POLITECNICO
MILANO 1863

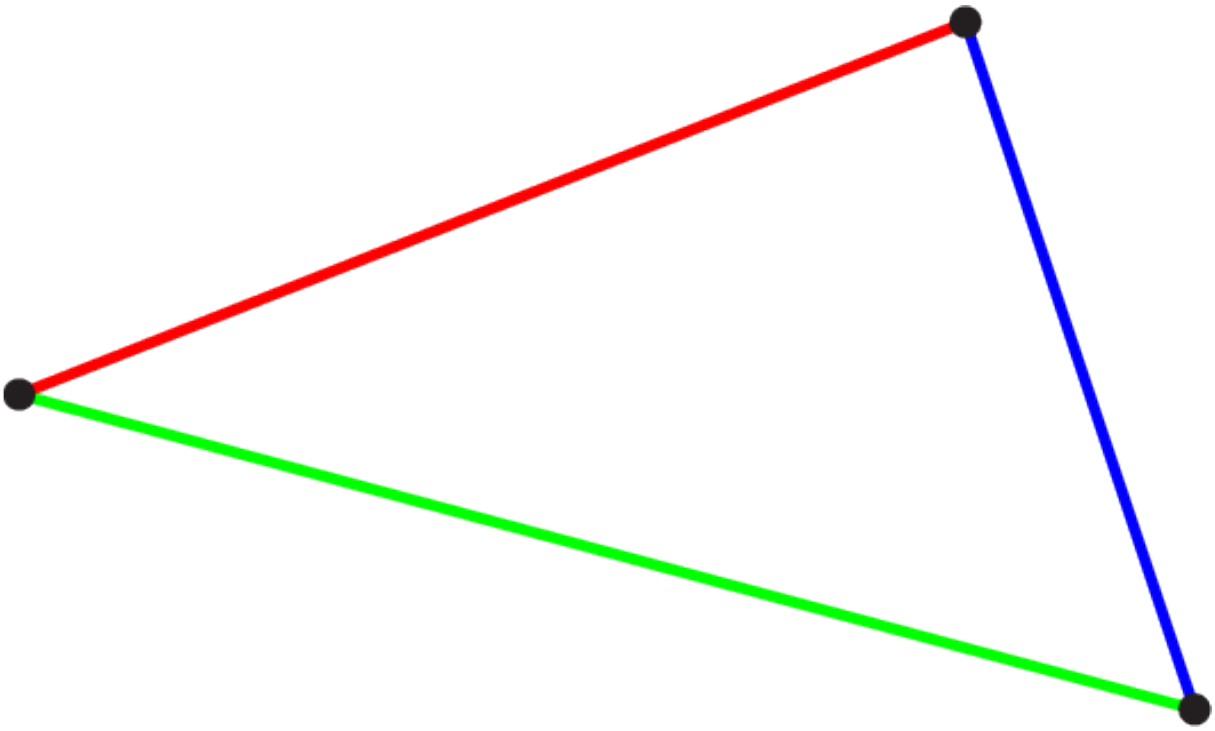
SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

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 work we propose two novel heuristics that work in pairs 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**) 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

Introduction

Structure

- 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.
- 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.
  - 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 coincisely 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, can

use this section 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 sections. 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.
- Eventually, in chapter 8 we will draw the main take-away points of our work, and produce a list of possible future development directions for the novel topics discussed in our thesis.

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

illumination and caustics. Some of the most accurate ray tracing algorithms can even simulate quantum effects of light [30].

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 [4]. 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 [17], in particular a very accurate algorithm called bi-directional path tracing [20]. 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

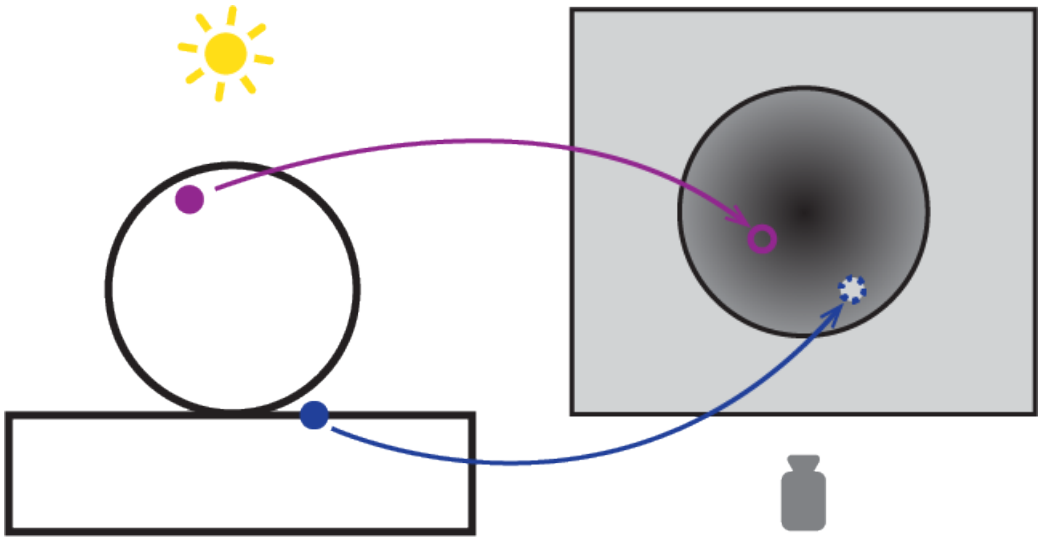


Figure 1.1: The first figure is from the main camera PoV, the second one from the light source PoV. The second figure represents depth: the closer a point is to the light source, the darker. The blue point is in shadow, because the corresponding point in the shadow map is further away than the stored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

every 16.5ms to achieve 60 fps, which can be considered today's standard by many PC videogames.

This starking 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 [5].

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. These optimizations don't alter 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 tracing[1].

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 [46]. 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.

[1] In 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.

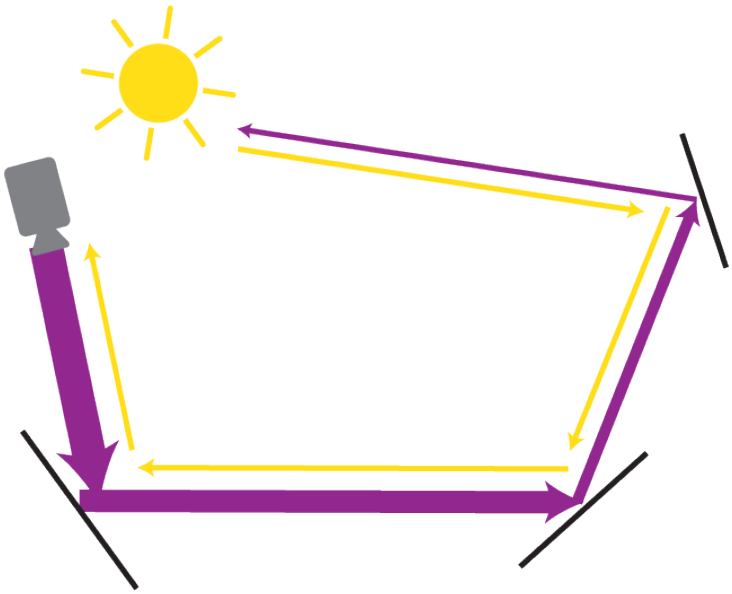


Figure 1.2: How rays are cast in backward ray tracing.

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 *importons*, which can be considered the dual concept of *photons* [6]. Thanks to the light reciprocity principle [31], which states that light transmits in the same way in both directions, tracing photons or importons 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 the 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 [14], 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.

(a) Point light (b) Plane area light (c) Spotlight

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 $BRDF$ [45]. The $BRDF$ tells us how much of the incoming light is reflected towards the direction $\bar{\omega}_o$.

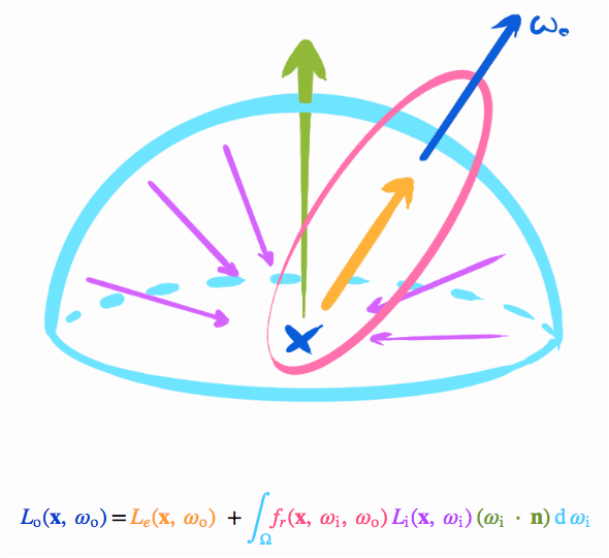


Figure 1.4: A visual representation of the Kajiya rendering equation from [3].

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 $L \cdot \cos(\alpha)$.

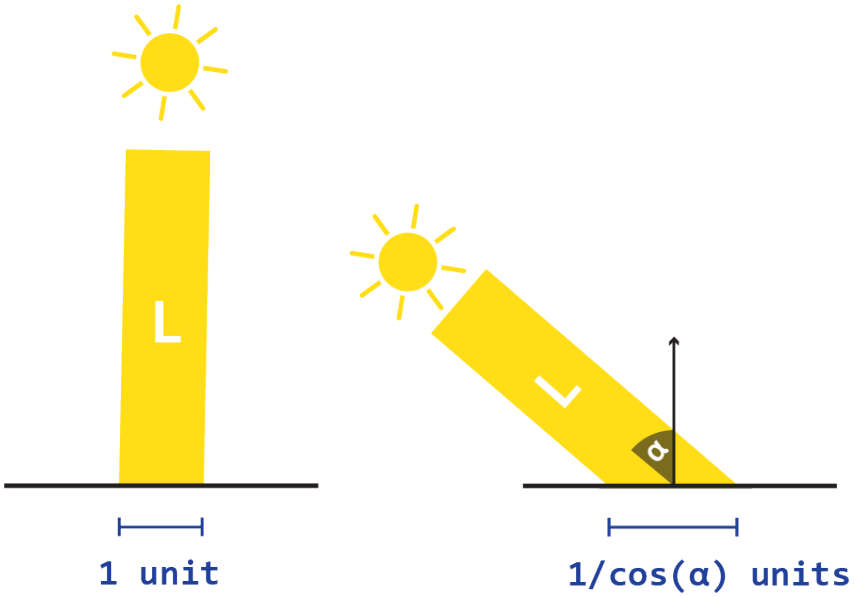


Figure 1.5: The geometry term.

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 $\overline{\omega_i}$, we can cast a probe ray R_1 from point $\overline{x}$ toward $\overline{\omega_i}$. The probe ray will hit another point $\overline{y}$. Now, since we want to compute how much light R_1 is carrying towards $\overline{x}$, we have to resolve the rendering equation at point $\overline{y}$ and with outgoing direction $-\overline{\omega_i}$, namely $L_o(\overline{y}, -\overline{\omega_i})$. 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).

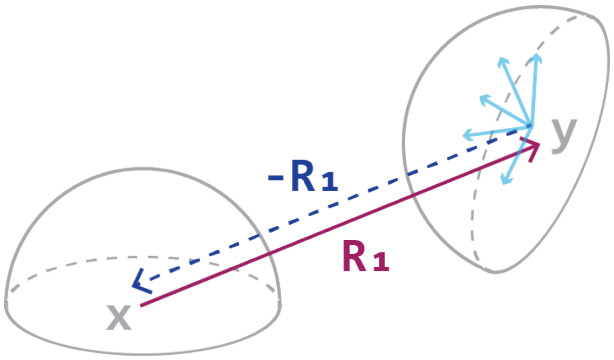


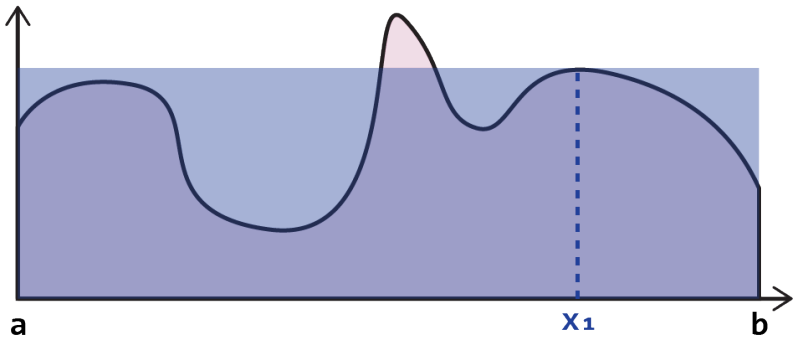
Figure 1.6: Recursiveness of the integral term of the Kajiya rendering equation.

The process we've just described only returns the amount of light incoming to $\overline{x}$ from an arbitrary direction $\overline{\omega_i}$. 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.

