POLITECNICO
MILANO 1863

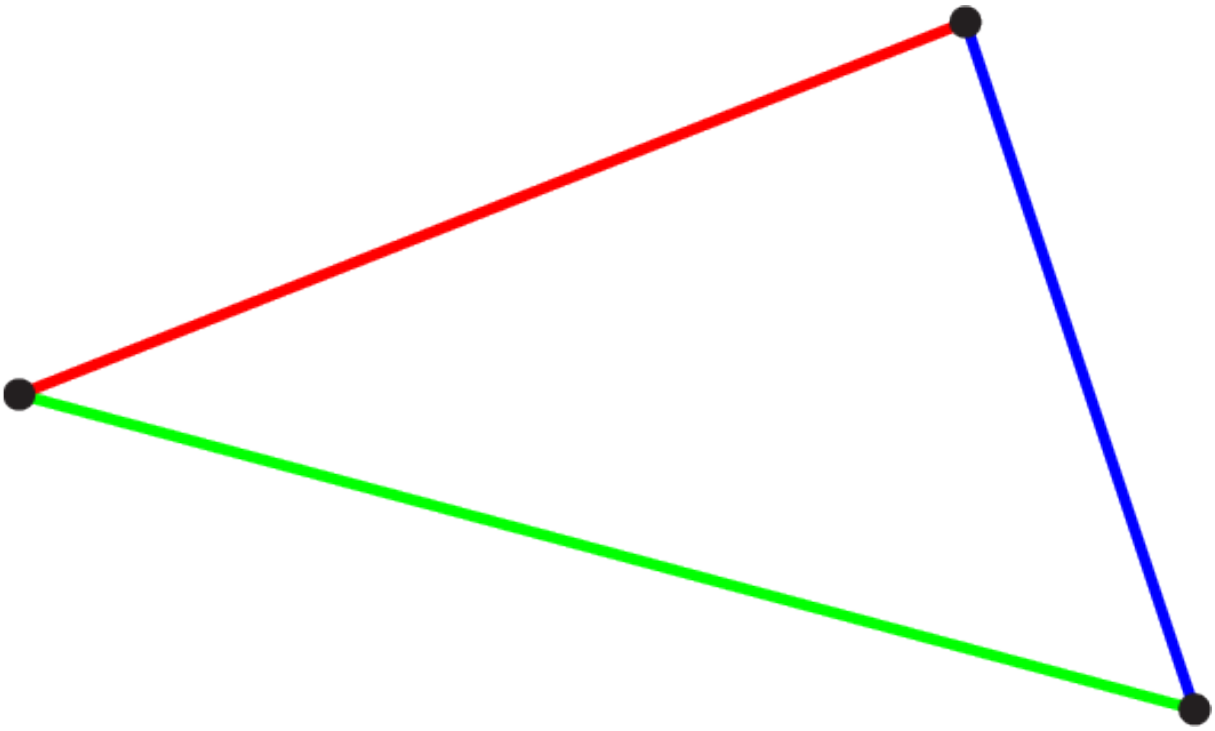
SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

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

Why ray tracing?

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

TODO same in chapter 1 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;
6. Color the pixel of the camera hit by the ray based on the energy of the ray.

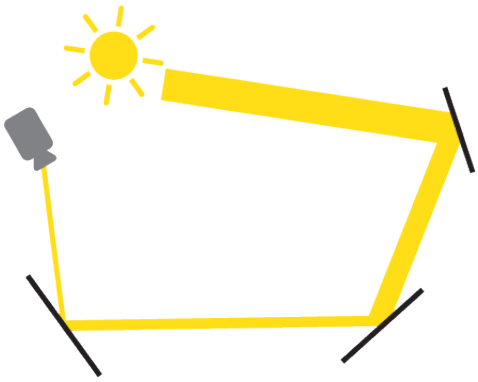


Figure 1: The width of the yellow ray represents the amount of energy carried. After each intersection some energy is absorbed.

Since ray tracing mimics the behavior of light in the real world, the technique can be directly used to simulate complex light effects that would require the use of ad-hoc and

approximate methods if we use 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 most intuitive ones used to generate shadows, called shadow mapping. 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, and 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.

We need optimizations

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. Of course, it is impossible to track the path of every photon emitted by a light source, therefore, even in ray tracing algorithms, some approximations must be made. The nature of the approximations depends on the specific ray tracing algorithm used. For example, in many techniques falling under the name of *backward ray tracing* [1], rays don't start from the light sources, but from the camera, and gain energy when they hit a light source. At this point the path described by the ray is followed backward and the energy hitting the pixel of the camera is computed. In this way the number of rays is greatly reduced, because all the rays hit the camera, therefore none is wasted. On the other hand, some phenomena (such as caustics) cannot be realistically simulated.

