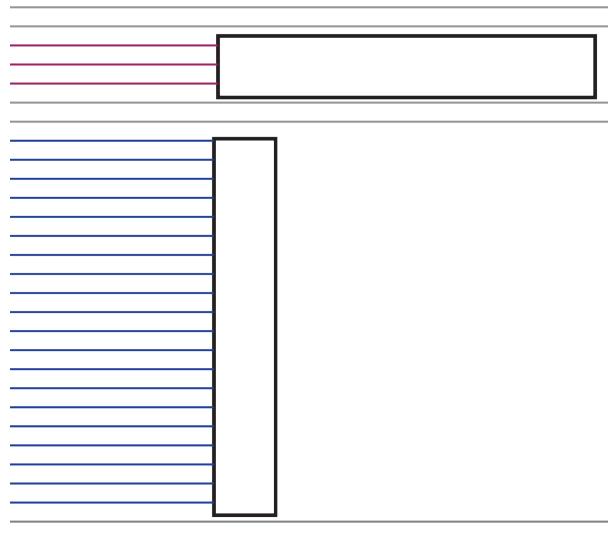


Figure 2.2: The two AABBs have the same hit probability according to SAH, but one of the two gets hit many more times in a region of parallel rays.

The first novel contribution of this work is based exactly on this simple observation, and on the refutation of the last hypothesis of surface area heuristic. In the next sections we

will describe two relevant ray distributions that organically form in the scenes, due to the use of direct light importance sampling.

In the past other works on this same topic have been written, but the proposed solution was based on a very different approach. For example, in this paper [14] Yan Gu et al. proposed to modify an already existing BVH based on some statistics harvested during the previous frames. In summary, if a node is rarely hit, its subtree is collapsed, and all the triangles in the subtree are directly placed in the node, which becomes a leaf. Conversely, frequently hit leaves become internal nodes, and are further divided.

In our approach, instead, we try to directly build a BVH aware of the ray distribution. To achieve this we propose a new heuristic to calculate the probability that a node is hit by a ray: the Projected Area Heuristic (PAH).

2.2. Parallel Ray Distribution

One of the relevant ray distributions that can naturally arise in a ray traced scene, where light importance sampling is used, is the distribution where rays are all parallel among themselves. This ray distribution usually arises in proximity of planar area light sources.



Figure 2.3: Parallel rays distributions arise in proximity of planar area light sources.

In a 3-dimensional region with such a ray distribution, we can estimate the probability an AABB is hit by a ray by computing the area projected by the AABB on the plane having a normal vector parallel to the rays.

Let us first analyze the case where the rays are parallel to one of the 3 cartesian axis, let's say z . In this case, the extension of an AABB in the z direction is irrelevant, because it is impossible for a ray to hit the box on any face other than the xy ones. Therefore, only the area of the xy face is related to the probability that the AABB is hit.

We can generalize this idea to the case where the rays are parallel to an arbitrary direction by orthographically projecting the AABB on the plane normal to the rays. The area of the orthographic projection is directly related to the probability a ray hits the AABB.

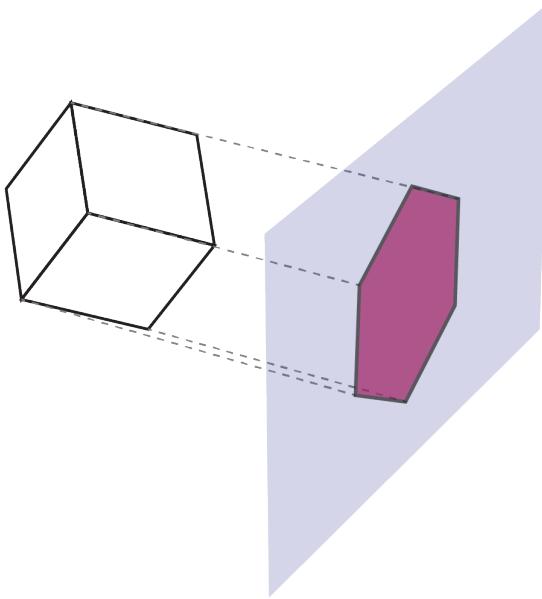


Figure 2.4: Orthographic projection of an AABB.

The details of our implementation will be extensively discussed in section 5.

2.3. Point Ray Distribution

Another ray distribution commonly found in proximity of point lights is what we call the point ray distribution. In this distribution, rays all pass through the same point in 3D space, creating a radial pattern over a solid angle, as it is shown in the figure below.

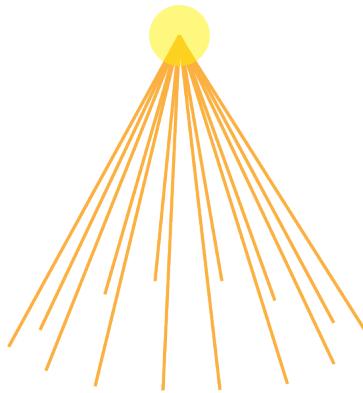


Figure 2.5: Point rays distributions arise in proximity of point light sources.

Also in this case we can observe a relationship between the probability an AABB in this region is hit by a ray, and the area of the projection of said AABB. However, this time, the projection to estimate the hit probability is the perspective projection. The point all the

2| Projected Area Heuristic



rays in this region pass through can be interpreted as the focal point of the perspective. The projection plane can be placed at any distance from the focal point. Indeed, while it is true that by placing the plane at different positions we would get projections of different absolute sizes, it is also true that what we are really interested in is the relative area among the AABBs projected on the same plane. Indeed, as it is possible to see from the formula in section 1.3.3 that we will report here:

$$p(\text{ray hits } K) = \frac{\text{Area}_K}{\text{Area}_{\text{root}}}$$

The hit probability p is computed as the ratio between two areas.

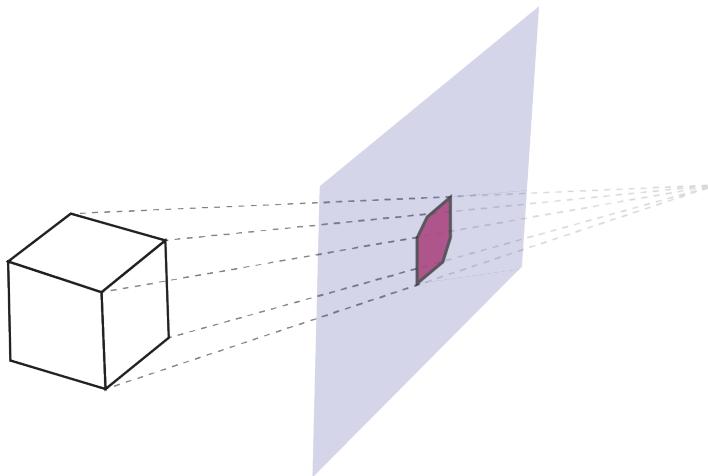


Figure 2.6: Perspective projection of an AABB.

2.4. The Projected Area Heuristic

In the last two sections (2.2 and 2.3) we have seen how computing the projected area where there are certain ray distributions can be a better estimate for the probability that a ray hits an AABB, compared to the surface area. In simple words, this happens because some parts of the AABB are less likely (if at all) to be hit by a ray depending on their orientation, compared to the rays present in that 3D region.

With this notion, it is now easy to define a heuristic to be used during BVH construction and another one for BVH evaluation. Indeed we can use the exact same formulae used for the surface area heuristic, but with the projected area in place of the surface area of the AABBs:

$$\text{Cost}_{PAH}(K) = \frac{\text{ProjArea}_K}{\text{ProjArea}_{\text{root}}} \cdot \#\text{triangles}_K \cdot 1.2$$

2| Projected Area Heuristic



$$Cost_{PAH}(BVH) = \sum_{L \in leaf\ nodes} \frac{ProjArea_L}{ProjArea_{root}} \cdot \#triangles_L \cdot 1.2 + \sum_{I \in internal\ nodes} \frac{ProjArea_I}{ProjArea_{root}} \cdot 2 \cdot 1$$

With this heuristic it is possible to build a BVH for a portion of the scene where there is a certain ray distribution. If there is a parallel ray distribution, the orthographic projected area will be used. Instead, if there is a point ray distribution, perspective projection should be used.

However, in a scene, there likely is more than a single and global ray distribution. It is possible that ray distributions are localized in definite regions of the scene. This means that, rather than building a single and global BVH valid for all the scene, it is necessary to build partial and localized BVHs for specific regions. This is the main topic of the next chapter 3.

2| Projected Area Heuristic

46

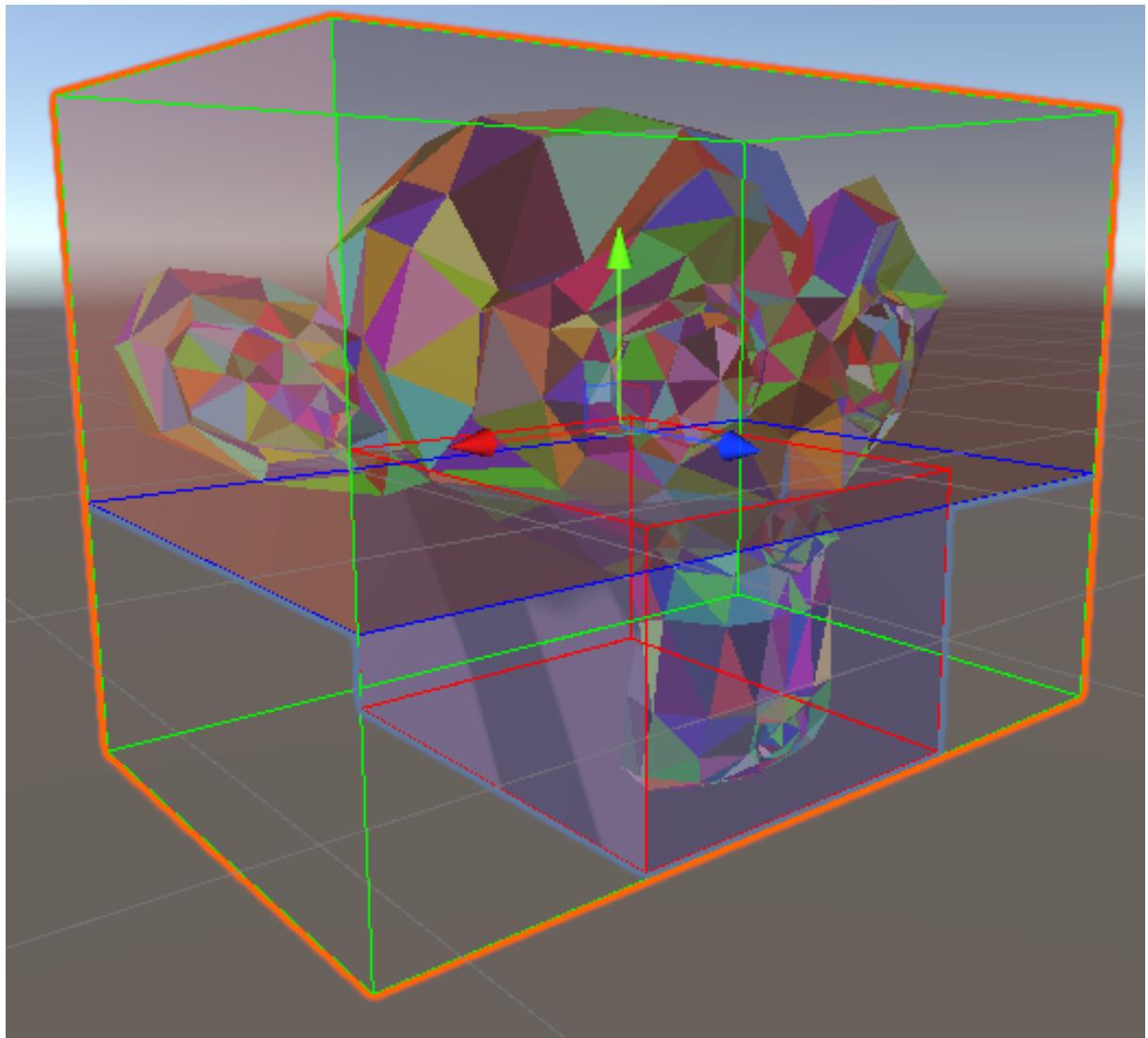


Figure 2.7: A BVH seen in our Unity visualizer. The father node is green, and its two children are red and blue.

3 | Multiple Influence Areas and the Top Level Structure

In this chapter we will show how it is possible to deal with a scene where there are multiple localized ray distributions. We will first introduce the problem in detail and with visual examples, then we will define the concept of ray affinity to region where a relevant ray distribution is present. Eventually we will show how a scene with multiple ray distributions can be traversed by a ray. In this chapter we won't explain all the implementation details of the solution we propose, that will instead be discussed in section 5.5.

As we have touched on in the last chapter, while ray tracing a scene, it is likely that more than a single relevant ray distribution appears. It is even more likely that, even if only one relevant distribution is present, it doesn't cover the entirety of the scene. For cases like these, it is necessary to be able to recognize whether a ray is affine to a certain ray distribution present in a scene. In case the ray is affine, we can use the BVH built with PAH on that part of the scene to find the first intersection. In case it is not affine, we must fall back to a global BVH, covering the whole scene. The definition of the term *affine* is given in the next section (3.1).

From this point on, we will refer to a 3D region of a scene where a relevant ray distribution is present as influence area. If the ray distribution is parallel (section 2.2), we call the region *plane influence area*. If the ray distribution is radial (section 2.3), the influence area is qualified as *point influence area*. Each influence area has an associated BVH (also called local BVH) built with PAH, where the projection method (orthographic or perspective) is the corresponding one to the ray distribution of the influence area.

Influence areas must first be localized in the scene. To do so, we decided to encode a plane influence area as an oriented bounding box (section 5.3.1). The rays start from one of the faces of the OBB, and are perpendicular to it. For what concerns point influence areas, the most natural bounding volume to localize them is 3D is a frustum (section 5.3.4). Rays start from the near plane of the frustum, and go toward the far plane. The

3| Multiple Influence Areas and the Top Level Structure

48

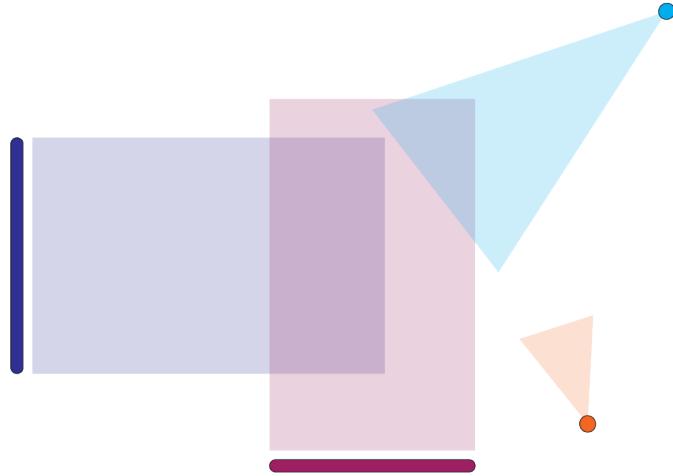


Figure 3.1: A scene with multiple relevant ray distributions.

direction of any of the ray can be obtained by connecting the origin of the ray on the near plane to the focal point of the frustum.

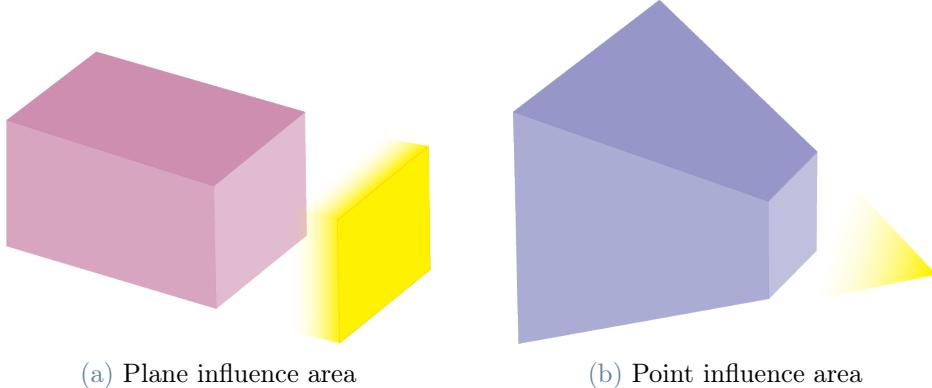


Figure 3.2: Plane influence areas are described by OBBs, whereas point influence areas are described by frustums.

3.1. Ray-Influence Area Affinity

The first necessary condition to have a ray that is affine to an influence area, is that the origin of the ray is inside the influence area. To do so it is possible to use the algorithms described in appendices A.6 and A.7, but can be made more efficient, as described in section 3.3.

The second necessary condition is that the direction of the ray fits with the direction of the rays in the influence area. This is extremely simple to verify for plane influence areas, where all the rays must have the same direction. Verifying this second condition for point

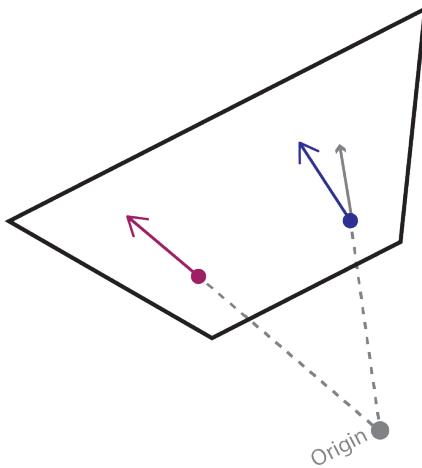


Figure 3.3: Visualization of the method to verify if the direction of a ray is affine with a point influence area.

influence areas is more complex, but can be achieved by connecting the origin of the ray with the focal point, and checking that the direction of this vector is parallel with the direction of the ray, as it is possible to visualize in figure 3.3.

It is possible to introduce a tolerance in the verification of the second condition. The bigger the tolerance, the more rays will be affine, but if the tolerance gets too large also rays that don't benefit from the BVH built with PAH will traverse it.

If a ray is affine to an influence area, it can use the BVH associated with the influence area during traversal.

3.2. Local and Global BVHs

Since influence areas are localized in specific regions of the scene, it would be wasteful to build the associated PAH BVH on all the geometry present. The associated BVH is therefore built by considering only the triangles inside the influence area. It is important to note that a triangle is considered inside the influence area if at least one of its vertices or edges is inside. This conservative definition of the *inside* concept is necessary to avoid errors in the traversal. Indeed, if we used a less restrictive definition, such as by considering the barycenter, it would be possible to find the wrong closest hit while tracing a ray, as it is possible to observe in figure 3.4.

3| Multiple Influence Areas and the Top Level Structure

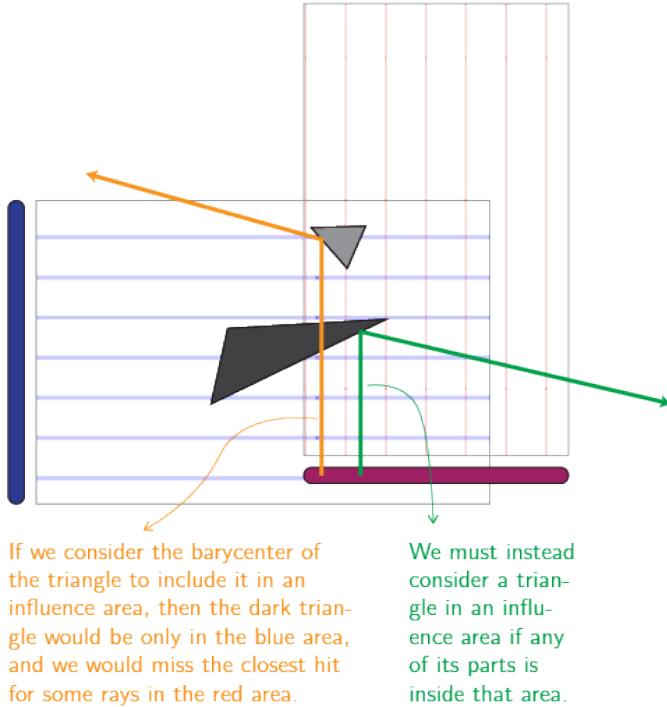


Figure 3.4: Example of why it is necessary to use a conservative approach.

Influence areas do not guarantee to cover the whole scene. It is indeed more frequent that regions of the scene don't have any particular ray distribution. In this case, those triangles that are not inside any influence area, would not be part of any BVH, and would therefore be invisible to rays during traversal. For this very reason, it is necessary to build a global BVH for the whole scene. In our implementation, this BVH is built by using surface area heuristic.

When a ray traverses a local BVH and no hit is found, it is necessary to traverse the global BVH. It is indeed possible to find a hit with triangles that are not inside the influence area of the local BVH, as it can be observed in figure 3.5. However, if a ray is found in the local BVH, it is guaranteed for it to be the closest one, as shown in figure 3.4.

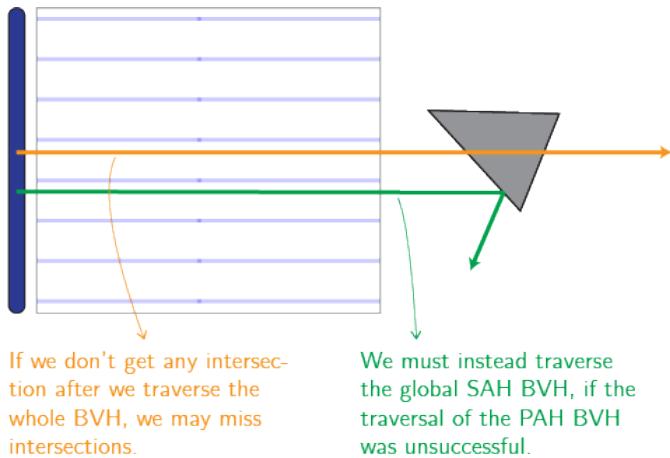


Figure 3.5: Example of how it is possible that a hit is found in the global BVH, but not in the local one.

3.3. Top Level Structure

At the beginning of this chapter, we stated that the first affinity criterion between a ray and an influence area is that the origin of the ray is inside the influence area. We have also seen how BVHs associated with influence areas contain only triangles that are inside the influence area. It is therefore of paramount importance to have a fast method to detect whether a point is inside a certain 3D region of the scene.

In our implementation we tried to tackle this problem by using two different approaches. We will now briefly describe them from a theoretical point of view. All the details of our implementation can be found in section 5.5.

The first approach is to use algorithms described in appendices A.6 and A.7, and each time we want to check if a point is inside any influence area, iterate over all the influence areas present in the scene. The complexity of this method linearly depends on the amount of influence areas present in the scene. Moreover, the algorithms to find out if a point is inside an OBB or a frustum are not cheap, especially in a scenario where they must be used for every single ray. On the other hand, all the influence areas can be easily stored in an array, therefore there is no overhead in building an ad-hoc structure.

On the opposite side of the spectrum there is the second method we propose. In this case we build an octree, where each leaf contains the influence areas that overlap with it. Building such an octree is not trivial, and it is diffusely described in section 5.5.2. However, once the octree is built, it is extremely cheap to find out what are the influence areas containing any given 3D point. It is indeed sufficient to traverse the octree and read

3| Multiple Influence Areas and the Top Level Structure

the data in the leaf. Considering that it is not necessary to have extremely fine-grained octrees to achieve good results with PAH, we consider octrees the better choice as top level structure. More on the performance can be found in chapter 6.

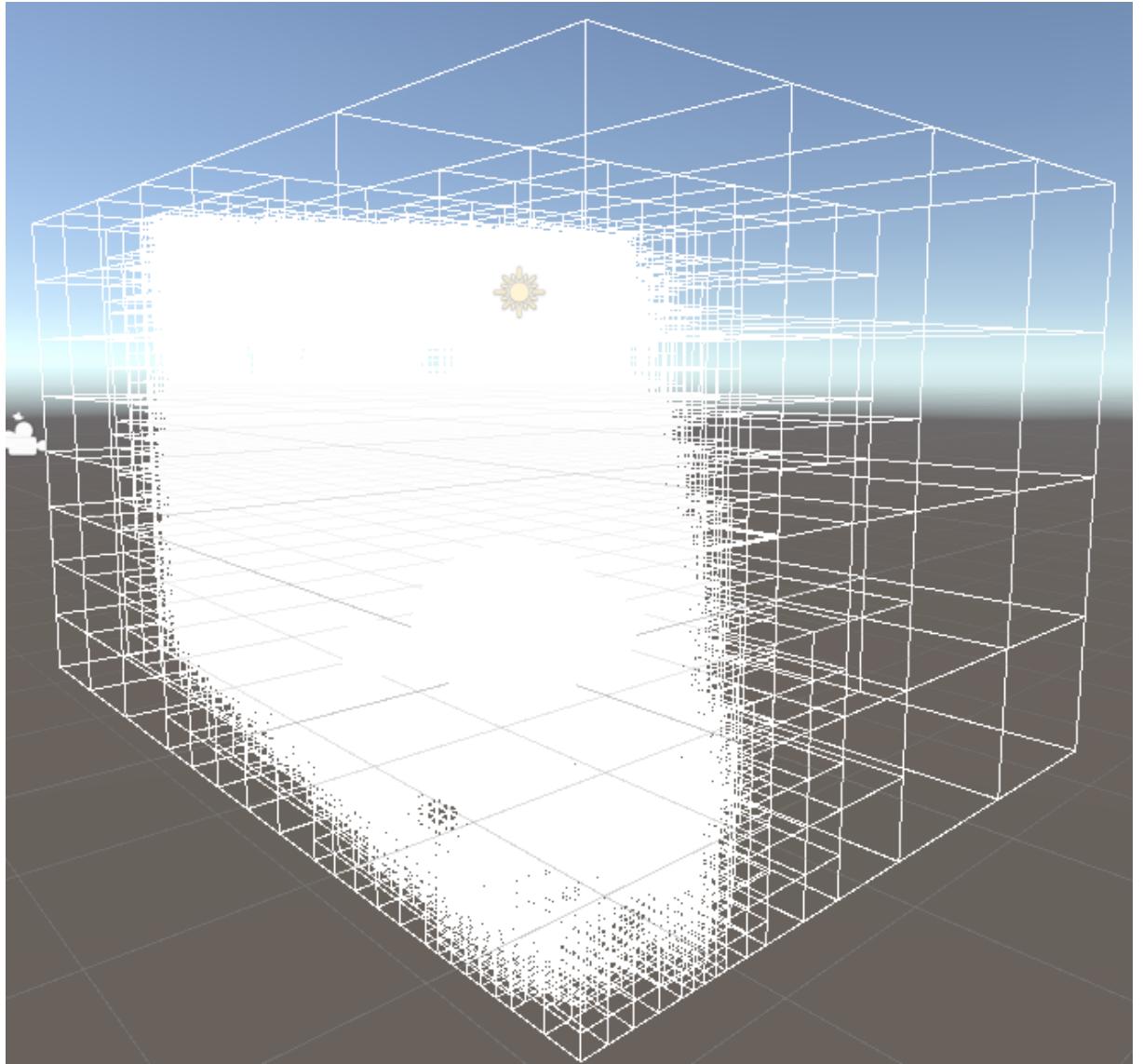


Figure 3.6: The wireframe of the octree built on a single point influence area.

4 | Splitting Plane Facing Heuristic

In chapter 2 we described the first heuristic we propose in this thesis. With PAH it is possible to estimate better the probability a ray hits AABBs in a region of the scene where relevant ray distributions are present. This enables us to have more accurate information on how to split a node at each level, during BVH construction. However, as we stated in section 1.3.3, another important step in building high-quality BVHs is the choosing of the orientation of the splitting plane. In this chapter we will propose a novel method to select the splitting plane, whose main aim is to reduce the overlapping of the projected area of the children nodes. Its name is Splitting Plane Facing Heuristic (SPFH).

As extensively discussed in section 1.3.3, in today's BVH builders, during the triangles splitting phase, usually only one plane orientation is selected. With this strategy it is possible to trade build time for accuracy. Indeed, fewer splitting planes must be attempted at each step, but, at the same time, it is possible to miss some relevant splits. Moreover, since the majority of BVHs use AABBs, the possible splitting plane orientations among which to choose one are only 3, parallel to each cartesian axis.

Before continuing to the next paragraph, it is important to talk about the language we will use. In particular, the expression *cut along the K axis* means that the plane we are using to split the AABB into two children has a normal that is parallel to the specified axis K . Furthermore, we use the expression *K axis plane* to denote a plane whose normal is parallel to the K cartesian axis.

The most used heuristic to choose the splitting plane orientation, is to cut along the longest dimension of the AABB. The main idea behind this method is that, the longer the axis of the AABB, the more chances there are to cut off space effectively.

Our idea instead, based on [1], is to use the information about the ray directions in a specific region of the scene, in order to divide a node such that it is less likely that its two children can be hit by the same ray. We can quantify the likeliness that two AABBs are hit by the same ray, by computing the projection of the overlapped area between the

4| Splitting Plane Facing Heuristic

two. The projection, as we described in chapter 2, is an orthographic projection in case of a plane influence area, and a perspective projection in case of a point influence area.

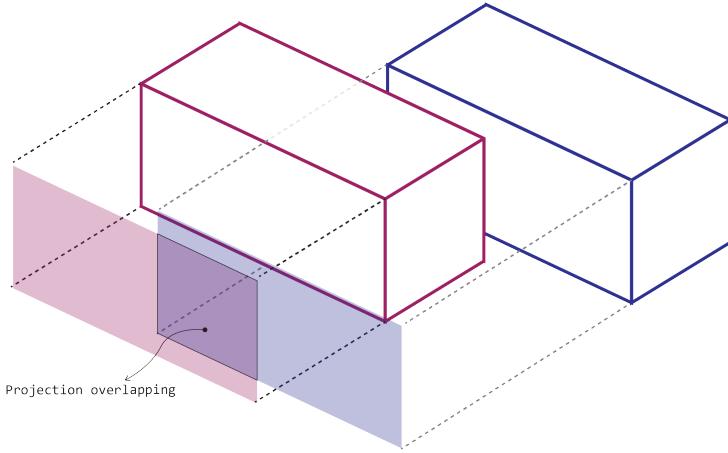


Figure 4.1: Overlapping projected area between two nodes' AABBs.

It is possible to compute the overlapping area between two AABBs projections with the algorithm described in appendix A.9. However, doing so for each possible triangles split at each BVH level would be too expensive from a computational point of view.

We therefore decided to use a heuristic to find out which is the best cartesian axis to cut along, so that it is most likely to have splits with not much overlapping. We found out that, the more the ray direction of the influence area is perpendicular to the normal of the splitting plane, the less likely it is for the two children's AABBs to have a large overlapping area.

We can first analyze the simplest case (figure 4.3), where there is a plane influence area with rays direction parallel to the z axis. In this case, the ray direction is perpendicular to the x and y cartesian axes, which define two possible splitting plane orientations. Instead, it is parallel to the z cartesian axis, which defines the last possible plane orientation (considering we use the optimization where only the 3 cartesian planes are used). If we consider a uniform distribution of the triangles in the father node, it is clear that, if we divide its AABB with the z axis plane, it is extremely likely for a single ray to hit both of the children. At the same time, it is almost impossible for a ray to hit both children if we use the other two planes to split the father node's AABB. Indeed, in this last case, the only possible source of overlapping is given by the fact that triangles are not point-like entities, but have an extension; therefore it is possible that the splitting plane, which partitions the triangles based on their barycenter, splits the vertices of a triangle in both the half-spaces generated by the plane, as we discussed in section 1.3.3 and as it is possible to observe in figure 4.2.

4| Splitting Plane Facing Heuristic

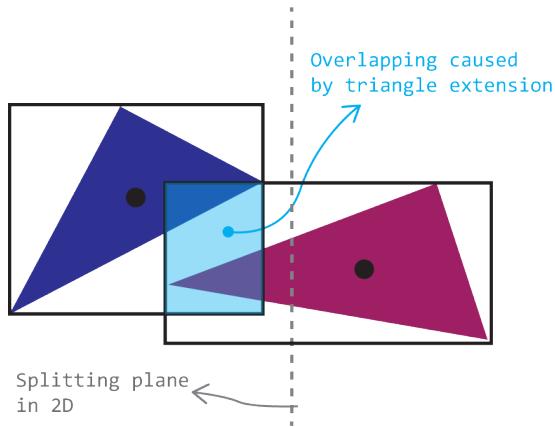


Figure 4.2: Overlapping caused by the triangle extension.