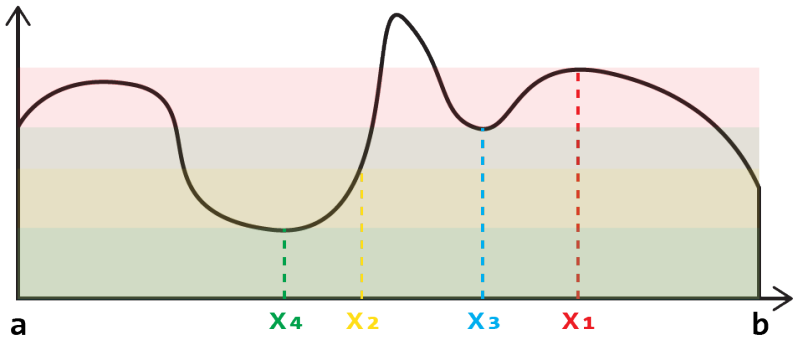
1.2.2. Monte-Carlo Integration

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 [35] 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 $\overline{ab}$ and height $f(k_0)$ as $\overline{ab} \cdot f(k_0)$. 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_rs, we'll usually get a better 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 hame 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 [37] [26]. 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:

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

In order to go from line 2 to line 3, it is important to remember the law of the uncounscious statistician (LOTUS) [32], 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 [37] [26]:

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

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

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

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 [27].

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.

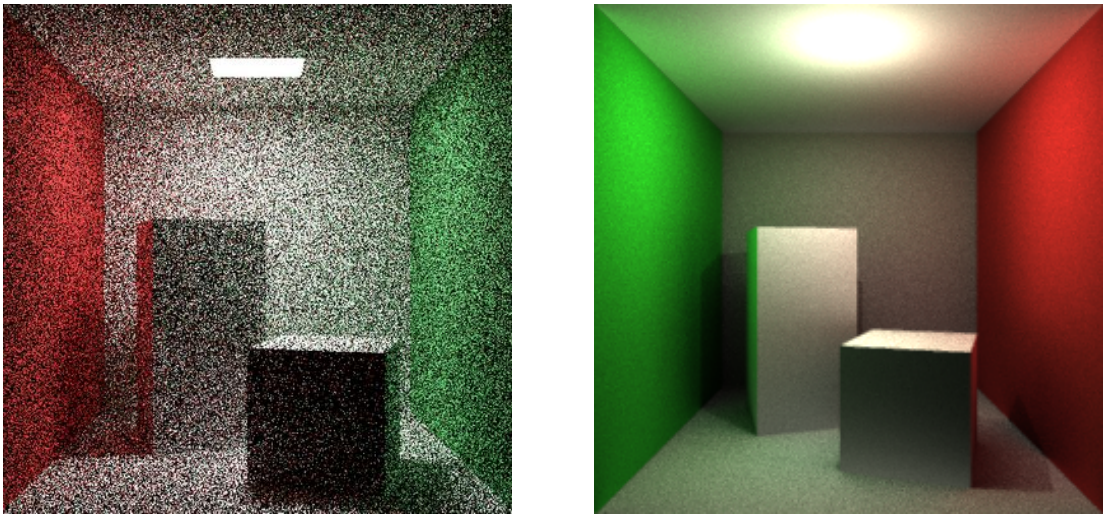
We can now calculate the variance and standard deviation of the Monte-Carlo estimator $\sigma[\langle F^N \rangle]$ in order to show the convergence rate of this technique. Here we use the random variable Y as $\frac{f(X)}{p(X)}$:

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

In the above proof [36] 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 more and more accurately. 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.



(a) Noisy image (b) Less noisy 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.

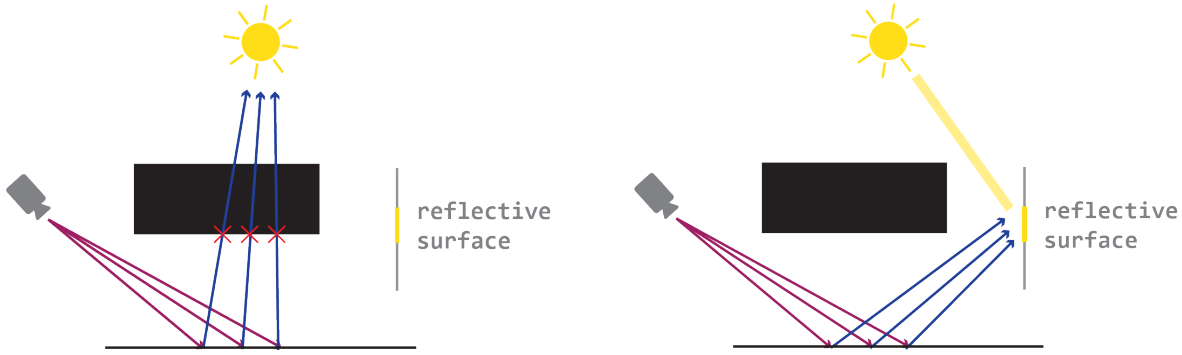
BRDF sampling The sampling PDF is proportional to the BRDF. BRDFs can assume complex analytical forms, therefore sampling the BRDF is not always easy.

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*: [22].

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 [39], 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 [9].



(a) Next event estimation (b) Path guiding

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). Since the technique is an extension to importance sampling, it will be described in appendix B.

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 [40]. 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 [16], signed distance fields [8] and implicit functions that work best with the ray marching algorithm [13], 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 trianglular meshes via rasterization, even though in the last few years their architecture has been made more flexible.

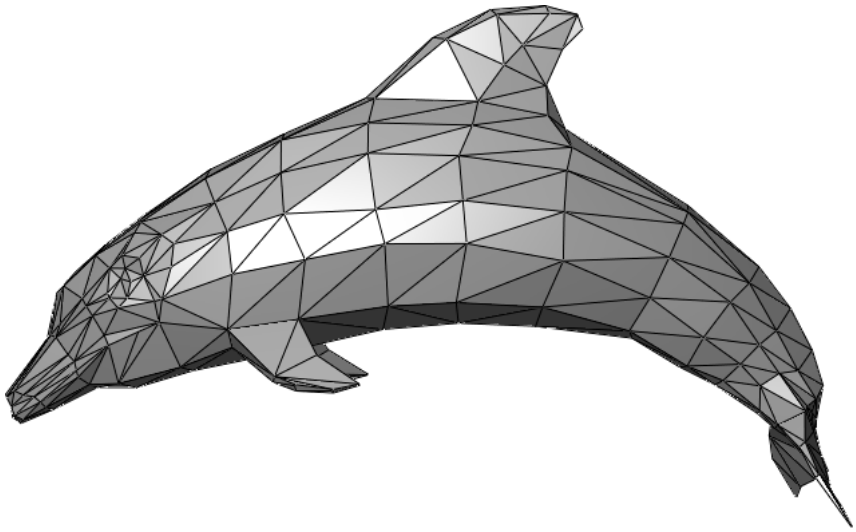


Figure 1.10: A triangular mesh.

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 context 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 [7] or more recent methods such as Nanite [15].

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 one[2].

The naive way of detecting the intersection is the brute-force solution to iterate 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, because, 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

[2]In 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.

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.

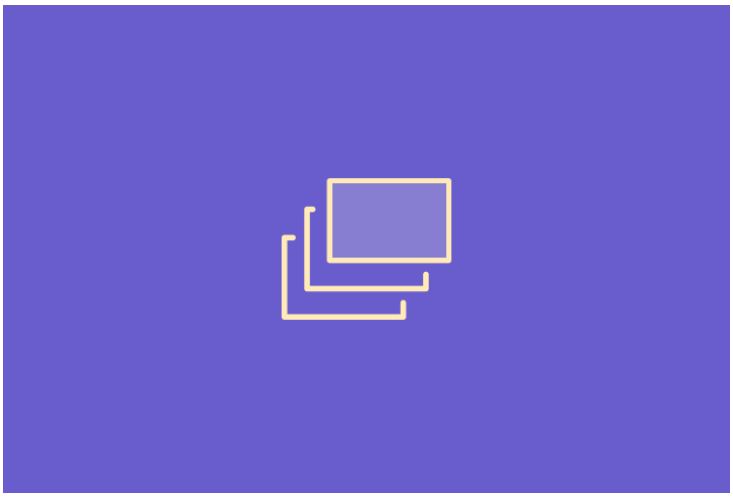


Figure 1.11: The uniform grid in 2D.

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 [18].

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 level[3], 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.

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

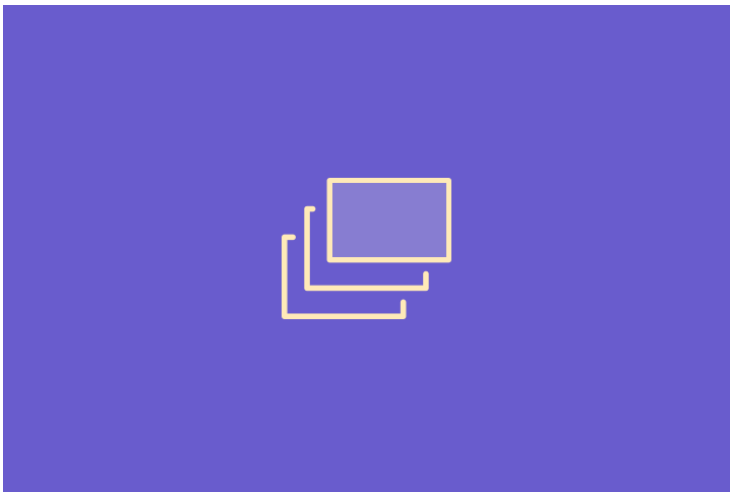


Figure 1.12: A 2D octree.

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.

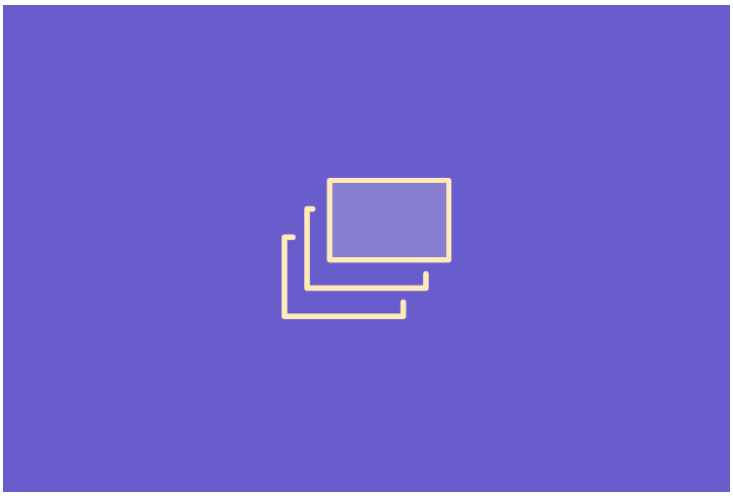


Figure 1.13: How an octree adapts to a *teapot in the stadium* kind of scene.

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.

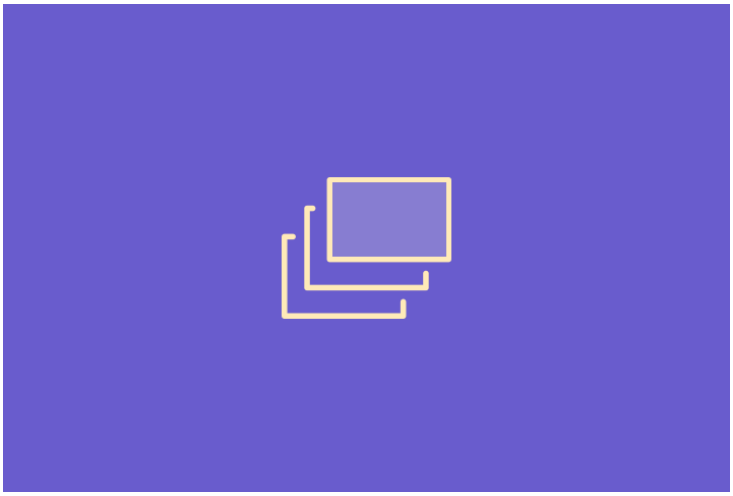


Figure 1.14: A kd-tree in 2D.

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 [41] is a binary tree[4] 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 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 thightly. 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.

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

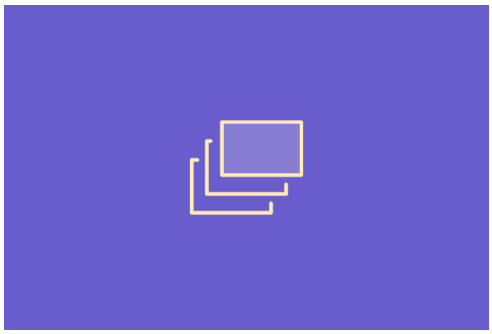


Figure 1.15: Bounding circle (2D bounding sphere) can present large slack spaces.

At the opposite side of the spectrum there are polygonal bounding volumes. In this case they are able to thightly 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 parallelepiped. 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 [42], 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 [44].

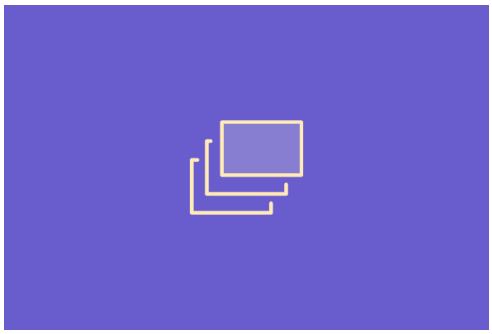


Figure 1.16: A 2D OBB.