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

One optimization that is foundational to modern ray tracing and relevant to our work, is the use of the **Monte-Carlo** integration method, and, in particular, a variance reduction

[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.

technique called **importance sampling**. We will introduce the concept in this section in an oversimplified way, and then explain it from a mathematical standpoint in **??**.

When a ray hits a point on a surface, it may bounce in any direction, based on the material. The most intuitive bounce is a perfect reflection, but, since at a microscopic level surfaces are never perfectly planar, it is possible for the incoming ray to bounce in any direction. This phenomenon is described in a probabilistic way by the bidirectional reflective distribution function (BRDF). Each material is described by a BRDF, and each BRDF makes it more likely for a ray coming from a certain direction to bounce to another range of directions (for example, for a perfectly reflective material, the BRDF will return a probability of 1 for a ray to bounce to the perfect reflection direction).

(a) Perfectly reflective surface.

(b) Non-reflective surface.

Figure 2: In figure (a) all the rays coming from a direction are reflected towards the same direction. In (b), instead, the surface is microscopically rough, 2 rays coming from the same direction could bounce to 2 very different directions. Under each figure there is the corresponding graph of its BRDF.

Therefore, if we wanted to accurately simulate the light behavior after a ray hits a material, and compute how much light is reflected towards a specific direction (the one of the incoming ray), we would need to send a probe ray to every direction of the hemisphere centered on the hit point, get back the light energy carried by each probe ray, and compute an average based on the probability of each probe ray given by the BRDF. In short, we would need to integrate the light energy function over the hemisphere. This exact same process would then need to be replicated when each probe ray hits an object, recursively, until each ray in the scene hits a light source or gets completely absorbed.

This process, of course, is not feasible, but it can be approximated by the Monte-Carlo method with fewer samples. The idea is that, instead of probing each direction of the hemisphere with a ray, we probe just a small number (often just one) of directions. In most cases the estimate of the light incoming to the point will not be accurate, but, granted that there is a high enough number of incoming rays hitting the neighborhood of the point, the incoming light average will indeed be accurate.

In order to get the most out of the probe rays we cast out of a hemisphere, we would like to send them in directions that contribute the most to the final value we are trying to calculate, namely the light reflected toward the direction of the incoming ray. This is the base concept behind importance sampling. In ray tracing there are two common ways to achieve this:

BRDF sampling Probe rays are cast in directions where the BRDF returns a high weight. In this way the energy the probe rays carry is multiplied by a value close to 1; on the other hand, the energy can be a low value.

Light sampling Probe rays are cast directly towards light sources. In this way they will likely carry a lot of energy (unless an obstacle is hit), but the BRDF weight they will be multiplied with may be small.

It is even possible to combine the two techniques with a method called multiple importance sampling (MIS).

The use of importance sampling generates artifacts in the ray distribution on the scene. Rays' directions will no longer be distributed uniformly, but, due to light sampling, more rays will tend to go towards light sources.

Rays intersections

One of the problems that is common to all the algorithms of the ray tracing family is to find the intersection between a ray and the geometry of the scene.

The objects in the scene are usually meshes, which are a collection of geometric primitives. In many real-world scenarios, primitives are simple triangles, described by 3 vertices. Therefore, the problem of intersecting a ray with the scene is reduced to the problem of intersecting a ray with a collection of triangles.

Given an algorithm to find out if a ray hits a triangle in A.3, the naive way of retrieving the closer triangle hit by a ray would be to perform the ray-triangle test on all the triangles present in the scene. Such an algorithm has a complexity of $\mathcal{O}(n \cdot m)$ where n is the number of triangles and m the number of rays.

Acceleration structures have been developed to speed up the process. The state-of-the-art acceleration structure used in ray tracing is the **bounding volume hierarchy** (BVH).

In summary, a BVH is a binary tree[2] where each node wraps some of the triangles in the scene in a bounding volume, usually an axis-aligned bounding box (AABB). Given a node A wrapping the triangles in the set T_A, the two children of A, called B and C wrap

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

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. This means that, if the BVH is balanced, the the complexity to find an intersection between a ray and the scene is $\mathcal{O}(log_2(n) \cdot m)$.