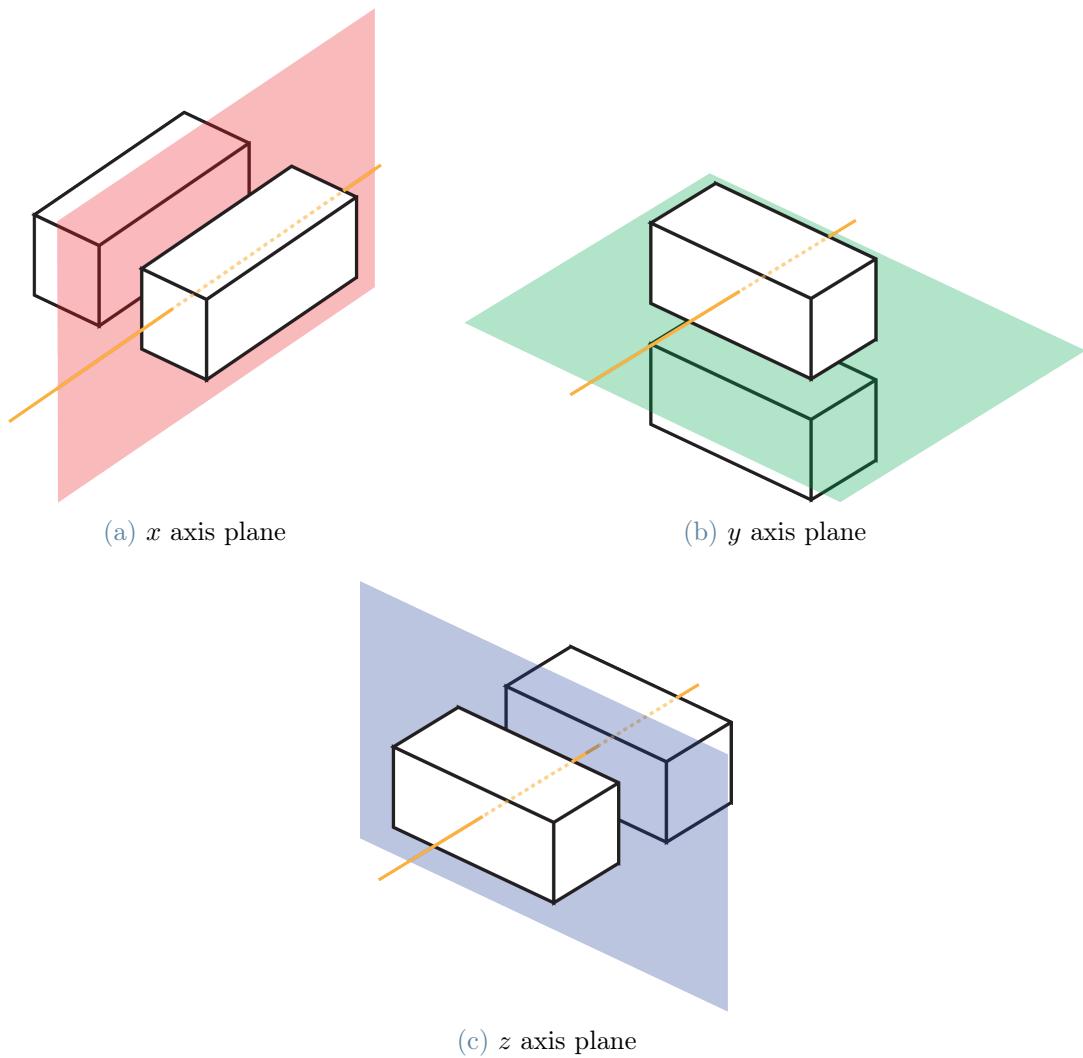


Figure 4.3: Visual illustration of the possible splitting plane orientations and how a ray can hit the children AABBs.

4| Splitting Plane Facing Heuristic



The same reasoning can be carried out with influence areas with arbitrary rays direction. However, it is important to understand that, if there isn't a clear cartesian axis the rays are perpendicular to, using SPFH to choose the splitting plane direction becomes useless. Let's imagine a case in 2 dimensions where the rays have an angle of 45° . In this case, if we adapt the reasoning in the last paragraph to the 2D scenario, we can see that choosing to cut along the x or y axes makes no difference. In such a case it is better to fall back to a known heuristic, such as the longest axis one.

For this reason, in our implementation, we decided to assign a quality value to each one of the three possible orientations of the splitting plane. We then use it to decide which is the best orientation to choose, or, in case none is good enough, to fall back to the longest axis heuristic.

The function to compute the quality value for the three cartesian axes takes into account how much the influence area's rays direction is parallel to each one of them. Given that the ray direction is named v :

$$\begin{aligned} \text{quality of } x \text{ axis} &= \frac{|v_x|}{|v_x| + |v_y| + |v_z|} \\ \text{quality of } y \text{ axis} &= \frac{|v_y|}{|v_x| + |v_y| + |v_z|} \\ \text{quality of } z \text{ axis} &= \frac{|v_z|}{|v_x| + |v_y| + |v_z|} \end{aligned}$$

In other words, we compute how much each cartesian axis direction is parallel to the rays direction, in percentage. In this way we can clearly quantify how much rays are parallel to each cartesian axis. At this point, it is possible to set a threshold over which a cartesian axis is deemed *too much parallel* to the direction of the rays to be a good choice for the splitting plane. If all the cartesian axes are over the threshold, we can fall back to the longest axis heuristic.

It is also possible that two cartesian axes have a very low quality value. In the example of the simplest case scenario, that can be found a few paragraphs above, the quality values are 0% for the x and y axes and 100% for the z axis. In this case, since both x and y are good candidates for the splitting plane, one is chosen arbitrarily. After a split is found, if the cost of the two children nodes is above a certain threshold, then also the second axis is tried. The threshold can be dynamically adjusted based on the quality of the axes in contention. For example, if instead of a quality value of 0%, as in the example above, the two axes had a quality value of 10%, then the threshold could be adjusted to a higher value, because it is less likely to find a good split with a quality value of 10% rather than

4| Splitting Plane Facing Heuristic

0%.

The same concept can be applied to the use of the fallback method. Before we stated that the fallback method is used when the quality values of all the axes are above a certain static threshold. However, it is also possible to set our implementation so that the fallback method is also used when, after a split, the two children nodes have a cost over a customizable fallback threshold.

A more in-depth explanation of our algorithm can be found in section 5.1.1.

Thanks to the customizable thresholds it is possible to make our algorithm more or less accurate in the choosing of the splitting plane. Having a more accurate algorithm can yield higher-quality BVHs, at the cost of a higher construction time.

Another important factor in the effectiveness of SPFH is the type of influence area. Until now we talked about *the rays direction of the influence area*. If this definition can be unequivocably given to plane influence areas, where all the rays face the same direction (chapter 3), the same cannot be stated for point influence areas, where the rays face many different directions. In the case of point influence area, the central direction can be considered as the direction of all the rays. However, if the point influence area have a large spread in its rays' directions, then this approximation doesn't hold. In such cases, it is not possible to use SPFH effectively, and it is therefore advised to fall back to the longest area heuristic.

The discussion about the quality of the BVHs built with the splitting plane facing heuristic (SPFH) can be found in chapter 6.

5 | Implementation

In this chapter we will analyze how we built the framework that allowed us to test the novel techniques we explained in previous chapters, and to compare the results with other methods. We will explain all the implementation details we skipped in previous chapters, however we won't dwell on the performance, which is the main subject of chapter 6.

At the beginning of the development we used to write prototypes in C#, in order to be able to immediately visualize the results in Unity. However, as we were convinced of the validity of our ideas, we decided to write code in C++, in order to have more control over it and be independent from a specific engine for visualization purposes. The code of our project can be found on this GitHub repository: [TODO](#).

Furthermore, it is possible to read the documentation of our code, generated with Doxygen¹, at this address: [TODO](#). We decided to provide the documentation in order to help future developments. As explained in the next chapters, we tried to make the framework generic and flexible, in order to be able to test different algorithms without completely rewriting the system.

In parallel with the development of the framework to test our algorithms, we also developed an independent visualizer in Unity. The visualizer receives a JSON generated by the C++ framework, and shows the generated data structures over a scene (usually the BVHs). We decided to build the Unity visualizer in order to help us graphically visualize the results of our algorithms. This approach proved very useful to fix bugs and identify pitfalls in our approaches. We won't describe the Unity visualizer in this thesis, but the code can be found at this address: [TODO](#).

Despite the framework being written in C++, in the next sections we will present the algorithms in a language-agnostic way: our focus is on the concepts rather than the idiosyncrasies of a specific language.

¹<https://www.doxygen.nl/>

5.1. BVH

In this section, we will present the concepts about the class that in our implementation is responsible for representing, building and traversing a BVH.

The main data members of our *Bvh* class are the following:

```
class Bvh
    root : Node
    influenceArea : InfluenceArea*
    computeCost : function*
    chooseSplittingPlanes : function*
    shouldStop : function*
    properties : Properties[]
```

We store the influence area the BVH is associated to. The pointer can be left *null* in case the BVH is not associated with any influence area, such as for the scene-global BVH.

The three functions are customizable, and are customization point for the BVH construction, as explained in the algorithm in section 1.3.3.

We also store the root node of the BVH. Each node is a member of the *Node* class, whose main members are:

```
class Node
    aabb : Aabb
    leftChild : Node*
    rightChild : Node*
    triangles : Triangle
```

Since a BVH is a complete tree, a leaf node can be detected by the absence of its children.

An important note is that, in our framework, we are also interested in getting information about the timing and performance of the BVH builder. For this purpose it is possible to add time information to a *Node*, collected during construction, by enabling a C++ macro. The information in each node can later be analyzed, as it is explained in section 5.2.

The *properties* are values upon which depends how the BVH is built. We will now list and explain some of them:

5| Implementation



maxLeafCost If a node has a cost (PAH or SAH depending on the cost strategy used) less than this threshold, it is a leaf.

maxLeafArea If a node has an area (projected or surface depending on the cost strategy used) less than this threshold, it is a leaf.

maxLeafHitProbability If the hit probability (section 1.3.3) of this node is less than this threshold, it is a leaf.

maxTrianglesPerLeaf If a node has fewer triangles than this threshold, it is a leaf.

maxLevels If a node is at a level higher than this threshold, it is a leaf.

bins How many cuts to try to split a node into its children. A higher value generates more accurate BVHs, but is also more expensive (section 1.3.3).

maxNonFallbackLevels If a node is at a level higher than this threshold, the specified fallback strategies (usually SAH and longest splitting plane heuristic) will be used to split it into children. This can be used to avoid using an expensive strategy (such as PAH) even at deep levels, where there is less to gain.

splitPlaneQualityThreshold [0, 1]. If the quality of the split plane (chapter 4) is less than this threshold, two things can happen: if a satisfactory split plane has already been found (look at the next two properties), it is used; if not, use the fallback strategy to find the splitting plane. A low value will leave the algorithm find the best splitting plane more times, but will slow the construction down. More details in section 5.1.1.

acceptableChildrenFatherHitProbabilityRatio Defined as $\frac{\text{leftChildHit\%} + \text{rightChildHit\%}}{\text{fatherHit\%}}$.

This value is used to determine if a splitting plane cut is acceptable when no more planes with the minimum quality are present. The value is used to approximate the overlapping of the two children nodes: in the ideal case (no overlapping) this value is less than 1. A low value forces the algorithm to use the fallback strategy more often, therefore will also slow the construction down. More details in section 5.1.1.

excellentChildrenFatherHitProbabilityRatio Same as above, but in this case this value is compared with the best children found after each splitting plane cut: if such value is lower than this threshold, no more splitting planes are attempted, even if they had the required quality. This allows an early out during the search for the splitting plane. More details in section 5.1.1.

A *Bvh* can be built with the member *build* function, which takes an array of *Triangles* representing the scene as argument. The function has already been described in section

5| Implementation

1.3.3. Here we limit ourselves to remember that the function builds the BVH recursively, and uses the three customization functions cited above to decide how to build the BVH.

To traverse a *Bvh* it is possible to use the *traverse* function, which takes a *Ray* as argument. The process will be detailed in section 5.1.2.

5.1.1. Splitting Plane Search Implementation

In this section we will analyze in more detail how the selection of the best splitting plane for each node is implemented.

The function *chooseSplittingPlanes* is required to return a list of cartesian axes, associated with their quality value. This function always returns all the three axes, even if the quality of some of them is extremely low. More info about this topic can be found in chapter 4.

In the *build* function of the *Bvh*, we iterate over the axes returned by the *chooseSplittingPlanes* function, and, for each one of them, we use the binned approach to look for a good way of splitting the node into two children. However, some of the properties listed above, can influence what axes returned by *chooseSplittingPlanes* are actually used:

Algorithm 5.1 How the *build* function chooses what axes to use to look for the best split.

```

1: ...
2: axes ← chooseSplittingPlanes(...)
3: for all ⟨axis, quality⟩ ∈ axes do
4:   bestCutSoFarQuality ←  $\frac{\text{bestLeftCostSoFar.hitProbability} + \text{bestRightCostSoFar.hitProbability}}{\text{fatherHitProbability}}$ 
5:   if bestCutSoFarQuality ≤ excellentChildrenFatherHitProbabilityRatio then
6:     break
7:   forceFallback ← false
8:   if quality < splitPlaneQualityThreshold then
9:     if bestCutSoFarQuality ≤ acceptableChildrenFatherHitProbRatio then
10:      break
11:   forceFallback ← true
12:   axis ← chooseSplittingPlanes(forceFallback, ...)[0]
13:   ...           ▷ start of the binned search phase with axis as splitting plane normal...

```

In simple words, if an axis is *excellent*, meaning that a very good triangle split has been found by using it, no more cuts are attempted, no matter what (line 5).

Instead, if no more axes are good enough (based on the threshold at line 8), then, if a cut

5| Implementation

has already been found in previous steps and its quality is *acceptable* (line 9), it is used as splitting plane. Otherwise the fallback method (always SAH in our test cases) is used to find a splitting plane (line 11 and 12).

It is recommended that the *build* function often looks for the best split by using just one plane, as we suggested in section 1.3.3. It is possible to achieve this by choosing a not-too-high threshold for what is considered an *acceptable* cut, or by rising the *splitPlaneQualityThreshold* to discern a good splitting plane from a bad one. Using more than one splitting plane should happen only in exceptional cases if this algorithm is to be used in a real-time scenario.

5.1.2. Traversal

The BVH traversal process is more straightforward than the build process. The main data structure used to traverse the BVH is the queue:

Algorithm 5.2 How the *build* function chooses what axes to use to look for the best split.

```

1: function TRAVERSE(ray)
2:   closestHitTriangle  $\leftarrow$  null
3:   closestHitDist  $\leftarrow \infty$ 
4:   toVisit  $\leftarrow$  queue
5:   toVisit.push(root)
6:   while not toVisit.empty() do
7:     current  $\leftarrow$  toVisit.pop()
8:     hitInfo  $\leftarrow$  collision(ray, current)
9:     if hitInfo  $\neq$  null and hitInfo.distance  $<$  closestHit then
10:      if current.isLeaf() then
11:        for all triangle  $\in$  current.triangles do
12:          triangleHitInfo  $\leftarrow$  collision(ray, triangle)     $\triangleright$  abbreviated to thi
13:          if thi  $\neq$  null and thi.distance  $<$  closestHit then
14:            closestHitTriangle  $\leftarrow$  triangle
15:            closestHitDist  $\leftarrow$  thi.distance
16:        else
17:          toVisit.push(current.leftChild)
18:          toVisit.push(current.rightChild)
19:      return  $\langle$  closestHitTriangle, closestHitDist  $\rangle$ 
```

In our implementation we are interested in analyzing whether the BVH cost is accurate.

For this reason we made some addition to the presented algorithm to artificially compute the real cost of the traversal of a BVH. In particular, we introduce a variable to keep track of the cost of the traversal which simulates the SAH or PAH. When we enter a leaf node, the value is incremented by `current.triangles.size() · 1`. When a non leaf node is hit the value is incremented by $2 \cdot 1.2$. It is possible to observe the similarities with what we discussed in section 1.3.3.

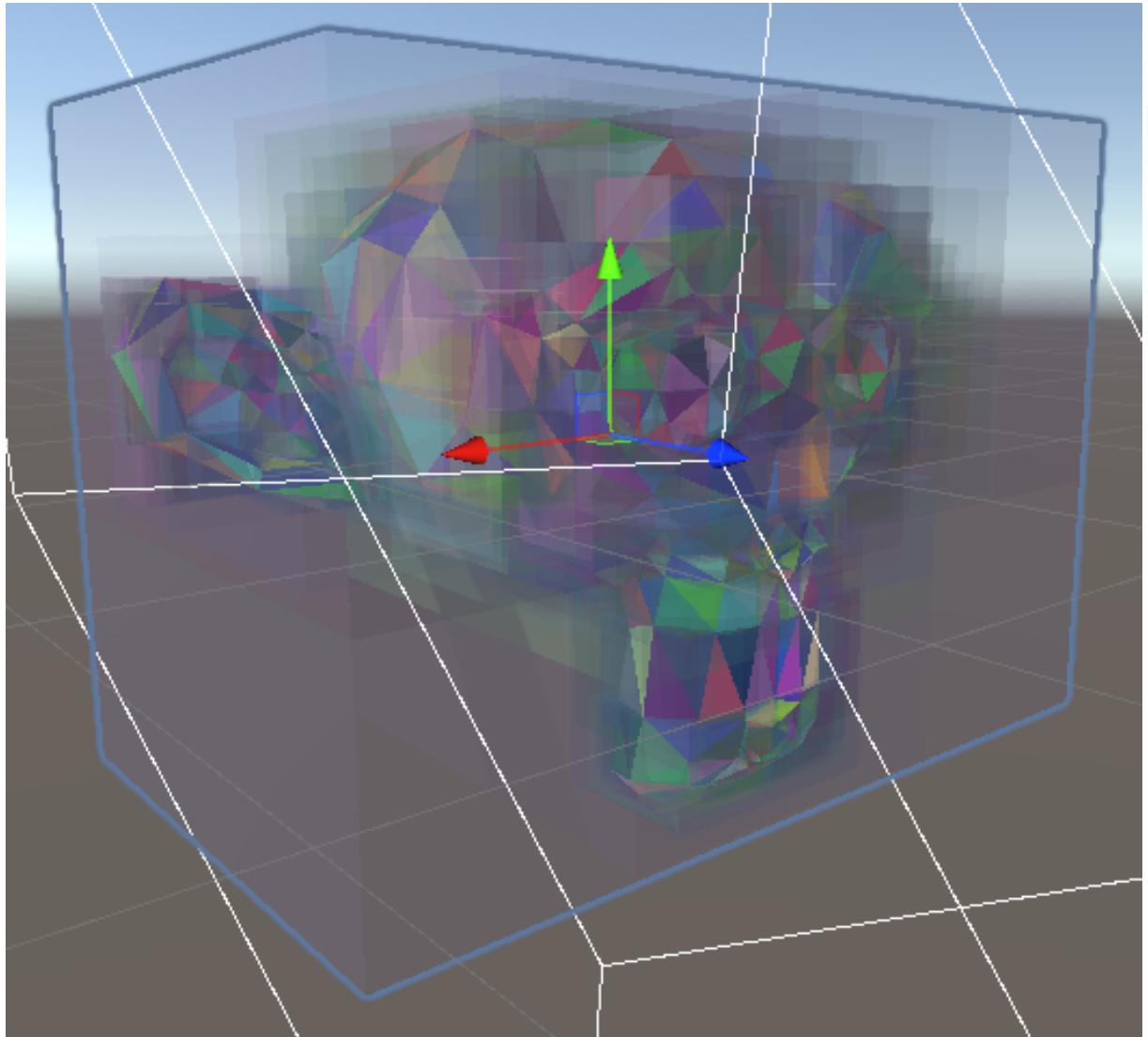


Figure 5.1: The BVH built over the Blender Suzanne mesh, as seen in our visualizer.

5.2. BVH Analyzer

In this section we will describe the class used to analyze a BVH and produce a JSON output, that can be passed to the Unity visualizer or to other analyzing tools.

One of the core points in our implementation of the BVH builder was to avoid as much as possible introducing overhead caused by procedures collecting data about the BVH. Nonetheless collecting information about the BVH was one of the most relevant interests of our research. To let these two necessities coexist, we decided to write a class that receives a BVH as input, along with some functions to analyze it, fully traverses it breadth-first, and outputs a JSON representing the BVH and the required information.

The most challenging part has been to let the user of the framework write its own data-collecting functions, and make it so that it was possible to collect virtually any statistic about the BVH by using these functions. At the end of the day we adopted an approach where the user must specify two functions for each category of data he or she wants to collect:

Per-node functions These functions are executed at each node during the traversal. They take as input the node itself and an object of a data type chosen by the user. This object is stored as a data member of the *BvhAnalyzer* class², therefore it is persistent across the invocations on different nodes.

Final actions These functions are executed at the end of the traversal of the BVH, and take as input the same object the corresponding per-node function uses. In this way, the per-node functions can store data on a *BVH-global* variable, and this data can be processed at the end of the traversal.

An example of per-node/final action pair is to calculate the PAH or SAH cost of a BVH: each per-node function computes the PAH or SAH cost for the node, appends it to the JSON of the current node, and adds its value to a global total cost. The final action reads the global total cost and outputs it to the JSON.

The same logic can be used, for example, to count the number of triangles (per-node and total), compute the siblings overlapping area (per-node and average, see chapter 4), save information about performance or simply count nodes.

5.3. Regions

In this section we will analyze some of the data structures we used to describe regions of 3D space. These structures are then used in many parts of the codebase, such as in the BVH, as influence areas or in the top level structure.

²To achieve this in practice we store a C++ tuple as data member of the *BvhAnalyzer*. At compile time, by leveraging template metaprogramming, the user can specify what data types he or she wants to store inside this tuple.

5| Implementation

All the presented structures derive from the base *Region* structure. This allows us to be able to interchange them easily. The *Region* structure contains the following methods to be overridden:

contains(point) Returns whether a point is inside the region. Used for AABBs for Regions top level structure (section 5.5.1), and for octree top level structure construction and traversal (section 5.5.2).

enclosingAabb() Returns the tightest AABB enclosing the region. Often used in pair with *contains(point)* as rejection test (look at section 5.3.3).

isCollidingWith(aabb) Returns whether the region is intersecting the specified AABB. Used in top level octree construction (section 5.5.2).

fullyContains(aabb) Returns whether the region is fully enclosing the specified AABB. Used in top level octree construction (section 5.5.2).

5.3.1. Oriented Bounding Box

An oriented bounding box is a region of 3D space enclosed in a rectangle parallelepiped. As mentioned in section 1.3.2, OBBs can be used as bounding volume in a BVH. While providing a good approximation of the volume occupied by the primitives in many situations, they are quite expensive to intersect with a ray, therefore are not often employed as bounding boxes in state-of-the-art BVH builders. In our implementation we use them to describe a plane influence area, as shown in section 5.4.1.

In our implementation we describe a bounding box in this way:

```
class Obb
    center : Vector3
    forward : Vector3*
    right : Vector3*
    up : Vector3
    halfSize : Vector3
```

We decided to store all the three main axes of the OBB because they are used in the SAT algorithm, as described in section A.5, and also in the algorithm to detect if a point is inside the OBB.

The *halfSize* member stores the extents of the OBB starting from the center and toward the 3 main directions.



5| Implementation

contains(point)

In order to detect if a point is inside an OBB, we project its distance vector from the center, via a dot product, to all the three main axes: if all three projections are shorter than the half size in the corresponding direction, the point is inside.

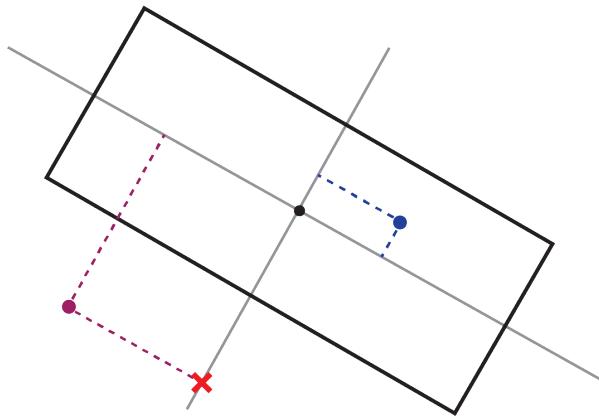


Figure 5.2: Check to detect if a point is inside an OBB.

enclosingAabb()

To compute the tightest enclosing AABB, we iterate over all the 8 vertices of the OBB and compute the maximum and minimum in every dimension.

isCollidingWith(aabb)

We use a simplified version the separating axis test (section A.5).

fullyContains(aabb)

To check if an AABB is fully contained in the OBB, we simply check that all the vertices of the AABB are inside the OBB, by using the *contains(point)* method.

5.3.2. Axis Aligned Bounding Box

An axis-aligned bounding box (AABB) is an OBB where the three main directions are constrained to be parallel to the three cartesian axes of the reference system. AABBs are the most used 3D region in our implementation. They are the key element of a BVH (section 5.1), and are also used in the octree top level structure (section 5.5.2).

The fact that AABB are, indeed, axis aligned, make them worse than OBBs as bounding volumes, however make them extremely fast to intersect with a ray in a GPU-friendly

5| Implementation

way (section A.1), and also extremely space-efficient to store:

```
class Aabb
    min : Vector3
    max : Vector3
```

AABBS can indeed be thought as a 3D analogous to a 1D segment. To describe the set of points of a segment, it is sufficient to know the start point and the end point. The same can be said for the AABB in 3 dimensions. We store the minimum and maximum point in all 3 dimensions, and every point within the min-max range in all 3 dimensions is, by definition, part of the AABB.

contains(point)

To check if a point is inside an AABB, it is sufficient to check that the point is inside the min-max range in all three directions.

enclosingAabb()

The AABB itself is the tightest enclosing AABB.

isCollidingWith(aabb)

The algorithm to check if two AABBs are colliding is described in section A.4. It is way less expensive than the separating axis test.

fullyContains(aabb)

To check if an AABB is fully contained in another AABB, it is enough to check that the minimum of the enclosing AABB is smaller in all dimensions than the minimum of the enclosed AABB; and that the maximum of the enclosing AABB is greater in all dimensions than the maximum of the enclosed AABB:

$$\left\{ \begin{array}{l} A.\min_x \leq B.\min_x \\ A.\max_x \geq B.\max_x \\ A.\min_y \leq B.\min_y \\ A.\max_y \geq B.\max_y \\ A.\min_z \leq B.\min_z \\ A.\max_z \geq B.\max_z \end{array} \right.$$

5.3.3. AABB for OBB

AABBS for OBBs are a data structure that optimize the ray-OBB intersection test cost in some scenarios, at the expense of a bigger memory footprint. As the name suggests, an AABB for OBB is an OBB where the tightest enclosing AABB is cached the first time it is computed:

```
class AabbForOobb
    obb : Oobb
    aabb : Aabb
```

In our implementation, virtually every time we could use an *Oobb*, we instead use an *AabbForOobb*. In all the predicate functions derived from *Region*, the *Aabb* implementation works as refutation test. It means that a negative response from the *Aabb* implementation implies a negative response from the *Oobb* implementation too. In symbols:

$$\neg Aabb.function \implies \neg Oobb.function$$

The condition is only sufficient, but not necessary. It is thus needed to fall back to the more expensive *Oobb*'s implementations in case the *Aabb*'s implementation give a positive response.

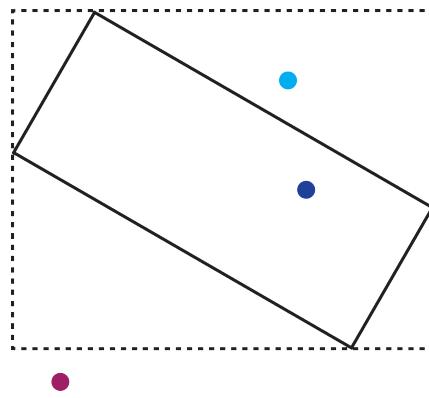


Figure 5.3: If the point (purple) is outside the AABB, it is necessarily outside the OBB too. The inverse (blue and light blue) is not guaranteed.

5.3.4. Frustum

A frustum represents the 3D region of space contained in a truncated rectangle pyramid. A frustum is used in computer graphics to represent the area of the scene visible from a

5| Implementation

pinhole camera. Due to its main usage, the two parallel planes of the frustum are called near plane and far plane.

A frustum can be defined by specifying its orientation in space, an origin point, the distance of the near and far planes from the origin point along the orientation direction, and a horizontal and vertical fields of view.

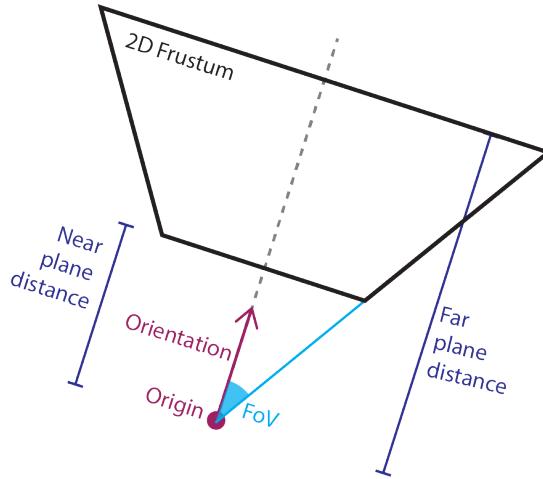


Figure 5.4: How a frustum can be defined.

Moreover, we can define a perspective projection matrix associated with the frustum. With this matrix it is possible to project a point in the frustum to its near plane. This aspect is particularly relevant in our implementation, since we use frustums to define the region of relevance of point influence areas (section 5.4.2), and we must have a method to project points of a point influence area to compute PAH.

In our implementation the frustum is described as an object of the following class:

```
class Frustum
    vertices : Vector3 []
    edgesDirections : Vector3 []
    facesNormals : Vector3 []
    projectionMatrix : Matrix4x4
    enclosingAabb : Aabb
```

Even though it is redundant to store vertices, edges and faces, we opted to compute and store them during the initialization phase, because they are often used in the *Frustum* interface, such as in the *isCollidingWith(aabb)* function.

5| Implementation



We also decided to store the matrix because we use frustums as point influence areas, therefore we often need to be able to project points to the near plane, such as during the computation of PAH. Furthermore, in the *contains(point)* function, it is necessary to project the point to detect whether it is inside the frustum, as explained in section A.7.

Analogous to what happens with the *AabbForObb* structure, also with the *Frustum* we cache the tightest enclosing AABB and use its implementation of the predicate functions derived from *Region* as refutation tests. However, if the refutation fails, the *Frustum* implementations can be even more expensive than the *Obb* ones.

contains(point)

After the AABB refutation test fails, we use the algorithm described in section A.7 to detect whether a point is inside the frustum.

enclosingAabb()

To compute the tightest enclosing AABB, we iterate over all the 8 vertices of the frustum and compute the maximum and minimum in every dimension. However, after the first time it is computed, the AABB is cached.

isCollidingWith(aabb)

After the AABB refutation test failure, we use a simplified version the separating axis test (section A.5) to detect if the frustum is intersecting an AABB.

fullyContains(aabb)

In case the AABB refutation test fails, we check that all the vertices of the AABB are inside the frustum, by using the *contains(point)* method.