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.

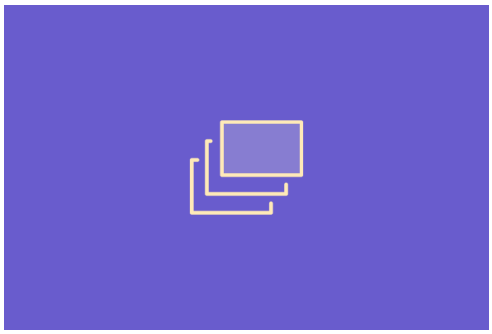


Figure 1.17: A 2D AABB.

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.

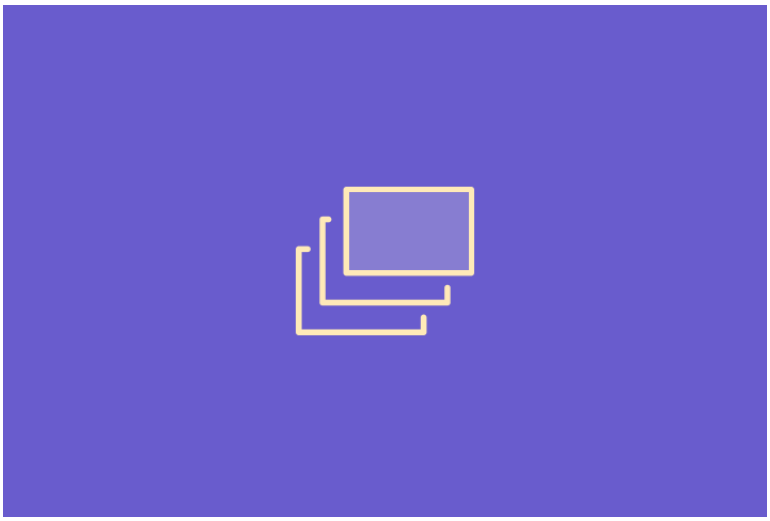


Figure 1.18: A 2-dimensional BVH.

Despite it is not clear that BVHs' raw performance is better than kd-trees', especially on large scenes [38], 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 [19].

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 [33] and [43].

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 [29]: $C_n = \frac{(2n)!}{(n+1)!\cdot n!}$. 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 fo building a BVH can be summarized in these steps:

Algorithm 1.1 Summarized BVH construction algorithm.

```
1:  function BuildBvh(triangles)
2:      leftNode ← [ ]
3:      rightNode ← [ ]
4:      cost ← ∞
5:      loop cost < threshold or all possible splits attempted
6:          [leftNodeTmp, rightNodeTmp] ← SplitTriangles(triangles, howToSplit)
7:          costTmp ← ComputeCost(leftNodeTmp, rightNodeTmp)
8:          if costTmp < cost then
9:              leftNode ← leftNodeTmp
10:             rightNode ← rightNodeTmp
11:             cost ← 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.

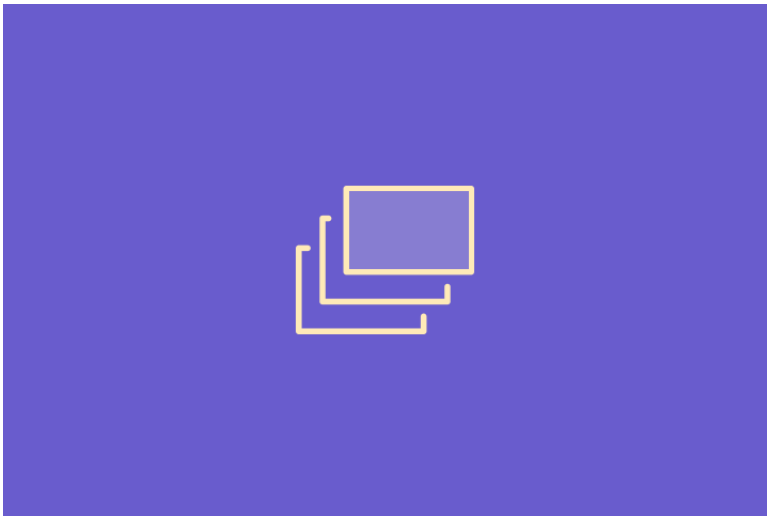


Figure 1.19: Triangles splitting via arbitrary plane.

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 splitted, 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 binnin it is possible to lose some cuts, but the most relevant ones are always found.

After generating the two subsets of triangle, 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 ← (∞, ∞, ∞)
3:     max ← (−∞, −∞, −∞)
4:     for all t ∈ triangles do                ▷ t.v_k is the k^th vertex of the triangle t
5:         min.x ← Min(min.x, t.v_0.x, t.v_1, t.v_2.x)
6:         max.x ← Max(max.x, t.v_0.x, t.v_1, t.v_2.x)
7:         min.y ← Min(min.y, t.v_0.y, t.v_1, t.v_2.y)
8:         max.y ← Max(max.y, t.v_0.y, t.v_1, t.v_2.y)
9:         min.z ← Min(min.z, t.v_0.z, t.v_1, t.v_2.z)
10:        max.z ← Max(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), 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:

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

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:

$$p(ray\ hits\ K) = \frac{Area_K}{Area_{root}}$$

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.

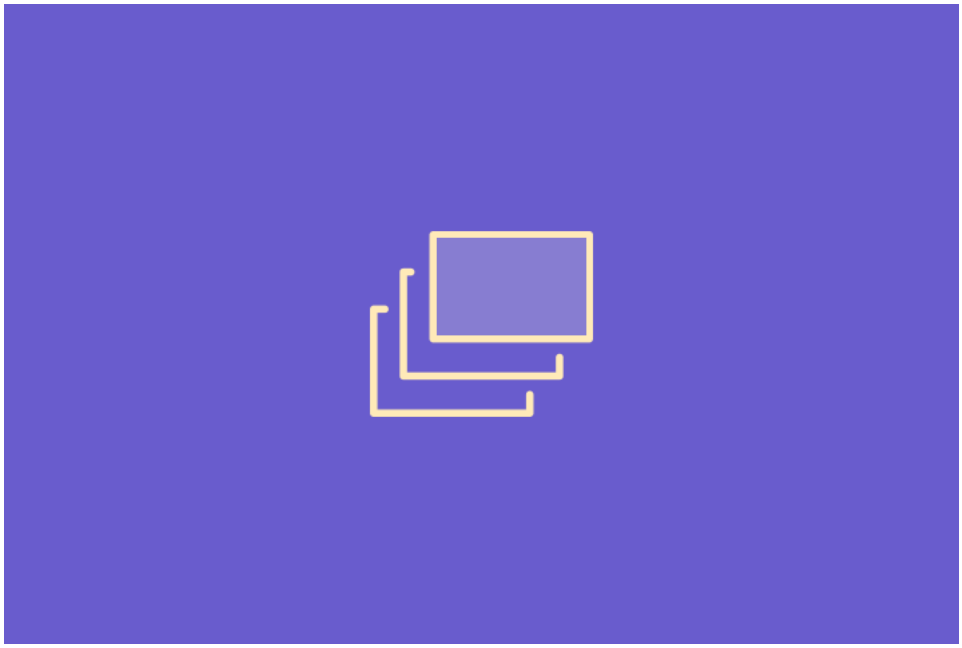


Figure 1.21: Visual relationship between AABB area and hit probability in 2D, under the assumption rays are uniformly distributed.

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.

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:

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

TLAS and BLAS

TODO

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

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 [10], 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.

(a) Without importance sampling (b) With importance sampling

Figure 2.1: With importance sampling rays are not distributed uniformly in the scene.

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).

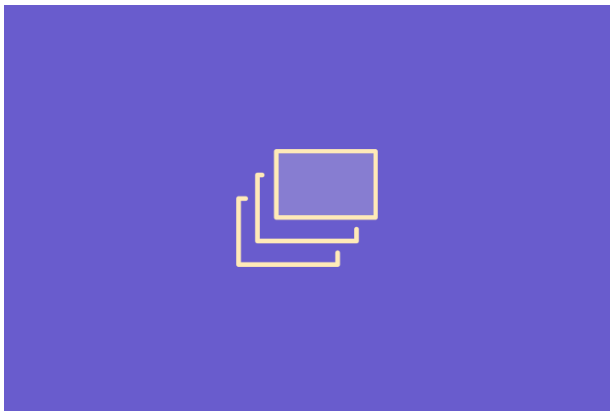


Figure 2.2: A flat but long AABB in a region of rays parallel to its long side.

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 [12] 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 plane area light sources.

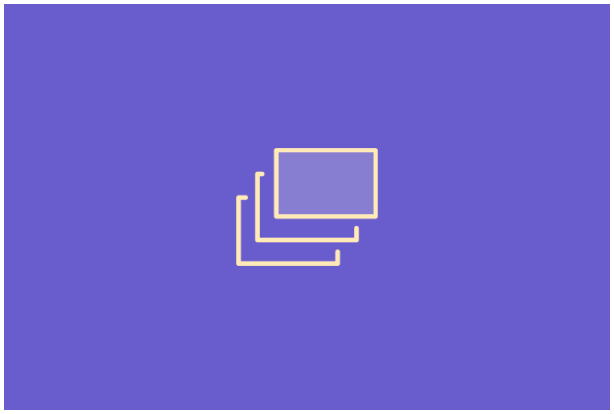


Figure 2.3: Parallel rays distributions arise in proximity of plane area light sources.

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.

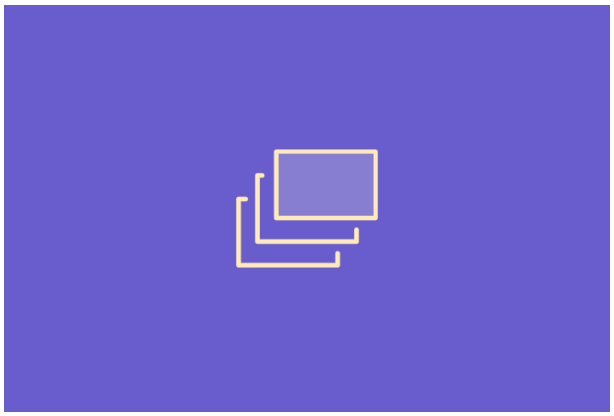


Figure 2.4: If rays are parallel to a cartesian axis, the extension of the AABB in that dimension is irrelevant.

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.

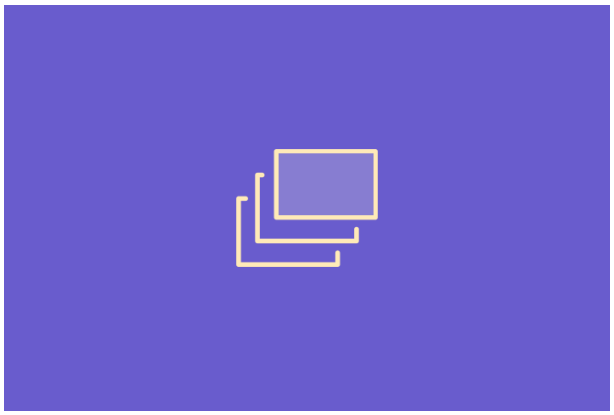


Figure 2.5: Orthographic projection of an AABB.

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

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.

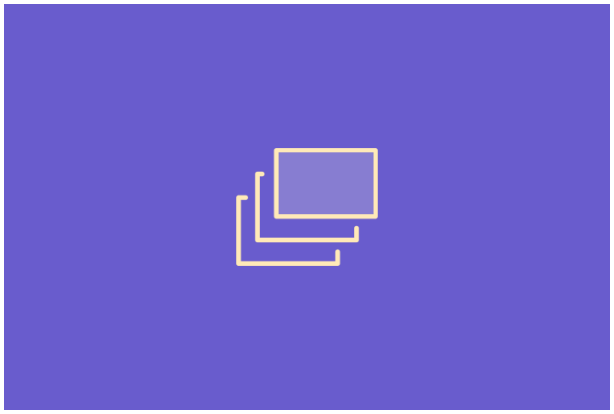


Figure 2.6: Point rays distributions arise in proximity of point light sources.

Also in this case we can observe a relationship between the probability an AABB in this area 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 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:

$$p(ray\ hits\ K) = \frac{Area_K}{Area_{root}}$$

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

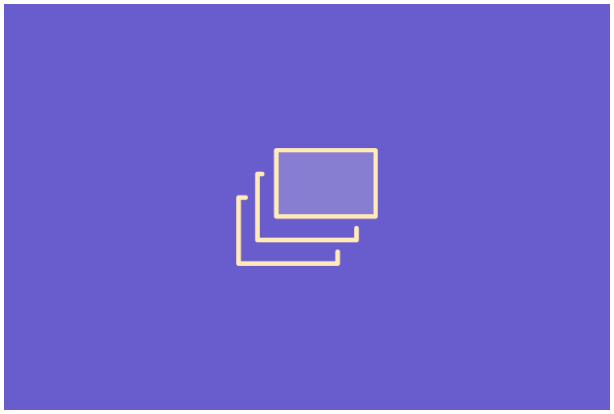


Figure 2.7: Perspective projection of an AABB.

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:

$$Cost_{PAH}(K) = \frac{ProjArea_K}{ProjArea_{root}} \cdot \#triangles_K \cdot 1.2$$

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

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.

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

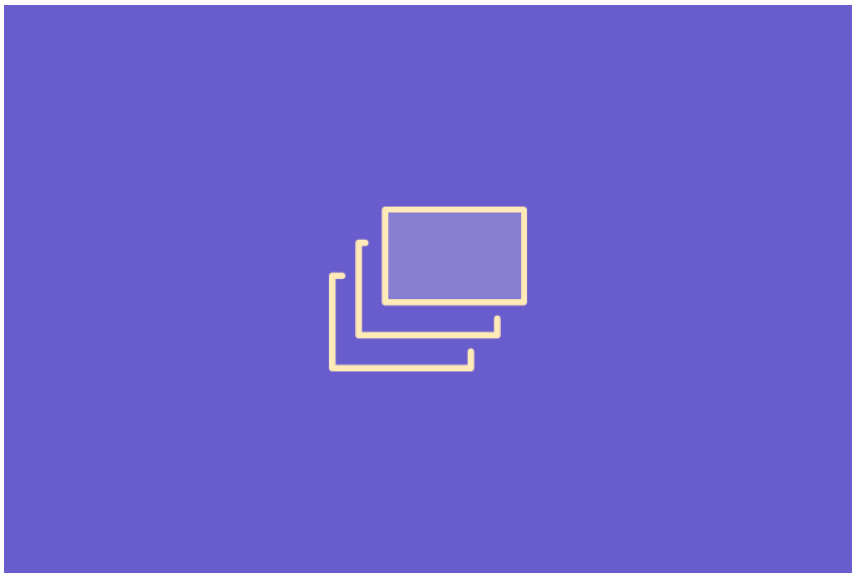


Figure 3.1: A scene with multiple relevant ray distributions.

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.

(a) Plane influence area (b) Point influence area

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

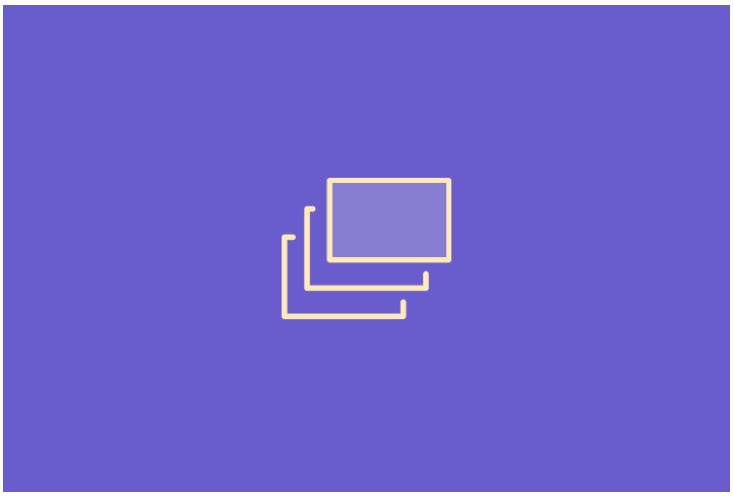


Figure 3.3: Visualization of the method to verify if the direction of a ray is affine with a point influence area.

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 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.

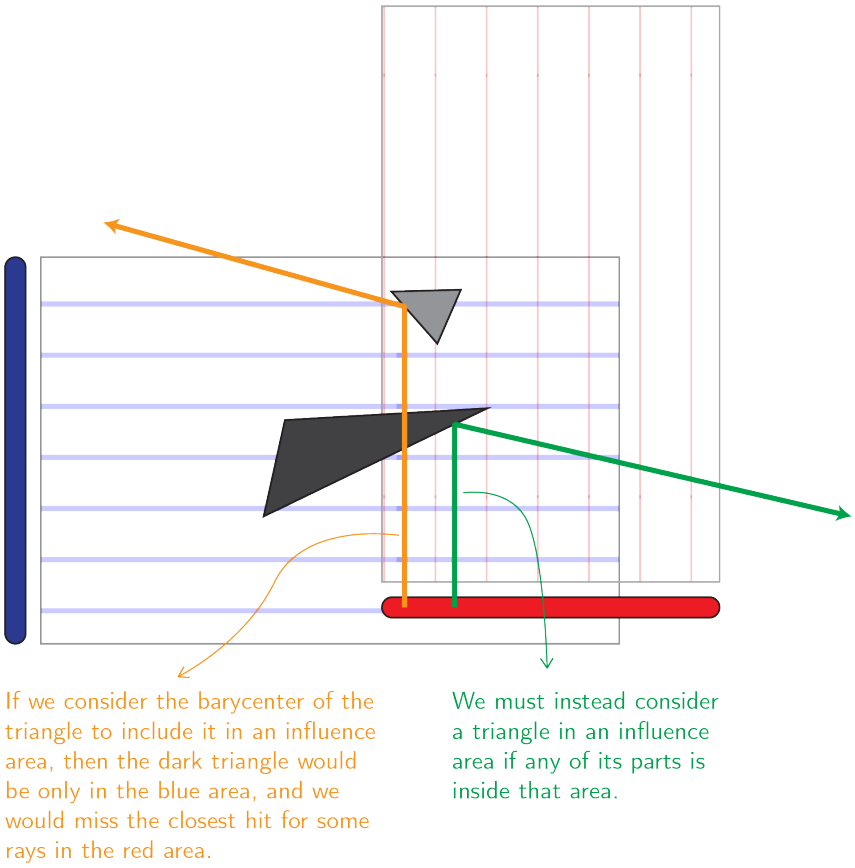


Figure 3.4: Example of why it is necessary to use a conservative approach.

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 distribuition. 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.

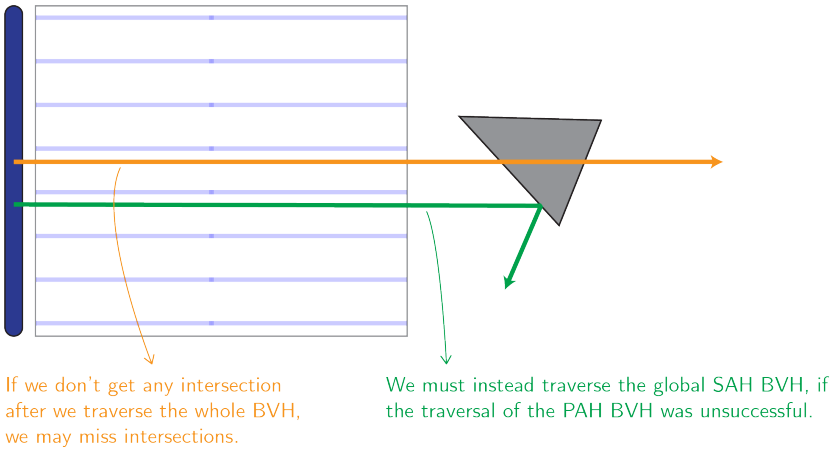


Figure 3.5: Example of how it is possible that a hit is found in the global BVH, but not in the local one.

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 succifient to traverse the octree and read 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.

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 less 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 uses 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. This happens because, under the assumption that triangles are uniformly distributed in the volume of the AABB, the longer the range, the smaller the density of triangles along the direction of the range.

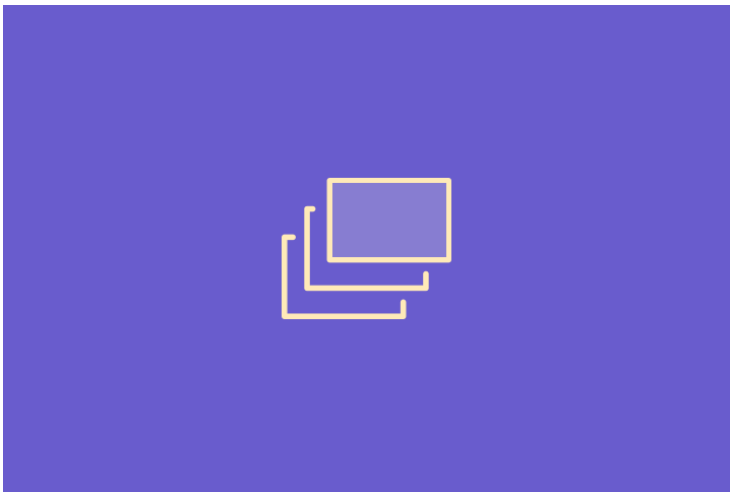


Figure 4.1: A lot of space is cut off by using longest axis splitting.

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 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.

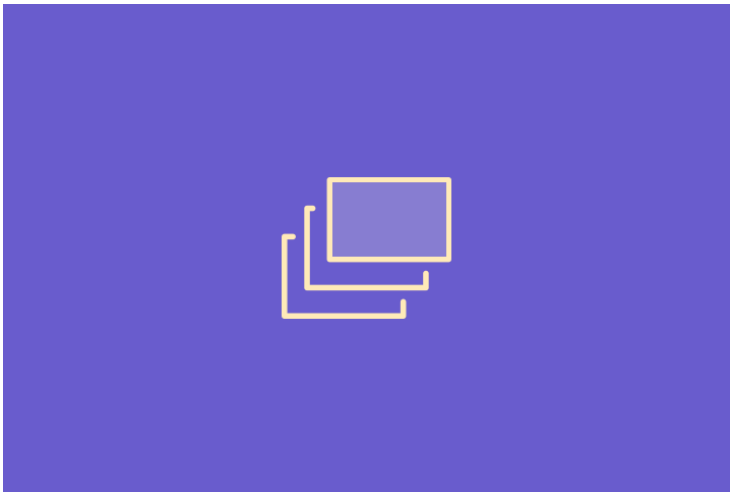


Figure 4.2: Overlapping projected area between two nodes' AABBs.

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 4.4), 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 4.3.

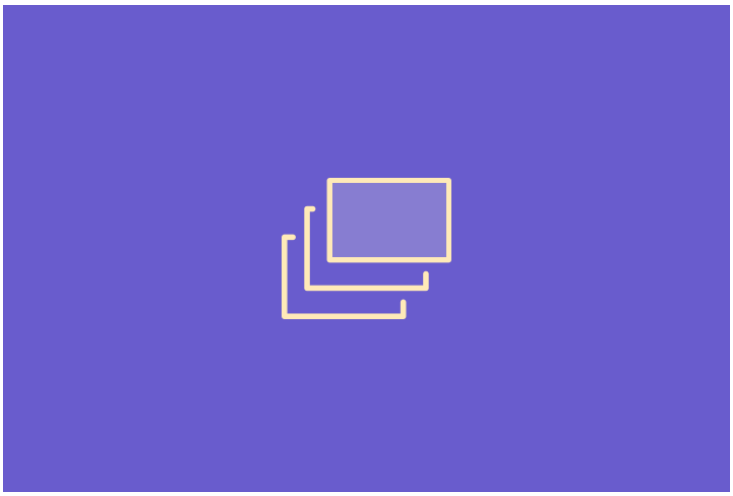
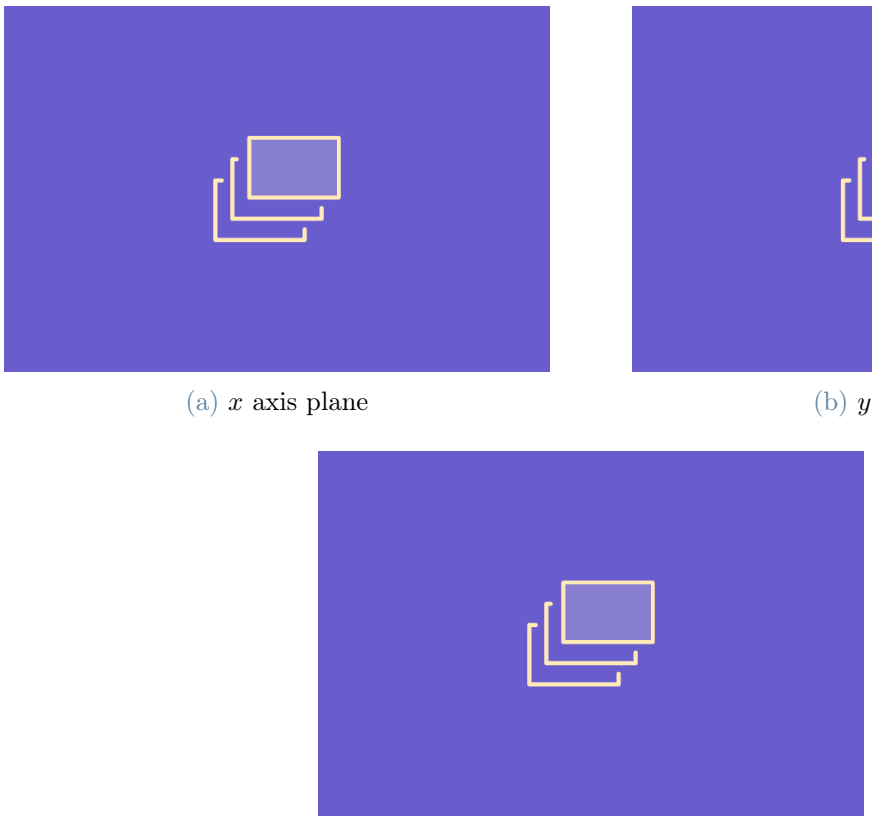


Figure 4.3: Overlapping caused by the triangle extension.



(a) x axis plane

(b) y axis plane

(c) z axis plane

Figure 4.4: Visual illustration of the possible splitting plane orientations and how a ray can hit the children AABBs.

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:

$$quality\ of\ x\ axis = \frac{|v_x|}{|v_x| + |v_y| + |v_z|}$$
$$quality\ of\ y\ axis = \frac{|v_y|}{|v_x| + |v_y| + |v_z|}$$
$$quality\ of\ z\ axis = \frac{|v_z|}{|v_x| + |v_y| + |v_z|}$$

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 0%.

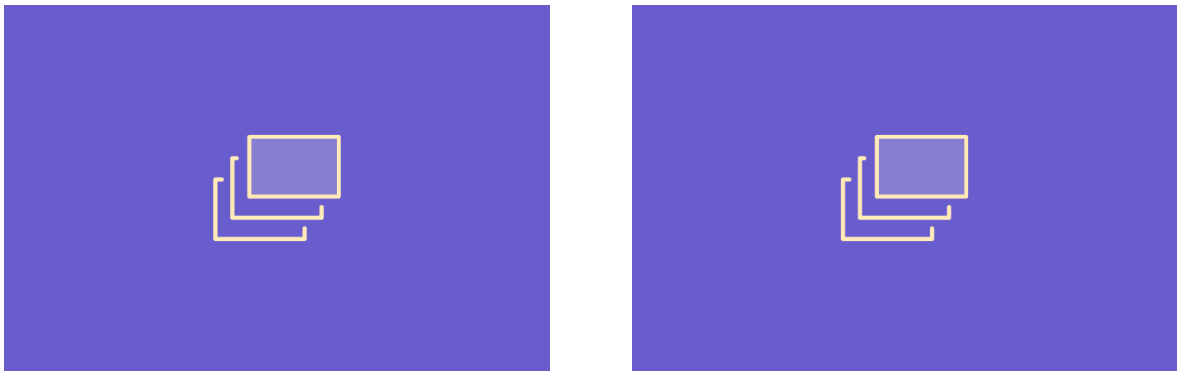
The same concept can be applied to the use of the fallback method. Before we stated that the fallback method is used when the quality values of all the axes are above a certain static threshold. However, it is also possible to set our implementation so that the fallback method is also used when, after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



(a) Good approximation (b) Bad approximation

Figure 4.5: For spread out point influence areas, approximating the rays' directions to the central direction is not a good approximation.

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 prototipes 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 Doxygens[1], 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 *properies* are values upon which depends how th BVH is built. We will now list and explain some of them:

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 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 (section **??**).

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 $\frac{leftChildHit\% + rightChildHit\%}{fatherHit\%}$. 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

1.3.3. Here we limit ourself 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 ← (bestLeftCostSoFar.hitProbability + bestRightCostSoFar.hitProbability) / 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

has alredy 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 ← null
3:      closestHitDist ← ∞
4:      toVisit ← queue
5:      toVisit.push(root)
6:      while not toVisit.empty() do
7:          current ← toVisit.pop()
8:          hitInfo ← collision(ray, current)
9:          if hitInfo ≠ null and hitInfo.distance < closestHit then
10:             if current.isLeaf() then
11:                 for all triangle ∈ current.triangles do
12:                     triangleHitInfo ← collision(ray, triangle)      ▷ abbreviated to thi
13:                     if thi ≠ null and thi.distance < closestHit then
14:                         closestHitTriangle ← triangle
15:                         closestHitDist ← thi.distance
16:             else
17:                 toVisit.push(current.leftChild)
18:                 toVisit.push(current.rightChild)
19:     return ⟨closestHitTriangle, closestHitDist⟩
```