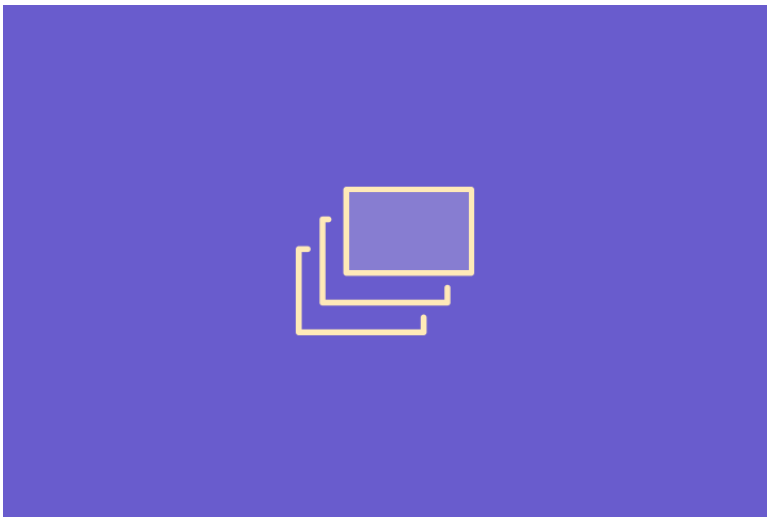


Figure 3: A 2-dimensional BVH.

Given the definition and the advantages of a BVH over the naive algorithm, we now have to build a BVH starting from the primitives in the scene. Building the optimal BVH is an NP-complete problem since, for each level of the BVH, we would have to try all the possible ways to subdivide the triangles into 2 AABBs, recursively, until all the branches are leaves; then measure the quality of the BVHs and take the best one.

In a real-world scenario a greedy algorithm is used instead, where, at each level, some of the possible ways to subdivide the triangles are tried, and the best one with reference to a cost metric function is greedily taken. This means that, even if in the next level it is found out that the split in a previous level was not optimal, the algorithm proceeds without changing the previous decision.

The quality of the trees built with such an algorithm depends on many factors, but two of the most relevant ones are the cost function and how the algorithm tries to split the triangles at each level.

The surface area heuristic

The vast majority of the state-of-the-art algorithms to build BVHs uses the **surface area heuristic (SAH)** as cost metric. SAH is extremely simple and fast: the cost of a node

K is proportional to the probability that a random ray intersects the node $p(hit\ K)$, and the number of primitives the node contains $\#T_K$:

$$SAH(K) = p(hit\ K) \cdot \#T_K$$

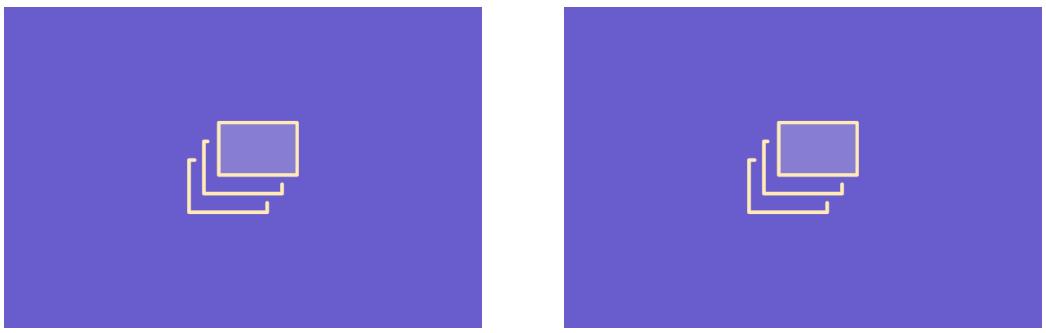
The probability that a node is hit by a random ray is proportional to its area: $p(hit\ K) = \frac{A_K}{A_tot}$ where A_tot is the area of the AABB enclosing the whole scene. Computing the area of an AABB is trivial.

SAH is based on three hypotheses:

1. All the rays hit AABB enclosing the whole scene.
2. All the rays start from outside the scene.
3. Rays are distributed in a uniform random way on their 6-dimensional space (3 dimensions for the position of their origin and 3 dimensions for their direction).

None of these hypotheses is satisfied in a real-world scenario, and, in particular, we will focus on the third one.

As we noted above, due to the use of importance sampling, rays are not distributed uniformly in the scene. In particular, due to light sampling, in proximity of light sources, rays have directions pointing towards the light source. Based on the light source type, we have different types of ray distributions, as it is possible to appreciate in the images below.



