Universidade Federal de Pernambuco

Centro de Informática

Engenharia da Computação

Study of Path Tracing and Coherent Ray Tracing Techniques

Rodrigo Brayner Lyra

Advisor: Prof. Dr. Silvio de Barros Melo

Recife, 2022

Rodrigo Brayner Lyra

Study of Path Tracing and Coherent Ray Tracing Techniques

A B.Sc. Dissertation presented to the Centro de Informática of Universidade Federal de Pernambuco in partial fulfillment of the requirements for the degree of Bachelor in Computer Engineering.

Concentration Area: Computer Graphics

Advisor: Prof. Dr. Silvio de Barros Melo

Recife, 2022

RESUMO

Operações de acesso à memória são, de longe, as mais custosas em arquiteturas de processadores modernos. Para mitigar o gargalo de von neumann, arquitetos de hardware desenvolveram hierarquias de memória que exploram conceitos como *localidade espacial* e *localidade temporal*, para guardar regiões da memória recentemente usadas em memórias de acesso rápido, denominadas memória cache. Porém, para aproveitar ao máximo os seus benefícios, programadores necessitam organizar os dados e algoritmos de seus programas para maximizar a localidade de dados. Técnicas de renderização vêm sendo desenvolvidas desde meados dos anos 1980. As indústrias cinematográfica e de animação utilizaram métodos de rasterização para renderizar os seus filmes. Essas técnicas processam cada objeto de forma sequencial e garante um acesso coerente à memória. Porém, esse tipo de algoritmo se limita a calcular apenas a iluminação local, o que gera imagens menos realistas. O crescente poder computacional dos hardwares modernos, as décadas de pesquisa e melhorias das técnicas baseadas em física e as otimizações de rastreamento de raios levaram a indústria cinematográfica a adotar em suas produções, algoritmos de *path tracing* baseados em física. Considerando que esses algoritmos inerentemente causam acesso randômico à memória, a busca por técnicas que possibilitem o seu acesso de forma coerente é um dos principais tópicos de pesquisa para a otimização dos métodos baseados em física, sendo tema de diversas abordagens e técnicas desenvolvidas durante anos. Nas últimas décadas, foram propostas inúmeras técnicas de paralelização, vetorização e reordenação das computações envolvidas nos algoritmos de *path tracing*. Este trabalho busca realizar uma revisão de algumas dessas técnicas que foram importantes para a evolução e a recente adoção da renderização baseada em física nos grandes estúdios de filmes e de animação; bem como descrever o processo de criação de um *path tracer* desenvolvido especialmente para este trabalho.

Palavras-chave: Computação Gráfica, Integração Monte-Carlo, *Path Tracing*, Coerência de Raio, Memória Cache.

ABSTRACT

Memory access operations are by far the most costly on modern computer architectures. To mitigate such a problem, hardware architects have developed memory hierarchies that exploit concepts such as *spatial locality* and *temporal locality* to store recently used memory regions in fast-access memories, called cache memory. However, to take full advantage of its benefits, programmers need to organize the data and algorithms of their programs to access memory sequentially. So, although physically based rendering techniques have been developed since the mid-1980s, for decades the film and animation industries have used rasterization methods to render their movies. Those techniques process each object sequentially and ensure coherent memory access. However, such algorithm is limited to calculating only local lighting, which results in less realistic images. Recently, the increasing computational power of modern hardware, decades of research and improvements in physically based techniques and ray tracing optimizations have led the film industry to adopt physically based path tracing algorithms in their productions. Considering that such algorithms inherently cause random access to memory, the search for techniques that allow access to memory in a coherent way is one of the main research topics for the optimization of physically based methods, being the subject of several approaches and techniques developed over the years. In the last decades, several techniques for parallelization, vectorization and reordering of the computations involved in path tracing algorithms have been proposed. This work aims to review some of those techniques that are important for the evolution and recent adoption of physically based rendering in major film and animation studios; as well as to describe the process of creating a path tracer developed especially in this work.

Keywords: Computer Graphics, Monte Carlo Integration, Path Tracing, Ray Coherence, Cache Hierarchy

LIST OF ACRONYMS

PBR	Physically Based Rendering
BSDF	Bidirectional Scattering Distribution Function
BRDF	Bidirectional Reflectance Distribution Function
BTDF	Bidirectional Transmittance Distribution Function
CPU	Central Processing Unit
IC	Integrated Circuit
DRAM	Dynamic Random Access Memory
MMU	Memory Management Unit
SIMD	Single Instruction Multiple Data
SSE	Streaming SIMD Exntensions
AVX	Advanced Vector Extensions
BVH	Bounding Volume Hierarchy
AABB	Axis Aligned Bounding Box
ISA	Instruction Set Architecture

SUMARY

1 Introduction

Computer Graphics is a vast research field in computer science in which the main goal is to synthesize an image based on some geometric input data. The process of synthesizing this image is called *Rendering.* This task may vary from drawing an user interface for a mobile or desktop application on the screen of a device, to rendering an entire movie with complex 3D scenes with billions of polygons and thousands of light sources using global illumination and complex, realistic materials. Techniques that accurately simulate the physical interactions between surfaces and light are known as Physically Based Rendering (PBR) techniques.