5.4. Influence Areas

Influence areas, as described in chapter 3, in our implementation are those objects responsible for defining a 3D region of space where there is a relevant ray distribution. The main task of an influence area is to provide an interface that can be queried to retrieve information related to its associated ray distribution. The most important methods, present in the base *InfluenceArea* class, are:

getProjectedArea(aabb) Returns the area projected by an AABB, based on the

associated ray distribution of the influence area: an orthographic projection for a plane influence area, and a perspective projection for a point influence area. Used during PAH based BVH construction.

getProjectedHull(aabb) Returns an array of 2D points, representing the projection of the contour points of an AABB as seen from the point of view of the influence area. Its use can be observed in section 6.2.2.

getRayDirection(aabb) Returns the direction of a ray intersecting the center of the specified AABB. Used in BVH construction when SPFH is adopted.

isDirectionAffine(ray) Returns whether the ray direction is affine to the underlying ray distribution. Affinity is a concept explained in section 3.1, and is used during BVH construction, if SPFH is employed.

Moreover, each *InfluenceArea* holds a *Region* as data member. The *Region* is used to define the 3D volume of space where the relevant ray distribution is present.

In the next sections we will explain the details of the functions listed above for two concrete influence areas: plane and point ones.

5.4.1. Plane Influence Area

As explained in chapter 3, plane influence areas describe regions of space where a distribution of parallel rays is present. As detailed in section 2.2, we can associate an orthographic projection to this kind of distribution, and use it to better estimate the probability an AABB is hit by a ray, by using PAH.

In order to build a *PlaneInfluenceArea*, it is necessary to specify a plane (namely a direction, a point it passes through and a 2D extent to specify the size of the plane) and a forward size, which describes how long the influence area is along the plane normal direction. Starting from this information, it is possible to create an *Obb* delimiting the region of 3D space where the relevant ray distribution is located. It is also possible to generate an orthographic projection matrix and a view matrix, that can then be multiplied to a view-projection matrix.

Given the extents of the plane (*top*, *bottom*, *right* and *left*) and the range in the forward direction (0 to *far*), the orthographic projection matrix can be built in this way [60].

5| Implementation

$$\begin{bmatrix} \frac{2}{right-left} & 0 & 0 & \frac{left+right}{2} \\ 0 & \frac{2}{top-bottom} & 0 & \frac{top+bottom}{2} \\ 0 & 0 & \frac{-2}{far} & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.2)$$

With this matrix we transform the rectangle parallelepiped viewing volume to the canonical view volume, which is the unit cube.



Figure 5.5: Visualization of the orthographic projection matrix: it shrinks the rectangle parallelepiped viewing volume to the canonical view volume.

The view matrix can be calculated starting from the *eye* position (the point the plane passes through), a *forward* direction (the normal of the plane), and a *worldUp* direction, which in our implementation is always $\langle 0, 1, 0 \rangle$.

We want to create a matrix to transform a point's coordinates from the global coordinate system, to the coordinate system centered on the *eye* position and looking at the *forward* direction. To do so, we must first compute the *right* and *up* directions starting from the *forward* and *up_{world}* directions, and then translate the origin of the global system to the *eye* position.

We know that the *right* direction is perpendicular to the *forward-up_{world}* plane, therefore we can use a cross product to compute it: $right = up_{world} \times forward$.

Then, we can use the same logic to compute the *up* direction: $up = forward \times right$.

5| Implementation

Finally, we must compute the translation vector in the new coordinate system:

$$\text{translation} = \langle \text{right} \cdot \text{eye}, \text{up} \cdot \text{eye}, \text{forward} \cdot \text{eye} \rangle$$

The view matrix now is:

$$\begin{bmatrix} \text{right}_x & \text{right}_y & \text{right}_z & -\text{translation}_x \\ \text{up}_x & \text{up}_y & \text{up}_z & -\text{translation}_y \\ \text{forward}_x & \text{forward}_y & \text{forward}_z & -\text{translation}_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.3)$$

getProjectedArea(aabb)

In order to compute the area orthographically projected by an AABB to a plane, we can observe that, regardless of the orientation of the plane, the projected hull can be subdivided into three parallelograms³, as it is possible to observe in figure 5.6. The algorithm to compute the projected area follows the following steps:

1. Choose any vertex of the AABB, let's say vertex A .
2. Select the adjacent vertices to A , called B , C and D .
3. Project the 4 vertices to the plane: $A' = \text{viewProjectionMatrix} \cdot A$.
4. Compute the $\overrightarrow{A'B'}$, $\overrightarrow{A'C'}$ and $\overrightarrow{A'D'}$ vectors.
5. Compute the areas of the three parallelograms. To do so it is possible to use the fact that the length of the cross product of two vectors equals the area of the parallelogram they generate: $\text{Area parallelogram} = |v \times u|$.
6. Sum the three areas.

This method of computing the area consists in only projecting 4 points and computing 3 cross products. The alternate method would be to compute the external hull of the AABB as seen by the plane point of view (explained in next section), projecting all the contour points and computing the area of the resulting 2D polygon, by using the algorithm in section A.10.

³It is possible that some parallelograms degenerate to a segment, for example in the case where the plane is parallel to one of the faces of the AABB. The computations handle this case, since the area of a degenerate parallelogram is 0.

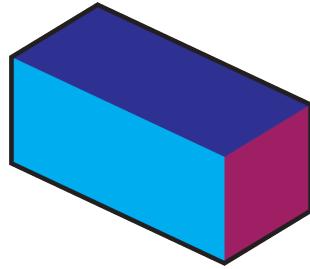


Figure 5.6: The orthographically projected hull of an AABB can be subdivided in 3 parallelograms.

getProjectedHull(aabb)

The projected hull of an AABB is defined as a 2D polygon formed by the silhouette of the AABB projected to a plane. We can define each point that, after the projection, is part of the hull as contour points; if instead a projected point is not part of the hull, and is thus internal to it, we call it internal point (figure 5.7).

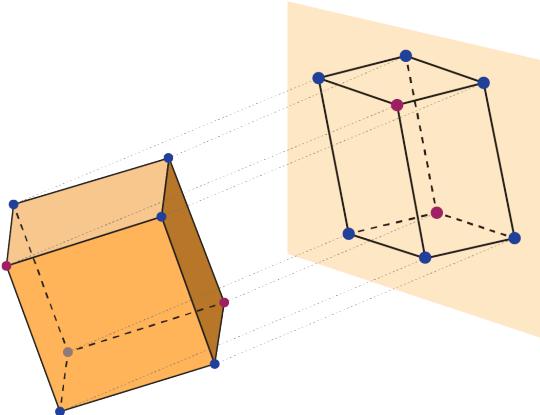


Figure 5.7: The projected hull of an AABB. The blue points are contour points, the purple points are internal points.

In order to compute the projected hull, we first have to detect what points of the AABB will be contour points after the projection. This is to avoid projecting more points (the internal points) than necessary. We can observe that, given a plane where to project the AABB, the points making up the projected hull can be determined by simply looking at the normal direction to the plane. In particular, it is sufficient to know the sign of each component of the normal direction to define what points will be contour points, and what points will be internal ones.

In our implementation, we hold a dictionary-like structure, called *hullTable*, which, at each element, stores a list of vertices. The list of vertices stored as elements of the

5| Implementation

dictionary tells us what are the contour points of a projected AABB if it is observed by a certain point of view relative to the AABB itself. For example, if the AABB is seen from a point of view directly above the center of the AABB and looking directly down, the projected hull will have as contour points the vertices of the top face of the AABB. If instead the AABB is seen from the top-left, then the contour points will be those of the top and left faces.

In order to be able to perform a fast look-up of the *hullTable*, we decided to implement the structure as an array, with a particular indexing rule. The indexing rule states that each position of the array is a binary encoding of the relative position from which the AABB is looked-at. We show the rule in the following table:

bit	5	4	3	2	1	0
look from	back	front	top	bottom	right	left

For example, at index 9, which in binary is 001001, we will store the projected contour points in the case the AABB is seen from top and left. Of course, in the *hullTable* array there will be some empty positions, since it is impossible, for example, to look at an AABB from top and bottom at the same time.

In order to be able to identify the vertices of an AABB, we use the conventional layout shown in figure A.4 in section A.5.1.

With the indexing of the *hullTable* array in place, it is now possible to compute the index of the array where the contour points are stored, only based on the normal direction of the projection plane, as shown in the following algorithm:

Algorithm 5.3 Given the direction of the normal to the projection plane, returns the corresponding index in the *hullTable*

```

1: function FINDHULLTABLEINDEX(dir)
2:   i  $\leftarrow$  0
3:   if dir.x  $>$  0 then  $|i| = 1$ 
4:   else if dir.x  $<$  0 then  $|i| = 2$ 
5:   if dir.y  $>$  0 then  $|i| = 4$ 
6:   else if dir.y  $<$  0 then  $|i| = 8$ 
7:   if dir.z  $>$  0 then  $|i| = 32$ 
8:   else if dir.z  $<$  0 then  $|i| = 16$ 
9:   return i

```

The found contour points can then be projected via the view-projection matrix.

5| Implementation

getRayDirection(aabb)

Regardless of the AABB, the ray direction is always parallel to the forward direction of the *Obb* stored as *Region*.

isDirectionAffine(ray)

To check if the ray direction is affine to this influence area, as defined in section 3.1, we must check that the ray direction and the forward direction of the *Obb* are parallel. It is possible to introduce a tolerance while checking for the parallelity of the two directions.

5.4.2. Point Influence Area

A point influence area describes a region of the scene where there is a radial distribution of rays. As shown in section 2.3, the rays part of a point influence area can be seen as the rays generated by a pinhole camera. For this reason, a perspective projection can be associated to such a ray distribution, and it can be used to project the AABBs of the BVH associated with this influence area, to calculate the projected area heuristic. As anticipated in section 5.3.4, a *Frustum* is the ideal bounding volume to delimit the area of relevance of a point influence area.

To create a *PointInfluenceArea*, it is required to specify a *Pov* and a pair of distances along the *Pov* forward direction, where the near and far planes of the *Frustum* will be placed. A *Pov* is a class that represents a point of view. As such, it stores an eye position, a forward direction, and a pair of horizontal and vertical fields of view:

```
class Pov
    eye : Vector3 []
    forward : Vector3 []
    fovX : Matrix4x4
    fovY : Aabb
```

Given a *Pov* and a near and far planes, it is possible to directly compute the perspective matrix and the view matrix, that are then combined in a view-projection matrix via multiplication.

The process to compute the view matrix has already been explained in section 5.4.1, thus it won't be repeated here.

To compute the perspective projection matrix, we need the bounds of the frustum the

5| Implementation

projection matrix is associated with. We already have the *near* and *far* planes, but we lack the *right*, *left*, *top* and *bottom* limits. We can compute them starting from the horizontal and vertical fields of view, as it is possible to visualize in figure 5.8:

$$right = near \cdot \tan\left(\frac{fovX}{2}\right)$$

$$left = -right$$

$$top = near \cdot \tan\left(\frac{fovY}{2}\right)$$

$$bottom = -top$$

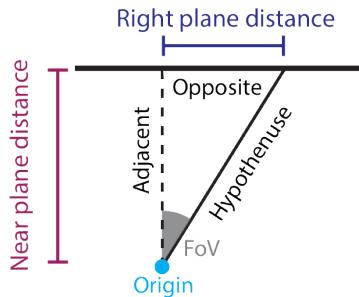


Figure 5.8: Geometrical visualization of how to compute the right plane starting from the horizontal field of view and the near plane. $\tan(\alpha_{FoV}) = \frac{\text{opposite}}{\text{adjacent}}$

Now we can calculate the perspective projection matrix [60]:

$$\begin{bmatrix} \frac{2 \cdot \text{near}}{right-left} & 0 & -\frac{right+left}{right-left} & 0 \\ 0 & \frac{2 \cdot \text{near}}{top-bottom} & -\frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & \frac{far+near}{far-near} & -\frac{2 \cdot far \cdot near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad (5.4)$$

getProjectedArea(aabb)

Differently from the orthographic case, there are no shortcuts to compute the area of the perspective projection of an AABB. We must compute the contour points of the projected hull, and calculate the area of the resulting polygon.

The first step is discussed in the next section; how to compute the area of a 2D polygon is detailed in section A.10.

5| Implementation

getProjectedHull(aabb)

The idea to compute the projected hull of an AABB in the case where the projection is a perspective, is extremely similar to the one discussed in section 5.4.1. Indeed, even in this case, we first have to detect what points of the AABB, after the perspective projection, will end up being contour points. The only difference is that now, this set of points can be computed by knowing the relative position of the eye and the AABB; in this case the forward direction of the *Pov* is irrelevant.

For this reason, we can use the same structure we used for *PlaneInfluenceAreas* with *PointInfluenceAreas* too, namely the *hullTable*. In this case, though, when we need to compute what index of the *hullTable* to read, we must use a different algorithm:

Algorithm 5.4 Given the eye position and the AABB, returns the corresponding index in the *hullTable*

```

1: function FINDHULLTABLEINDEX(eye, aabb)
2:   i  $\leftarrow$  0
3:   if eye.x  $<$  aabb.min.x then i = i|1
4:   else if eye.x  $>$  aabb.max.x then i = i|2
5:   if eye.y  $<$  aabb.min.y then i = i|4
6:   else if eye.y  $>$  aabb.max.y then i = i|8
7:   if eye.z  $<$  aabb.min.z then i = i|32
8:   else if eye.z  $>$  aabb.max.z then i = i|16
9:   return i
```

What the above algorithm does, is to check if the eye position is inside the min-max range of the AABB in each dimension. If it is inside in a specific dimension, we deduce that from the eye position we cannot see the faces that are perpendicular to the considered dimension direction. If we repeat this process for all three dimensions, we obtain the set of visible faces, and therefore the contour points. This idea can be visualized in figure 5.9.

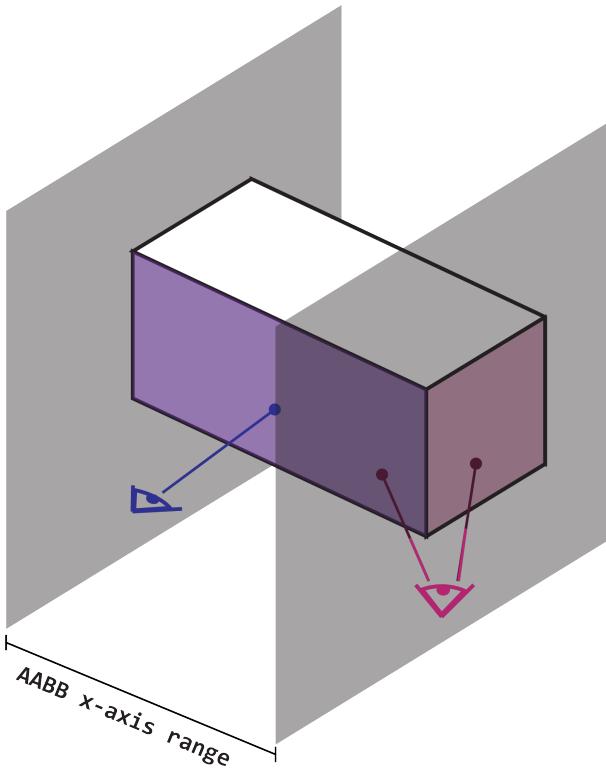


Figure 5.9: Considering the x dimension, the blue eye is inside the range of the AABB, whereas the purple one is not. The purple eye can also see the right face.

Once the contour points are detected, it is possible to project them with the view-projection matrix.

getRayDirection(aabb)

In case of radial rays distribution, it is not possible to obtain a univocal direction for the rays that hit an AABB, because an AABB is not a point-like entity. However, obtaining even an approximate rays direction is important in the splitting plane facing heuristic (SPFH), as described in chapter 4. We therefore decided, in our implementation, to approximate the direction of the rays that hit an AABB with the direction of the ray that hits the center of said AABB:

$$\text{ray direction} = \text{center}_{\text{AABB}} - \text{eye}$$

isDirectionAffine(ray)

As already discussed in section 3.1, to verify that a ray direction is affine to a point influence area, we can connect the origin of the ray with the eye position, and verify that



this vector is parallel to the direction of the ray. Also in this case it is possible to introduce a tolerance. This method only works if the origin of the ray is inside the frustum of the point influence area. However, since the first affinity condition is exactly this one, it is never a problem.

$$\text{origin}_{\text{ray}} - \text{eye} \parallel \text{direction}_{\text{ray}}$$

5.5. Top Level Structure

In this section we talk about the implementation details of the top level structure, the topic we described in section 3.3.

In summary, the top level structure is the component of our framework responsible to manage the influence areas in the scene and, given a 3D point, returning a list of influence areas the point belongs to. It is employed to verify the first affinity condition (section 3.1) whenever a ray is traced. It is therefore extremely important for the top level structure to be efficient in finding out what influence areas are present in a given position of the scene.

In our implementation we propose two top level structures. The first one has zero overhead during the building phase, but it is slower in returning the influence areas a point belongs to. The second one has some overhead during the building phase, but it is extremely fast during the traversal phase.

5.5.1. AABBs for Regions

The following top level structure has a very low complexity, and uses the technique presented in section 5.3.3 as its only optimization. In the AABBs for regions data structure, all the influence areas are stored in a plain array. During the traversal phase, the array is iterated over, and the *contains(point)* interface function of the region corresponding to the currently iterated influence area is used to check if the current point is affine to the influence area.

Since this top level structure uses a simple array, there is no overhead in creating it given the influence areas present in the scene. Moreover, it doesn't present any approximation of the 3D regions of the influence areas. On the other hand, employing it each time we have to detect what influence areas a ray is possibly affine with is not ideal, because the complexity of such a task grows linearly with the number of influence areas present in the scene. Furthermore, the algorithms of the *contains(point)* functions, discussed in

section A.7, are too expensive to be used in this scenario. The optimization of caching the AABB of the 3D region of the influence areas and using it for rejection tests improves performance, but doesn't change the algorithm complexity.

A more accurate analysis of the performance of this top level structure can be found in section 6.3.

5.5.2. Octree

The top level structure we present in this section is based upon an octree. As we have already explained in section 1.3.1, an octree is a tree where each node has exactly 8 children, which partition the volume enclosed by the father into 8 octants with the same shape and the same volume. An illustration of a quadtree (a 2-dimensional octree) can be found in figure 1.12.

Octrees are often used to partition 3-dimensional space recursively and adaptively, such that a leaf node contains the information about the volume of space it represents. This means that the volume of a leaf node must be uniform with reference to the information the octree is built upon. A classic example of the use of quadtrees, is in image compression [41]: the image is recursively subdivided into cells, until a cell contains only one color (figure 5.10). In this case, the color is the information upon which the quadtree is built, and it must therefore be uniform in leaf nodes.

5| Implementation

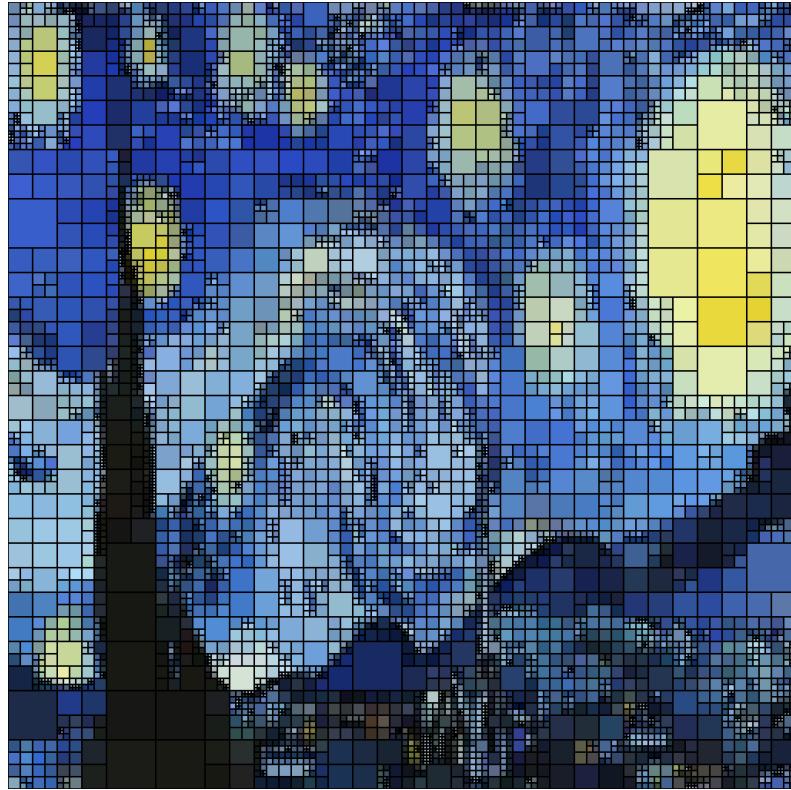


Figure 5.10: The image is recursively subdivided, until a cells contains pixels of just one color (or a maximum depth is reached).

Our idea is similar to the one of the image compression, however we work in a 3-dimensional environment, and the information upon which our octree is built are the influence areas' regions. We want our leaf nodes to enclose a volume in whose range there are the same influence areas' regions. This means that a leaf can enclose a volume in which no regions are present, a volume where there is just one region, or a volume where there is the same combination of regions, as it is possible to observe in figure 5.12.

5| Implementation

83

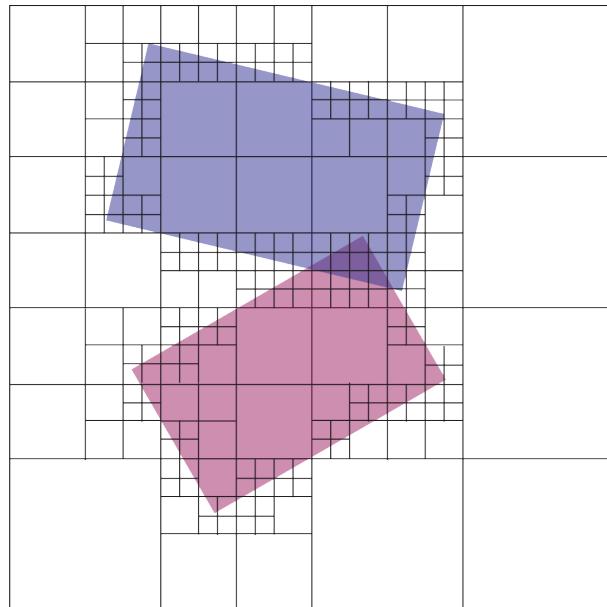


Figure 5.11: An example in 2 dimensions of a quadtree built upon influence areas.

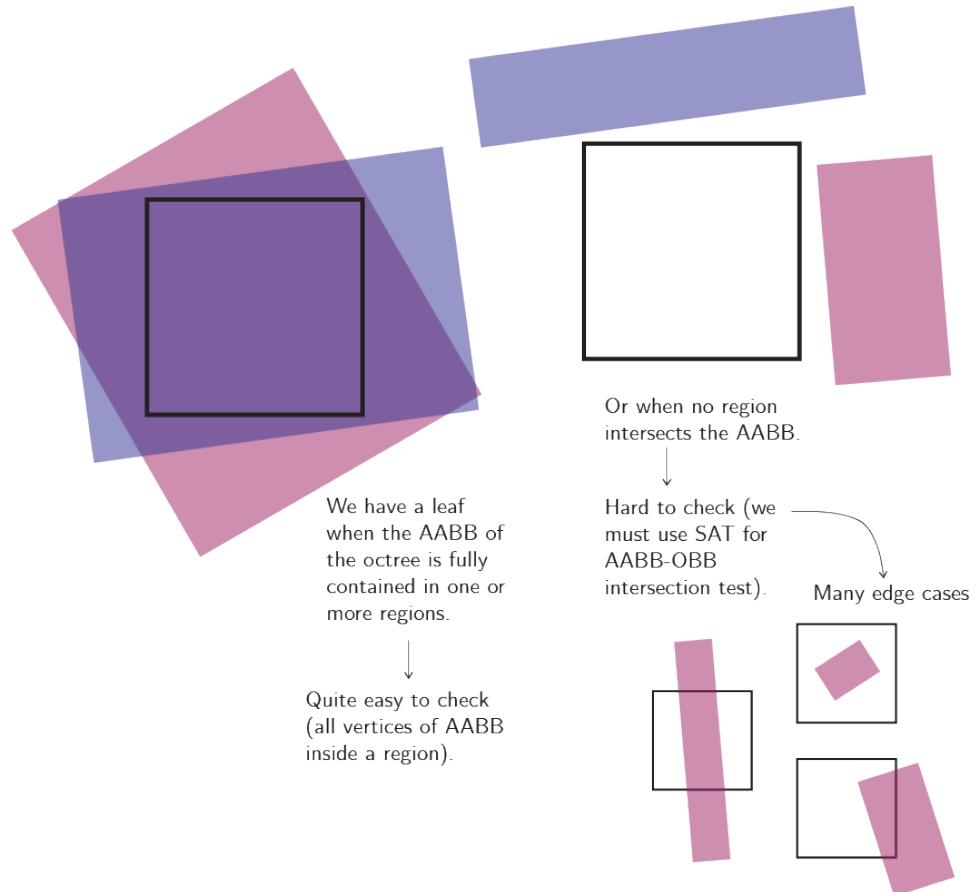


Figure 5.12: Different cases of the relative position of an octree cell and the influence areas' regions

5| Implementation

The algorithm for building the top level octree can be resumed in the following flowchart:

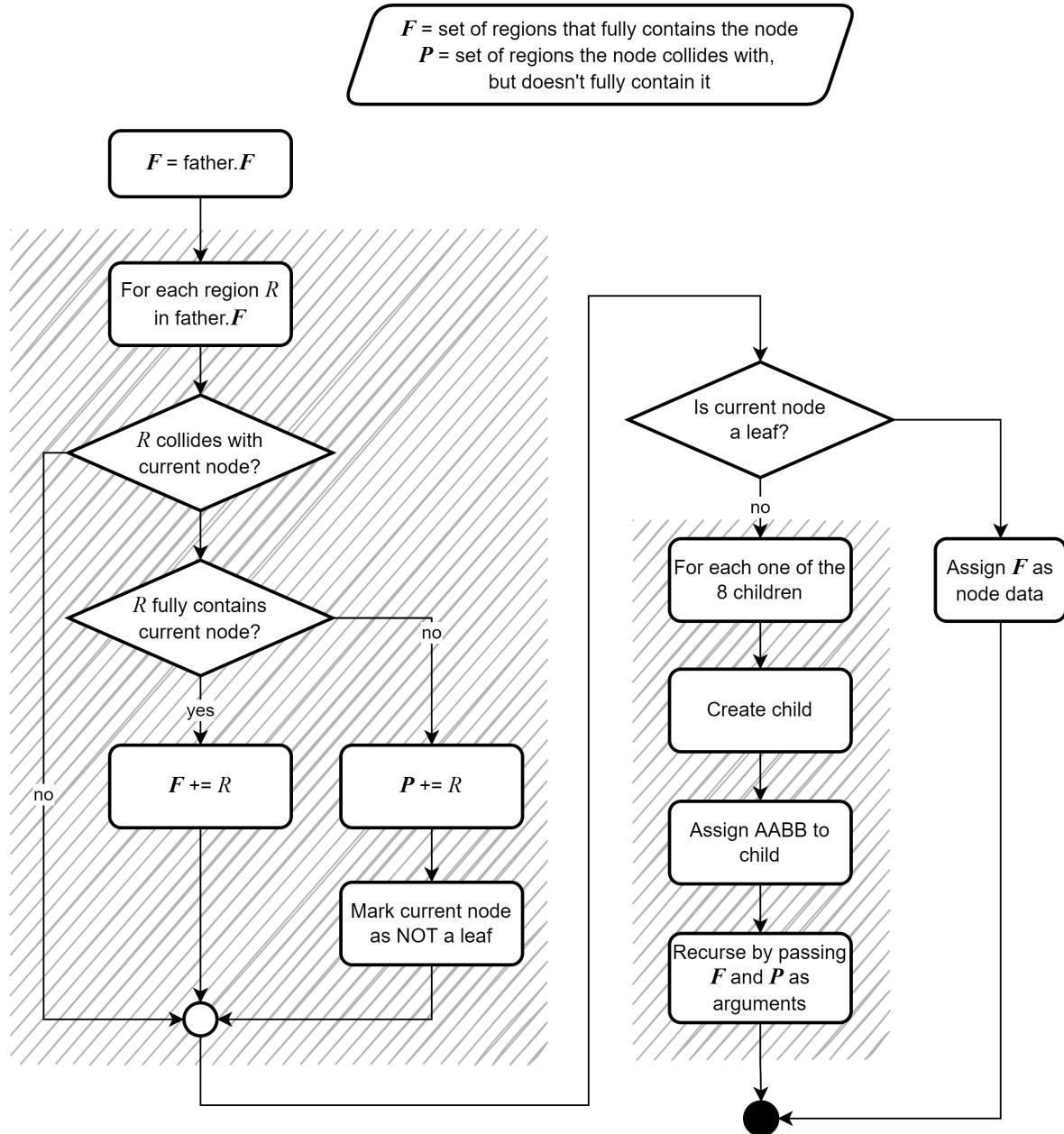


Figure 5.13: Algorithm to build the top level octree.

Each node of the octree has this structure:

```

class Node
  boundaries : Aabb []
  data : Bvh * []
  children : Node[8]
  isLeaf : bool
  
```

5| Implementation

It is possible to note that the region of 3D space pertaining to the node is represented by an *Aabb*, and that the data is a list of *Bvh* pointers, which are the BVHs associated with the influence areas present in the boundaries of the node. Only leaf nodes have a filled *data* member, whereas internal nodes have no *data*.

The *children* array has an indexing following the layout in the following figure:

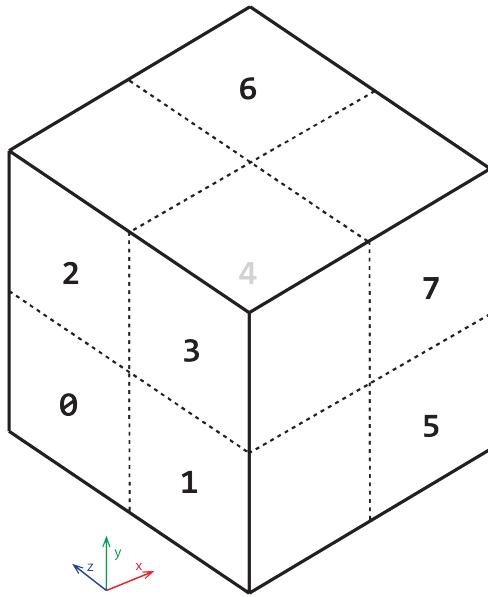


Figure 5.14: The indexing layout of the children of an octree node.

In order to detect if a region R is overlapping with the current octree node, represented by an AABB, it is possible to use the algorithm detailed in section A.5, which uses the separating axis test.

To check if a region R fully contains the current node's AABB, we can use the *Region* interface method called *fullyContains(aabb)* (section 5.3). In particular, *Regions* can be *Obbs* in case of plane influence areas, or *Frustums* for point influence areas.

In the algorithm presented in figure 5.13 we stop when a node is marked as a leaf, and a node is marked as a leaf when it is fully contained by all the influence areas it overlaps with. Since the nodes are represented by AABBs, and the influence areas by OBBs and frustums, it is many case impossible to reach a point where the recursive calls of the algorithm end. It would require an infinite amount of subdivisions. For this reason, in our implementation, it is possible to set a maximum depth for the top level octree. Once the maximum depth is reached, a node is by definition a leaf node, and it contains as *data* all the influence areas that even partially overlap with it. This means that in our implementation we are using a conservative approximation of the influence areas: influence

5| Implementation



areas as represented in the octree can have a bigger volume than the real influence areas, but never a smaller one. Consequently, it is possible that a ray is considered affine to a certain influence area even though it doesn't respect the first affinity condition (section 3.1). As we will analyze in chapter 6, this is not a problem. Indeed we will experimentally show that even with a low maximum depth of the octree, it is possible to achieve the desired results during ray-scene traversal.

Once the octree is built, it is possible to use it to detect which regions are present in a given point of the scene.