(a) Point light generated distribution (b) Area light generated distribution

Figure 4: The ray distributions generated by point and area lights.

Area lights tend to generate regions where rays are parallel. This means that we can better estimate the probability that a ray intersects an AABB by computing its projected

area on the plane perpendicular to the bundle of rays, instead of the surface area of the AABB. Such a projected area can be computed with an orthographic projection.

Point lights, instead, tend to generate a bundle of lines passing through a point, and therefore the projected area can be computed with a perspective projection.

In this work we aim to show how this way of computing the cost function for the construction of a BVH can produce higher quality BVHs in some situations. We named this novel approach as **projected area heuristic (PAH)**.

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

High Level Overview

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 [12].

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 [2]. 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.

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, in particular a very accurate algorithm called path tracing. 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 trac-

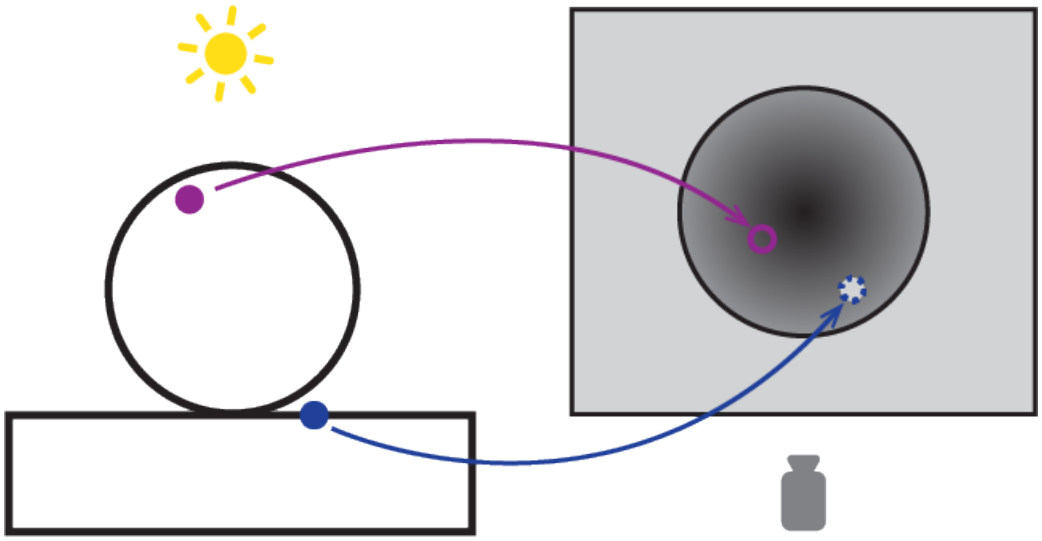


Figure 1.1: The first figure is from the main camera PoV, the second one from the light source PoV. The second figure represents depth: the closer a point is to the light source, the darker. The blue point is in shadow, because the corresponding point in the shadow map is further away than the stored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

ing, a frame must be produced every 33ms to achieve 30 frames per second (fps), and every 16.5ms to achieve 60 fps, which can be considered today's standard by many PC videogames.

This 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.

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 [3].

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 in this second family. This technique is called importance sampling, and will be discussed in section 1.2.

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 **??**. 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.

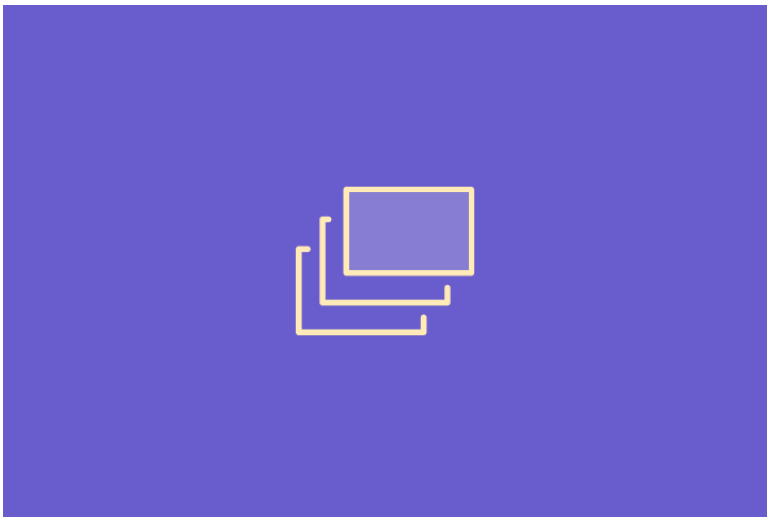


Figure 1.2: How rays are cast in backward ray tracing.