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 $current.triangles.size() \cdot 1$. 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.

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 to introduce 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 cohexist, 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* class[2], 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

[2]To 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.

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.

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.

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 corrsponding direction, the point is inside.

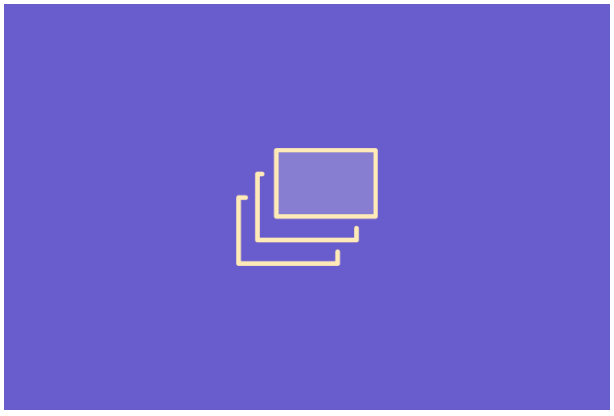


Figure 5.1: Check to detect if a point is inside an OBB.

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

---

class *Aabb*

min : *Vector*3

max : *Vector*3

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 the 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:

$$\begin{cases} A.min_x \leq B.min_x \\ A.max_x \geq B.max_x \\ A.min_y \leq B.min_y \\ A.max_y \geq B.max_y \\ A.min_z \leq B.min_z \\ A.max_z \geq B.max_z \end{cases}$$

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 *AabbForObb*
obb : *Obb*
aabb : *Aabb*

In our implementation, virtually every time we could use an *Obb*, we instead use an *AabbForObb*. 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 *Obb* implementation too. In symbols:

$$\neg Aabb.function \implies \neg Obb.function$$

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

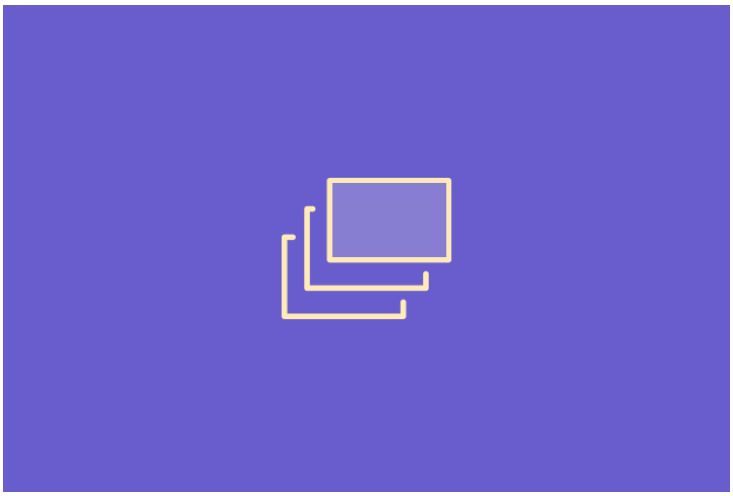


Figure 5.2: If the point (purple) is outside the AABB, it is necessarily outside the OBB too. The inverse (blue and light blue) is not guaranteed.

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 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.

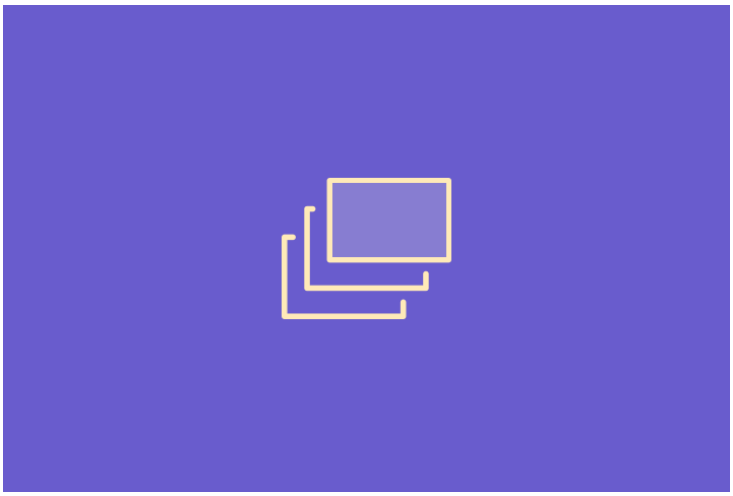


Figure 5.3: How a frustum can be defined.

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 relevantce 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.

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 separanting 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. Used during TODO(it is used during tests to cull the the projected root AABB to the near plane size)

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 [34].

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

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

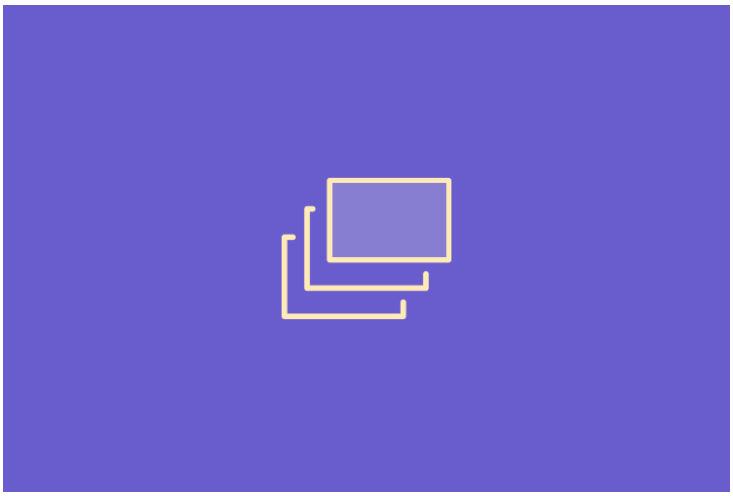


Figure 5.4: Visualization of the orthographic projection matrix: it shrinks the rectangle parallelepiped viewing volume to the canonical view volume.

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: $up = forward \times right$.

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

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

.

The view matrix now is:

$$\begin{bmatrix} right_x & right_y & right_z & -translation_x \\ up_x & up_y & up_z & -translation_y \\ forward_x & forward_y & forward_z & -translation_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.3}$$

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 parallelograms[3], as it is possible to observe in figure 5.5. 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: $A' = viewProjectionMatrix \cdot A$.

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

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: $Area\ parallelogram = |v \times u|$.
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.

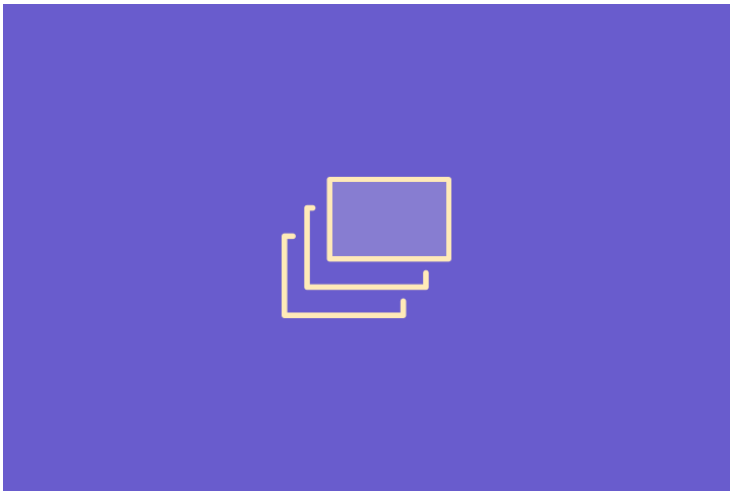


Figure 5.5: The orthographically projected hull of an AABB can be subdivided in 3 parallelograms.

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 5.6).

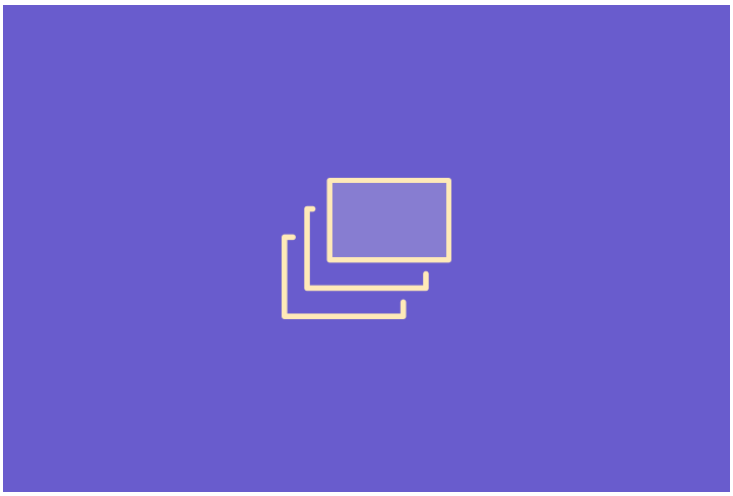


Figure 5.6: The projected hull of an AABB. The blue points are contour points, the purple points are internal points.

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 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 ← 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.