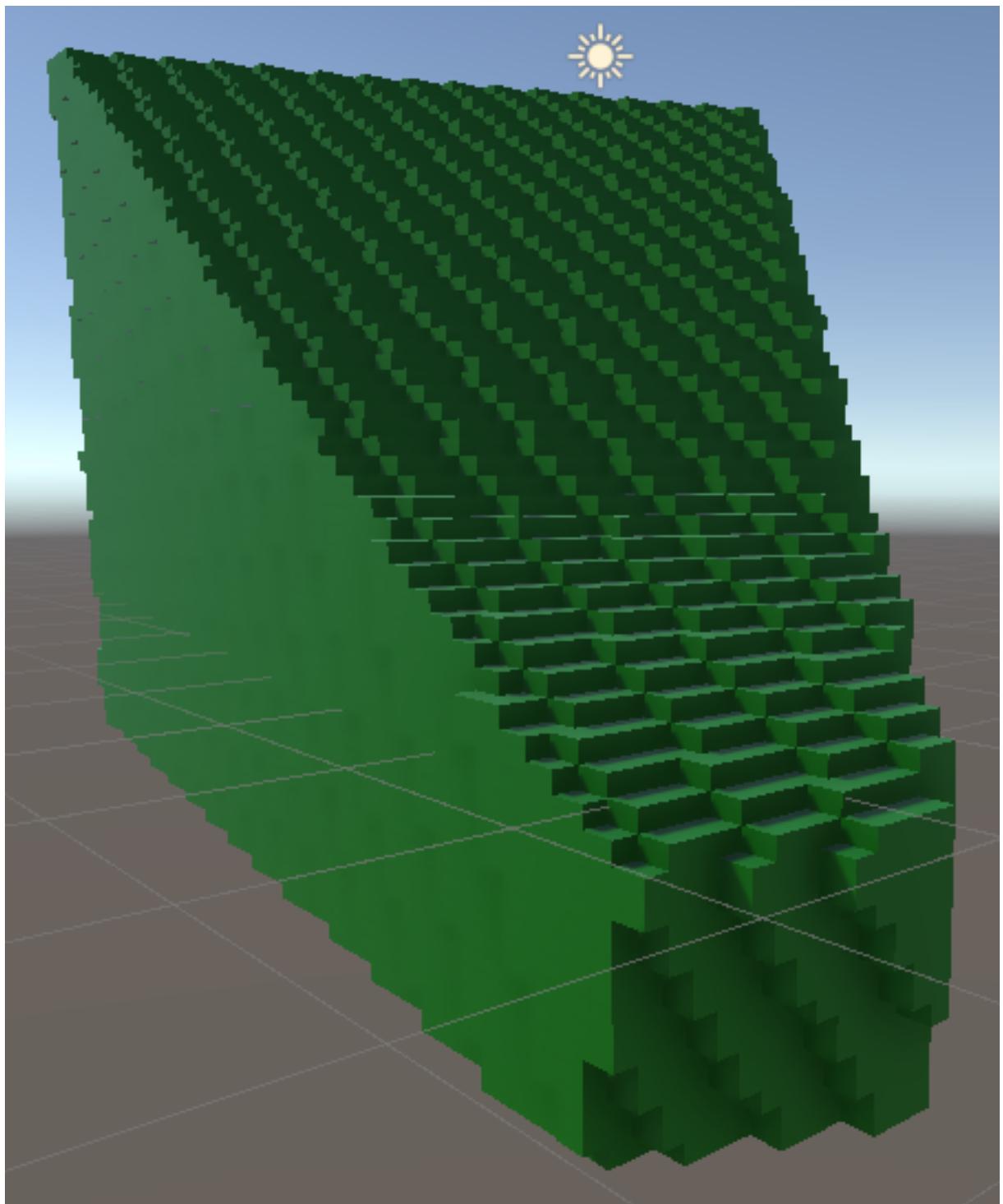


Figure 5.15: 5-level octree built on a single point influence area.

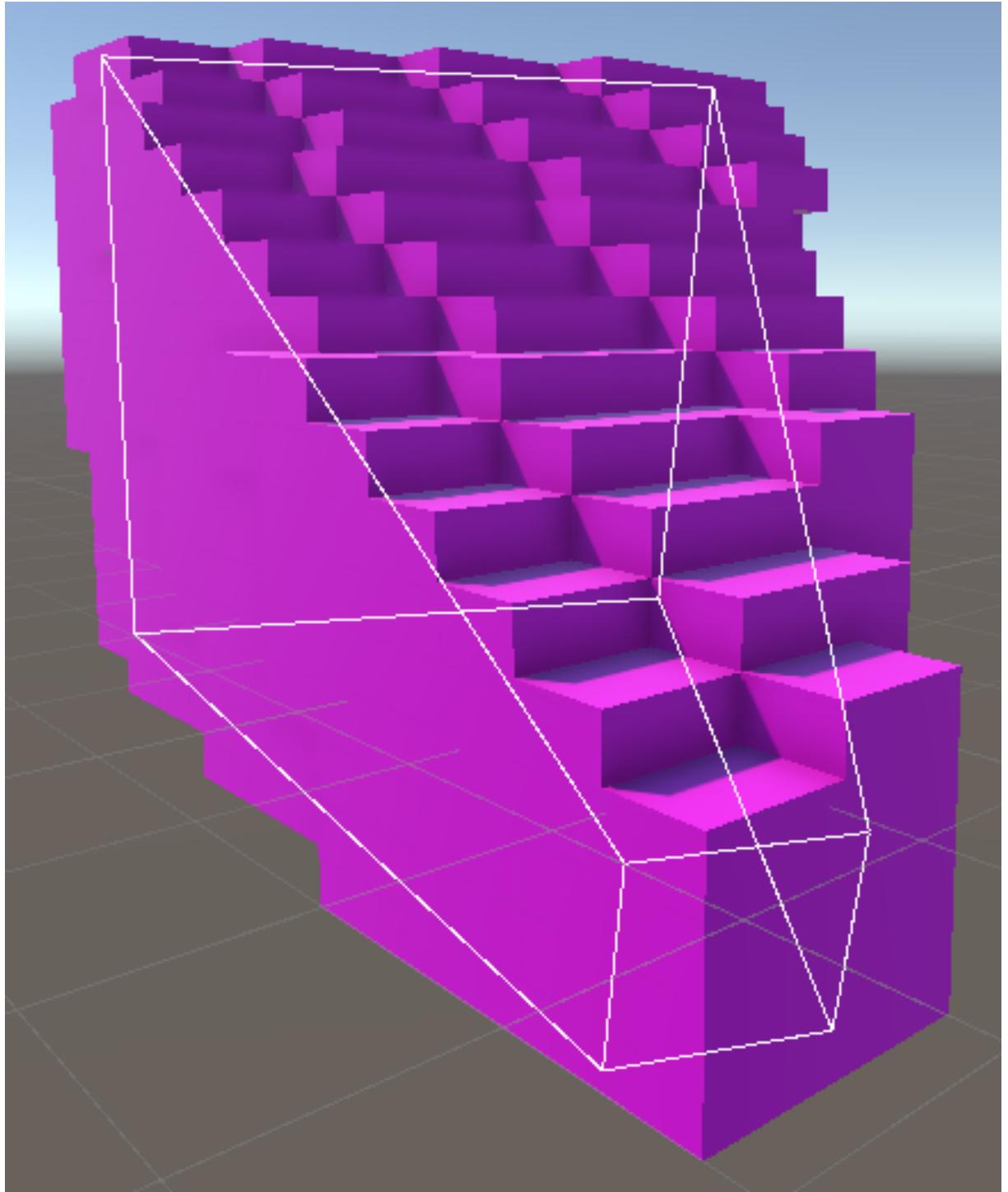


Figure 5.16: 4-level octree built on a single point influence area.

5.6. Additional Structures

In the previous sections of this chapter we described the main structures used in our framework to test the validity of the theoretical claims of this thesis. In the framework

5| Implementation

many other structures and algorithms are present. However, since these other structures are not as strongly related to the theory of our thesis as the ones we described above, we will only briefly list some of them and their responsibilities.

More information about their usage can be found at `TODO`.

5.6.1. Test Scene

In the previous sections we have seen what are the structures used to represent a BVH, an influence area and a top level structure. The *TestScene* class is the glue among all these other structures. It is possible to create a *TestScene* by specifying the *InfluenceAreas* present in the scene, the top level structure type to use, and a list of *RayCasters* (section 5.6.2).

The method *build(triangles)* can then be used to build the top level structure based on the influence areas. After the top level structure is built, all the BVHs associated with the influence areas are built, by considering the array of triangles passed as argument as the geometry present in the scene. Eventually the global PAH BVH is built. It is also possible to specify a *BvhAnalyzer* (section 5.2) to get information about the BVHs built.

The method *traverse()* can be used to cast some rays based on the *RayCasters* specified during construction, and collect the results.

5.6.2. Ray Caster

A *RayCaster* is responsible to cast rays in the scene. It is possible to specify a region of the scene where the rays should have origin, a set of directions, and a random distribution to sample rays.

Some relevant *RayCasters* are:

Plane ray caster Rays start from a rectangular section of a plane, and are perpendicular to it (up to a customizable tolerance).

Square sphere cap ray caster Rays start from a point and are cast toward a square sector on the unit sphere surface. Our visual interactive representation can be tried at <https://www.geogebra.org/m/bzswwgvj>.

Circular sphere cap sampling Same as above, but the sphere cap the rays are cast toward is circular. Our visual interactive representation can be found at <https://www.geogebra.org/m/aqcepkrz>.

5| Implementation



It is possible to specify a point in space and a facing direction for any *RayCaster*, along with an amount of rays to generate.

5.6.3. CSV Exporter

CsvExporter class objects can be used to specify some of the attributes present in the JSON relative to the BVHs or the top level structure, and export it in the CSV format. Since the JSONs relative to the top level structures can become quite big (in our test cases we obtained JSONs with more than 20000 lines), having a class to extract only the relevant information for a specific analysis has proven invaluable during the experimentation phase.

6 | Experimental Results

In this chapter we will present the results we obtained by performing experimental tests on our theories. The experiments have been carried out by using our framework, presented in chapter 5. Our experiments aim at simulating some possible situations our techniques may face in a real-world application. We will detail our methodologies and the reason behind the choice of the selected test cases. We also try to provide mathematical indicators, or at least empirical reasoning, about why a certain method works better than another one, given a specific scenario.

Our tests are aimed at comparing how our heuristics change the behavior of a simple and standard BVH builder, compared to the state-of-the-art heuristics, and based on the BVH properties we adopt (section 5.1). This means that our focus during the testing phase is not to build the fastest possible BVH builder. Our reasoning behind this choice, is that we want to reduce complexity at the bare minimum, in order to be able to isolate what the real focus of this work is, namely the advantages and disadvantages of PAH and SPFH compared to SAH and LSPH.

If we tried to build an extremely performant BVH builder, we would have needed to introduce a lot of low-level optimizations. For example, we would have needed to take into account memory access patterns to increase data locality, as introduced in this paper [58]; or we would have needed to build an algorithm running on the GPU. Getting these optimizations right would have required a lot of very specific expertise and trial and error, therefore we would have needed to use an existing BVH builder and modify it, losing the full control over it, which was one of our guiding principles. Furthermore, introducing low-level optimizations can often lead to unintuitive behaviors and difficult to correctly analyze scenarios, which would have made it more difficult to carry out a really objective comparison between the heuristics. Most of these optimizations are not strictly related to a specific way a BVH is built, and can therefore be implemented for state-of-the-art BVH builders as well as for ours.

6| Experimental Results

6.1. Test Scenes

We will now present the eight test scenes and the reasons why we selected each one of them.

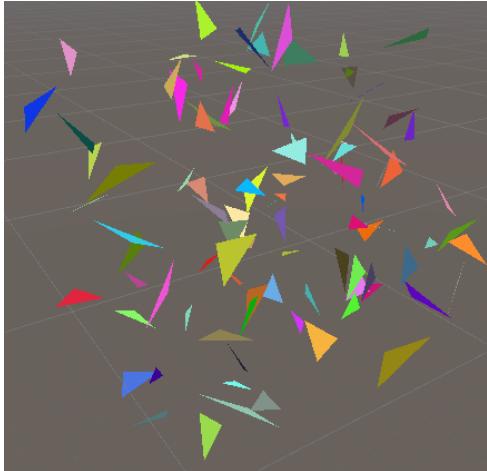


Figure 6.1: Random 100

100 triangles placed in a random uniform pattern in a parallelepiped volume. Triangles can overlap, and are similar in size. The main goal of this scene is to study PAH in a uniform scene, and SPFH in a scene where it is difficult to cut off space.

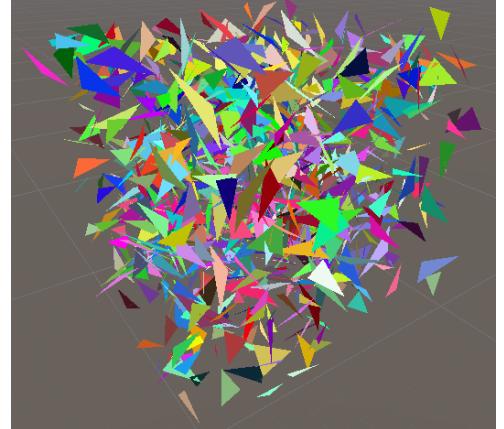


Figure 6.2: Random 1000

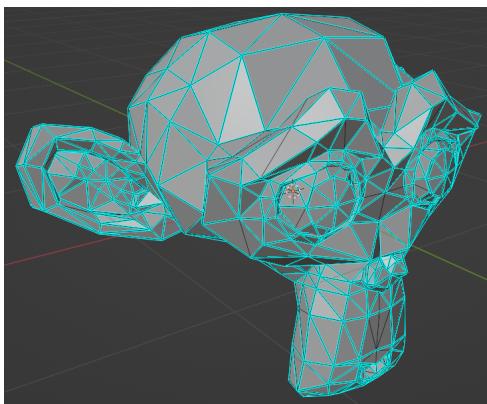


Figure 6.3: Suzanne

Suzanne represents a small and low-poly object, with continuous surfaces almost everywhere. The goal is to try PAH on stand-alone objects, and SPFH again in a scenario where it is difficult to effectively cut off space.

6| Experimental Results

The cottage presents a scene with a big but regular object. Triangles can reach very big dimensions, therefore we use it to test PAH in a scenario with a few big triangles. The same challenge is presented for SPFH: the presence of big triangles makes it difficult to create splits with small overlaps.

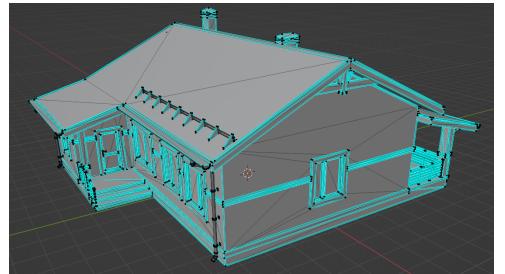


Figure 6.4: Cottage

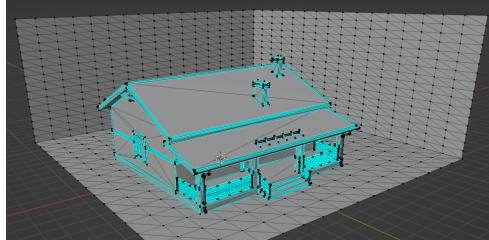


Figure 6.5: Cottage Walls

Exactly the same scene as above, but this time all the rays hit a geometry primitive without the need of using the fallback BVH. The aim is to test how much using the fallback BVH deteriorates the performance of our technique.

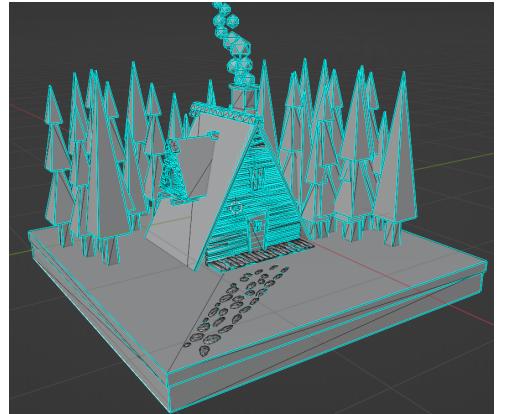


Figure 6.6: Wood

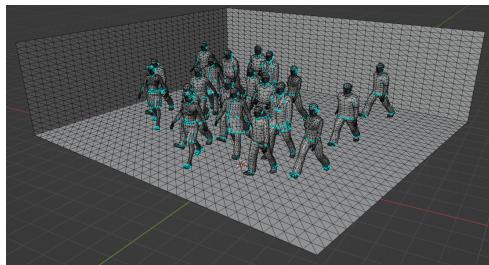


Figure 6.7: Crowd

The crowd test scene presents many human-like people, with a rather high polycount. The scene was primarily used to observe the behavior of SPFH in a detailed scene, and also to analyze PAH in a scenario with many organic and complex shapes, where AABBs tend to overlap.

6| Experimental Results

Sponza is a scene with a very high polycount compared to the previous ones. It presents the interior of a building, where there are detailed objects with various forms. This is the test case that most closely simulates a real-world scenario.

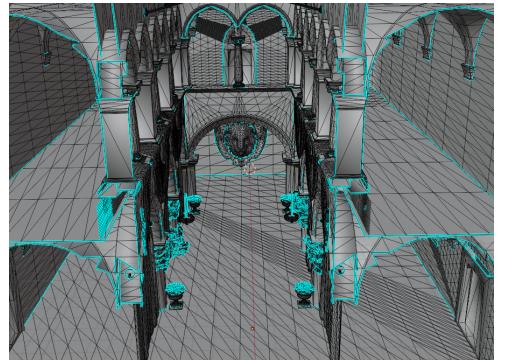


Figure 6.8: Sponza

We decided to test each one of these scenes in a scenario with a single influence area covering most of the geometry. The influence areas we used were both plane and point influence areas, and we decided to carry out tests with different orientations of the influence areas. In particular, we chose four different orientations for each scene and for both plane and point influence areas:

Parallel In this case the main rays direction of the influence area is parallel to one of the three cartesian axes. This scenario should, in principle, be the best one in general for AABB-based BVHs, and in particular for SPFH.

15 degrees The main rays direction of the influence area is parallel to one of the three cartesian planes, and forms an angle of 15 degrees with one of the two axes generating the parallel plane. In this scenario, SPFH has a clear best splitting plane (parallel to the rays), and has a clear second-best option (the plane forming a 15 degrees angle with the rays).

45 degrees The main rays direction of the influence area is parallel to one of the three cartesian planes, and forms an angle of 45 degrees with one of the two axes generating the parallel plane. SPFH has only one clear best direction for splitting a node (parallel to the rays). This means that, based on the BVH properties chosen (section 5.1), it may often happen to use the fallback strategy, namely SAH and LSPH.

Oblique The main rays direction is not parallel to any of the three cartesian planes. This scenario is the most challenging for SPFH, because it is possible that all three splitting planes are not a good option. For this reason, this scenario can be the best suited to tune the properties of the BVH.



6.2. Comparison with State of the Art Heuristics

In this section we will present the tests we performed to compare our heuristics with state-of-the-art heuristics. We will explain why we carried out each experiment and comment on the results.

6.2.1. PAH versus SAH Precision

The first test we performed is aimed at comparing the precision of projected area heuristic to surface area heuristic. As explained in section 5.1.2, we have developed a function to compute the real cost of traversing a BVH for a specific ray. We can then use this method to compute the average cost of traversing the BVH for a certain amount of rays.

In this test, we build a BVH with PAH over a certain influence area. Then, we use a ray caster (section 5.6.2), associated with the same influence area, to generate the rays to traverse the scene with. We use these rays to compute the average real traversal cost of the BVH, and compare it with the cost predicted by using PAH. We express this comparison as percentage error:

$$\text{error}_{PAH} = \left| \frac{\text{real cost} - \text{estimated cost}}{\text{estimated cost}} \right|$$

Then we build a BVH with SAH, and use the same ray caster to traverse it. We compute the average real traversal cost and the cost estimated by SAH. We then compute the error_{SAH} in the same way as above.

To present the results, we decided to aggregate data based on the influence area used (plane or point), and the orientation of the influence area (parallel, 15°, 45° and oblique).

6| Experimental Results

96

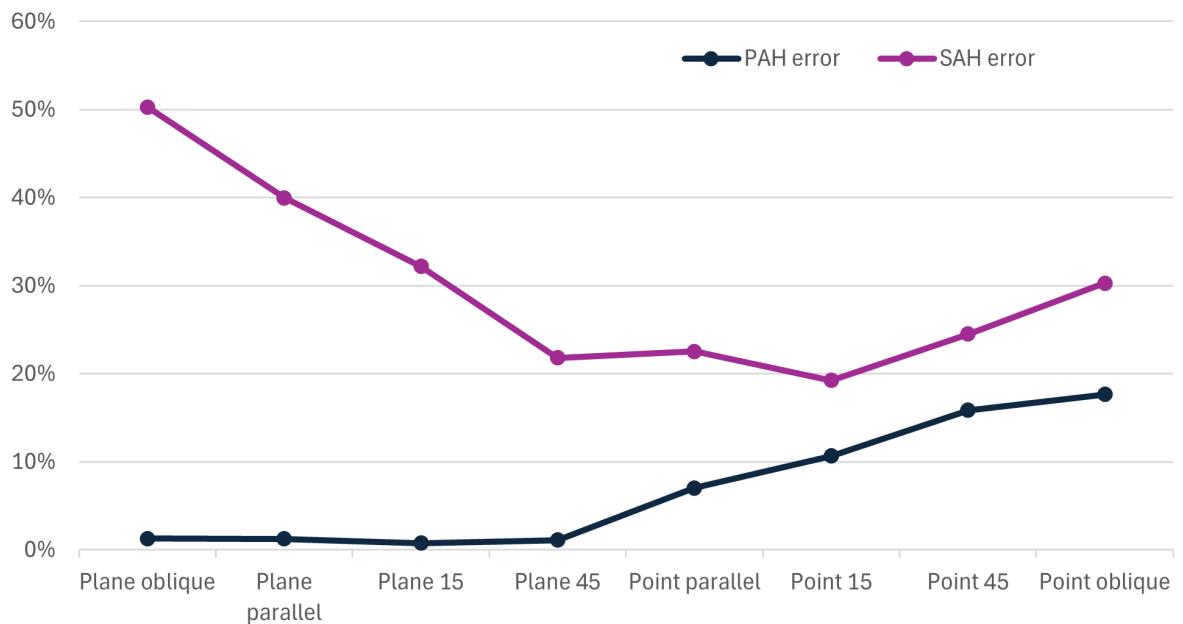


Figure 6.9: Accuracy of PAH compared to SAH.

As it is possible to observe from the chart 6.9, the error of PAH is always smaller than SAH's. On average PAH error is 7%, and SAH's 36%.

It is also possible to observe that PAH is more precise when the influence area is a plane influence area. In this case the average error is 2%, compared to the 12% error in case of point influence areas.

The results we obtained are in line with our thesis: since in the test scene the ray distribution is not uniform, but follows a parallel or radial pattern, the second SAH hypothesis doesn't hold. Our heuristic, instead, uses as ray distribution hypothesis that rays naturally organize themselves in parallel or radial patterns, due to light importance sampling (section 1.2). Since, in our test cases, we artificially added parallel and radial distributions, PAH is able to estimate better the cost of the BVH.

6.2.2. Projected Area Culling

During the first iteration of the previous test, we got worse results for PAH: the average error for scenes with plane influence areas was 12%, and with point influence areas 23%. The reason behind these results was a subtle error in the way the projected area of the AABBs on the near plane of the viewing volume was computed.

As we have explained in sections 5.4.1 and 5.4.2, we can use orthographic and perspective projections to project the AABB to the near plane. However, the 2D hull resulting from

6| Experimental Results

the projection, can have some of its parts outside the near plane, as it can be observed in figure 6.10.

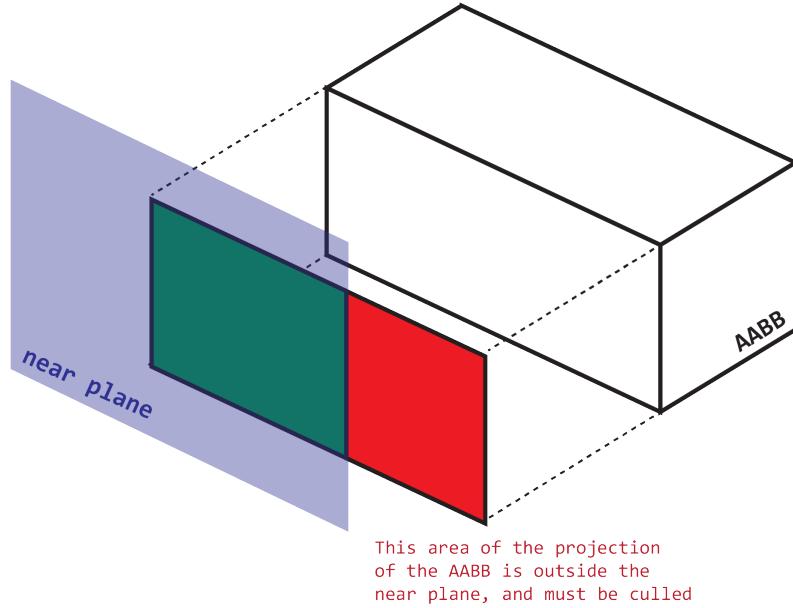


Figure 6.10: The 2D projected hull of the AABB is not fully contained in the viewing area of the near plane.

This leads to calculating projected areas that are bigger than the area of the near plane. This eventuality drives to errors in computing the hit probability (section 1.3.3) of the AABBs of the BVH. Indeed, the parts of the 2D hull that are outside of the near plane, cannot by definition be hit by a ray affine to the influence area associated to their BVH. However, by including them in the computation of the projected area of the AABB, they actually contribute to the calculation of the hit probability of the AABB, raising its value.

In order to mitigate this issue, we need to find a way to exclude the parts of the projected hull of an AABB that are outside the near plane. To do so we can apply culling, explained in section A.9. When we compute the projected hull of an AABB, before calculating its area, we cull it with the near plane. The near plane vertices can be easily computed, indeed, since after their projections the vertices of the AABB are in clipping space, then the near plane is a unit square.

6.2.3. PAH and SPFH versus SAH and LSPH

With the tests present in this section, we compare the building and traversal of BVHs built with the two heuristics we propose in this work, to BVHs built with the-state-of-the-art heuristics, namely surface area heuristic (SAH) and longest splitting plane heuristic

6| Experimental Results



(LSPH).

The set-up for this test is exactly the same described in section 6.2.1. We build a BVH with PAH and SPFH over a certain influence area. Then, we use a ray caster (section 5.6.2) associated with the same influence area to generate the rays to traverse the scene with. We use these rays to compute the average real traversal cost of the BVH. Eventually, we build a BVH over the same geometry with SAH and LSPH, and compute the average real cost in the same way. We then compute these values:

- How much faster, in percentage, the BVH built with PAH and SPFH is to traverse compared to the state-of-the-art one: $faster_{PAH+SPFH} = \frac{cost_{SAH+LSPH} - cost_{PAH+SPFH}}{cost_{PAH+SPFH}}$. This value assumes positive values if the BVH built with our heuristics is faster to traverse, and negative ones otherwise.
- How much, in percentage, our *full structure* is faster than the state-of-the-art BVH. With the term *full structure* we refer to the process of traversing the PAH and SPFH BVH, and, possibly, traversing the global fallback BVH (chapter 3), in case the search in the PAH BVH hasn't found any intersection: $faster_{full} = \frac{cost_{SAH+LSPH} - (cost_{PAH+SPFH} + cost_{fallback})}{cost_{PAH+SPFH} + cost_{fallback}}$.
- The average time per node to compute its cost, both for PAH and SAH.
- The average time per node to compute the best splitting plane, both for SPFH and LSPH.

$faster_{PAH+SPFH}$ and $faster_{full}$ are indicators agnostic to the hardware used. Whereas the average times depends on the hardware on which the algorithm is run on. Another option we took into consideration was to perform a low level analysis on the assembly instructions each heuristic is translated into by the compiler. However, as we already stated in the introduction to this chapter, our focus is to compare the heuristics, therefore we are not interested in the absolute values, but in their relative magnitudes. For this reason, we opted to use the raw run time as indicator.

The properties (section 5.1) for the PAH BVHs are set to default values we found out work well empirically. In section 6.4 we carry out a comparison among different choices of these values:

6| Experimental Results



property	value
<i>maxLeafCost</i>	0
<i>maxLeafArea</i>	0
<i>maxLeafHitProbability</i>	0
<i>maxTrianglesPerLeaf</i>	2
<i>maxLevels</i>	100
<i>bins</i>	40
<i>maxNonFallbackLevels</i>	100
<i>splitPlaneQualityThreshold</i>	0.4
<i>acceptableChildrenFatherHitProbabilityRatio</i>	1.3
<i>excellentChildrenFatherHitProbabilityRatio</i>	0.9

In this section we mainly take into account the $faster_{PAH+SPFH}$ indicator, as the $faster_{full}$ is analyzed in section 6.2.5.

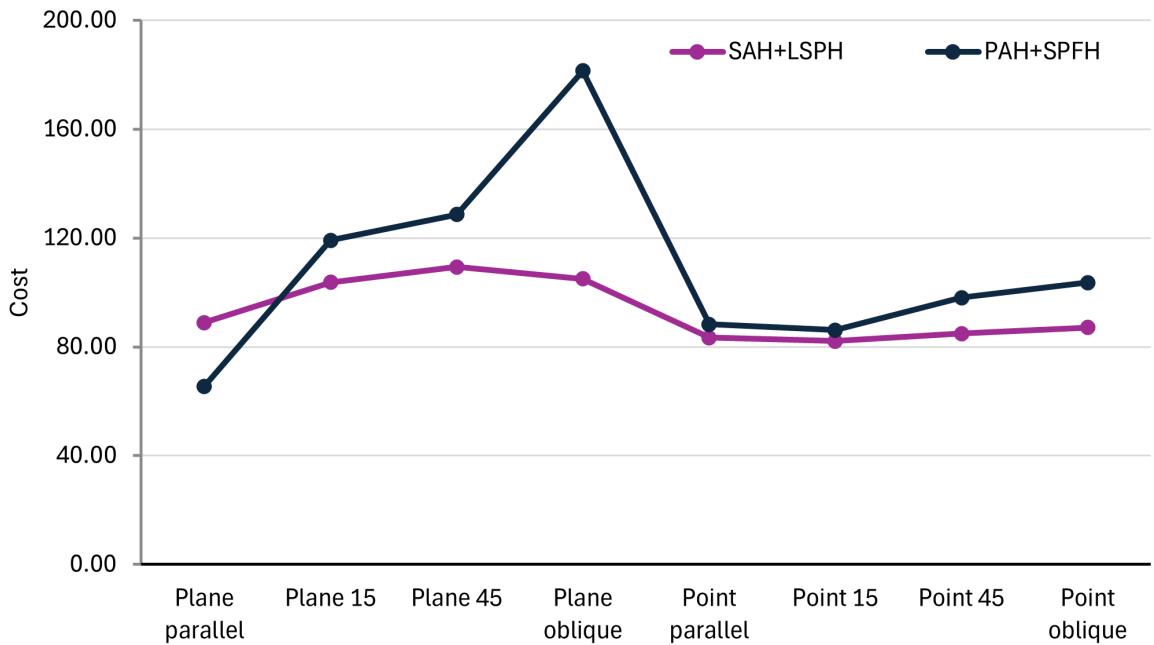


Figure 6.11: Cost of a BVH built with PAH and SPFH compared to one built with SAH and LSPH.

As it is possible to observe in the chart 6.11, the results of our combined heuristics are better than state of the art heuristics ones only in the test case of the plane influence area with rays parallel to a cartesian axis. Results are comparable to state of the art ones also in the case of rays *almost parallel* to a cartesian axis. The most relevant reason behind these results is overlapping, and we will talk about it in section 6.2.8.