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.

A way of looking at this algorithm is to think we are tracing *importons*, which can be considered the dual concept of *photons* [4]. Thanks to the light reciprocity principle [13], which states that light transmits in the same way in both ways, 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 makes it more likely that a ray hits 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. We will see how this technique generates artifacts in the ray distribution in the scene, which is one of the hypotheses of our thesis.

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 [6], 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 such a quantity we need to know these parameters:

- $L_e(\bar{x}, \bar{\omega}_o)$ 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 $\bar{\omega}_i$.

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.

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

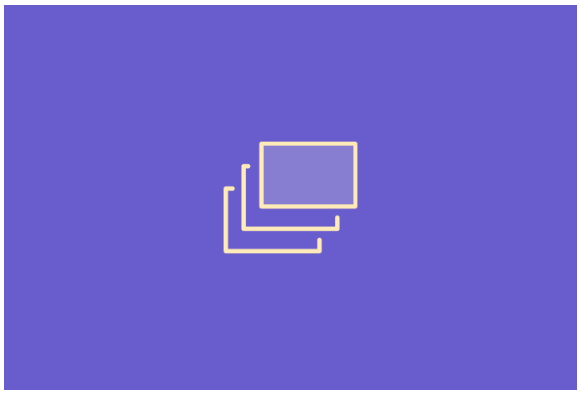


Figure 1.3: A visual representation of the integral term of the rendering equation.

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

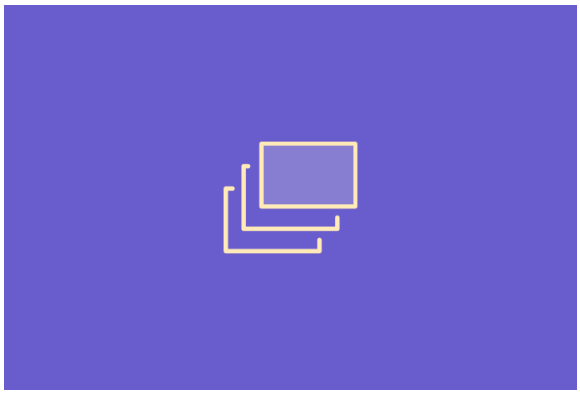


Figure 1.4: The geometry term.

In this integral term, 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 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).

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.

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 is a method by which it is possible to compute the numerical integral of a function by using random numbers. 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. We can formalize this process with this formula:

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

Where $X_i \sim \frac{1}{b-a}$ is a uniform random variable in the $[a, b]$ interval.

$\langle F^N \rangle$ is referred to as the basic Monte-Carlo estimator. 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), 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 the 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:

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

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

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

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 it is estimating. The rate of convergence is proportional to the variance σ^2, this means that, in order to reduce the estimator error by half, 4 times more samples are needed.

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. This is particularly relevant in our case, since we are working in a 3-dimensional domain.

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.

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.

Variance Reduction Techniques

1.3. Ray Tracing Acceleration Structures

2 | Chapter two

Chapter 2

Bibliography

[1] T. Akenine-Möller, E. Haines, N. Hoffman, A. Pesce, M. Iwanicki, and S. Hillaire. *Real-Time Rendering 4th Edition*. A K Peters/CRC Press, Boca Raton, FL, USA, 2018. ISBN 978-1-13862-700-0. Chapter 22.2.

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

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

[4] 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).

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

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

[7] 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.

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

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

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

[11] 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).

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

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

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 [9].

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.