PBR methods are widely used in movies and video games today, but the rendering process and techniques used in these media are quite different. While movies use offline rendering techniques, video games use real-time rendering techniques. Since games are interactive applications, players want them to run smoothly in their devices, the games rendering system must generate at least 30 frames per second on the screen to give the human brain an impression of a fluid animation. Offline rendering, on the other hand, involves computationally expensive methods often used to generate film-quality images. Since movies are rendered in a wide time window, much more complex scenes and rendering techniques are used, allowing each frame to be processed for hours. Although not all offline rendering techniques use PBR, it is the most common approach nowadays. While many Physically Based Rendering research topics converge between these two rendering modes, this work focuses on Path Tracing, a Physically Based Rendering technique for offline rendering, and from now on the term will be used in this context.

Physically Based Rendering comprises simulating the physics of light and its interactions with surfaces and media to generate an image similar or even indistinguishable from real photography. Among several PBR rendering techniques developed over the years, Path Tracing received a recent general adoption by the film and animation industries. The algorithm comprises generating several paths for each pixel, representing light rays arriving at the camera, and simulates all the bounces and illumination contributions from that ray until it reaches a light source. The level of geometric complexity and lighting quality that can be achieved with state-of-the-art Path Tracing is shown in fig. 1.

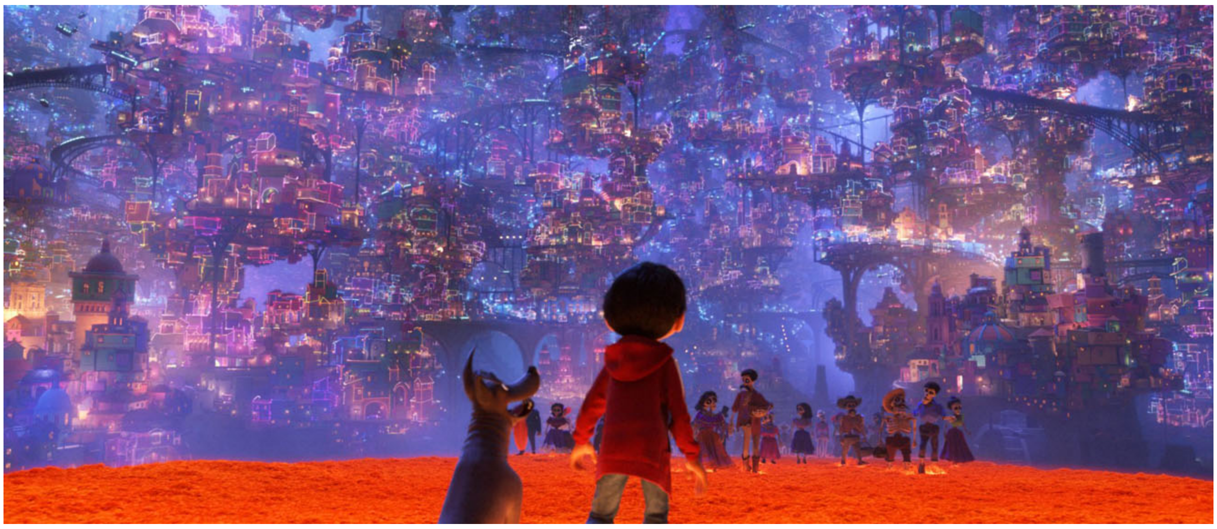


Figure 1 – Shot of the city of the dead from the Pixar's movie *Coco* (©2017 Disney•Pixar), rendered using Path Tracing with the RenderMan rendering engine. This scene contains a very high amount of geometry visible from the camera position, but the most incredible feature of this shot is handling around 8.2 million light sources. Rendering such an amount of light was not possible when the film was still being planned, and the Pixar's research and development team had to create novel algorithms to accomplish this task.

Source: FxGuide website article: "RenderMan's Visuals for Coco".

One of the main obstacles for the adoption of Physically Based Rendering in graphics applications like films or games is the high computational cost for generating such images. A real time application such as a game, which usually runs at 30 frames per second, can take a maximum time of $33ms$ to render a frame. A full feature movie with two hours of duration and 24 frames per second would take one year just for rendering if each frame takes 3 minutes to render. In the 1980s, an image generated from path tracing would take hours to render on a high-end computer. On top of that, global illumination techniques require that the whole scene is present in the system main memory, because a ray need to test intersection with all the objects in the scene. These limitations made the early adoption of physically based rendering techniques impossible for the entertainment industries, which opted for faster and more flexible architectures.