6| Experimental Results



Another way of looking at this data, is to compare the average number of intersections with nodes and triangles each ray has during the traversal of the BVH. The presented numbers of intersections per ray is referred to the total number of nodes and triangles a ray collides with, and doesn't stop when the closest hit is found. This allows us to estimate better some qualities of the BVH, such as the effect overlapping has on it, and is less susceptible to the particular layout of a scene (such as a scene with a blocking object near the camera).

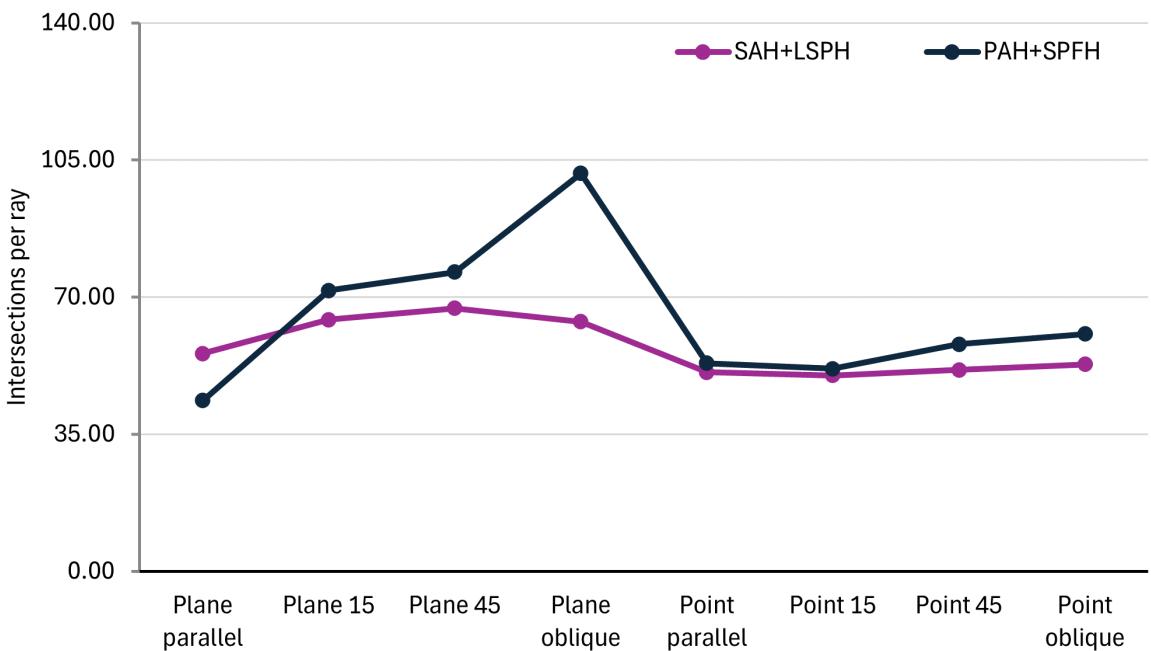


Figure 6.12: Number of intersections per ray during traversal.

As it is possible to see, the data on the intersections very closely approximates the data on the BVH cost, while being a less abstract concept to understand. These results can only be obtained in a portion of the scene where a parallel or point distribution is present.

6.2.4. Building Performance

For what concerns the building phase of the BVHs, the timing results can be observed in charts 6.13 and 6.14. We measured the time of each relevant phase of the BVH building by storing the timestamp of entrance in the relevant section, and subtracting it by a timestamp obtained at the end of the profiled section.

6| Experimental Results

101

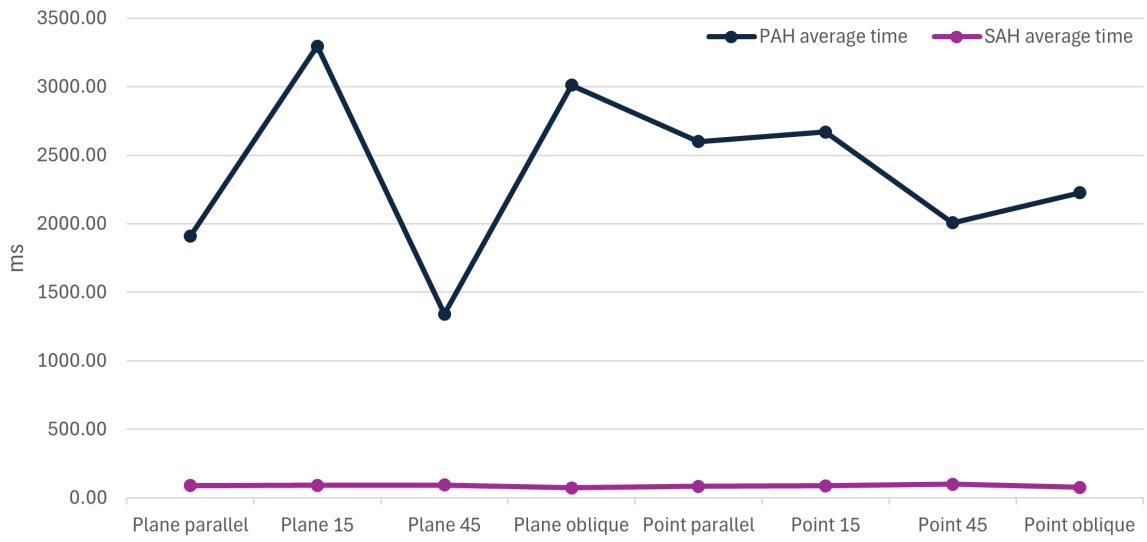


Figure 6.13: Average time per node spent inside the compute cost function for PAH and SAH.

As it is possible to observe, PAH is on average an order of magnitude slower than SAH. This is expected, since SAH can be computed with 3 multiplications and 3 sums, whereas with PAH a projection of at least 4 3D points and 3 cross products are needed, as explained in section 5.4.1. The dip in the planar 45 degrees case can be explained by remembering how algorithm works: in the 45 degrees case there is only one relevant plane by which to split the triangles of a node (chapter 4). This means that the fallback method, namely SAH, is used more frequently than in other scenarios; since SAH is faster to compute than PAH, the average time for the BVH is reduced.

It is important to remember that our implementation runs on the CPU and is not optimized. However, considering the nature of the operations used in PAH, namely projections obtained with matrix multiplications, it is reasonable to think that its performance could improve if it was tailored and run on a parallel architecture, like a GPU. At the moment we don't have any experimental evidence this is the case, and this is only a supposition based on an analogy with the algorithms that work best on a GPU. A future direction of development should be to port the algorithm to compute PAH on a GPU architecture.

For what concerns SPFH, it is possible to note from chart 6.14 that its performance is in line with the performance of LSPH. Also in this case, this is an expected outcome, since, in both algorithms, 3 divisions and 3 comparisons are needed to compute the quality values of each splitting plane, as explained in chapter 4.

6| Experimental Results

102

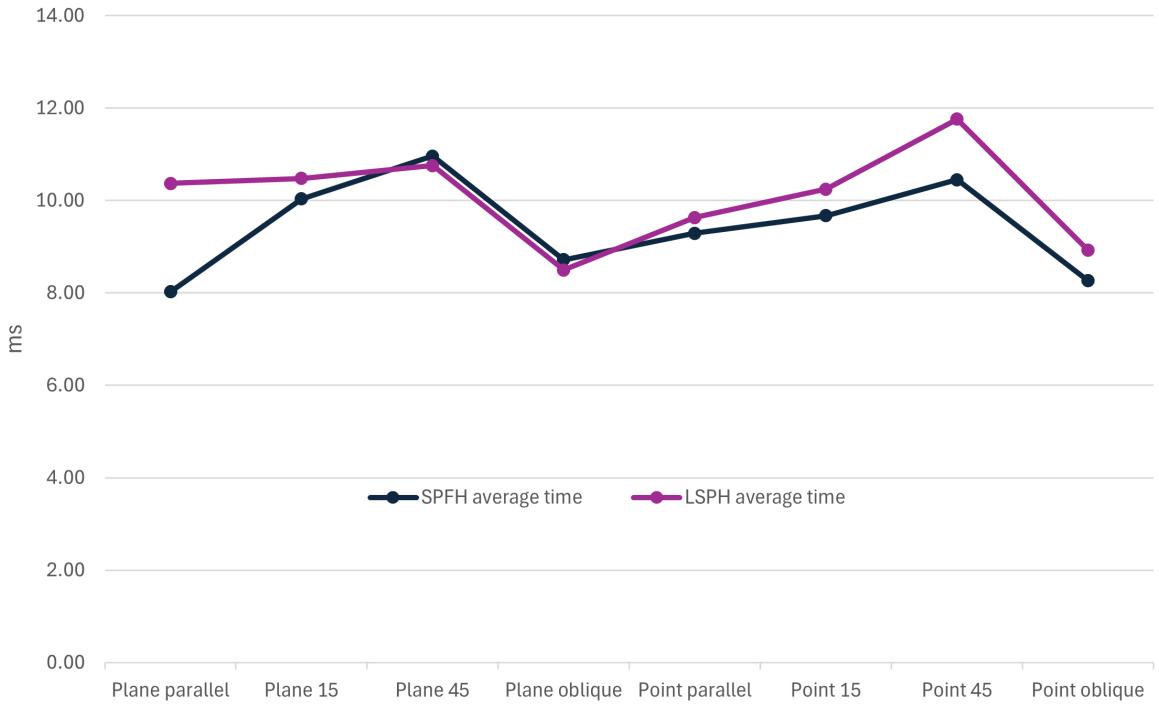


Figure 6.14: Average time spent inside the compute splitting plane function for SPFH and LSPH.

We also considered to analyze the amount of splitting plane orientations that are attempted at each node with SPFH and a *splitPlaneQualityThreshold* of 0.4, as stated in table 6.2. On average with SPFH 2.1 splitting plane orientations are attempted. Whereas, with LSPH, 1.5 splitting planes orientations are attempted. With LSPH we adopted a *splitPlaneQualityThreshold* of 0.99.

In section 6.4 we will analyze how the *splitPlaneQualityThreshold* and other properties affect the building speed and traversal performance of the BVHs.

6.2.5. Breakeven Hit Percentage

In the previous section we focused on the raw performance of BVHs built with PAH and SPFH, by analyzing the values of the $faster_{PAH+SPFH}$ indicator in various situations. However, in a real case scenario, when a search in the BVH associated with an influence area fails, a search in the global BVH must be performed, as described in section 3.2. For this reason, the *faster full* indicator is better at predicting if BVHs built with our heuristics are actually faster to traverse than BVHs built with SAH and LSPH.

We found out that the results are directly related to how many of the rays hit the geometry present in the BVH associated with the influence area. We therefore attempted to formalize this concept, and find a formula to detect when it is worth it to build a local

6| Experimental Results

BVH based on the hit percentage:

$$\begin{aligned}
 & \#rays \cdot cost_{PAH+SPFH} + \#rays \cdot (1 - hit\%) \cdot cost_{SAH+LSPH} < \#rays \cdot cost_{SAH+LSPH} \\
 & cost_{PAH+SPFH} + (1 - hit\%) \cdot cost_{SAH+LSPH} - cost_{SAH+LSPH} < 0 \\
 & cost_{PAH+SPFH} - hit\% \cdot cost_{SAH+LSPH} + cost_{SAH+LSPH} - cost_{SAH+LSPH} < 0 \\
 & cost_{PAH+SPFH} < hit\% \cdot cost_{SAH+LSPH} \\
 & hit\% > \frac{cost_{PAH+SPFH}}{cost_{SAH+LSPH}}
 \end{aligned}$$

In the first line, on the left we sum the cost of the rays traversing only the BVH associated with the influence area, and the rays traversing it and the global BVH; on the right there is the cost of the rays only traversing the global BVH built with SAH and LSPH. Using the local BVH is worth it only if the quantity on the left is smaller than the quantity on the right. We find out that, it is worth it to use a local BVH, only if the hit percentage of the rays with geometry part of the local BVH is bigger than the ratio between the cost of the local BVH and the cost of the global BVH.

This formula is useful at approximating the required hit percentage, because all the needed parameters can be computed before traversing the BVH. Indeed it is possible to use the estimated costs of the local and global BVHs to get the breakeven hit percentage. In section 6.2.1 we showed how good is PAH at predicting the cost of a BVH.

However, the main flaw of this formula is that we assume that the cost of traversing the global BVH for the rays that miss the local one, is the same as the cost of the full global BVH. This is not always the case, indeed, in many scenarios, the rays that miss the local BVH will have some peculiar characteristics. For example, as it is possible to observe in figure 6.15, all they rays missing in the local BVH are concentrated in a section of the scene, and therefore hit a specific part of the global BVH, which cannot be accurately approximated by the whole BVH.

6| Experimental Results

104

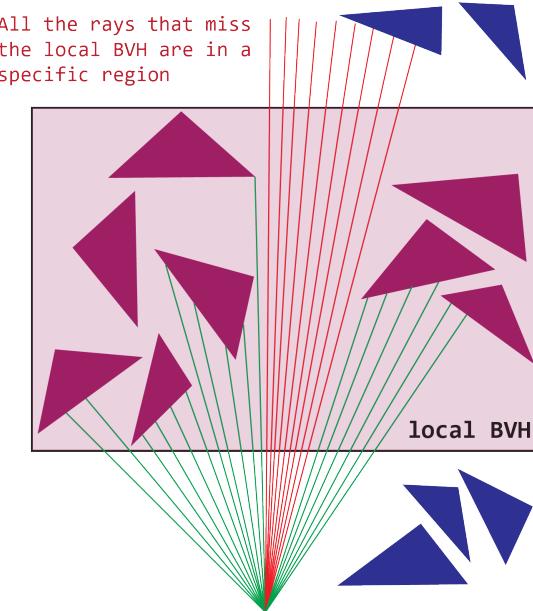


Figure 6.15: The rays that miss in the local BVH are all concentrated in a section of the scene.

For this reason, instead of using $cost_{SAH+LSPH}$ in the formula to compute the breakeven hit percentage, we would need to use a more specific cost value, computed only for the rays that miss the local BVH. We call this value $cost_{miss}$:

$$\begin{aligned} \#rays \cdot cost_{PAH+SPFH} + \#rays \cdot (1 - hit\%) \cdot cost_{miss} &< \#rays \cdot cost_{SAH+LSPH} \\ cost_{PAH+SPFH} + (1 - hit\%) \cdot cost_{miss} - cost_{SAH+LSPH} &< 0 \\ cost_{PAH+SPFH} - hit\% \cdot cost_{miss} + cost_{miss} - cost_{SAH+LSPH} &< 0 \\ hit\% > \frac{cost_{PAH+SPFH} + cost_{miss} + cost_{SAH+LSPH}}{cost_{miss}} \end{aligned}$$

As it is possible to observe in chart 6.16, this new formula is accurate at predicting the breakeven hit percentage value. As the difference between the cost of the traversal of the scene with our method and the state-of-the-art method comes close to 0, the delta between the calculated breakeven hit percentage and the measured hit percentage tends to go to 0 too. For example, for the test case of the *Wood* scene with a point influence area with rays parallel to a cartesian axis, the difference in traversal cost between our method and the state-of-the-art method is 0%, and the difference between the computed breakeven hit percentage (55%) and the measured hit percentage (56%) is just 1%, meaning that the computed breakeven hit percentage is accurate for this test case.

However, in order to compute it, the scene must be traversed first, so that it is possible to obtain a value for $cost_{miss}$.

6| Experimental Results

105

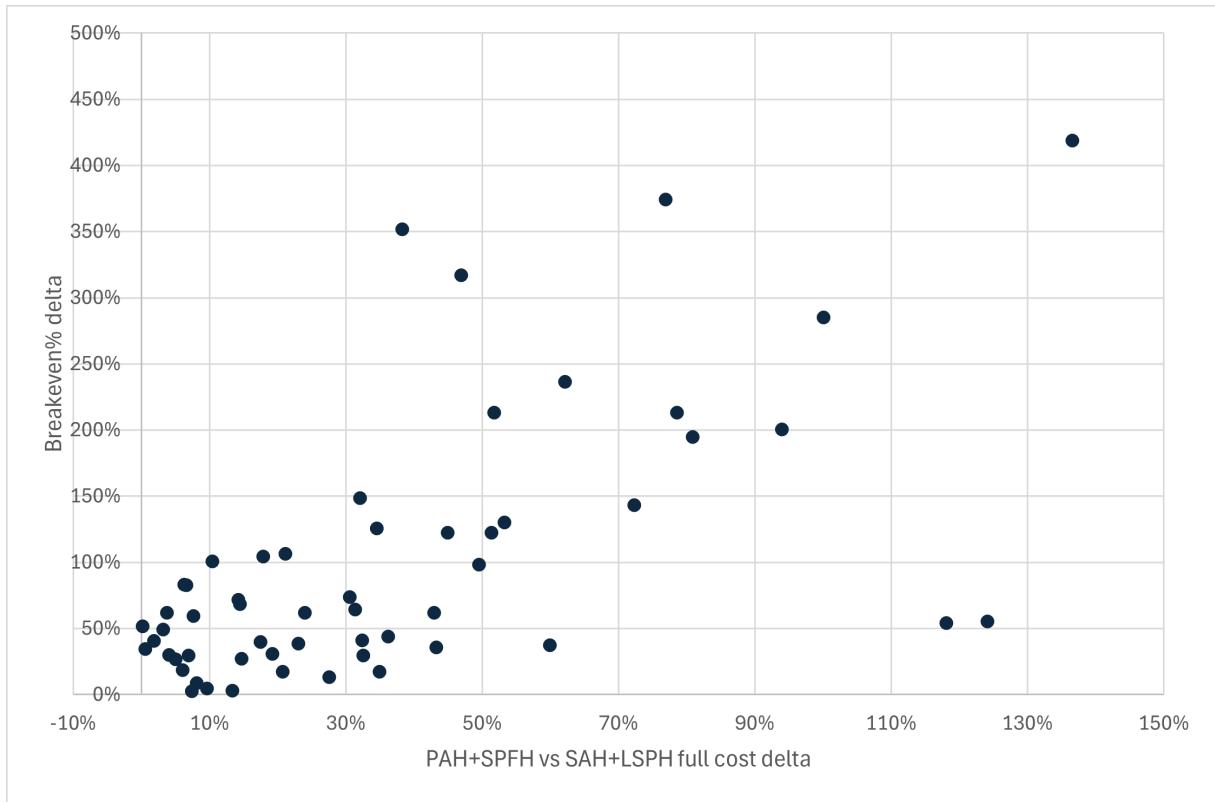


Figure 6.16: As the difference in traversal cost of our method and the state-of-the-art method goes to 0, also the difference between the calculated breakeven hit percentage and the measured one tends to 0.

6.2.6. SPFH in Isolation

In this section we analyze the performance of splitting plane facing heuristic used to build BVHs that adopts SAH as cost heuristic. In other words we analyze the performance of SPFH in isolation.

The reason behind this specific test case is that, as we observed in section 6.2.3, the building performance of SPFH is not worse than the performance of LSPH. On the other hand, PAH is much slower than SAH, at least in our naive implementation. Therefore, with this test case, we want to verify whether it can be worth it to use SPFH without PAH to build BVHs with a higher quality, rather than BVHs built by using SAH and LSPH.

6| Experimental Results

106

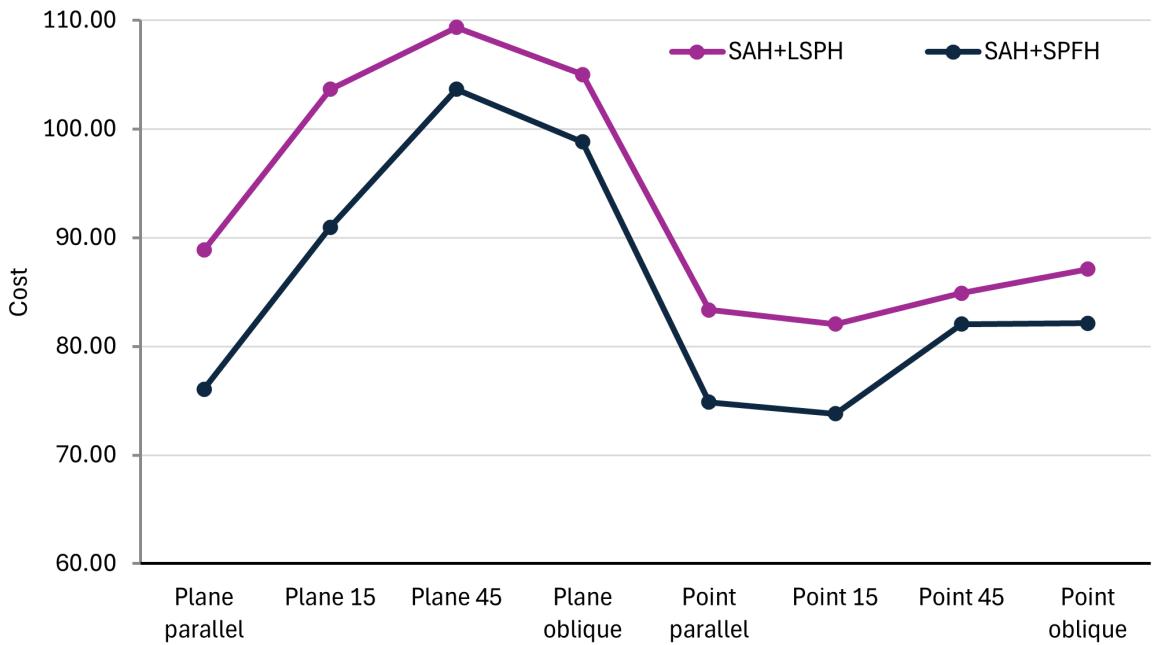


Figure 6.17: Comparison of how much faster SPFH and SAH is compared to SPFH and PAH.

Unexpectedly, the results suggest that using SPFH along with SAH, produces BVHs with a better quality than the BVHs built by using both of our two novel heuristics, and also better BVHs than state of the art heuristics.

As in the analysis in section 6.2.3, the results are better in scenes where the rays are parallel or *almost parallel* to one cartesian axis. Moreover, SPFH seems to work best with plane influence areas.

As it is possible to observe, using SFPH with influence areas with oblique rays, produces BVHs of similar quality to the BVHs produced by LSPH. This is expected, since, as explained in section 5.1.1, SPFH falls back to using LSPH in case an acceptable split is not found, and this happens often when the rays are not parallel to any cartesian axis.

The reason why SPFH works well can be explained by the overlapping amount between the children of a node. This topic is discussed in section 6.2.8.

6.2.7. PAH in Isolation

Given the unexpected results presented in the previous section, we decided to further investigate, and performed analyses on the quality of BVHs built with PAH and LSPH.

6| Experimental Results

107

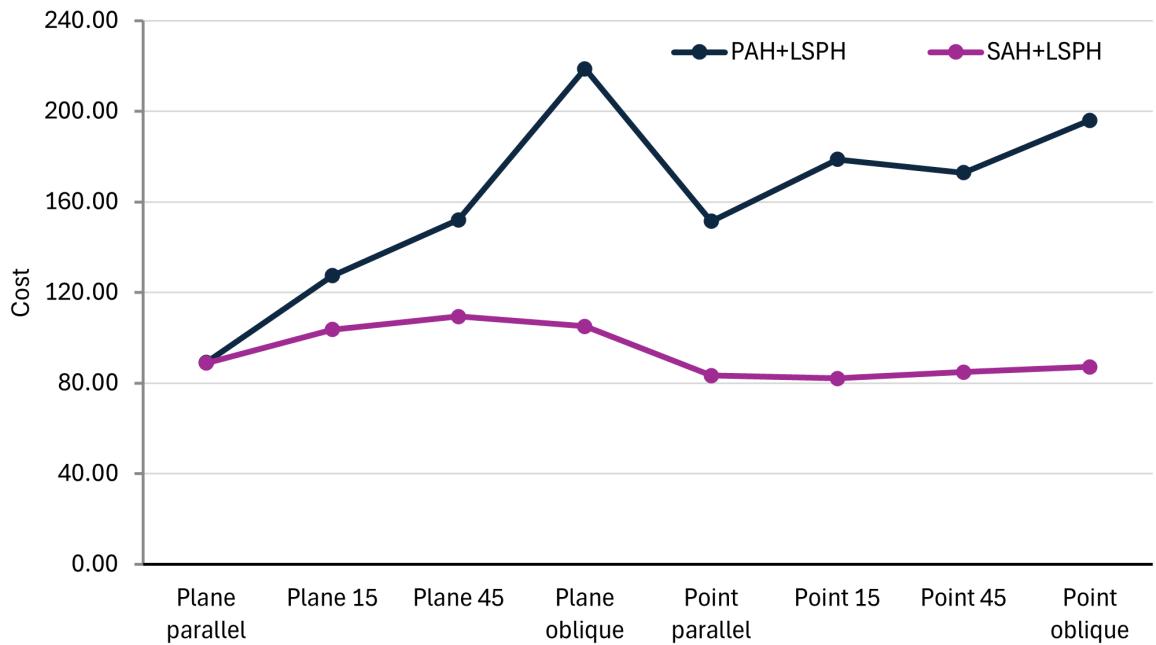


Figure 6.18: Comparison of how much faster PAH and LSPH are compared to SAH and LSPH.

The results in chart 6.18 show that projected area heuristic alone, produces BVHs with a worse quality than surface area heuristic, and also worse than using our two heuristics combined. By observing with our Unity visualizer tool the BVHs built with PAH and LSPH, we noticed a recurrent issue which partially explains why PAH doesn't perform well with LSPH. In principle, by using PAH we can adopt a more accurate cost metric than SAH. Therefore, by minimizing it, we expect to reach a better result. However, if the splitting plane orientation doesn't allow for splits without overlapping, what happens is that PAH splits the father node into an extremely small child and an extremely big one, as it can be observed in figure 6.19.



Figure 6.19: Issue when using PAH with overlapping children. Father node is green, big child is blue and small child is red.

By doing so, the projected area is indeed locally minimized. However, the split is not ideal, because it creates a large imbalance in the BVH. When we use SPFH this happens

6| Experimental Results



less frequently, and the greedy minimizing of the projected area produces better results.

6.2.8. Siblings Overlapping

This paper [1] from Alia, Karras and Laine discusses how overlapping in 3-dimensional space among BVH nodes leads SAH to underestimate the BVH cost. The tests presented in this section show how, also in the scenarios where PAH and SPFH can be used, overlapping can be a good predictor of the quality of a BVH. In our case, the overlapping is not calculated in 3-dimensional space, but in the 2-dimensional space where the AABBs get projected. We will eventually show how SPFH is able to reduce overlapping compared to LSPH.

We decided to measure the overlapping between two sibling nodes at different depths of the BVH. Given that we average BVH maximum depth, among all the test cases, is 20, we decided to take samples at levels 4, 7, 10, 15 and 100. The first relevant fact we noted is that, regardless of the cost metric used to build the BVH, the average overlapping percentage among sibling nodes is related to the quality of the final BVH, as it can be observed in the scatter chart 6.21.

We measure the overlapping between siblings by first projecting their AABBs to the plane of the influence area associated to the BVH. At this point we get two convex hulls, which we first cull against whe near plane, as explained in section 6.2.2. Now we can use the same culling algorithm, explained in section A.9, to obtain the convex hull of the shape that is the intersection between the two hulls. Eventually we can compute the area of this hull.

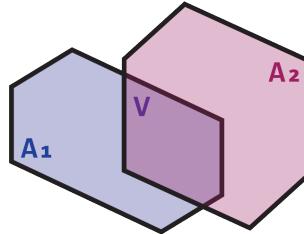


Figure 6.20: How to compute the overlapping between siblings.

If we name A_1 and A_2 the areas of the projected hulls of the AABBs, and V the area of the intersection shape between the two, we can compute the overlapping percentage as:

$$\text{Node overlapping percentage} = \frac{V}{\min(A_1, A_2)}$$

6| Experimental Results

This process is carried out in a *BvhAnalyzer* templated function of our framework (section 5.2).

To compute the average overlapping percentage, we sum together all the overlapping areas ($V_1 + V_2 + \dots$) of each pair of sibling, along with the area of the smallest of the two siblings ($A_{min1} + A_{min2} + \dots$), and we compute the total overlapping percentage in the same way:

$$\text{Total overlapping percentage} = \frac{V_1 + V_2 + \dots}{A_{min1} + A_{min2} + \dots}$$

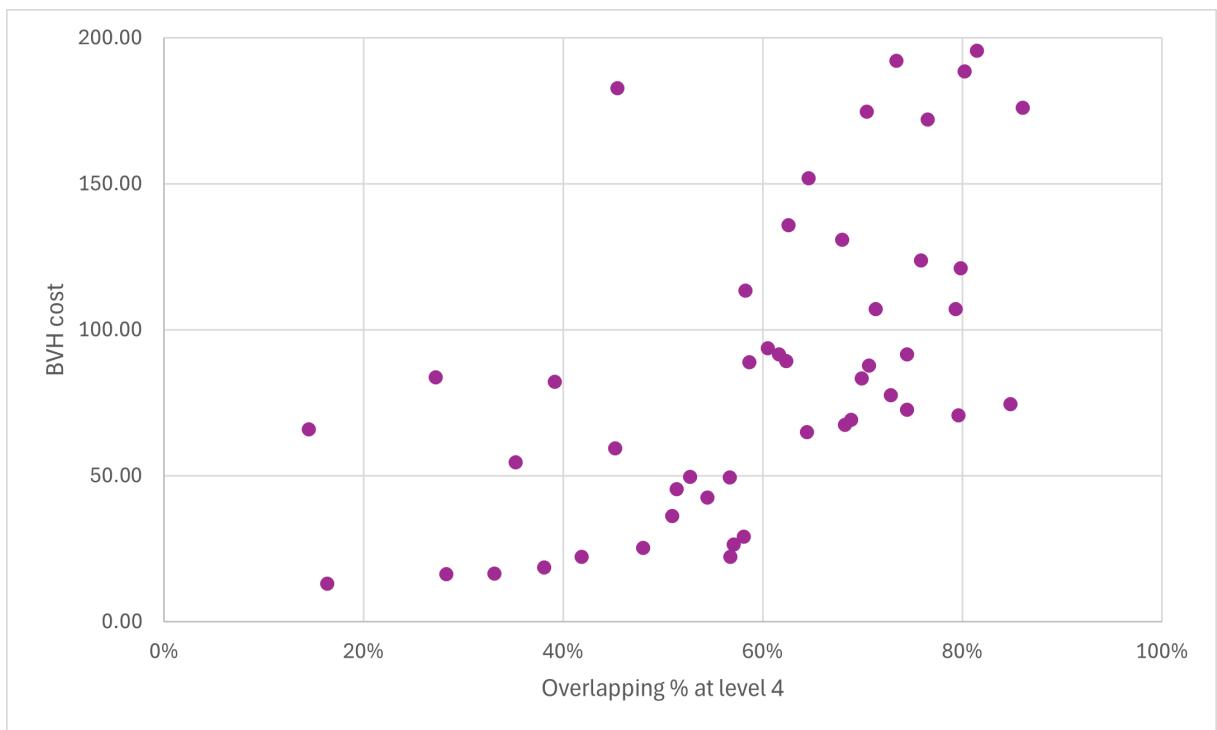


Figure 6.21: High overlapping leads to worse BVH cost.

This result confirms the findings in the paper from Alia, Karras and Laine, and extends it to our context. The intuitive reason behind this behavior is that, if the projections of two siblings do not overlap, it is impossible for a ray to hit both. Missing at least one of the two leads to a smaller number of intersection tests, because it is possible to discard a subtree of the BVH.

We also noted how SPFH can drastically reduce the overlapping percentage between siblings, compared to LSPH. This is the main reason why SPFH is very good at producing higher quality BVHs, as shown in section 6.2.6.