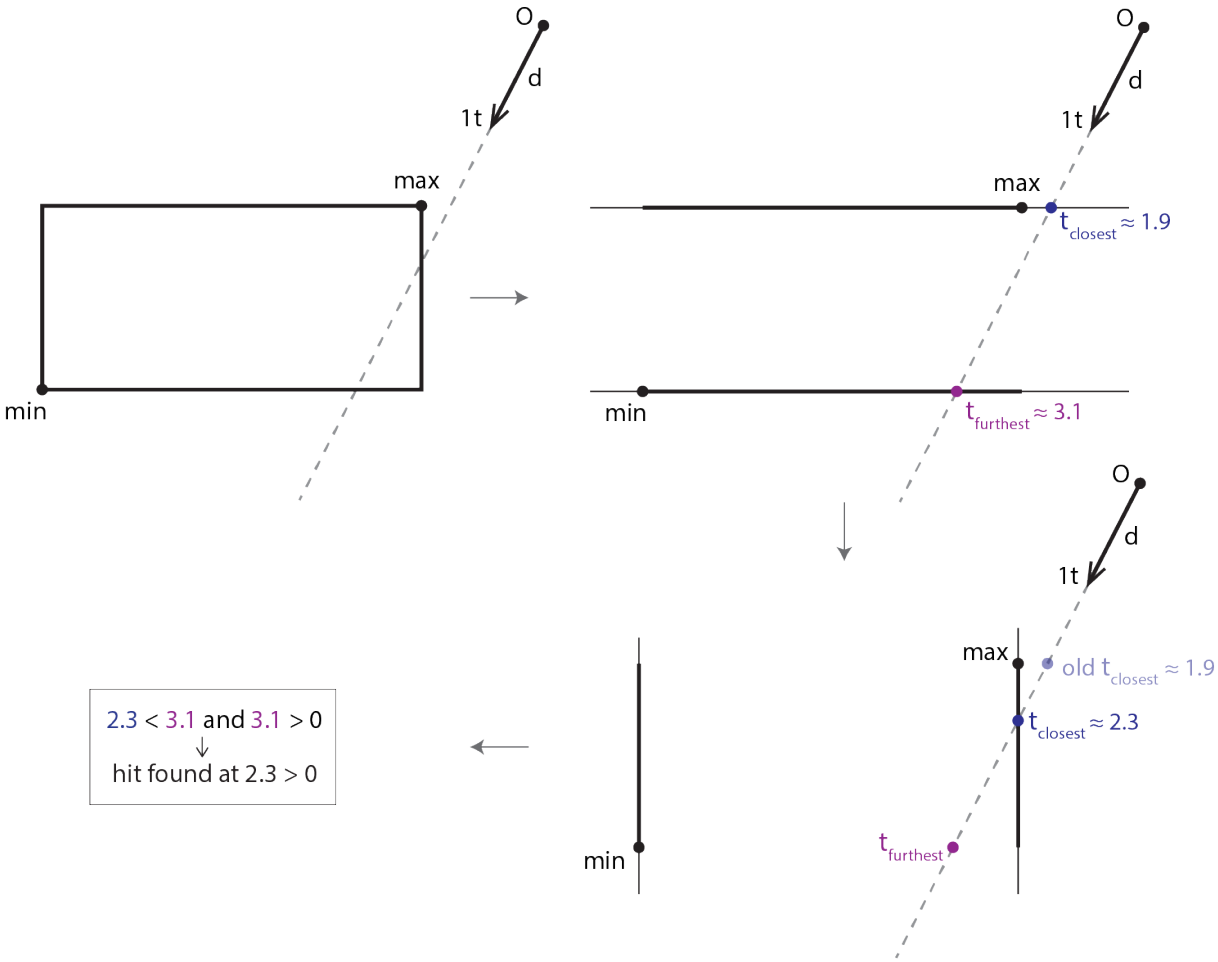


Figure A.1: Visual representation of the presented algorithm in 2 dimensions. An interactive simulation of this algorithm can be found at: `https://www.geogebra.org/m/np3tnjvb`.

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 [8].

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) [1]. 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.

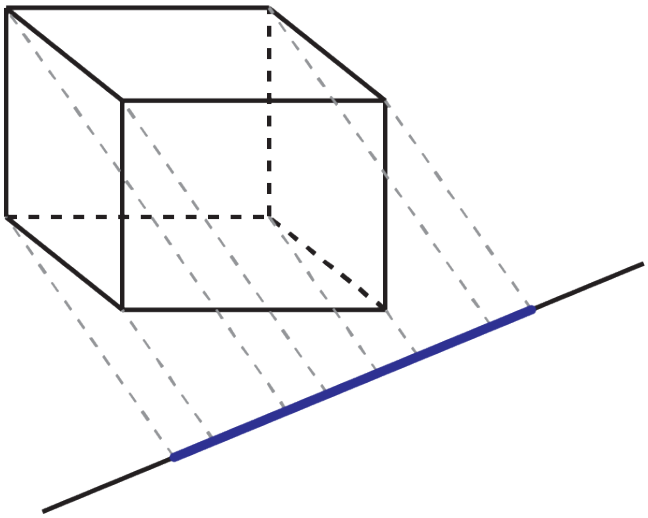
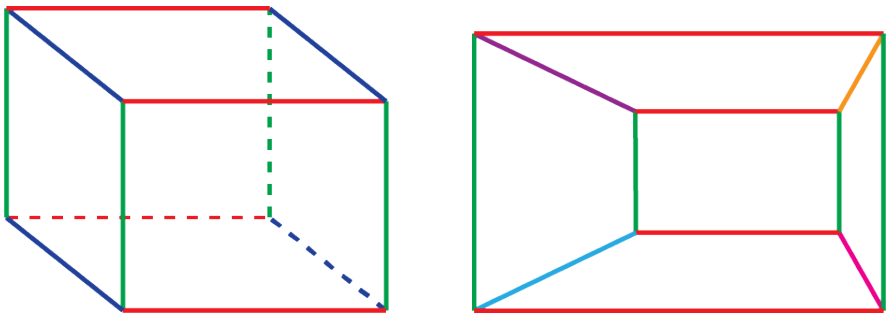