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 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 the horizontal and vertical fields of view, as it is possible to visualize in figure 5.7:

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

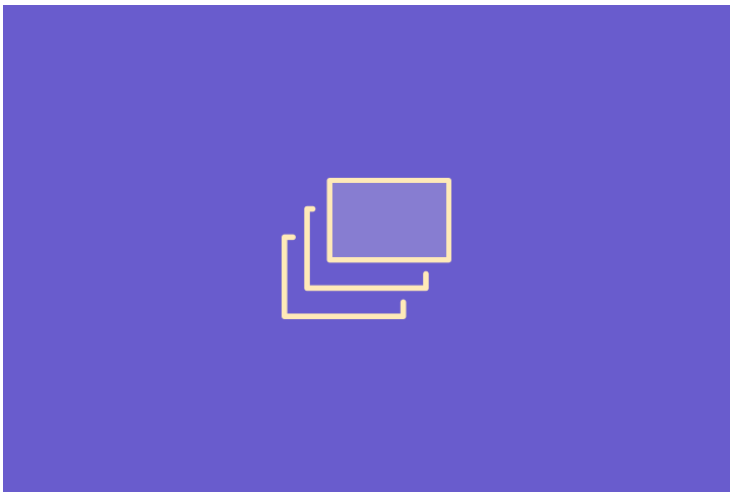


Figure 5.7: Geometrical visualization of how to compute the right plane starting from the horizontal field of view and the near plane.

Now we can calculate the perspective projection matrix [34]:

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

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.

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 sectiom 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 ← 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 algorith 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 5.8.

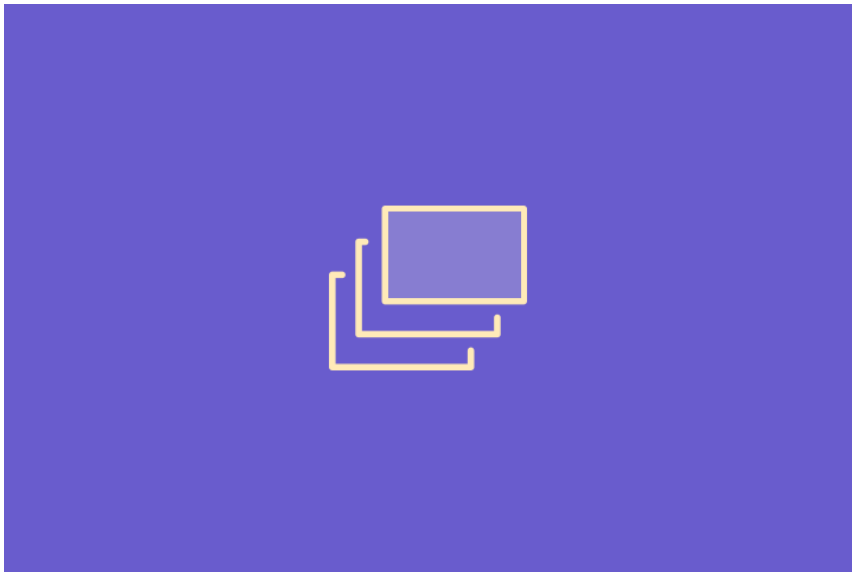


Figure 5.8: Considering the x dimension, the blue eye is inside the range of the AABB, whereas the purple one is not. The purple eye can also see the right face.

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:

$$ray\ direction = center_{AABB} - eye$$

isDirectionAffine(ray)

As already discussed in section 3.1, to verify that a ray direction if 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 is condition is exactly this one, it is never a problem.

$$origin_{ray} - eye /\!/ direction_{ray}$$

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 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 **??**.

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 [34]: the image is recursively subdivided into cells, until a cell contains only one color

(figure 5.9). In this case, the color is the information upon which the quadtree is built, and it must therefore be uniform in leaf nodes.

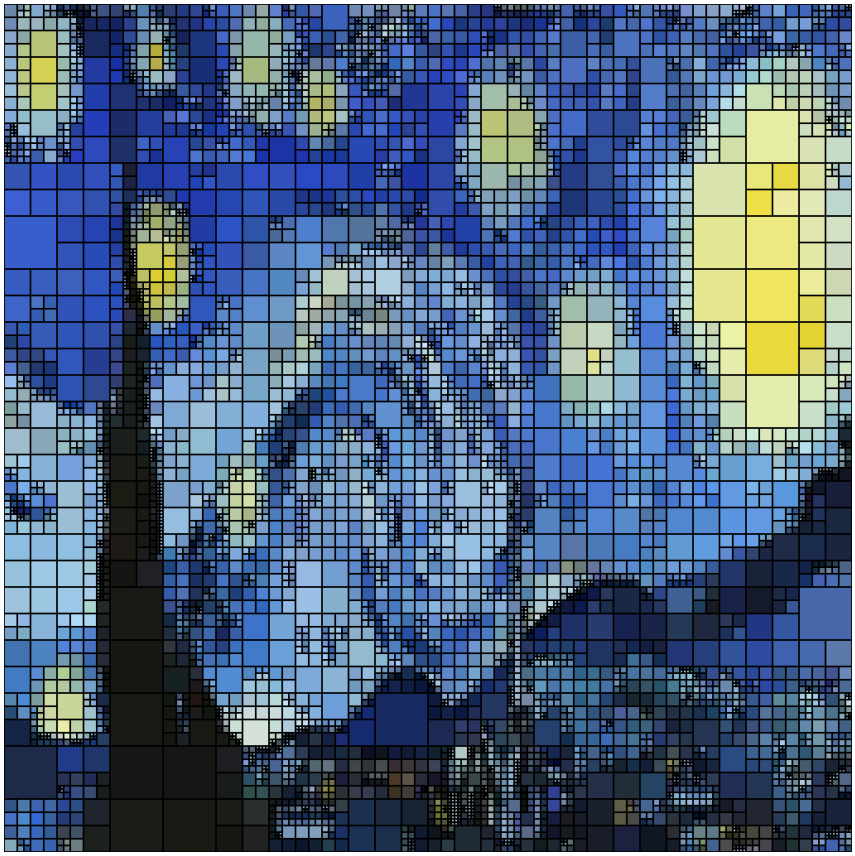


Figure 5.9: The image is recursively subdivided, until a cells contains pixels of just one color (or a maximum depth is reached).

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 where 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 5.11.

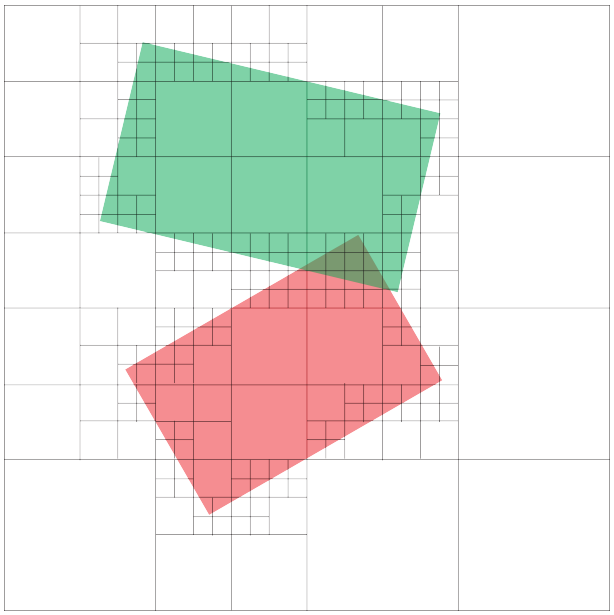


Figure 5.10: An example in 2 dimensions of a quadtree built upon influence areas.

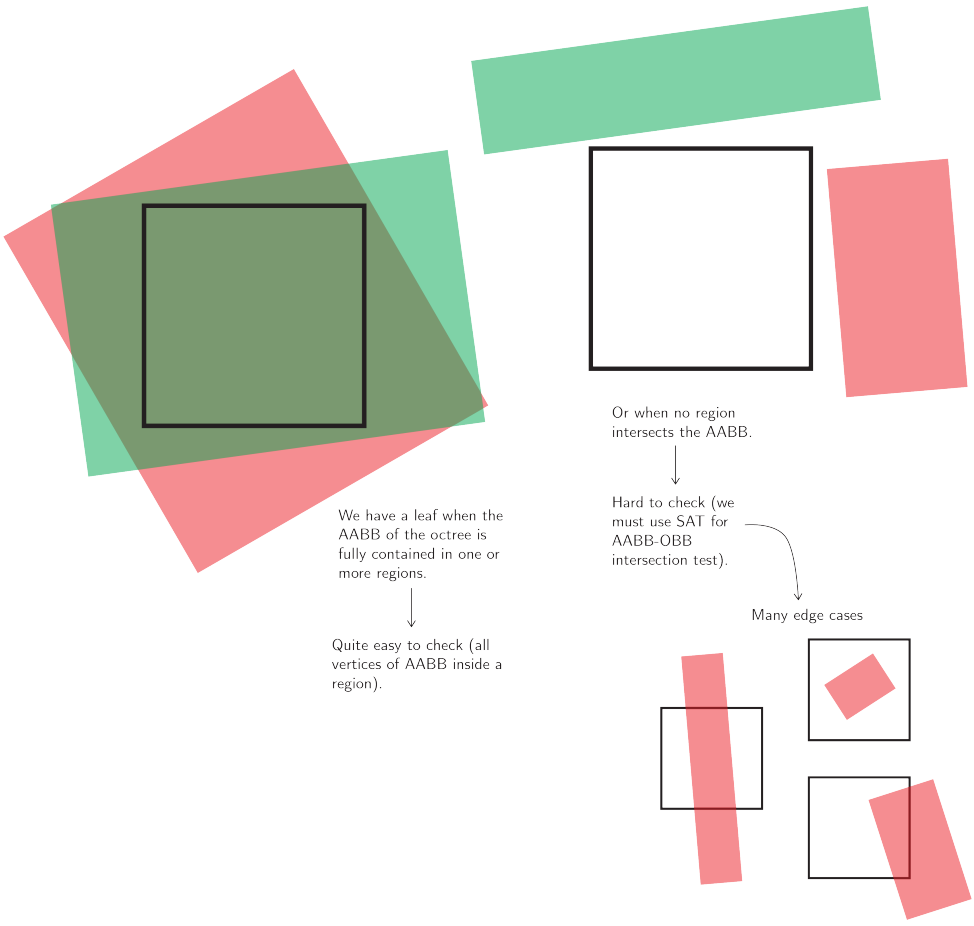


Figure 5.11: Different cases of the relative position of an octree cell and the influence areas' regions

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

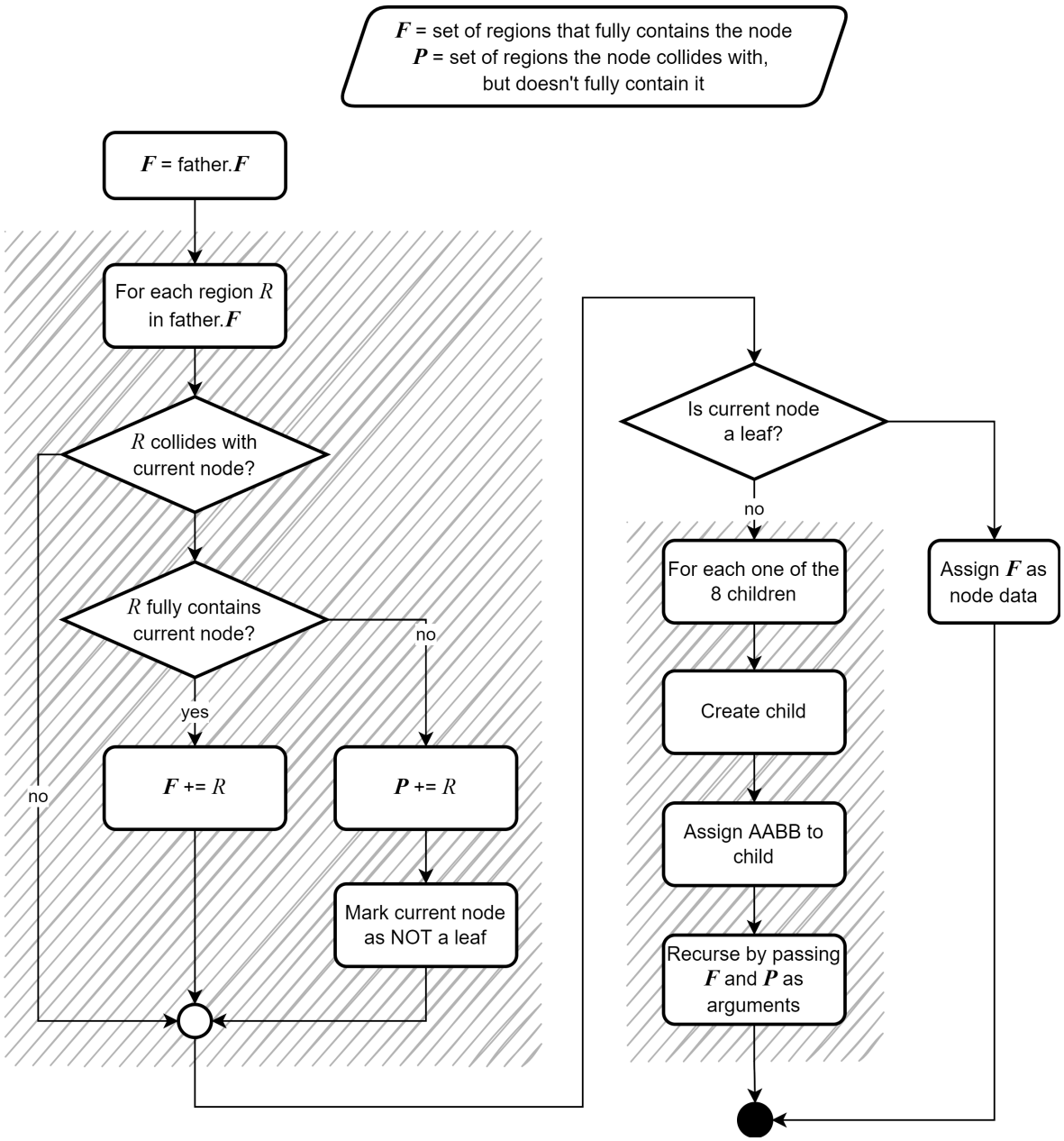


Figure 5.12: Algorithm to build the top level octree.