6| Experimental Results

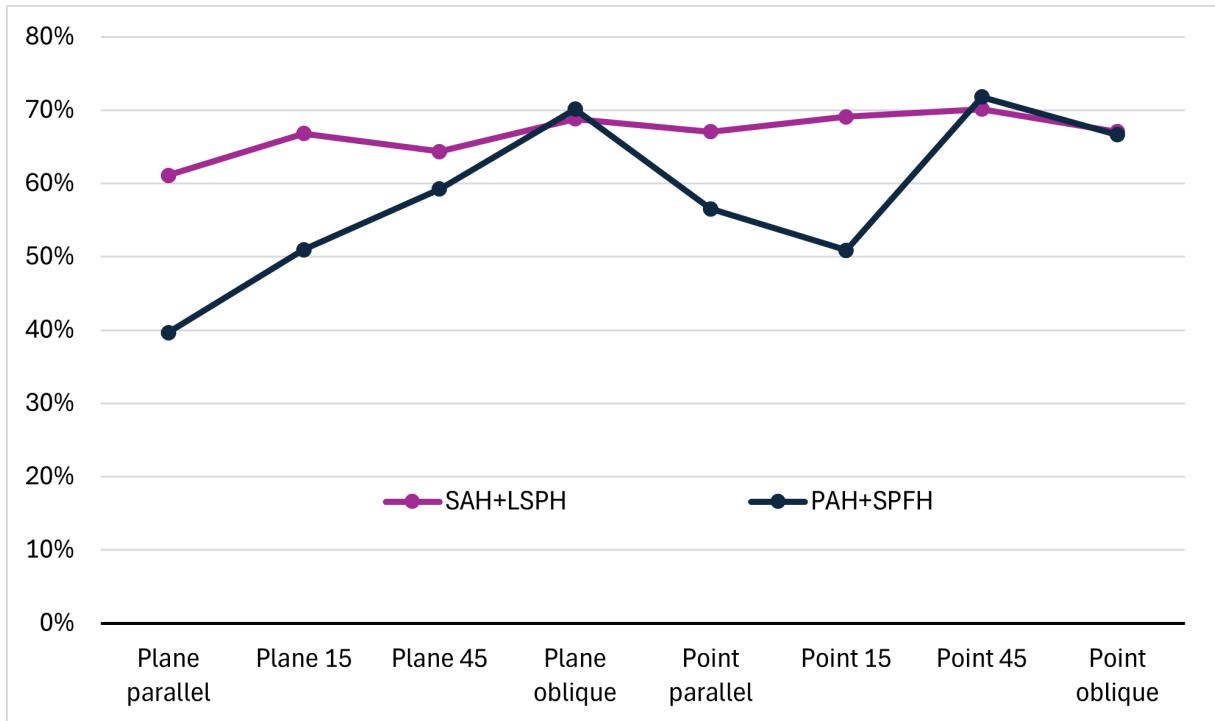


Figure 6.22: Overlapping percentage at BVH level 4.

From this diagram it is possible to note how SPFH (59%) is better than LSPH (66%) at minimizing the overlapping. SPFH works better when rays tend to be parallel to one cartesian axis. This can be associated with two factors:

- As we have seen in section 6.2.6, SPFH falls back to LSPH more often in oblique test cases;
- Since BVHs are made up of AABBs, if the rays are parallel to a cartesian axis, it is possible to maximize overlapping reduction, as it can be better observed in figure 6.23.

6| Experimental Results

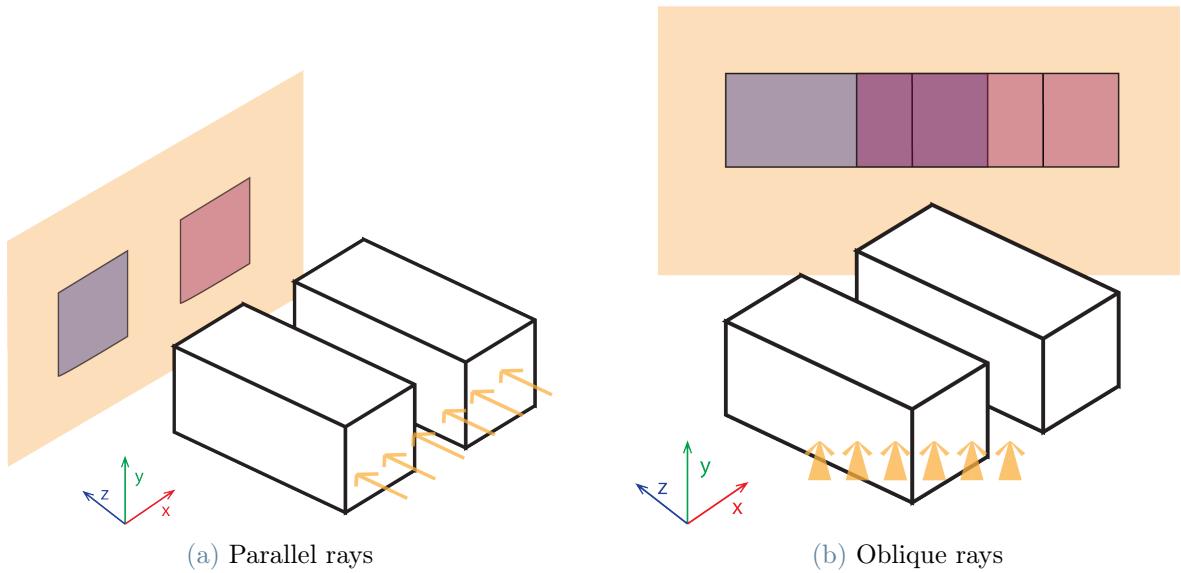


Figure 6.23: Overlapping reduction works best if the rays are parallel to a cartesian axis.

It is also interesting to see how SPFH is more effective at reducing overlapping in the first levels of the BVH, whereas it is worse than LSPH on deeper levels.

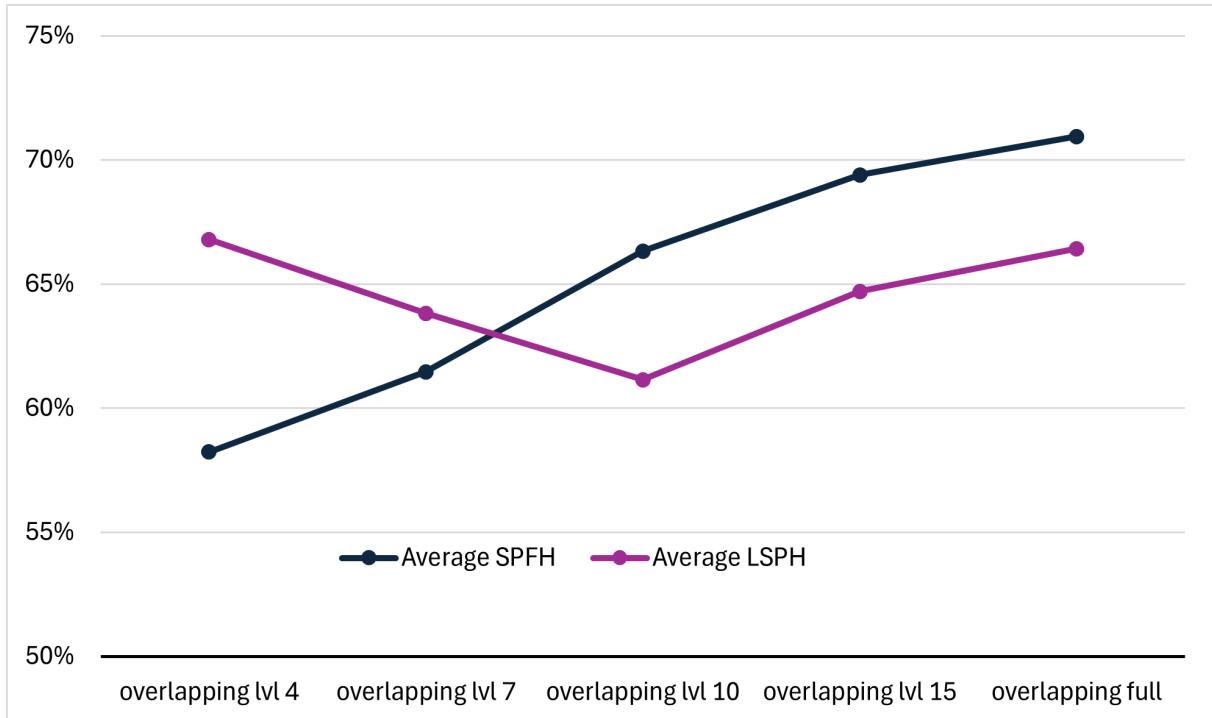


Figure 6.24: Overlapping percentage by level.

This fact can be exploited, and it is therefore possible to use SPFH only on the higher levels of a BVH, and then directly fallback to LSPH. With this optimization it is pos-

6| Experimental Results



sible to use more quality-oriented properties for the BVH construction, such as a lower *splittingPlaneQualityThreshold* or *acceptableChildrenFatherHitProbabilityRatio*. We will talk about this in section 6.4.

6.3. Octree versus AABBs for Regions Top Level Structures

In this chapter we will present a comparison between the two top level acceleration structures we implemented in our framework, namely the octree and the AABBs for regions (section 5.5). We will also analyze how many levels the octree must have in order to approximate in an acceptable way the influence areas it is built on, and how the approximation affects the traversal of the contained BVHs.

In order to compare the octree to the AABBs for regions structure, we created a specified quantity of influence areas. Half of these influence areas are plane, and the other half are point influence areas. We then build both the octree and the AABBs for regions structures over these influence areas. Eventually we uniformly sample 1000 points in the region where the influence areas are placed, and query both the top level structures for the influence areas a specific point is contained into.

In our first test we compare the average time required by the two top level structures to return the result of our queries. We decided to also compare octrees with different amount of levels. The quantity of influence areas for this first testing scenario is 20, equally divided between plane and point ones.

6| Experimental Results

113

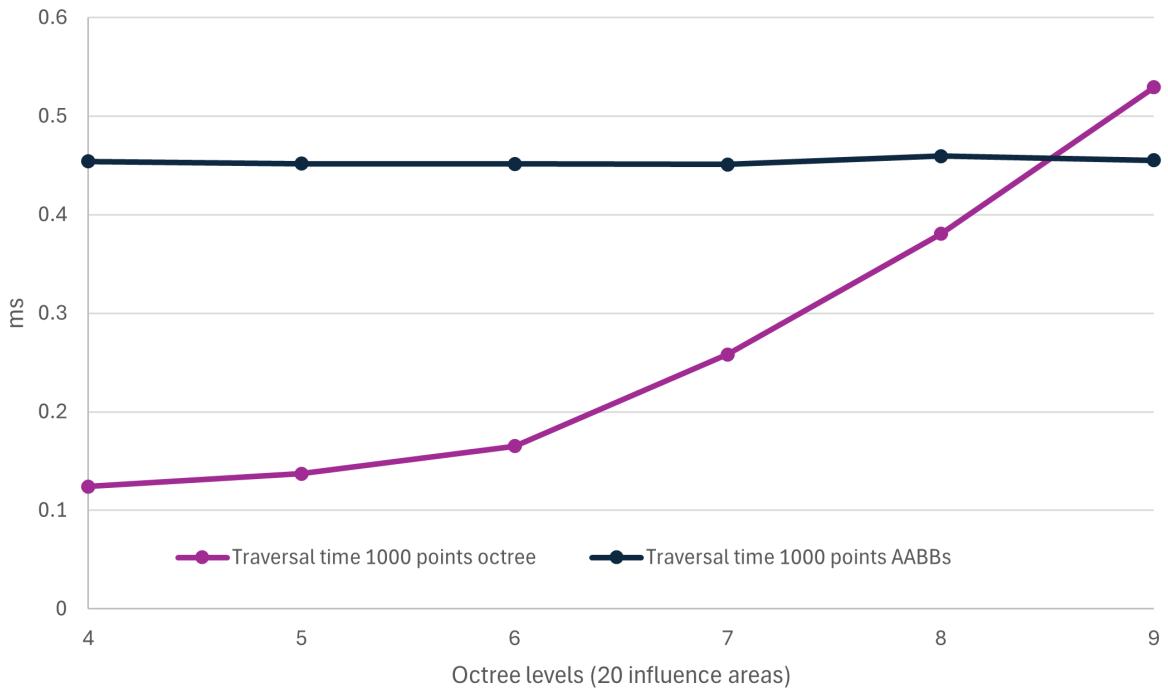


Figure 6.25: Average time required to complete the queries by octrees of different levels, and AABBs for regions.

In chart 6.25 we can observe how the required time grows as the number of the levels of the octree increases. The growth gets faster as the number of levels increases: this is expected, since in a octonary tree the number of nodes can be obtained by this formula: $\#nodes = 8^n - 1$ where n is the number of levels. This means that, if we increase the number of levels from $n - 1$ to n , the tree will have 8^{n-1} nodes more, and the time to traverse it will exponentially grow.

Instead, the time for the AABBs for regions structure remains constant, since the structure doesn't have the concept of level. In case of 20 influence areas, the breakeven point between the two structures is between the 8th and 9th level, meaning that it is not worth it to use an octree if it has more than 8 levels.

The octree is the structure that scales better with the amount of influence areas. In order to show this, we decided to build octrees with 7 levels over an increasing amount of influence areas:

6| Experimental Results

114

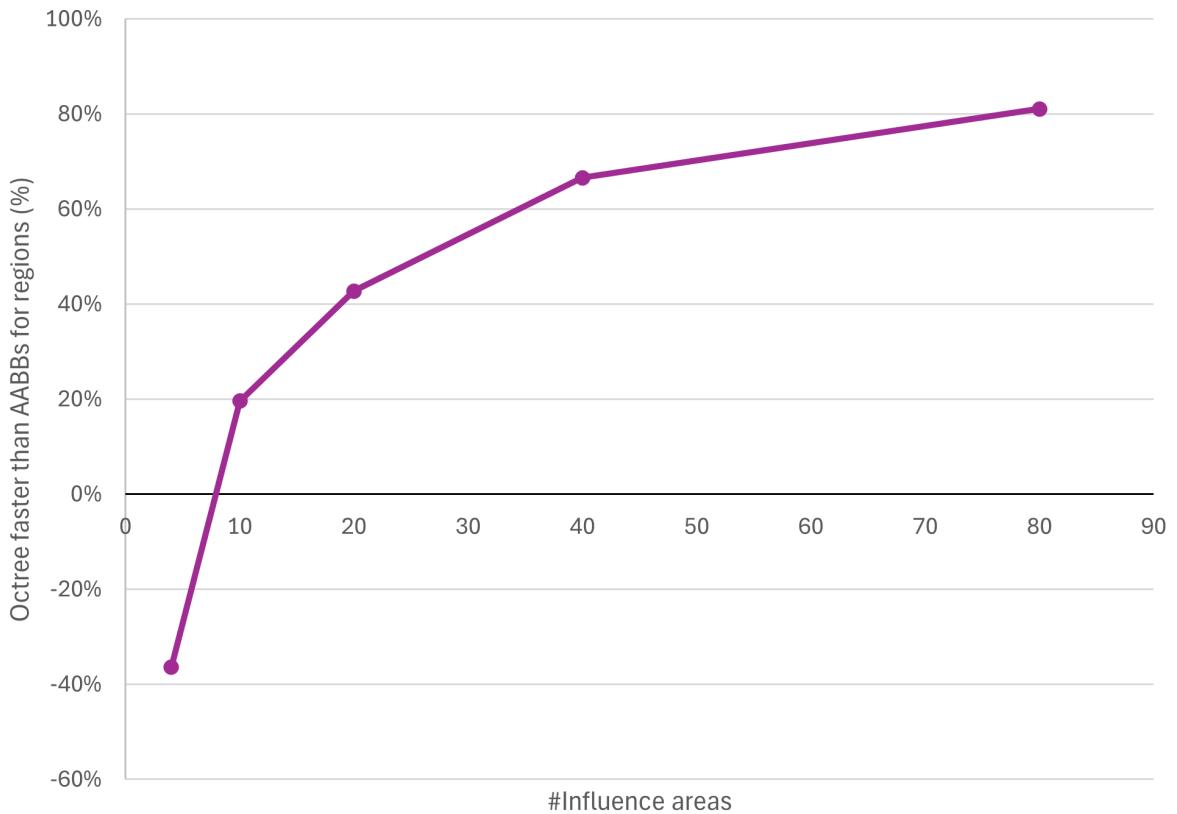


Figure 6.26: How much faster the octree is to traverse compared to the AABBs for regions built over the same influence areas.

As it can be observed in chart 6.26, with a small amount of influence areas (4), an octree with 7 levels is 36% slower than the AABBs for regions structure at detecting the influence areas a point is contained into. As the amount of influence areas increases, the AABBs for regions structure becomes slower and slower, while the octree keeps its query time constant. This is expected, since the query time of the octree is independent from the amount of influence areas, and only depends on the number of levels. Already with 10 influence areas, the octree is 20% faster, and the discrepancy keeps increasing.

Last, we wanted to quantify the error an octree introduces. The octree is indeed an approximation of the influence areas. As we mentioned in chapter 3.3, it is a conservative approximation, meaning that the volume of the influence areas is overestimated, but never underestimated.

6| Experimental Results

115

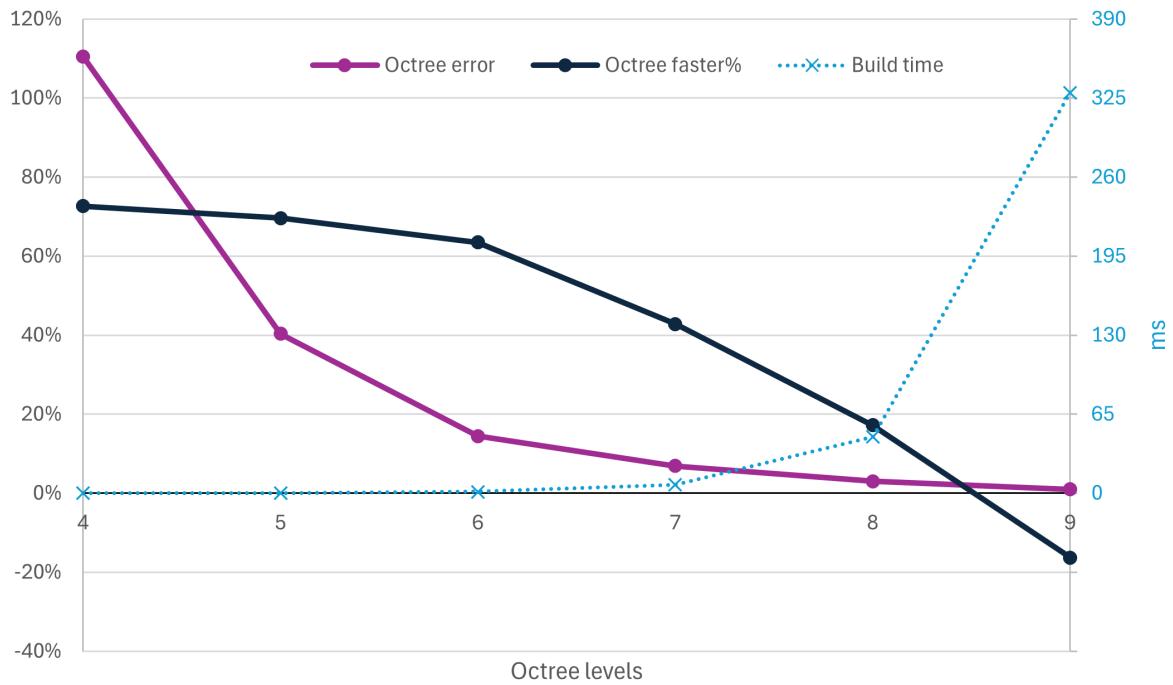


Figure 6.27: Analysis of the error an octree with a given amount of levels introduces in its queries, and how much faster it is compared to the AABBs for regions structure.

In chart 6.27 we show the average error of the octrees queries in percentage, and how much faster they are compared to the AABBs for regions structure. An octree with 4 levels has an error percentage of 110%, in particular of the 9136 influence areas the points were considered contained into, 4795 were false positives. The error decreases exponentially, and already with 5 levels it is 40%, while the octree is only 3 percentage points slower compared to the one with 4 levels.

From the chart is is possible to observe that the maximum difference between how faster is the octree and its error percentage is reached with an octree of 6 levels. However, in our test cases (section 6.2.3) we adopted octrees with 5 levels, in order to reduce the building times. Indeed, as it can be observed in the chart above, the building time grows exponentially, which is expected given that the number of nodes in the tree also grows exponentially, as mentioned a few paragraphs above.

The last analysis we carried out concerns the average traversal cost of the BVHs as related to the approximation error of the underlying octree. If the approximation of the octree is too coarse, too many rays that are not affine to a certain influence area will be cast towards it regardless. This leads to searching the ray-scene intersection in a BVH that doesn't follow the PAH and SPFH hypotheses. Furthermore, it is more likely for such a

6| Experimental Results

116

ray to completely miss the BVH, and fall back to the global BVH, wasting time. The experimental results show that with a 4 level octree the average cost of traversing the BVH for the scenes *Cottage*, *Random1000*, *Wood*, *Suzanne* is 100.5, and increases to 115.5 if we consider also the fallback searches. With an octree with 5 levels the costs go down respectively to 88.7 and 101.4. However, if we use a 6 levels octree the costs only decrease to 87.6 and 99.8. All the analyses were performed on octrees built over 20 influence areas.

	4 levels	5 levels	6 levels
Cost w/o fallback	100.5	88.7	87.6
Cost with fallback	115.5	101.4	99.8

Table 6.1: Octree levels and BVH traversal cost.

6.4. Testing Different Properties for PAH and SPFH BVH Construction

In this section we will briefly analyze how, by modifying some properties during the building phase of a BVH (table 6.2), the resulting BVH changes. In particular we will focus on the *splitPlaneQualityThreshold*, *acceptableChildrenFatherHitProbabilityRatio* and *excellentChildrenFatherHitProbabilityRatio* properties, and measure the BVH quality and the amount of nodes built by using the fallback heuristics. All the results are presented in an aggregated form, where the average value of all the used test cases is considered. We will run the analysis on all the test cases where both PAH and SPFH are used.

The base values for the properties listed above are the following:

<i>splitPlaneQualityThreshold</i>	0.4
<i>acceptableChildrenFatherHitProbabilityRatio</i>	1.3
<i>excellentChildrenFatherHitProbabilityRatio</i>	0.9

Table 6.2: Base values for properties.

With these values we get an average BVH quality of 110.5. The amount of attempted splitting planes per node is 1.8: this value is computed as follow:

$$\text{attempted splitting planes} = \frac{\text{total splitting planes attempted}}{\#\text{internal nodes}}$$

The fallback splitting method, namely using LSPH and SAH, is used 0.34 times per node. The fallback method is used when there are no more splitting planes with a quality above *splitPlaneQualityThreshold* and no split with a hit ratio value less than *acceptableChildrenFatherHitProbabilityRatio* has already been found. A better explanation of this process can be found in section 5.1.1.

First, we tried to lower the *splitPlaneQualityThreshold*. The result we expect from this change is to build BVHs with a better quality, at the cost of an increased build time. Indeed, with a less strict threshold on the minimum quality of a splitting plane, more splitting planes will be attempted while trying to divide a node into its two children. The results we obtained fulfill our expectation: the average cost of the BVHs is now 88.4, with an amount of attempted splits per node of 2.4. Moreover, the fallback method is used fewer times, because, by attempting more splitting planes, it is more likely to find a good split: the amount of nodes generated with the fallback method is now only the 13%.

We then tried to raise the *splitPlaneQualityThreshold*, and, even in this case, we obtained results in line with the expectations. The average BVH cost increased to 148.3, while the amount of attempted splitting planes decreased to 1.5 per node. The fallback method is used 0.68 times per node.

The next set of tests we performed keeps the *splitPlaneQualityThreshold* constant, while increasing and decreasing the *acceptableChildrenFatherHitProbabilityRatio* and *excellentChildrenFatherHitProbabilityRatio* properties.

We first tried to decrease the two properties mentioned above to values of 1.1 and 0.7 respectively. We expect that, with a more demanding threshold to consider a split as *good enough*, the quality of the BVH will increase at the cost of having more nodes generated with the fallback heuristics. The experimental results confirm our hypotheses. Indeed, the average BVH cost decreases to 69.2, while 40% of the nodes are created with LSPH and SAH. Also, the amount of splitting planes attempted increases to 2.0 splitting planes per node.

Eventually we raised *acceptableChildrenFatherHitProbabilityRatio* to 1.5 and *excellentChildrenFatherHitProbabilityRatio* to 1.1. The results are in line with our expectations: the average cost of a BVH is now increased to 124.6, but only 29% of the nodes are created with the fallback heuristics. The average amount of attempted splitting planes per node is decreased to 1.7.

In table 6.3 it is possible to look at all the results described above in a condensed form.

6| Experimental Results

	Case 0	Case 1	Case 2	Case 3	Case 4
<i>splitPlaneQualityThreshold</i>	<u>0.4</u>	0.1	0.8	<u>0.4</u>	<u>0.4</u>
<i>acceptable.C.F.HitProb.Ratio</i>	<u>1.3</u>	<u>1.3</u>	<u>1.3</u>	1.1	1.5
<i>excellent.C.F.HitProb.Ratio</i>	<u>0.9</u>	<u>0.9</u>	<u>0.9</u>	0.7	1.1
Average BVH cost	110.5	88.4	148.3	69.1	124.6
Avg. splitting planes attempted	1.8	2.4	1.5	2.0	1.7
Fallback percentage	0.34	0.13	0.68	0.41	0.29

Table 6.3: Summary of our experiments with different properties values. Underlined values are the default ones.

6.5. Memory Footprint

In this section we try to estimate the memory consumption of the objects needed to support our heuristics, and compare them with the memory footprint of a standard BVH.

Estimate of the Cost of a BVH

In order to estimate the memory occupied by a BVH, we first have to estimate the number of nodes in it. On average the BVHs of the test cases have 20.9 levels, which we can approximate to 21. A binary tree with 21 level, in case it is balanced, has $2^{21} - 1 = 2097151$ nodes, half of which are leaf nodes.

In our implementation, a node is represented by a structure (section 5.1) holding an *Aabb*, two pointers to the children nodes and a list of triangles. We decided to keep the list of triangles in internal nodes for debugging and visualization reasons, however it is only necessary in leaf nodes. For the sake of this analysis, we will only consider leaf nodes containing a list of pointers to triangles.

An *Aabb* is represented by a pair of 3-dimensional vectors of floating point numbers, therefore it has a weight of $4B \cdot 3 \cdot 2 = 24Bytes$.

Assuming that floating points are stored in 4 bytes, and pointers in 8 bytes, an internal node has a weight of $24B + 2 \cdot 8B = 40Bytes$.

To compute the weight of a leaf node, we have to add the size of the array of triangles' indices. Since, in our test cases, the algorithm stopped when the number of triangles in a node was less than 3, we can assume¹ that all the leaf nodes carried 2 triangles. In this

¹It is also possible to have a leaf node with more than 2 triangles, in case it was impossible to split due to the relative position of the triangles' centers.

6| Experimental Results

case their weight is $40B + 2 \cdot 4B = 48B$.

We can now compute the cost of a balanced BVH of 18 levels as:

$$(2^{21-1} - 1) \cdot 40B + 2^{21-1} \cdot 48B \approx 92.27MB$$

Estimate of the Cost of an Influence Areas

When a BVH has an associated influence area, we must add the cost of it to the one of the array. A *PlaneInfluenceArea* is made up of a *Plane*, a 4×4 view projection matrix (used to accelerate the projection process), and an *AabbForObb* (section 5.3.3).

The *Plane* holds two 3D floating-point vectors, one for the passing point and one for the normal (24Bytes). The *AabbForObb* contains 6 3D floating-point vectors, respectively for the center, forward, right and up directions, and the half size in all directions. Moreover it contains an *Aabb* for the rejection tests: the total cost is $6 \cdot 3 \cdot 4B + 24B = 96Bytes$.

The total weight of a *PlaneInfluenceArea* is $24B + 96B + 4 \cdot 4 \cdot 4B = 184Bytes$.

A *PointInfluenceArea* is made up of a *Pov* and a *Frustum*. The *Pov* contains two 3D floating-point vectors for the viewing direction and position, and 2 floats for the *x* and *y* fields of view (32Bytes).

The *Frustum* a more heavy structure, because it contains redundant information to improve ray-frustum collision detection (see section 5.3.4). It contains 4×4 view-projection matrix, an array of 6 3D directions for the face normals, an array of 6 3D vectors for the edges and 8 3D vertices. Furthermore it contains an *Aabb* for rejection tests.

The total cost is $32B + 4 \cdot 4 \cdot 4B + 6 \cdot 3 \cdot 4B + 6 \cdot 3 \cdot 4B + 8 \cdot 3 \cdot 4B = 336Bytes$.

The average cost of an influence area is therefore: $\frac{184B + 336B}{2} = 260Bytes$.

Estimate of the Cost of an Octree

In order to compute the size of all the structures needed for our heuristics, we first have to make some assumptions on the number of influence areas present in the scene. We will first analyze the case where just one influence area is present in the scene, and then the case were 20 influence areas are present in the scene.

We also compare both the cases where the octree or the AABBs for regions top level structure is used (section 3.3).

In case the octree is used, its cost can be computed in a similar way to the BVH. Since

6| Experimental Results



we showed that octrees with 5 levels provide the right compromise between speed and precision (section 6.3), we assume a 5 level octree. A 5 level octree has $8^5 - 1 = 32767$ nodes, half of which are leaves.

A node of an octree holds an *Aabb*, 8 pointers to its children and, in case it is a leaf, an array of pointers to the *Bvhs* included in the leaf. This means that an internal node has a weight of $24B + 8 \cdot 4B = 56Bytes$. A leaf node, assuming it contains just one *Bvh*, has a cost of $56B + 4B = 60B$.

Therefore, the cost of the octree is:

$$(8^{5-1} - 1) \cdot 56B + 8^{5-1} \cdot 60B \approx 1.9MB$$

For what concerns the AABB for regions structure, it is simply an array of BVH.

Putting All Together

We can not put everything together and calculate the cost of our structures. In case there is only one influence area covering most of the scene, the total cost of our structure is:

$$\begin{aligned} \text{Total cost} &= \text{Cost local BVH} + \text{Cost influence area} + \text{Cost global BVH} + \text{Cost octree} \\ &= 92.27MB + 260B + 92.27MB + 1.9MB \approx 186.44MB \end{aligned}$$

If we used only the global BVH, the cost would have been only $11.53MB$, which means that using our heuristics is costing 2.16 times more.

In case there are 20 influence areas, we assume that they contain roughly the same amount of triangles, and that only 50% of all the triangles in the scene are contained in at least one influence area. Before we assumed that the BVH has 21 levels, is balanced and each leaf contains 2 triangles.

This means that the total number of triangles in the scene is $2^{21-1} \cdot 2 = 2097152$. Only half of these are included in a BVH, therefore each BVH roughly contains $\frac{2097152}{2 \cdot 20} \approx 50000$ triangles. A balanced BVH with 50000 triangles has an amount of levels equals to²:

²The division by 2 comes from the fact that each leaf contains 2 triangles on average, as we explained above.

6| Experimental Results

$$\left\lceil \log_2 \left(\frac{50000}{2} \right) + 1 \right\rceil = 16$$

A BVH with 14 levels has a cost of $(2^{16-1} - 1) \cdot 40B + 2^{16-1} \cdot 48B \approx 2.88MB$.

We can now compute the total cost of the structures needed to use our heuristics in this scenario:

$$\begin{aligned} \text{Total cost} &= (\text{Cost local BVH} + \text{Cost influence area}) \cdot 20 + \text{Cost global BVH} + \text{Cost octree} \\ &= (2.88MB + 260B) \cdot 20 + 186.44MB + 1.9MB \approx 191.22MB \end{aligned}$$

Since the 20 BVHs contains a smaller amount of nodes the cost of each BVH is greatly reduced. In particular, the dominant part of the cost of the BVH increases exponentially with the number of nodes. This relation allow us to use our structures even in cases with many influence areas of modest size.

7 | Conclusion

In this chapter we will briefly talk about some of the potential use cases of the novel techniques presented in this work, and possible development directions. We will conclude our thesis with a summary of the presented methods and some considerations.