Figure A.2: The projection of an AABB on an axis.

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

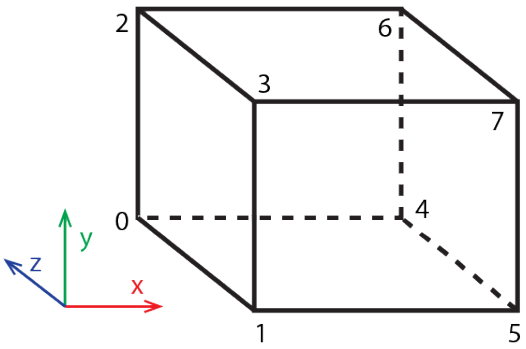
(b) Frustum: 6 different edges directions

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:



(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 [5].

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

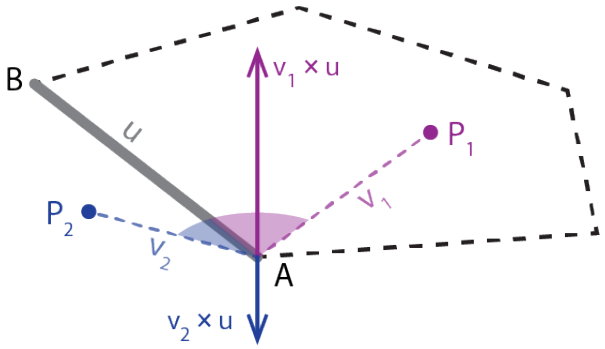


Figure A.4: Visualization of cross product.

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 ([10]):