Each node of the octree has this structure:

class *Node*

$boundaries : Aabb\ [\,]$

$data : Bvh*\ [\,]$

$children : Node[8]$

$isLeaf : bool$

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:

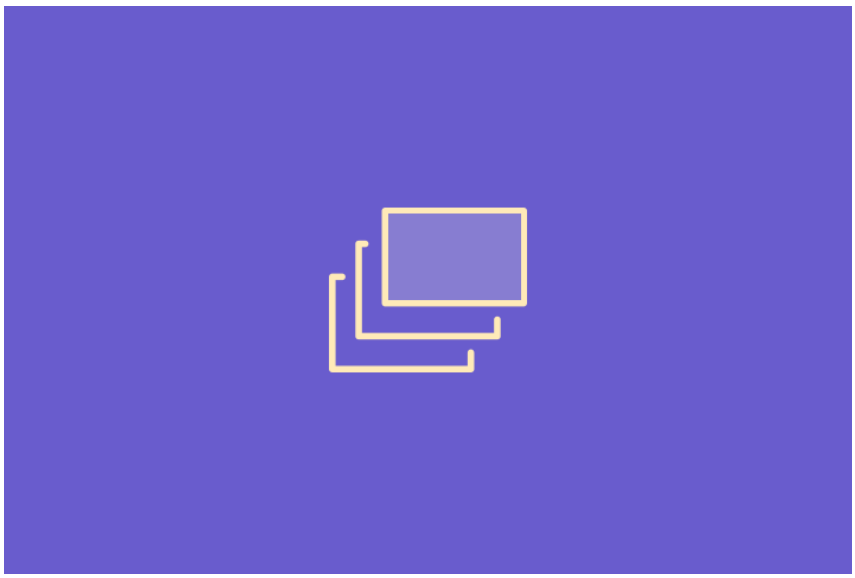


Figure 5.13: The indexing layout of the children of an octree node.

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 *Obb*s in case of plane influence areas, or *Frustum*s for point influence areas.

In the algorithm presented in figure 5.12 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 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 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.

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 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 of 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 *RayCaster*s 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`.

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 invaluably helpful during the experimentation phase.

6 | Experimental Results

7 | Use Cases

8 | Conclusion and Future Developments

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. Intersection test methods. In *Real-Time Rendering 4th Edition*, chapter 22. 2023.

[3] Anonymous. Difference betwen rendering equation, lighting model, ray tracing, global illumination and shadows? `https://computergraphics.stackexchange.com/questions/9241/difference-betwen-rendering-equation-lighting-model-ray-tracing-global-illumi`. Accessed: (18/04/2024).

[4] S. Brabec, T. Annen, and H.-P. Seidel. Practical shadow mapping. *Journal of Graphics Tools*, 7(4):9–18, 2002.

[5] J. Burgess. Rtx on—the nvidia turing gpu. *IEEE Micro*, 40(2):36–44, 2020.

[6] 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).

[7] 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.

[8] J. Cole. Signed distance fields. `https://jasmcole.com/2019/10/03/signed-distance-fields/`. Accessed: (11/05/2023).

[9] A. Dittebrandt, J. Hanika, and C. Dachsbacher. Temporal sample reuse for next event estimation and path guiding for real-time path tracing. 2020.

[10] B. Fabianowski, C. Fowler, and J. Dingliana. A cost metric for scene-interior ray origins. In *Eurographics (Short Papers)*, pages 49–52, 2009.

[11] G. Gribb and K. Hartmann. Fast extraction of viewing frustum planes from the world-view-projection matrix. *Online document*, 2001.

[12] 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.

[13] J. Hart, D. Sandin, and L. Kauffman. Ray tracing deterministic 3-d fractals. *SIGGRAPH '89*, 1989.

[14] J. T. Kajiya. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 143–150, 1986.

[15] B. Karis, R. Stubbe, and G. Wihlidal. Nanite: A deep dive. SIGGRAPH '21: Advances in Real-Time Rendering in Games.

[16] A. E. Kaufman and K. Mueller. Overview of volume rendering. *The visualization handbook*, 7:127–174, 2005.

[17] 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.

[18] D. Kirk and J. Arvo. Improved ray tagging for voxel-based ray tracing. In *Graphics Gems II*, pages 264–266. Elsevier, 1991.

[19] 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.

[20] E. P. Lafortune and Y. D. Willems. Bi-directional path tracing. 1993.

[21] 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.

[22] P. Moreau and P. Clarberg. Importance sampling of many lights on the gpu. In *Ray Tracing Gems*, chapter 18. 2019.

[23] S. Owen. Ray-plane intersection. `https://education.siggraph.org/static/HyperGraph/raytrace/rayplane_intersection.htm`, 1999. Accessed: (11/01/2024).

[24] S. Owen. Ray-box intersection. `https://education.siggraph.org/static/HyperGraph/raytrace/rtinter3.htm`, 2001. Accessed: (10/01/2024).

[25] S. Oz. Intersection of convex polygons algorithm. https://www.swtestacademy.com/intersection-convex-polygons-algorithm/. Accessed: (20/03/2024).

[26] 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).

[27] 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).

[28] 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, . Accessed: (09/01/2024).

[29] S. Roman et al. *An introduction to Catalan numbers*. Springer, 2015.

[30] L. P. Santos, T. Bashford-Rogers, J. Barbosa, and P. Navrátil. Towards quantum ray tracing. *IEEE Transactions on Visualization and Computer Graphics*, 2024.

[31] A. Shelankov and G. Pikus. Reciprocity in reflection and transmission of light. *Physical Review B*, 46(6):3326, 1992.

[32] J. Soch. Proof: Law of the unconscious statistician. https://statproofbook.github.io/P/mean-lotus.html. Accessed: (09/09/2023).

[33] 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.

[34] TODO. Todo. *TODO*, TODO(TODO):TODO, TODO.

[35] E. Veach. Chapter 2: Monte carlo integration. https://www.ime.usp.br/~jmstern/wp-content/uploads/2020/04/EricVeach2.pdf.

[36] E. Veach. *Robust Monte Carlo methods for light transport simulation*. Stanford University, 1998.

[37] E. Veach. *Robust Monte Carlo methods for light transport simulation*. Stanford University, 1998.

[38] 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.

[39] 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.

[40] 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.

[41] H. Weghorst, G. Hooper, and D. P. Greenberg. Improved computational methods for ray tracing. *ACM Transactions on Graphics (TOG)*, 3(1):52–69, 1984.

[42] 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).

[43] D. Wodniok and M. Goesele. Construction of bounding volume hierarchies with sah cost approximation on temporary subtrees. *Computers & Graphics*, 62:41–52, 2017.

[44] 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.

[45] C. Wynn. An introduction to brdf-based lighting. *Nvidia Corporation*, 4(3), 2000.

[46] M. Young. The pinhole camera. *The Phys. Teacher*, pages 648–655, 1989.

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 [24].

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 ($\overline{t_1}$ and $\overline{t_2}$) such that $r(\overline{t_{1,2}})$ 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 $\overline{t_{1,2}}$ values. We call the point with the smallest $\bar{t}$ value the *closest*, and the other one the $\overline{furthest}$.
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.

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.

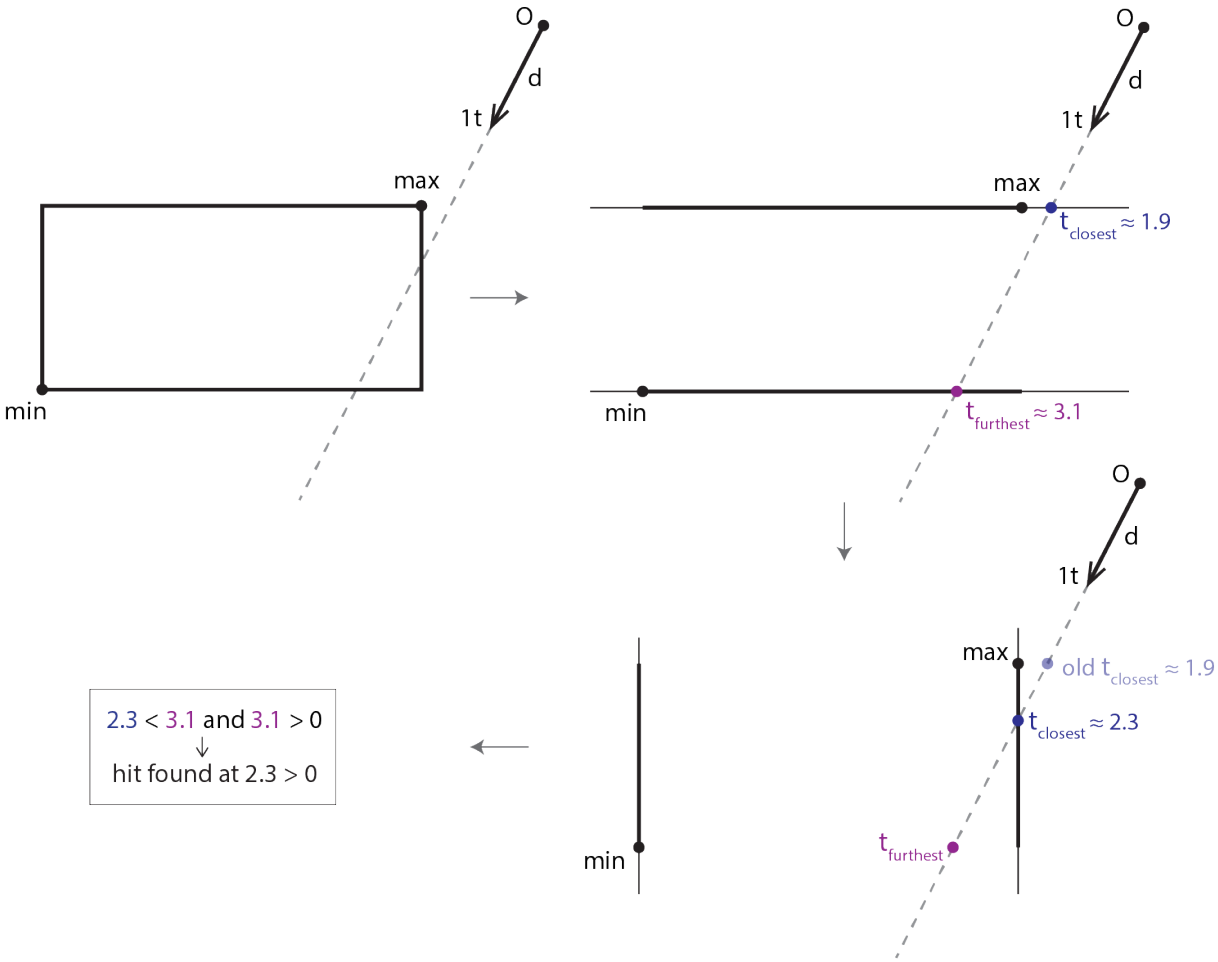


Figure A.1: Visual representation of the presented algorithm in 2 dimensions. An interactive simulation of this algorithm can be found at: `https://www.geogebra.org/m/np3tnjvb`.

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

function INTERSECT($ray, aabb$)
  $t1_x \leftarrow \frac{aabb.min.x - ray.origin.x}{ray.direction.x}$ ▷ *yz plane*
  $t2_x \leftarrow \frac{aabb.max.x - ray.origin.x}{ray.direction.x}$
  $tMin \leftarrow min(t1_x, t2_x)$
  $tMax \leftarrow max(t1_x, t2_x)$
  $t1_y \leftarrow \frac{aabb.min.y - ray.origin.y}{ray.direction.y}$ ▷ *xz plane*
  $t2_y \leftarrow \frac{aabb.max.y - ray.origin.y}{ray.direction.y}$
  $tMin \leftarrow max(tMin, min(t1_y, t2_y))$
  $tMax \leftarrow min(tMax, max(t1_y, t2_y))$
  $t1_z \leftarrow \frac{aabb.min.z - ray.origin.z}{ray.direction.z}$ ▷ *xy plane*
  $t2_z \leftarrow \frac{aabb.max.z - ray.origin.z}{ray.direction.z}$
  $tMin \leftarrow max(tMin, min(t1_z, t2_z))$
  $tMax \leftarrow min(tMax, max(t1_z, t2_z))$
  $areColliding \leftarrow tMax > tMin$ **and** $tMax \geq 0$
  $collisionDist \leftarrow tMin < 0 \,?\, tMax \,:\, tMin$
  return $\langle areColliding, collisionDist \rangle$

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 [23].