Also, according to Moore's law, the computational power of processors doubles every 18 months since the early 1960s. But since artists push new technologies to the limit and continuously improve production image quality when ever possible, James Blinn, a renowned computer scientist and computer graphics expert, stated that rendering time remains constant even with computers getting faster. This assertion was later called the *Blinn's Law* and turned out to be mostly true.

However, the scientists saw the future potential of pursuing a physically based simulation of light, specifically the path tracing algorithm. These techniques not only created beautiful photo-realistic images, but could help the industry to standardize its production processes and pipelines into a unified model, as at the time each company had its own architecture with incompatible specificities and workflows. Path tracing also does progressive rendering, meaning that artists can start the rendering, preview a result with noise but very close to production quality in little time, and either resume the process from where it stopped, or fixing any problem before generating unnecessary production-quality images and wasting valuable rendering time.

To accomplish this task, a lot of research done in the last decades focused on a variety of disciplines. Some delved deeper in specifically optimizing the rendering process by proposing novel rendering architectures and pipelines, hardware-specific optimizations, ray-primitive intersection tests, ray-tracing acceleration structures, SIMD shading, and SIMD ray tracing implementations. Other works sought to improve the overall image quality and visual effects while also sometimes bringing optimizations by reducing the amount of samples necessary for a noiseless image. Those works include proposals in the areas of reflection models, sampling theory, animation, camera effects, texturing, shape representations, among others.

This work provides an overview of Physically Based Rendering techniques, with an especial focus on Path tracing, and discuss some methods developed by researchers to speed up the shading and ray traversal steps. Chapter 2 covers the basic principles of physically based rendering and rendering techniques that will be the focus of the rest of the work. It also covers some topics on computer architecture and CPUs, since it is crucial to understand how those characteristics affect the implementation of high-performance applications. Chapter 3 of this work discusses the Path Tracing algorithm bottlenecks and sources of performance problems. It also covers several researches developed over the years for optimizing path tracing in modern processor architectures. Chapter 4 presents a path tracer developed as a practical exercise for this work. Although it does not support the advanced techniques described in the previous chapter yet, the implemented path tracer was capable of rendering considerably complex scenes with a high resolution and a high number of samples. Chapter 5 concludes this work, while also pointing at directions for future works, both in the theoretical and practical fields.

1.1 Goals

1.1.1 Main Goals

This work aims to do a theoretical and practical overview of the research in physically based rendering and path tracing, discussing the performance problems introduced by executing ray tracing in complex scenes, as well as to analyze the different approaches proposed by researchers to optimize and accelerate ray intersection tests and shading calculations. It also describes the process of implementing a path tracer.

1.1.2 Specific Goals

- Overviewing the previous works on ray tracing and physically based rendering.
- Elaborating and discussing the performance problems of path tracing as well as the researched techniques to mitigate them.
- Studying the coherent ray tracing algorithms and acceleration techniques for physically based rendering using path tracing.
- Presenting an implementation of a path tracer based on the architecture proposed by the reference book "Physically Based Rendering: From Theory to Implementation" (PHARR; JAKOB; HUMPHREYS, 2016).

2 Theoretical Background

2.1 Physically Based Rendering

Physically Based Rendering (PBR) stands for using physically correct light-surface and light-volume interactions in shading and lighting models (WILSON, 2020). It is not a strictly defined set of rules and standards that renderers must conform. Non-PBR techniques typically use low-cost computational approximations to achieve some kind of realism without a heavy penalty in performance. Such approximations have been used by the entertainment industry for a long time, but with the processing power of modern architectures and the development of many optimized techniques, PBR is now widely used by the industry. The term "Physically Based Rendering", although used for a long time in academic works, became popular with Pharr, Jakob e Humphreys (2016) book of the same name, which also won a Technical Achievement Academy Award "for their formalization and reference implementation of the concepts behind physically based rendering" (BENEDICT, 2014). This section will cover the key topics of PBR that will be used throughout this work.