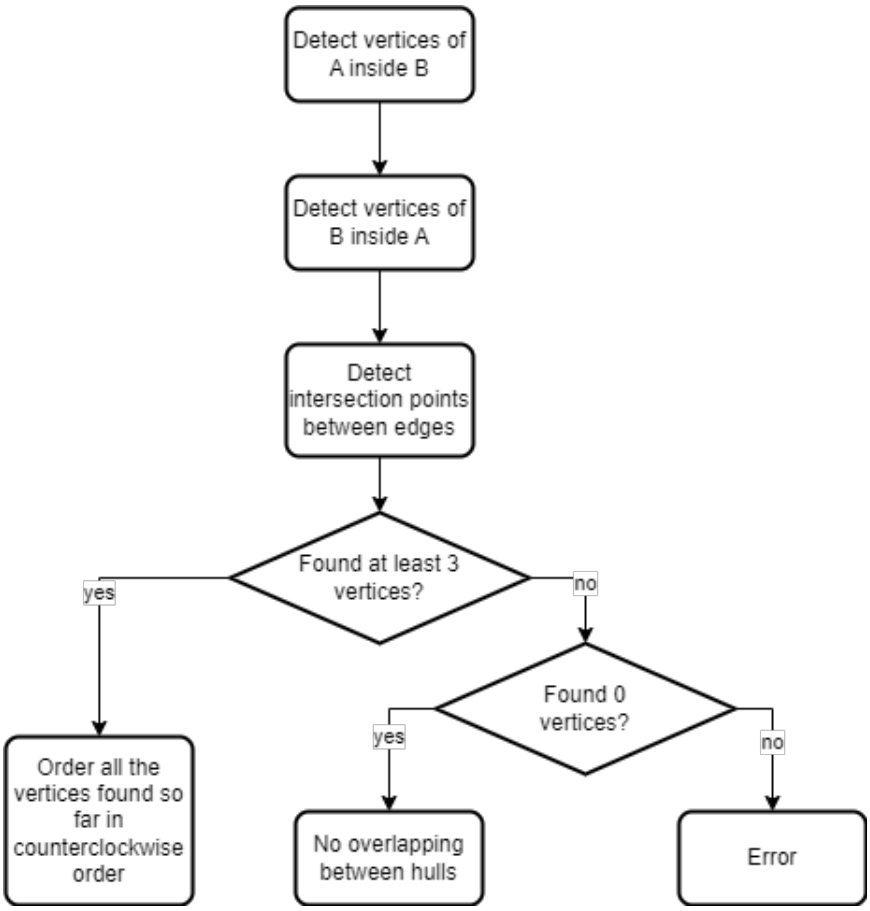


Figure A.5: General algorithm to find the overlapping region between two convex hulls called A and B.

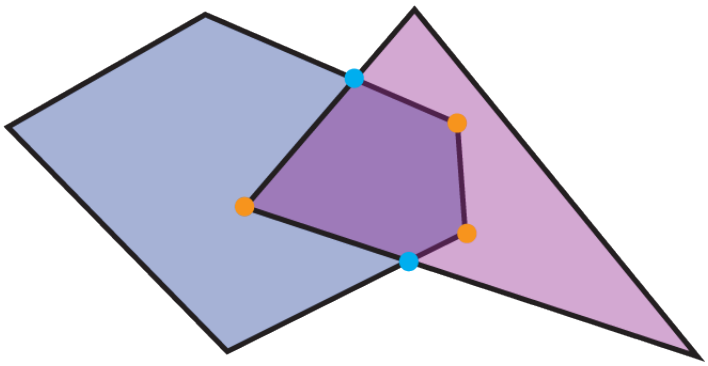


Figure A.6: Yellow vertices are found in the first 2 steps (vertices inside), whereas light blue vertices are found in the third step (edges intersections).

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 $\overline{PQ}$:

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

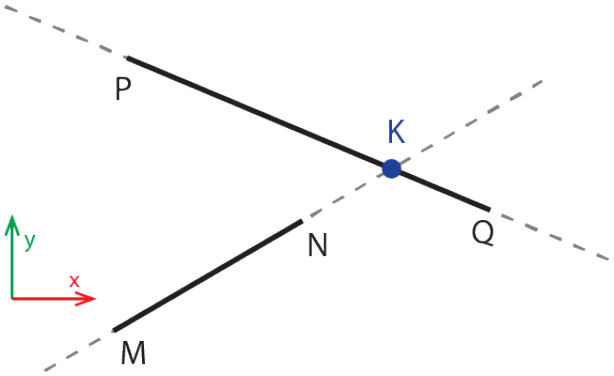


Figure A.7: In this example segments do not collide because $max(M_x, N_x) = N_x \not\geq K_x$ or $max(M_y, N_y) \not\geq K_y$

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.

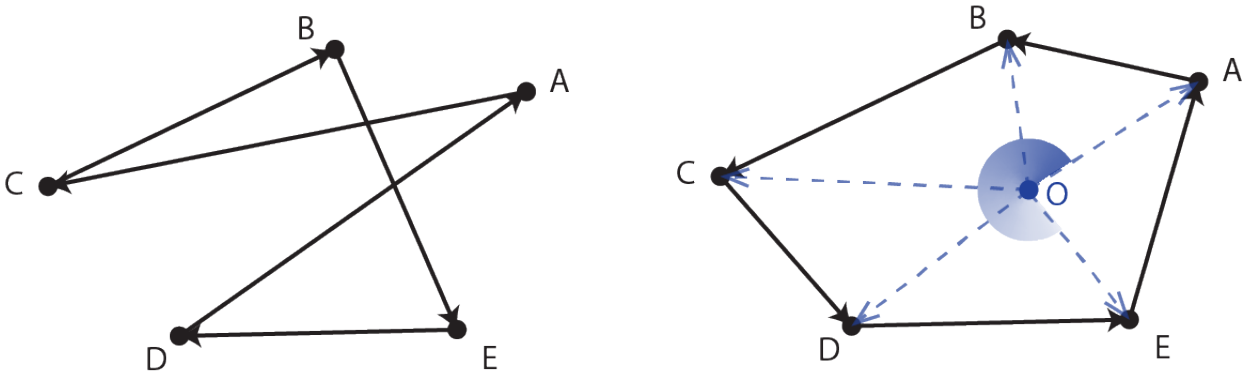


Figure A.8: Vertices of a convex hull before and after $atan2$ sorting.

[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 [7].

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 $P_0 = P_{n+1}$.

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.