7.1. Use Cases

As we said while presenting this thesis, bounding volume hierarchies are a key component of ray tracing. Their hierarchical structure is fundamental in accelerating the ray-scene intersection process, which is the core mechanism by which algorithms of the ray tracing family are able to render a scene. This means that an improvement in how BVHs work, inherently makes ray tracing faster.

In particular, our work is too general to find specific use cases, nonetheless we can summarize the strong points of our techniques, and some possible broad application contexts.

Both projected area heuristic and splitting plane facing heuristic work best when the rays of the influence area they are used on are parallel to a cartesian axis (section 6.2.3). Since this property can be known as soon as an influence area is discovered, it is possible to decide on-the-fly whether it is worth it to use our technique or not. For example, in a rendering scenario where there is a static light source, it is possible to immediately decide whether it is appropriate to build an influence area for it or not. In case of a dynamic light, things can become more difficult, as it would require to decide on a per-frame basis whether it is worth it to use the BVH associated with the influence area or not. In a real-time scenario this could prove too expensive, however more tests should be carried in different scenarios, in order to prove this claim.

Building BVHs with PAH has proven to be an order of magnitude more expensive than building them with SAH (section 6.2.3), therefore the technique is not suited for a real-time dynamic scenario. On the other hand, it can bring improvements in non-real-time use cases, where it can be worth it to trade one-time construction time for per-ray traversal time. Indeed, in non-real-time scenarios, usually there are many more rays to trace, on a



scene with more geometry. This means that even a small improvement in traversal time can potentially outweigh the increased construction time cost.

As we have seen, PAH in isolation doesn't bring benefits in many scenarios compared to SAH; on the other hand, splitting plane facing heuristic has proven to work well even without PAH. Furthermore, SPFH is not more expensive than the state-of-the-art LSPH, therefore it may be worth it to use it even in real-time scenarios. The drawback of using SPFH is that it can be effectively used only in specific regions of the scene, meaning that it is necessary to build the underlying top level structure (chapter 3). This fact can hinder the use of SPFH in a real-time scenario, especially in dynamic use cases, such as for dynamic lights.

All in all, without carrying out further tests, we think that our methods are at the moment more suited for non-real-time scenarios, where a faster-to-traverse BVH can reduce the rendering time more effectively.

7.2. Future Developments

The most important development of our work is to improve the algorithms and structures used. While in this thesis we showed PAH and SPFH validity from a theoretical point of view, it is now fundamental to test them in more concrete use cases before continuing the development in other directions.

In order to do so, we suggest to look at the state-of-the-art algorithms used to build BVHs with SAH and LSPH, and then try to change them to use PAH and SPFH. In principle, many of the optimizations used for standard heuristics should perform equally well even for our heuristics.

A different approach is instead needed for the top level structure. In this case it is not possible to perform a one-to-one translation from existing techniques, even though it may be worth it to study how TLAS and BLAS [25] are implemented.

Another important step would be to port the construction and traversal algorithms on parallel architectures. It may be beneficial to do this in two steps: first implementing a concurrent builder on the CPU, by using a multi-threaded approach. This could be helpful to get insights on what are the most critical points in the change of programming model. Eventually it is also important to transition to a fully parallel builder on the GPU. As shown in [18], building a BVH is a problem that can be effectively resolved by using a parallel approach, and doing so with our heuristics should be no different, except, probably, some considerations on the data layout to make the top level acceleration

structure as efficient as possible, especially during traversal.

In case the previous development produces acceptable results, the next step to make our techniques operative would be to create a method to detect influence areas autonomously. At the moment, we artificially create influence areas. While this approach could be used even in a real-world scenario, by adding influence areas where light sources are present, it could bring many improvements if it was possible to detect influence areas by giving as input the rays traced in previous frames. By doing so, there would be two major improvements: first, influence areas near light sources would be more accurate; seconds, it would be possible to detect influence areas even in parts of the scene far from light sources, such as near a reflective surface like a mirror. The first method that comes to mind to do so is clustering, but even other directions can be explored. In particular, it could be beneficial to try to recycle results from previous frames in order to make the influence area detections algorithms more lightweight and accurate.

Another possible development is to try a different top level structure. For example, one structure that could bring some benefits over the octree is the kd-tree [12].

7.3. Closing Thoughts

Designing and developing two new heuristics for BVH construction, in a field with such a long history of development was a challenge, and a very instructive one. In this last section we will only summarize our work, and evaluate it from a more subjective point of view.

Before starting the ideation of the two heuristics, we carried out a detailed phase of studying the state-of-the-art of the rendering methods used nowadays. The study started from a general overview of the most used real-time rendering techniques, from sources like [2], [34] and [28]. Then we focused more on ray tracing. First, we studied the mathematical theory behind ray tracing, in particular Monte-Carlo and variance reduction methods, culminating with the study of more recent papers on modern techniques, such as ReStir [56]. After this, we moved on to studying the more concrete part about acceleration structures, in particular BVHs. Starting from the basis and studying both the theory and the concrete structures, allowed us to put them together and find the small gap in the hypotheses of SAH.

We then started prototyping a very rough algorithm in Unity, in order to verify if our ideas could work at least in the most basic scenarios. By using Unity, we were also able to directly look at what we were developing, and this lead us to the creation of the second

7| Conclusion



heuristic, namely SPFH.

At this point, we started writing the actual code in C++, without forgetting how important it was to be able to have a visual tool to visualize our results. While writing the C++ code, we decided to try and build an actual framework, and document it, in order to make life easier both for use and for future researchers. This proved invaluablely important, as we were able to try out many different configurations with just small changes in the codebase.

The last part was to test our methods in more concrete scenes, and extrapolate some indicators in order to be able to understand and present our results.

Our work, while it has been tested on concrete scenes, is mostly a theoretical one. We showed how the novel heuristics we proposed can indeed work in some specific scenarios. However, we think that the way to actually use them is realistically long, but, as we described in section 7.2, we believe it is worth it to travel it.

Bibliography

- [1] T. Aila, T. Karras, and S. Laine. On quality metrics of bounding volume hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*, pages 101–107, 2013.
- [2] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire. Real-time rendering 4th edition. In *Real-Time Rendering 4th Edition*. CRC Press, 2023.
- [3] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire. Intersection test methods. In *Real-Time Rendering 4th Edition*, chapter 22. 2023.
- [4] Anonymous. Difference between rendering equation, lighting model, ray tracing, global illumination and shadows? <https://computergraphics.stackexchange.com/questions/9241/difference-between-rendering-equation-lighting-model-ray-tracing-global-illumination-and-shadows>. Accessed: (18/04/2024).
- [5] S. Brabec, T. Annen, and H.-P. Seidel. Practical shadow mapping. *Journal of Graphics Tools*, 7(4):9–18, 2002.
- [6] J. Burgess. Rtx on—the nvidia turing gpu. *IEEE Micro*, 40(2):36–44, 2020.
- [7] A. Celarek. Rendering: The rendering equation. https://www.cg.tuwien.ac.at/courses/Rendering/2020/slides/04_The_Rendering_Equation_v20200515.pdf. From TU Wien university, Accessed: (19/08/2023).
- [8] K. Chung, C.-H. Yu, D. Kim, and L.-S. Kim. Shader-based tessellation to save memory bandwidth in a mobile multimedia processor. *Computers & Graphics*, 33 (5):625–637, 2009.
- [9] J. Cole. Signed distance fields. <https://jasmcole.com/2019/10/03/signed-distance-fields/>. Accessed: (11/05/2023).
- [10] A. Dittebrandt, J. Hanika, and C. Dachsbacher. Temporal sample reuse for next event estimation and path guiding for real-time path tracing. 2020.

- [11] B. Fabianowski, C. Fowler, and J. Dingliana. A cost metric for scene-interior ray origins. In *Eurographics (Short Papers)*, pages 49–52, 2009.
- [12] D. Fussell and K. R. Subramanian. *Fast ray tracing using kd trees*. University of Texas at Austin, Department of Computer Sciences, 1988.
- [13] G. Gribb and K. Hartmann. Fast extraction of viewing frustum planes from the world-view-projection matrix. *Online document*, 2001.
- [14] Y. Gu, Y. He, and G. E. Blelloch. Ray specialized contraction on bounding volume hierarchies. In *Computer Graphics Forum*, volume 34, pages 309–318. Wiley Online Library, 2015.
- [15] J. Hart, D. Sandin, and L. Kauffman. Ray tracing deterministic 3-d fractals. *SIGGRAPH '89*, 1989.
- [16] J. T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150, 1986.
- [17] B. Karis, R. Stubbe, and G. Wihlidal. Nanite: A deep dive. SIGGRAPH '21: Advances in Real-Time Rendering in Games.
- [18] T. Karras. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. *The Eurographics Association*, 2012.
- [19] A. E. Kaufman and K. Mueller. Overview of volume rendering. *The visualization handbook*, 7:127–174, 2005.
- [20] A. Keller, L. Fascione, M. Fajardo, I. Georgiev, P. Christensen, J. Hanika, C. Eisenacher, and G. Nichols. The path tracing revolution in the movie industry. In *ACM SIGGRAPH 2015 Courses*, pages 1–7. 2015.
- [21] D. Kirk and J. Arvo. Improved ray tagging for voxel-based ray tracing. In *Graphics Gems II*, pages 264–266. Elsevier, 1991.
- [22] D. Kopta, T. Ize, J. Spjut, E. Brunvand, A. Davis, and A. Kensler. Fast, effective bvh updates for animated scenes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 197–204, 2012.
- [23] E. P. Lafontaine and Y. D. Willems. Bi-directional path tracing. 1993.
- [24] Y. Lee and W. Lim. Shoelace formula: Connecting the area of a polygon and the vector cross product. *The Mathematics Teacher*, 110(8):631–636, 2017.
- [25] M.-K. Lefrancois and P. Gautron. Dx12 raytracing tutorial. <https://developer.nvidia.com/doc/api/directx/dx12-raytracing-tutorial>.

Bibliography

- nvidia.com/rtx/raytracing/dxr/dx12-raytracing-tutorial-part-1. Accessed: (14/11/2023).
- [26] J. D. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6:153–166, 1990.
- [27] P. Moreau and P. Clarberg. Importance sampling of many lights on the gpu. In *Ray Tracing Gems*, chapter 18. 2019.
- [28] P. Moreau and P. Clarberg. *Ray Tracing Gems*. 2019.
- [29] S. Owen. Ray-plane intersection. https://education.siggraph.org/static/HyperGraph/raytrace/rayplane_intersection.htm, 1999. Accessed: (11/01/2024).
- [30] S. Owen. Ray-box intersection. <https://education.siggraph.org/static/HyperGraph/raytrace/rtinter3.htm>, 2001. Accessed: (10/01/2024).
- [31] S. Oz. Intersection of convex polygons algorithm. <https://www.swtestacademy.com/intersection-convex-polygons-algorithm/>. Accessed: (20/03/2024).
- [32] M. Pharr. On the importance of sampling. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*, pages 207–222, 2019.
- [33] M. Pharr, W. Jakob, and G. Humphreys. *Physically based rendering: From theory to implementation*. MIT Press, 2023. Chapter 8.4.
- [34] M. Pharr, W. Jakob, and G. Humphreys. *Physically based rendering: From theory to implementation*. MIT Press, 2023.
- [35] J.-C. Prunier. Monte carlo methods in practice. <https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice/monte-carlo-integration.html>, . Accessed: (22/08/2023).
- [36] J.-C. Prunier. Mathematical foundations of monte carlo methods. <https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-mathematical-foundations/quick-introduction-to-monte-carlo-methods.html>, . Accessed: (21/08/2023).
- [37] J.-C. Prunier. Ray-triangle intersection: Geometric solution. <https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/>

- [ray-triangle-intersection-geometric-solution.html](https://www.cs.cmu.edu/~baran/ray-triangle-intersection-geometric-solution.html), . Accessed: (09/01/2024).
- [38] S. Roman et al. *An introduction to Catalan numbers*. Springer, 2015.
- [39] L. P. Santos, T. Bashford-Rogers, J. Barbosa, and P. Navrátil. Towards quantum ray tracing. *IEEE Transactions on Visualization and Computer Graphics*, 2024.
- [40] A. Shelankov and G. Pikus. Reciprocity in reflection and transmission of light. *Physical Review B*, 46(6):3326, 1992.
- [41] E. Shusterman and M. Feder. Image compression via improved quadtree decomposition algorithms. *IEEE Transactions on Image Processing*, 3(2):207–215, 1994.
- [42] K. Sigmam. Inverse transform method. Columbia University. Lecture notes.
- [43] J. Soch. Proof: Law of the unconscious statistician. <https://statproofbook.github.io/P/mean-lotus.html>. Accessed: (09/09/2023).
- [44] M. Stich, H. Friedrich, and A. Dietrich. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 7–13, 2009.
- [45] E. Veach. Chapter 9: Multiple importance sampling. <https://graphics.stanford.edu/courses/cs348b-03/papers/veach-chapter9.pdf>, .
- [46] E. Veach. Chapter 2: Monte carlo integration. <https://www.ime.usp.br/~jmstern/wp-content/uploads/2020/04/EricVeach2.pdf>, .
- [47] E. Veach. *Robust Monte Carlo methods for light transport simulation*. Stanford University, 1998.
- [48] E. Veach. *Robust Monte Carlo methods for light transport simulation*. Stanford University, 1998.
- [49] M. Vinkler, V. Havran, and J. Bittner. Bounding volume hierarchies versus kd-trees on contemporary many-core architectures. In *Proceedings of the 30th Spring Conference on Computer Graphics*, pages 29–36, 2014.
- [50] J. Vorba, J. Hanika, S. Herholz, T. Müller, J. Křivánek, and A. Keller. Path guiding in production. In *ACM SIGGRAPH 2019 Courses*, pages 1–77. 2019.
- [51] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent ray tracing. In *Computer graphics forum*, volume 20, pages 153–165. Wiley Online Library, 2001.

7| BIBLIOGRAPHY



- [52] H. Weghorst, G. Hooper, and D. P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics (TOG)*, 3(1):52–69, 1984.
- [53] J. Wei. OBB generation via principal component analysis. <https://hewjunwei.wordpress.com/2013/01/26/obb-generation-via-principal-component-analysis/>. Accessed: (02/02/2024).
- [54] D. Wodniok and M. Goesele. Construction of bounding volume hierarchies with sah cost approximation on temporary subtrees. *Computers & Graphics*, 62:41–52, 2017.
- [55] S. Woop, C. Benthin, I. Wald, G. S. Johnson, and E. Tabellion. Exploiting local orientation similarity for efficient ray traversal of hair and fur. *High Performance Graphics*, 3, 2014.
- [56] C. Wyman, M. Kettunen, D. Lin, B. Bitterli, C. Yuksel, W. Jarosz, and P. Kozlow. A gentle introduction to restir path reuse in real-time. In *A Gentle Introduction to ReSTIR Path Reuse in Real-Time*. SIGGRAPH’23 Courses, 2023.
- [57] C. Wynn. An introduction to brdf-based lighting. *Nvidia Corporation*, 4(3), 2000.
- [58] S.-E. Yoon and D. Manocha. Cache-efficient layouts of bounding volume hierarchies. In *Computer Graphics Forum*, volume 25, pages 507–516. Wiley Online Library, 2006.
- [59] M. Young. The pinhole camera. *The Phys. Teacher*, pages 648–655, 1989.
- [60] C. Yuksel. Intro to graphics 06 - 3d transformations. <https://www.youtube.com/watch?v=1z1S2kQKXD8>, 2021. Accessed: (16/05/2023).

A | Collision and Culling Algorithms

A.1. Ray-AABB Intersection

The algorithm we used to detect intersections between a ray and an AABB is the branchless slab algorithm [30].

Given a ray in the form: $r(t) = O + t \cdot d$, where O is the origin and d the direction, the main idea of the algorithm is to find the 2 values of t (\bar{t}_1 and \bar{t}_2) such that $r(\bar{t}_{1,2})$ are the points where the ray intersects the AABB.

Since the object to intersect the ray with is an axis-aligned bounding box in the min-max form, the algorithm can proceed one dimension at a time:

1. First, it finds the intersection points of the ray with the planes parallel to the yz plane, and sorts them in an ascending order with reference to the corresponding $\bar{t}_{1,2}$ values. We call the point with the smallest \bar{t} value the *closest*, and the other one the *furthest*.
2. Then it does the same with the xz plane:
 - As closest intersection point, it keeps the furthest between the 2 closest intersection points found so far (the one with the yz plane and the one with the xz plane).
 - As furthest intersection point, it keeps the closest between the 2 furthest intersection points found so far.
3. Then it does the same with the xy plane.
4. Finally, an intersection is detected only in the case where the furthest intersection point actually has an associated \bar{t} value bigger than the one of the closest point found by the algorithm.

A| Collision and Culling Algorithms

132

5. The returned \bar{t} value is the smaller one, as long as it is greater or equal to 0; otherwise it means that the origin of the ray is inside the AABB, and one of the intersection points is *behind* the ray origin.

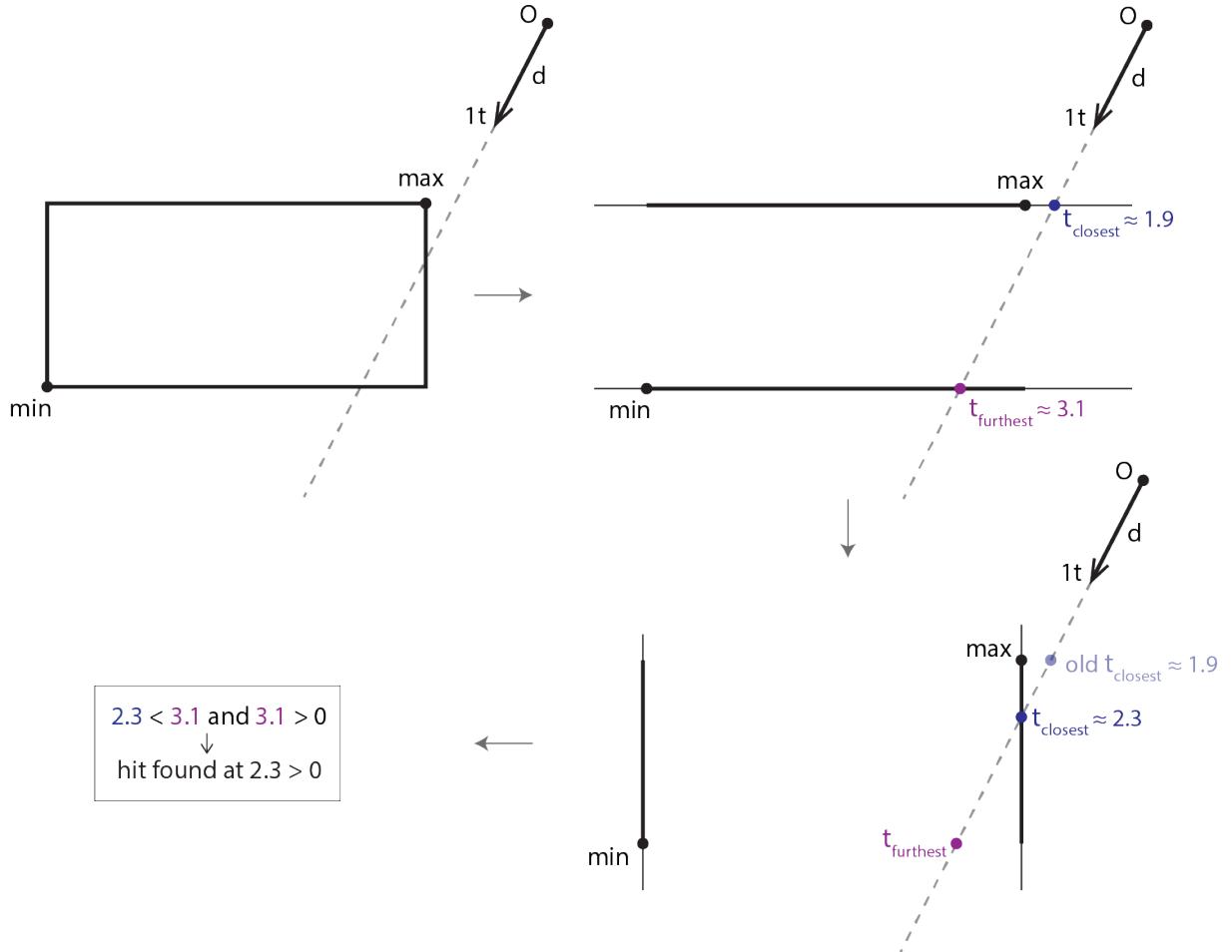


Figure A.1: Visual representation of the presented algorithm in 2 dimensions. An interactive simulation of this algorithm can be found at: <https://www.geogebra.org/m/np3tnjvb>.

Algorithm A.1 Ray-AABB branchless slab intersection algorithm in 3 dimensions

```

1: function INTERSECT(ray, aabb)
2:    $t1_x \leftarrow \frac{aabb.\min.x - ray.\origin.x}{ray.\direction.x}$   $\triangleright yz\ plane$ 
3:    $t2_x \leftarrow \frac{aabb.\max.x - ray.\origin.x}{ray.\direction.x}$ 
4:    $tMin \leftarrow \min(t1_x, t2_x)$ 
5:    $tMax \leftarrow \max(t1_x, t2_x)$ 
6:    $t1_y \leftarrow \frac{aabb.\min.y - ray.\origin.y}{ray.\direction.y}$   $\triangleright xz\ plane$ 
7:    $t2_y \leftarrow \frac{aabb.\max.y - ray.\origin.y}{ray.\direction.y}$ 
8:    $tMin \leftarrow \max(tMin, \min(t1_y, t2_y))$ 
9:    $tMax \leftarrow \min(tMax, \max(t1_y, t2_y))$ 
10:   $t1_z \leftarrow \frac{aabb.\min.z - ray.\origin.z}{ray.\direction.z}$   $\triangleright xy\ plane$ 
11:   $t2_z \leftarrow \frac{aabb.\max.z - ray.\origin.z}{ray.\direction.z}$ 
12:   $tMin \leftarrow \max(tMin, \min(t1_z, t2_z))$ 
13:   $tMax \leftarrow \min(tMax, \max(t1_z, t2_z))$ 
14:  areColliding  $\leftarrow tMax > tMin \text{ and } tMax \geq 0$ 
15:  collisionDist  $\leftarrow tMin < 0 ? tMax : tMin$ 
16:  return  $\langle \text{areColliding}, \text{collisionDist} \rangle$ 

```

It is interesting to note how, under the floating-point IEEE 754 standard, the algorithm also works when it is not possible to find an intersection point along a certain axis (i.e. when the ray is parallel to certain planes). Indeed, in such cases, the values $\overline{t_{1,2}}$ will be $\pm\infty$, and the comparisons will still be well defined.

A.2. Ray-Plane Intersection

For ray-plane intersection we decided to use this algorithm presented in the educational portal of the SIGGRAPH conference [29].

Given a ray in the form: $r(t) = O + t \cdot d$, where O is the origin and d the direction, and a plane whose normal n and a point P are known, we first check whether the plane and the ray are parallel, in which case no intersection can be found.

Then, if they are not parallel, we obtain the analytic form of the 3-dimensional plane:

$$A \cdot x + B \cdot y + C \cdot z + D = 0$$

In particular, we know a point P that is part of the plane, therefore we can obtain the D

A| Collision and Culling Algorithms



parameter:

$$\begin{aligned} A \cdot P_x + B \cdot P_y + C \cdot P_z + D &= 0 \\ \implies D &= -(A \cdot P_x + B \cdot P_y + C \cdot P_z) \end{aligned}$$

By definition, the vector formed by the parameters $[A, B, C]$ is perpendicular to the plane, therefore:

$$\begin{aligned} D &= -(n_x \cdot P_x + n_y \cdot P_y + n_z \cdot P_z) \\ \implies D &= -\langle n \cdot P \rangle \end{aligned}$$

Now that we have the parametric equation of the plane, we can force a point of the plane to also be a point of the ray:

$$\begin{aligned} A \cdot r(t)_x + B \cdot r(t)_y + C \cdot r(t)_z + D &= 0 \\ \implies A \cdot (O_x + t \cdot d_x) + B \cdot (O_y + t \cdot d_y) + C \cdot (O_z + t \cdot d_z) + D &= 0 \\ \implies t &= \frac{-\langle n \cdot O \rangle + D}{\langle n \cdot d \rangle} \end{aligned}$$

Finally, if the found \bar{t} value is negative, it means that the intersection point between the ray and the plane is *behind* the ray origin, therefore no intersection is found. Else the ray intersects the plane at point $r(\bar{t})$.

Algorithm A.2 Ray-plane intersection algorithm

```

1: function INTERSECT(ray, plane)
2:   d  $\leftarrow$  ray.direction
3:   O  $\leftarrow$  ray.origin
4:   n  $\leftarrow$  plane.normal
5:   P  $\leftarrow$  plane.point
6:   if  $\langle n \cdot d \rangle = 0$  then                                ▷ Ray is parallel to plane
7:     return  $\langle \text{false}, \_ \rangle$ 
8:   D  $\leftarrow -\langle n \cdot P \rangle
9:   t  $\leftarrow \frac{-\langle n \cdot O \rangle}{\langle n \cdot d \rangle}
10:  if t < 0 then                               ▷ Intersection point is behind ray origin
11:    return  $\langle \text{false}, \_ \rangle$ 
12:  else
13:    return  $\langle \text{true}, t \rangle$$$ 
```

A.3. Ray-Triangle Intersection

Once we have algorithms to check for ray-plane intersection (A.2) and for a point inside a 2D convex hull (A.8), we can combine them to check if a ray intersects a triangle and

to compute the coordinates of the intersection point:

1. Build a plane that has as normal the normal to the triangle, and as point any vertex of the triangle;
2. Use the ray-plane intersection algorithm (A.2) to find the coordinates of the point where the ray and the plane collide (if any);
3. Use the point inside 2D convex hull test (A.8) to determine if the intersection point is inside the triangle.

A.4. AABB-AABB Intersection

To detect a collision between 2 axis-aligned bounding boxes in the min-max form, it is sufficient to check that there is an overlap between them in all 3 dimensions. By naming the 2 AABBs as A and B we get:

$$\left\{ \begin{array}{l} A.\min_x \leq B.\max_x \\ A.\max_x \geq B.\min_x \\ A.\min_y \leq B.\max_y \\ A.\max_y \geq B.\min_y \\ A.\min_z \leq B.\max_z \\ A.\max_z \geq B.\min_z \end{array} \right.$$

A.5. Frustum-AABB Intersection

In order to detect an intersection between a frustum and an axis-aligned bounding box in the min-max form, we used a simplified version of the separating axis test (a special case of the separating hyperplane theorem) [3]. The simplification comes from the fact that we need to find the intersection of a frustum and an AABB, and not two 3D convex hulls, meaning that we can exploit some assumptions on the direction of the edges of the two objects, as we'll note below.

Before proceeding with the separating axis test, we first try a simpler AABB-AABB collision test, between the given AABB and the AABB that most tightly encloses the frustum. In case this *rejection test* gives a negative answer, we can deduce that the frustum and the AABB are not colliding. Otherwise, we must use the more expensive SAT.

The separating axis theorem in 3 dimensions states that 2 convex hulls are not colliding if and only if there is a plane that divides the space into 2 half-spaces, each fully containing one of the two convex hulls.

To find whether such a plane exists, we project the two convex hulls on certain axes, and check whether their 1D projections are overlapping. The theorem also states that if there is an axis where the projections are not overlapping it must be either:

- An axis perpendicular to one of the faces of the convex hulls, or
- An axis parallel to the cross product between an edge of the first convex hull and an edge of the second convex hull.

This consideration makes it possible to use the theorem in a concrete scenario.

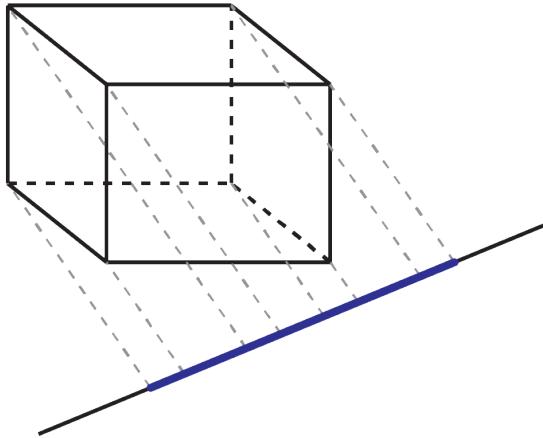


Figure A.2: The projection of an AABB on an axis.

In principle, given 2 polyhedra with 6 faces each (such as a frustum and an AABB), there should be $(6 + 6)_{normals} + (12 \cdot 12)_{cross\ products} = 156$ axis to check; but, since:

- The AABB has edges only in 3 different directions, and faces normals only in 3 different directions, and
- The frustum has edges only in 6 different directions, and faces normals only in 5 different directions

the number of checks is reduced to $(3 + 5)_{normals} + (3 \cdot 6)_{cross\ products} = 26$.

A| Collision and Culling Algorithms

137

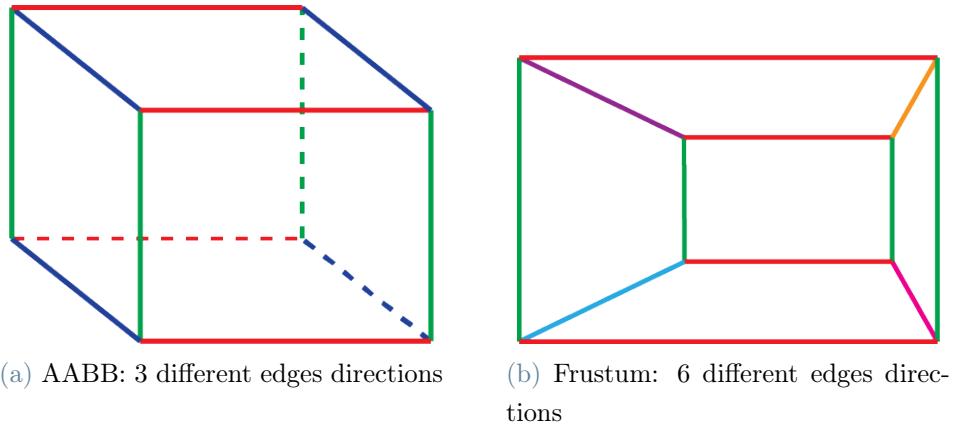
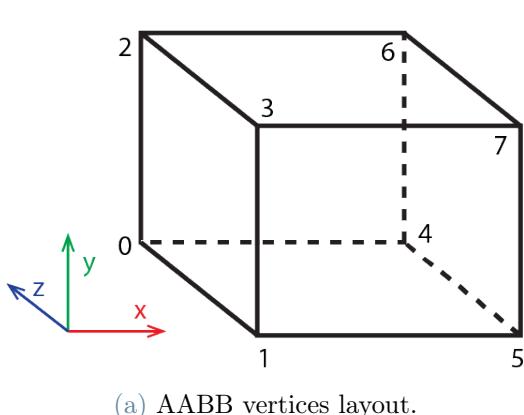


Figure A.3: In the figure the edges having the same direction are colored in the same color.

A.5.1. 1D Projections Overlapping Test

In order to detect if the 1D projections of the 3D hulls are overlapping, we identify the outermost points of each projection (namely $A_{min}, A_{max}, B_{min}, B_{max}$) and check that $B_{min} \leq A_{max}$ & $B_{max} \geq A_{min}$.