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

parameter:

$$A \cdot P_x + B \cdot P_y + C \cdot P_z + D = 0$$
$$\implies D = -(A \cdot P_x + B \cdot P_y + C \cdot P_z)$$

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

$$D = -(n_x \cdot P_x + n_y \cdot P_y + n_z \cdot P_z)$$
$$\implies D = -\langle n \cdot P \rangle$$

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:

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

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 ← ray.direction
3:      O ← ray.origin
4:      n ← plane.normal
5:      P ← plane.point
6:      if ⟨n · d⟩ = 0 then                    ▷ Ray is parallel to plane
7:          return ⟨false, _⟩
8:      D ← −⟨n · P⟩
9:      t ← −⟨n·O⟩ / ⟨n·d⟩
10:     if t < 0 then                          ▷ Intersection point is behind ray origin
11:         return ⟨false, _⟩
12:     else
13:         return ⟨true, t⟩
```

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:

$$\begin{cases} A.min_x \leq B.max_x \\ A.max_x \geq B.min_x \\ A.min_y \leq B.max_y \\ A.max_y \geq B.min_y \\ A.min_z \leq B.max_z \\ A.max_z \geq B.min_z \end{cases}$$

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) [2]. 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.

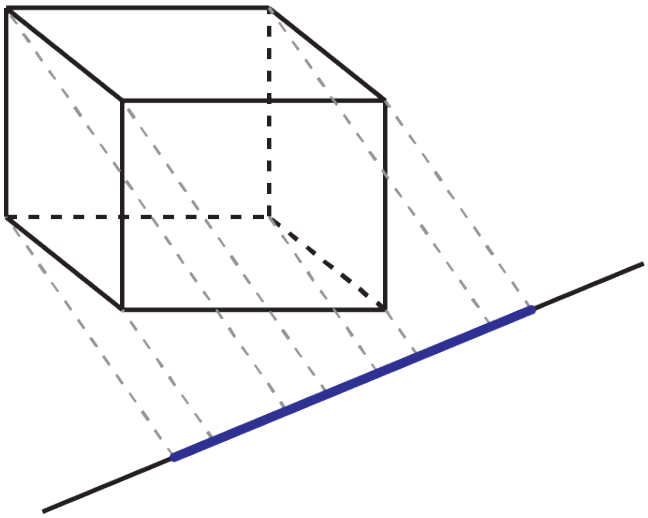
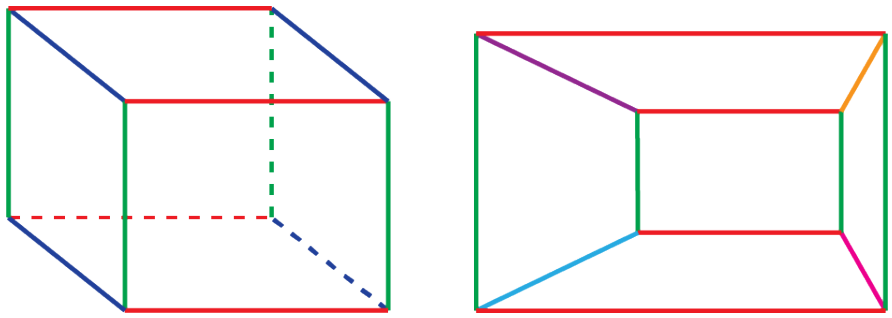


Figure A.2: The projection of an AABB on an axis.

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) AABB: 3 different edges directions

(b) Frustum: 6 different edges directions

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

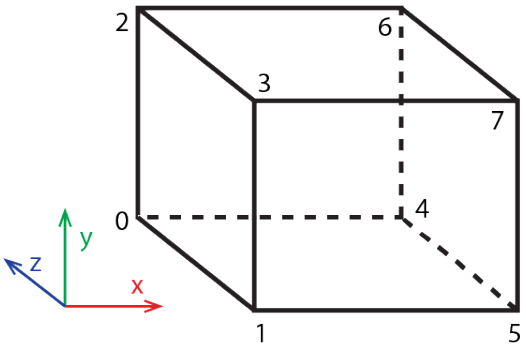
A.5.1. 1D Projections Overlapping Test

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:

Figure A.4

(a) AABB vertices layout.



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

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 ← (⊥ frustum faces) ∪ (⊥ AABB faces) ∪ (×edges)
5:     for all axis ∈ axesToCheck do
6:         frustumExtremes ← findFrustumExtremes(frustum, axis) ▷ Returns the
           vertices of the frustum that, after the projection, will be the extremes
7:         aabbExtremes ← findAabbExtremes(aabb, axis) ▷ Same as above, but uses
           the discussed optimization
8:         A_min ← ⟨aabbExtremes.first · axis⟩
9:         A_max ← ⟨aabbExtremes.second · axis⟩
10:        B_min ← ⟨frustumExtremes.first · axis⟩
11:        B_max ← ⟨frustumExtremes.second · axis⟩
12:        if !(B_min ≤ A_max & B_max ≥ A_min) then
13:            return false
14:    return true ▷ If we havent'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 [11].

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 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:

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

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 [28] 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:

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

[1] Based 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 $\langle 1, 1, 1 \rangle$

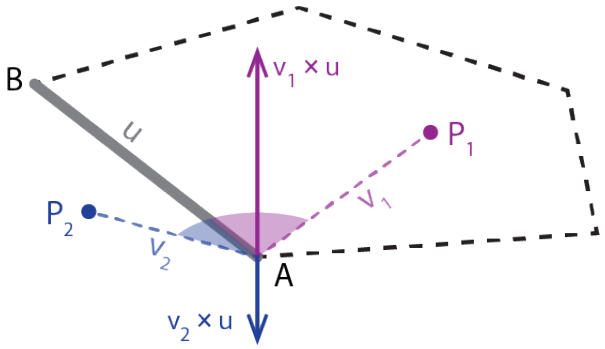


Figure A.5: Visualization of cross product.

Therefore the strategy to determine if a point is on the same side of 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 ← number of edges of hull
3:      u ← hull[0] − hull[N − 1]
4:      v ← P − hull[N − 1]
5:      ref ← u × v                          ▷ The reference cross product
6:      for 0 ≤ i < N do
7:          u ← hull[i + 1] − hull[i]
8:          v ← P − hull[i]
9:          cross ← u × v
10:         if ⟨ref · cross⟩ ≤ 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 ([25]):

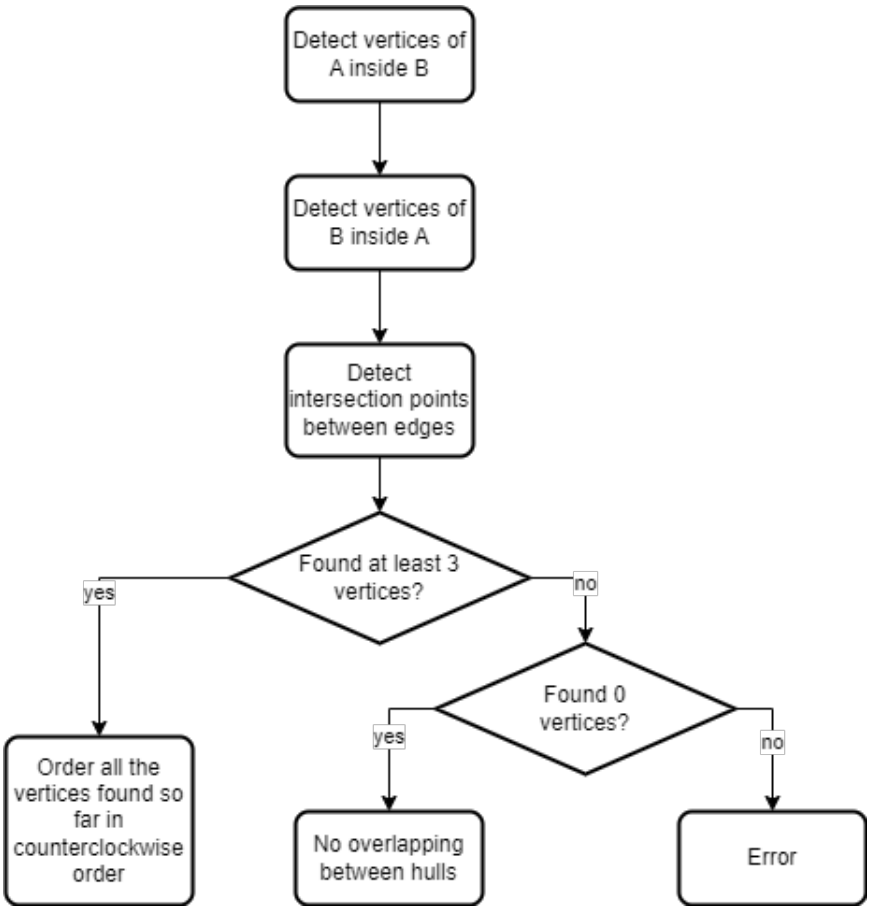


Figure A.6: General algorithm to find the overlapping region between two convex hulls called A and B.

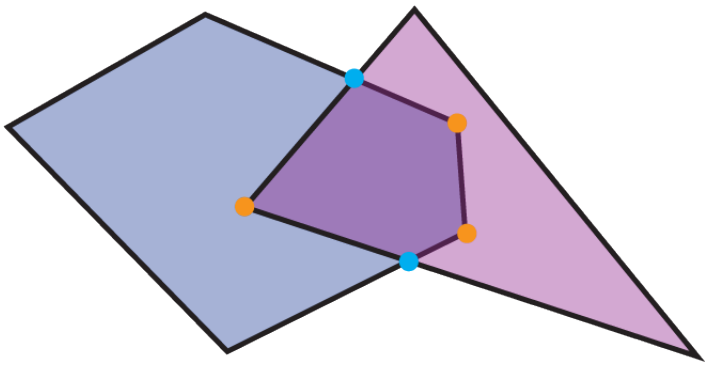


Figure A.7: Yellow vertices are found in the first 2 steps (vertices inside), whereas light blue vertices are found in the third step (edges intersections).

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:

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

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 $\overline{PQ}$:

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

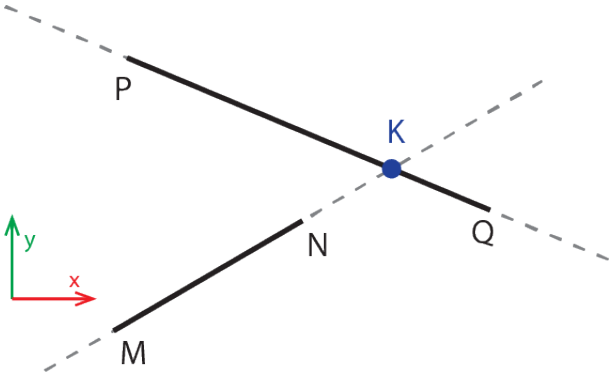


Figure A.8: In this example segments do not collide because $max(M_x, N_x) = N_x \not\geq K_x$ or $max(M_y, N_y) \not\geq K_y$

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 hull[2], 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 $atan2(\overrightarrow{OA_{ky}}, \overrightarrow{OA_{Kx}})$.

The $atan2(v_y, v_x)$ 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.

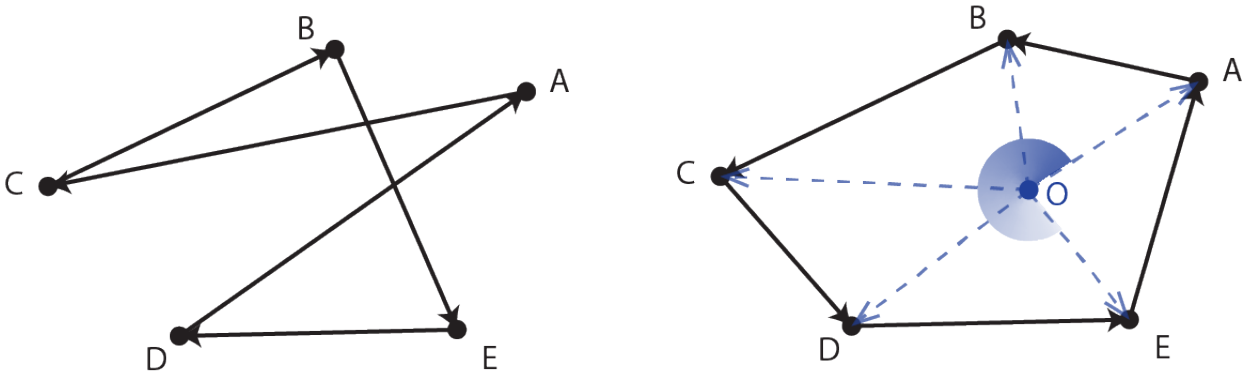


Figure A.9: Vertices of a convex hull before and after $atan2$ sorting.

[2]We 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 [21].

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

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

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

B | Multiple Importance Sampling

TODO

List of Figures

List of Tables

List of Symbols

Symbol	Description	Unit
alpha	symbol 1	km

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.