For the AABB another optimization is possible, where we detect what points will be the outermost after the projection, without actually projecting them, based on the direction of the axis:



axis direction	extremes
$x \geq 0 \ \& \ y \geq 0 \ \& \ z \leq 0$	1, 6
$x \leq 0 \ \& \ y \leq 0 \ \& \ z \geq 0$	6, 1
$x \geq 0 \ \& \ y \geq 0 \ \& \ z \geq 0$	0, 7
$x \leq 0 \ \& \ y \leq 0 \ \& \ z \leq 0$	7, 0
$x \geq 0 \ \& \ y \leq 0 \ \& \ z \leq 0$	3, 4
$x \leq 0 \ \& \ y \geq 0 \ \& \ z \geq 0$	4, 3
$x \geq 0 \ \& \ y \leq 0 \ \& \ z \geq 0$	2, 5
$x \leq 0 \ \& \ y \geq 0 \ \& \ z \leq 0$	5, 2

Figure A.4

Algorithm A.3 Ray-AABB branchless slab intersection algorithm in 3 dimensions

```

1: function INTERSECT(frustum, aabb)
2:   if !intersect(frustum.aabb, aabb) then                                ▷ AABB-AABB test
3:     return false
4:   axesToCheck  $\leftarrow (\perp \text{ frustum faces}) \cup (\perp \text{ AABB faces}) \cup (\times \text{edges})$ 
5:   for all axis  $\in$  axesToCheck do
6:     frustumExtremes  $\leftarrow$  findFrustumExtremes(frustum, axis) ▷ Returns the
vertices of the frustum that, after the projection, will be the extremes
7:     aabbExtremes  $\leftarrow$  findAabbExtremes(aabb, axis) ▷ Same as above, but uses
the discussed optimization
8:      $A_{min} \leftarrow \langle aabbExtremes.first \cdot axis \rangle$ 
9:      $A_{max} \leftarrow \langle aabbExtremes.second \cdot axis \rangle$ 
10:     $B_{min} \leftarrow \langle frustumExtremes.first \cdot axis \rangle$ 
11:     $B_{max} \leftarrow \langle frustumExtremes.second \cdot axis \rangle$ 
12:    if !( $B_{min} \leq A_{max}$  &  $B_{max} \geq A_{min}$ ) then
13:      return false
14:   return true ▷ If we haven't found any axis where there is no overlap, boxes are
colliding

```

A.6. Point inside AABB Test

To check if a point P is inside an axis-aligned bounding box in the min-max form, it is sufficient to compare its coordinates with the minimum and maximum of the AABB component-wise:

$$\begin{cases} min_x \leq P_x \leq max_x \\ min_y \leq P_y \leq max_y \\ min_z \leq P_z \leq max_z \end{cases}$$

A.7. Point inside Frustum Test

It is possible to detect whether a point is inside a 3-dimensional frustum by projecting it with the perspective matrix associated with the frustum and then comparing its coordinates, as suggested by [13].

Given the perspective matrix M associated with the frustum, we can project a point P and get: $P' = M \cdot P$; and perform the perspective division.

$P'' = \frac{P'}{P'_w}$ P'' is now in Normalized Device Coordinates (NDC) space, where the frustum

A| Collision and Culling Algorithms

is an axis-aligned bounding box that extends from $\langle -1, -1, -1 \rangle$ to $\langle 1, 1, 1 \rangle$ ¹.

It is now immediate to see that P is inside the frustum if and only if P'' is inside the AABB (see section A.6).

A simple optimization allow us to avoid the perspective division. Indeed, since in homogeneous coordinates:

$$\langle x', y', z', w' \rangle = \left\langle \frac{x'}{w'}, \frac{y'}{w'}, \frac{z'}{w'}, \frac{w'}{w'} \right\rangle = \langle x'', y'', z'', 1 \rangle$$

We can change the inequalities to check whether the point is inside the frustum from:

$$\begin{cases} -1 \leq x'' \leq 1 \\ -1 \leq y'' \leq 1 \\ -1 \leq z'' \leq 1 \end{cases} \quad \text{to:} \quad \begin{cases} -w' \leq x' \leq w' \\ -w' \leq y' \leq w' \\ -w' \leq z' \leq w' \end{cases}$$

We created a 2D visual demonstration of how it is possible to detect if a point is inside a frustum at <https://www.geogebra.org/m/ammj5mxd>.

A.8. Point inside 2D Convex Hull Test

Given a 2D convex hull in 3-dimensional space and a 3D point laying on the same plane as the hull, it is possible to use a simple inside-outside test [37] to determine whether the point is inside the convex hull.

The main idea is that the point lays inside the convex hull if and only if it is *to the right* (or *to the left*, depending on the winding order) of all the edges of the hull.

In order to determine the relative position of a point and an edge \overline{AB} , we can look at the cross product:

$$u \times v \text{ where } u = \overrightarrow{AB}, v = \overrightarrow{AP}$$

¹Based on the convention used, it is possible that the AABB in NDC space has a different size. For example, it is common an AABB extending from $\langle -1, -1, 0 \rangle$ to $\langle 1, 1, 1 \rangle$

A| Collision and Culling Algorithms

140

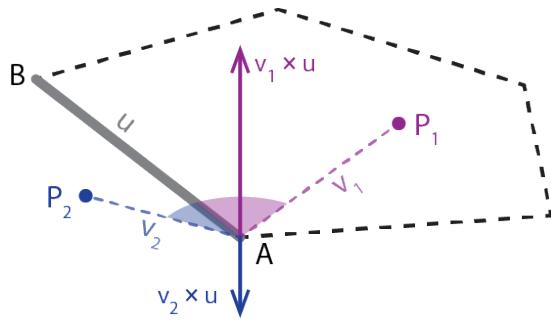


Figure A.5: Visualization of cross product.

Therefore, the strategy to determine if a point is on the same side as all the edges of the convex hull, is to compute a reference cross product, by choosing any of the edges, and then making sure that all the other cross products have the same direction. To check that 2 vectors have the same direction it is sufficient that their dot product is positive.

Algorithm A.4 Inside-outside test between a 3D point and a 2D convex hull.

```

1: function ISINSIDE( $P, hull$ )
2:    $N \leftarrow$  number of edges of  $hull$ 
3:    $u \leftarrow hull[0] - hull[N - 1]$ 
4:    $v \leftarrow P - hull[N - 1]$ 
5:    $ref \leftarrow u \times v$                                  $\triangleright$  The reference cross product
6:   for  $0 \leq i < N$  do
7:      $u \leftarrow hull[i + 1] - hull[i]$ 
8:      $v \leftarrow P - hull[i]$ 
9:      $cross \leftarrow u \times v$ 
10:    if  $\langle ref \cdot cross \rangle \leq 0$  then
11:      return false
12:   return true
```

A.9. 2D Convex Hull Culling

In order to find the overlapping region between two 2D hulls in 2-dimensional space, we can proceed as illustrated in the diagram below ([31]):

A| Collision and Culling Algorithms

141

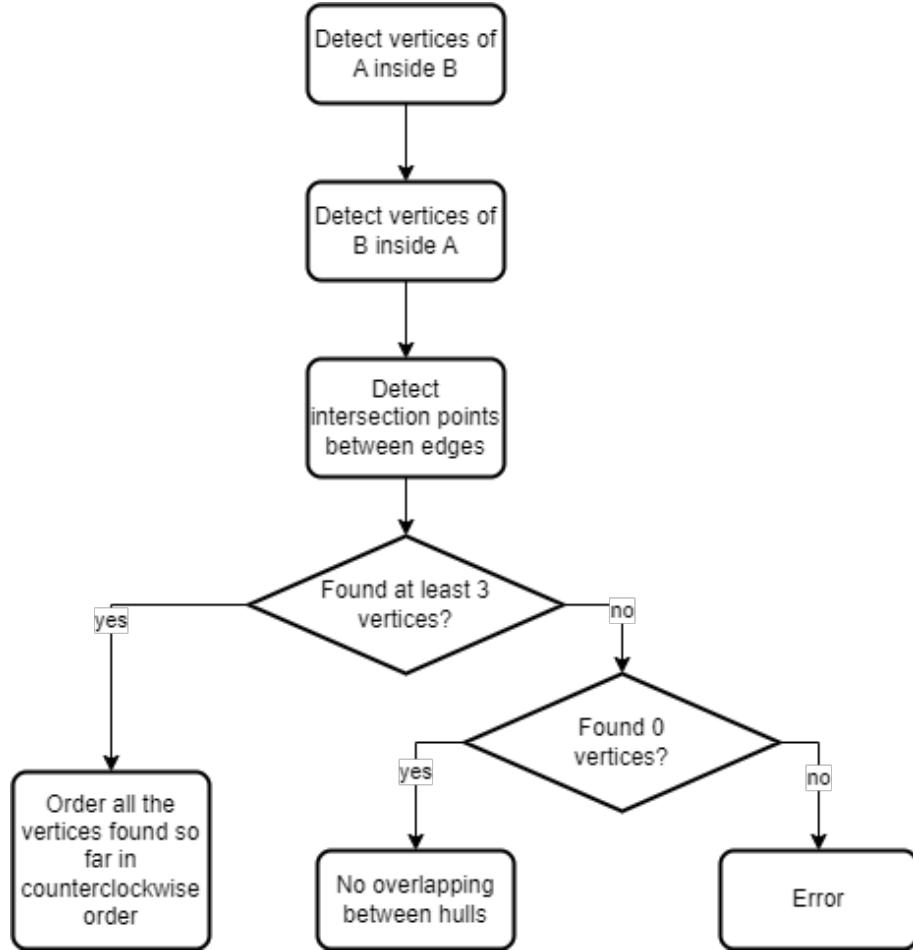


Figure A.6: General algorithm to find the overlapping region between two convex hulls called A and B .

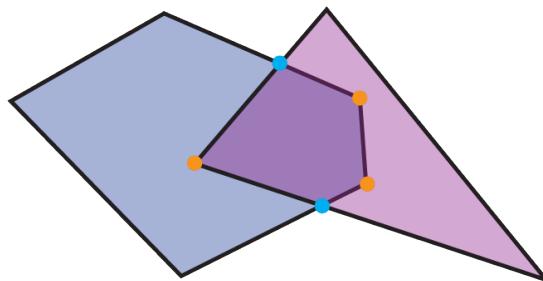


Figure A.7: Yellow vertices are found in the first 2 steps (vertices inside), whereas light blue vertices are found in the third step (edges intersections).

We'll now go through each phase and see the used algorithms.

A.9.1. Vertices inside convex hull

To find out what vertices of a hull are inside the other one, we simply looped over them and used the point inside convex hull test (A.8).

A.9.2. Edges intersections

To detect an intersection between two segments, we first have to compute the equation of the line the segment is lying on.

Given a segment \overline{PQ} , the underlying line has equation:

$$A \cdot x + B \cdot y = C$$

We can then compute the parameters of the line as:

$$\begin{cases} A = Q_y - P_y \\ B = P_x - Q_x \\ C = A \cdot P_x + B \cdot P_y \end{cases}$$

After we calculate the underlying line of both segments, with parameters $A_1, B_1, C_1, A_2, B_2, C_2$, we can compute the intersection point K of the two lines as:

$$\begin{cases} \Delta = A_1 \cdot B_2 - A_2 \cdot B_1 \\ K_x = \frac{B_2 \cdot C_1 - B_1 \cdot C_2}{\Delta} \\ K_y = \frac{A_1 \cdot C_2 - A_2 \cdot C_1}{\Delta} \end{cases}$$

We are now left with the task of verifying whether the found intersection point K is in between both segments' extremes. Let's call the first segment \overline{MN} and the second one \overline{PQ} :

$$\begin{cases} \min(M_x, N_x) \leq K_x \quad \& \\ \max(M_x, N_x) \geq K_x \quad \& \\ \min(M_y, N_y) \leq K_y \quad \& \\ \max(M_y, N_y) \geq K_y \quad \& \end{cases} \quad \text{And} \quad \begin{cases} \min(P_x, Q_x) \leq K_x \quad \& \\ \max(P_x, Q_x) \geq K_x \quad \& \\ \min(P_y, Q_y) \leq K_y \quad \& \\ \max(P_y, Q_y) \geq K_y \quad \& \end{cases}$$

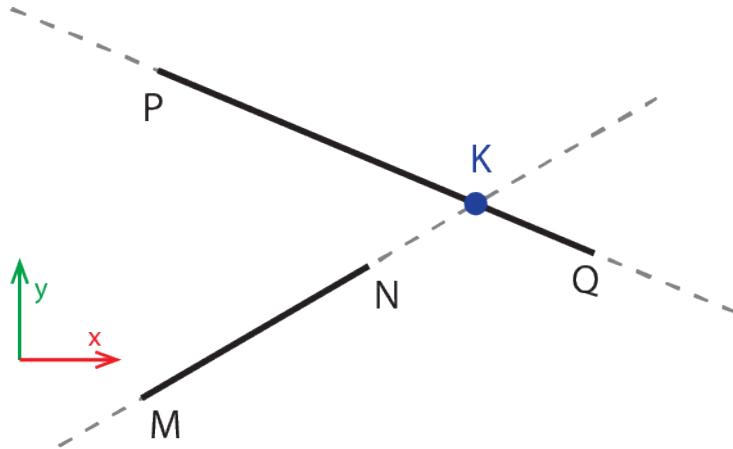


Figure A.8: In this example segments do not collide because $\max(M_x, N_x) = N_x \not\geq K_x$ or $\max(M_y, N_y) \not\geq K_y$

A.9.3. Vertices ordering

Given a set of unordered 2D points belonging to a convex hull, we want to sort them in a counterclockwise order, so that two consecutive vertices form an edge of the convex hull.

To do so we can compute the barycenter O of the set of points that, being them part of a convex hull², is necessarily inside the convex hull itself.

Now, for each vertex A_k we can calculate the vector $\overrightarrow{OA_k}$, and sort the vertices based on $\text{atan}2(\overrightarrow{OA_{ky}}, \overrightarrow{OA_{Kx}})$.

The $\text{atan}2(v_y, v_x)$ function returns the angle between the positive x-axis and the vector $v = \langle v_x, v_y \rangle$. Differently from the arctangent function, the returned angle ranges in the interval $(-\pi, \pi]$, therefore is well suited for our purpose of sorting the convex hull vertices.

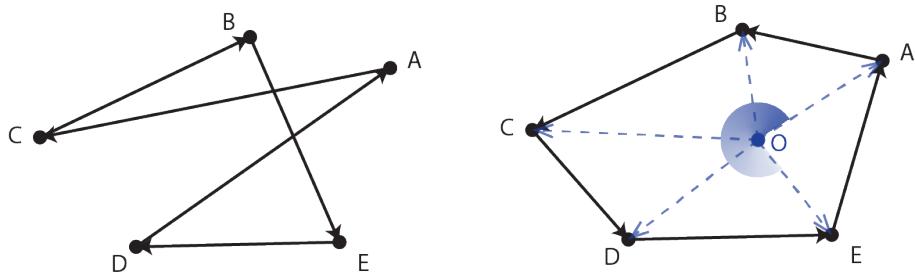


Figure A.9: Vertices of a convex hull before and after $\text{atan}2$ sorting.

²We can state that the vertices we found so far make up a convex hull because the overlapping of two convex hulls is necessarily a convex hull.

A.10. 2D Hull Area Computation

To calculate the area of a 2-dimensional hull we decided to use the Gauss's area formula, also known as the shoelace formula [24].

Given a polygon with vertices P_0, P_1, \dots, P_n , where each vertex has coordinates: $P_k = (x_k, y_k)$, its area can be found with this formula:

$$\begin{aligned} \text{Area} &= \left| \frac{1}{2} \cdot \left(\begin{vmatrix} x_0 & y_1 \\ y_0 & y_1 \end{vmatrix} + \begin{vmatrix} x_1 & y_2 \\ y_1 & y_2 \end{vmatrix} + \dots + \begin{vmatrix} x_{n-1} & y_n \\ y_{n-1} & y_n \end{vmatrix} + \begin{vmatrix} x_n & y_0 \\ y_n & y_0 \end{vmatrix} \right) \right| \\ &= \left| \frac{\sum_{i=0}^n (x_i \cdot y_{i+1} - y_i \cdot x_{i+1})}{2} \right| \end{aligned}$$

In the last formula we consider $P_0 = P_{n+1}$.

B | Sphere Cap Sampling

In this appendix we explain how we implemented the uniform random sampling from a sphere cap, given an angle γ and a radius R (figure B.1). This method is used to implement the circular sphere cap ray caster we talked about in section 5.6.2.

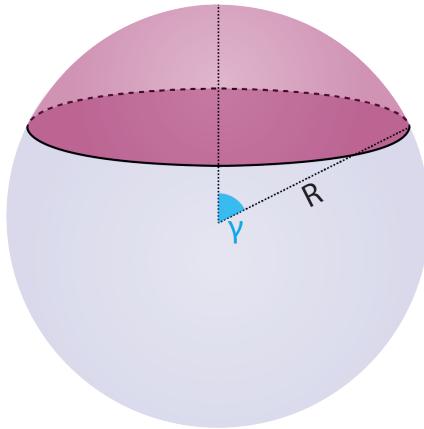


Figure B.1: A sphere cap of angle γ and radius R .

In order to sample uniformly from the sphere cap, the idea is to rotate a vector starting from the center of the sphere along two axes:

1. First, we rotate the vector along one of the axes perpendicular to it. We call this rotation *yaw*, and its values range from 0 to γ .
2. Then, we rotate the new vector we got along the axis parallel to the original vector. We call this rotation *roll*, and its values range from 0 to 2π .

The issue with this method is how we sample the first random rotation, the yaw. Indeed we cannot use uniform sampling. If we used uniform sampling, we would get more samples in the region close to the center of the sphere cap, and fewer and fewer samples the more we get far from it. The reason behind this can be easily intuited by looking at figure B.2: the more we step away from the center of the cap, the more the circumferences get bigger. If we used uniform sampling, there would be the same probability for the random sample

to be placed in any of the circumferences. However, this means that inner circumferences would have a bigger sample-density than outer ones.

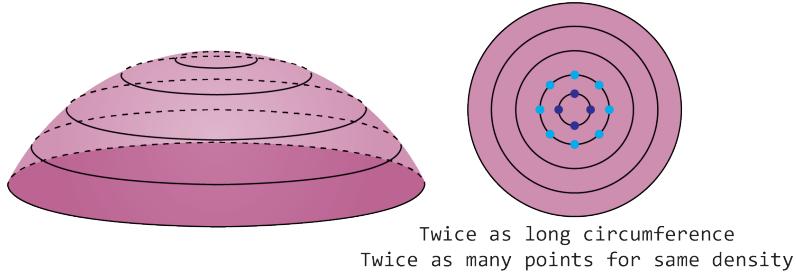


Figure B.2: The more we get far from the center of the sphere cap, the more the circles get big.

To solve this issue, we must sample the yaw with a PDF proportional to the rate of increase of the circumferences, which is a function of the yaw angle. In other words, since we want to keep the sample-density constant, if a circumference is twice as long as another one, it must receive twice the samples.

Given a sphere of ray R and oriented toward \vec{u} , the length of the circumference passing through the point of the sphere intersected by a vector forming with \vec{u} an angle γ , can be obtained by using this formula:

$$C(\gamma) = R \cdot \sin(\gamma) \cdot 2\pi$$

This means that the yaw angle must be sampled according to a PDF proportional to this function.

In order to do the sampling, we adopted the inverted CDF method [42]. This method can be summarized in four steps:

1. Normalize the function ($C(\gamma)$ in this case) to obtain the PDF.
2. Compute the CDF from the PDF.
3. Compute the inverse function to the CDF, called ICDF.
4. Now, to sample based on the original PDF, it is sufficient to obtain a uniform sample and plug it into the ICDF.

In our case, the first thing to do is to compute the normalization factor of $C(\gamma)$. To obtain it, it is sufficient to integrate the function from 0 to γ :

$$k_{norm}(\gamma) = \int_0^\gamma C(a)da$$

One problem is that we cannot compute this integral in our C++ code each time, because it would require too much computation effort. Therefore we decided to approximate the normalization factor with a similar closed-form function. In particular, with trial and error, we obtained this function, which approximates the integral function well, as it can be seen in figure B.3:

$$k_{approx}(\gamma) = 2.85 \cdot a^{1.81}$$

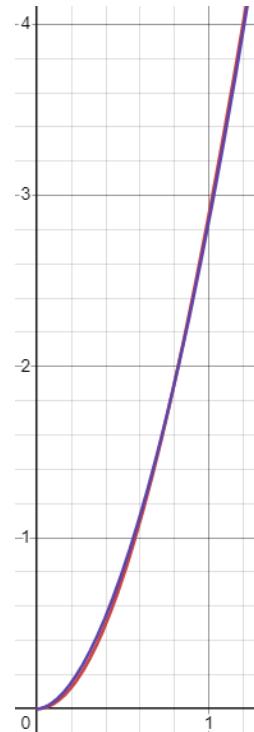


Figure B.3: In red the exact function, in purple the approximation.

We can now compute the PDF:

$$pdf(x) = \frac{C(x)}{k_{approx}(\gamma)}$$

The next step is to compute the CDF from the PDF. The CDF is obtained by integrating the PDF. In our case the CDF has form:

B| Sphere Cap Sampling

148

$$cdf(x) = -\frac{2\pi \cdot R}{k_{approx}(\gamma)} \cdot \cos(x) + \frac{2\pi \cdot R}{k_{approx}(\gamma)}$$

The third step is to compute the inverted CDF, which in our case is:

$$icdf(x) = \arccos\left(\frac{\frac{2\pi \cdot R}{k_{approx}(\gamma)} - x}{\frac{2\pi \cdot R}{k_{approx}(\gamma)}}\right)$$

Now that we obtained the ICDF, we can get uniform random samples $X_{uniform}$, and compute the random samples according to the PDF in this way:

$$X_{pdf} = icdf(X_{uniform})$$

Until now we have shown how to get the yaw rotation angle. The second step to sample a point uniformly from a sphere cap, is to rotate the new vector by a uniform random angle in the range o to 2π . The axis of rotation is parallel to the original vector.

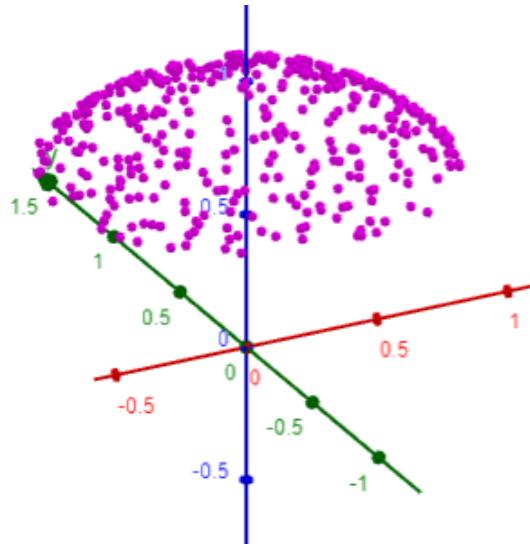


Figure B.4: Uniform random samples on a sphere cap.

List of Figures

1	Visualization of the process we described above.	1
2	Recursiveness of Kajiya's rendering equation.	3
3	A BVH built with our heuristics over a radial ray distribution, as displayed by our Unity visualizer.	4
4	How a node can be split. SPFH would choose the parallel plane	6
5	An octree as displayed by our Unity visualizer.	7
1.1	The first figure is from the main camera PoV, the second one from the light source PoV. The second figure represents depth: the closer a point is to the light source, the darker. The blue point is in shadow, because the corresponding point in the shadow map is further away than the stored depth.	10
1.2	How rays are cast in backward ray tracing.	12
1.3	Three possible types of light sources.	14
1.4	A visual representation of the Kajiya rendering equation from [4].	15
1.5	The geometry term.	15
1.6	Recursiveness of the integral term of the Kajiya rendering equation.	16
1.7	Monte-Carlo area approximation. When we have many samples, sometimes the area will be overestimated, some other times underestimated.	17
1.8	Noise in a ray traced image.	21
1.9	With NEE light is directly cast toward direct light sources. With path guiding, indirect illumination is taken into account in order to build a better sampling PDF, at a higher cost.	23
1.10	A triangular mesh.	25
1.11	The uniform grid in 2D.	27
1.12	A 2D octree (quadtree).	27
1.13	How an octree adapts to a <i>teapot in the stadium</i> kind of scene.	28
1.14	A kd-tree in 2D.	29
1.15	Bounding circle (2D bounding sphere) can present large slack spaces.	30
1.16	A 2D OBB.	31

1.17 A 2D AABB.	31
1.18 A 2-dimensional BVH.	32
1.19 Triangles splitting via arbitrary plane.	34
1.20 With binning it is possible to lose some cuts, but the most relevant ones are always found, for example the cut dividing the blue and purple triangles.	35
1.21 Visual relationship between AABB area and hit probability in 2D, under the assumption rays are uniformly distributed.	37
2.1 With importance sampling rays are not distributed uniformly in the scene, but some artifacts naturally arise.	41
2.2 The two AABBs have the same hit probability according to SAH, but one of the two gets hit many more times in a region of parallel rays.	41
2.3 Parallel rays distributions arise in proximity of planar area light sources.	42
2.4 Orthographic projection of an AABB.	43
2.5 Point rays distributions arise in proximity of point light sources.	43
2.6 Perspective projection of an AABB.	44
2.7 A BVH seen in our Unity visualizer. The father node is green, and its two children are red and blue.	46
3.1 A scene with multiple relevant ray distributions.	48
3.2 Plane influence areas are described by OBBs, whereas point influence areas are described by frustums.	48
3.3 Visualization of the method to verify if the direction of a ray is affine with a point influence area.	49
3.4 Example of why it is necessary to use a conservative approach.	50
3.5 Example of how it is possible that a hit is found in the global BVH, but not in the local one.	51
3.6 The wireframe of the octree built on a single point influence area.	52
4.1 Overlapping projected area between two nodes' AABBs.	54
4.2 Overlapping caused by the triangle extension.	55
4.3 Visual illustration of the possible splitting plane orientations and how a ray can hit the children AABBs.	55
5.1 The BVH built over the Blender Suzanne mesh, as seen in our visualizer.	63
5.2 Check to detect if a point is inside an OBB.	66
5.3 If the point (purple) is outside the AABB, it is necessarily outside the OBB too. The inverse (blue and light blue) is not guaranteed.	68
5.4 How a frustum can be defined.	69

5.5	Visualization of the orthographic projection matrix: it shrinks the rectangle parallelepiped viewing volume to the canonical view volume.	72
5.6	The orthographically projected hull of an AABB can be subdivided in 3 parallelograms.	74
5.7	The projected hull of an AABB. The blue points are contour points, the purple points are internal points.	74
5.8	Geometrical visualization of how to compute the right plane starting from the horizontal field of view and the near plane. $\tan(\alpha_{FoV}) = \frac{\text{opposite}}{\text{adjacent}}$. . .	77
5.9	Considering the x dimension, the blue eye is inside the range of the AABB, whereas the purple one is not. The purple eye can also see the right face. .	79
5.10	The image is recursively subdivided, until a cells contains pixels of just one color (or a maximum depth is reached).	82
5.11	An example in 2 dimensions of a quadtree built upon influence areas. . . .	83
5.12	Different cases of the relative position of an octree cell and the influence areas' regions	83
5.13	Algorithm to build the top level octree.	84
5.14	The indexing layout of the children of an octree node.	85
5.15	5-level octree built on a single point influence area.	87
5.16	4-level octree built on a single point influence area.	88
6.1	Random 100	92
6.2	Random 1000	92
6.3	Suzanne	92
6.4	Cottage	93
6.5	Cottage Walls	93
6.6	Wood	93
6.7	Crowd	93
6.8	Sponza	94
6.9	Accuracy of PAH compared to SAH.	96
6.10	The 2D projected hull of the AABB is not fully contained in the viewing area of the near plane.	97
6.11	Cost of a BVH built with PAH and SPFH compared to one built with SAH and LSPH.	99
6.12	Number of intersections per ray during traversal.	100
6.13	Average time per node spent inside the compute cost function for PAH and SAH.	101

6.14 Average time spent inside the compute splitting plane function for SPFH and LSPH.	102
6.15 The rays that miss in the local BVH are all concentrated in a section of the scene.	104
6.16 As the difference in traversal cost of our method and the state-of-the-art method goes to 0, also the difference between the calculated breakeven hit percentage and the measured one tends to 0.	105
6.17 Comparison of how much faster SPFH and SAH is compared to SPFH and PAH.	106
6.18 Comparison of how much faster PAH and LSPH are compared to SAH and LSPH.	107
6.19 Issue when using PAH with overlapping children. Father node is green, big child is blue and small child is red.	107
6.20 How to compute the overlapping between siblings.	108
6.21 High overlapping leads to worse BVH cost.	109
6.22 Overlapping percentage at BVH level 4.	110
6.23 Overlapping reduction works best if the rays are parallel to a cartesian axis.	111
6.24 Overlapping percentage by level.	111
6.25 Average time required to complete the queries by octrees of different levels, and AABBs for regions.	113
6.26 How much faster the octree is to traverse compared to the AABBs for regions built over the same influence areas.	114
6.27 Analysis of the error an octree with a given amount of levels introduces in its queries, and how much faster it is compared to the AABBs for regions structure.	115
A.1 Visual representation of the presented algorithm in 2 dimensions. An interactive simulation of this algorithm can be found at: https://www.geogebra.org/m/np3tnjvb	132
A.2 The projection of an AABB on an axis.	136
A.3 In the figure the edges having the same direction are colored in the same color.	137
A.4	137
A.5 Visualization of cross product.	140
A.6 General algorithm to find the overlapping region between two convex hulls called A and B	141

A.7	Yellow vertices are found in the first 2 steps (vertices inside), whereas light blue vertices are found in the third step (edges intersections)	141
A.8	In this example segments do not collide because $\max(M_x, N_x) = N_x \not\geq K_x$ or $\max(M_y, N_y) \not\geq K_y$	143
A.9	Vertices of a convex hull before and after <i>atan2</i> sorting.	143
B.1	A sphere cap of angle γ and radius R	145
B.2	The more we get far from the center of the sphere cap, the more the circles get big.	146
B.3	In red the exact function, in purple the approximation.	147
B.4	Uniform random samples on a sphere cap.	148

List of Tables

6.1	Octree levels and BVH traversal cost.	116
6.2	Base values for properties.	116
6.3	Summary of our experiments with different properties values. Underlined values are the default ones.	118

List of Symbols

Symbol	Description
PAH	Projected Area Heuristic
SAH	Surface Area Heuristic
SPFH	Splitting Plane Facing Heuristic
LSPH	Longest Splitting Plane Heuristic
BVH	Bounding Volume Hierarchy
AABB	Axis-Aligned Bounding Box
OBB	Oriented Bounding Box
SAT	Separating Axis Test

Ringraziamenti

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ultricies integer quis auctor elit sed vulputate mi. Accumsan sit amet nulla facilisi morbi. Suspendisse potenti nullam ac tortor vitae purus faucibus. Ultricies lacus sed turpis tincidunt id. Sit amet mauris commodo quis imperdiet. Arcu bibendum at varius vel. Venenatis urna cursus eget nunc. Mus mauris vitae ultricies leo integer malesuada nunc vel. Sodales neque sodales ut etiam sit. Pellentesque dignissim enim sit amet venenatis urna cursus eget nunc. Condimentum mattis pellentesque id nibh tortor id aliquet lectus. Ultrices gravida dictum fusce ut placerat orci nulla pellentesque dignissim. Faucibus pulvinar elementum integer enim neque. Morbi tincidunt augue interdum velit euismod in pellentesque massa.

A diam maecenas sed enim ut sem viverra aliquet eget. Viverra aliquet eget sit amet tellus cras. Tellus at urna condimentum mattis pellentesque. Quis viverra nibh cras pulvinar. Posuere morbi leo urna molestie at elementum. Aenean euismod elementum nisi quis eleifend quam. In hac habitasse platea dictumst vestibulum rhoncus. Nullam non nisi est sit amet facilisis magna etiam tempor. Neque laoreet suspendisse interdum consectetur libero. Vitae auctor eu augue ut lectus arcu bibendum. Ipsum consequat nisl vel pretium lectus quam. Velit dignissim sodales ut eu sem. Odio morbi quis commodo odio. Lectus nulla at volutpat diam. Neque gravida in fermentum et sollicitudin ac. Nunc non blandit massa enim nec dui nunc. Quisque id diam vel quam elementum pulvinar etiam non quam. Consequat id porta nibh venenatis cras sed felis. Vitae justo eget magna fermentum iaculis eu non diam. Mi sit amet mauris commodo.