2.1.1 BSDF

It is general knowledge that what our eyes see are light rays that reach the retina coming either from light interaction with the surface of objects or light emitted from some surface. Every surface is composed of some matter, therefore they interact with the incoming light in different ways. A mirror, for example, reflects light rays in the opposite direction of the incoming light, shown as the outgoing orange arrows in fig. 2, and thus the viewing angle defines what objects you will see in the reflection. Transparent materials such as glasses, on the other hand, transmit most of the light it receives into its surface, while also reflecting part of the incoming light. The light transmission causes a distortion in the light direction based on the *Snell's law*, which accounts for the refractive index of the surface and the angle of the incoming light. In contrast, surfaces like concrete, for example, have a uniform color at all viewing angles, thus means that the light rays arriving at a point in the concrete surface is transmitted into the surface and scattered

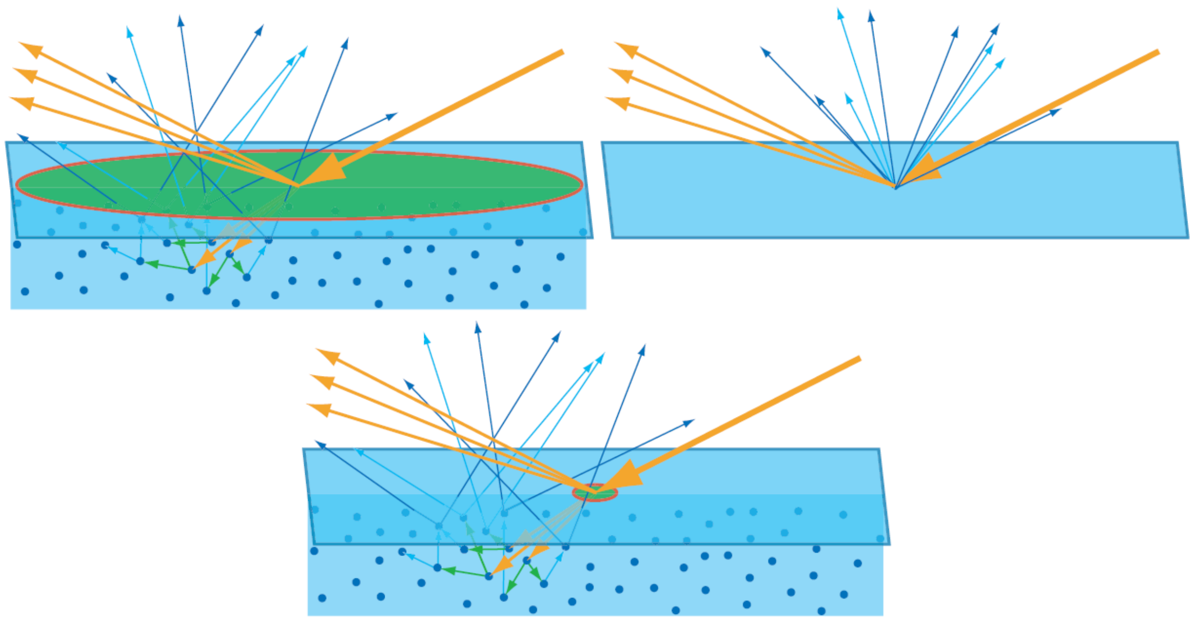
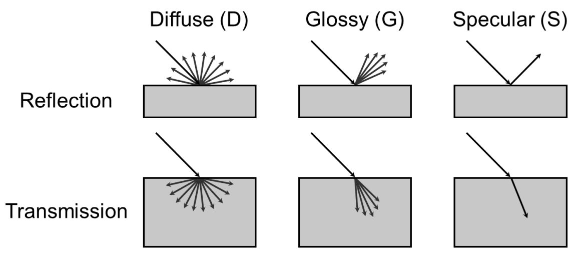


Figure 2 – The upper left image illustrate both the specular reflection as the outgoing orange rays, and the *Subsurface Scattering* effect, which consists of light entering the surface, being scattered and partially absorbed inside the surface, and then finally finding its way out. For surfaces where the distance between the ray exit and entry points is smaller than a pixel (green area in the upper left image), we can simply consider that the outgoing light comes from the entry point, as shown in the upper right image. In the bottom image, the distance from the exit point to the entry point of rays is larger than a pixel, needing advanced subsurface scattering shading techniques to be simulated.

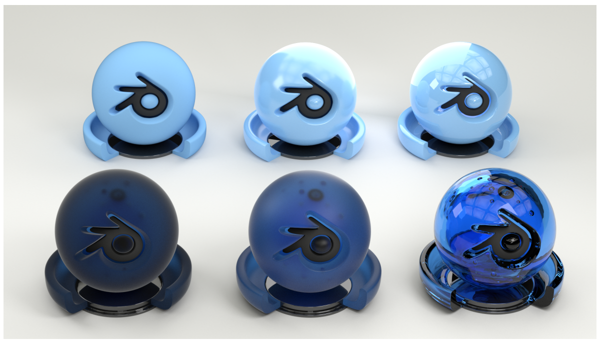
Source: Real-Time Rendering, 4th Edition (AKENINE-MLLER; HAINES; HOFFMAN, 2018).

inside it until it exits the surface in a random point and direction close to where it entered, a phenomenon called *Subsurface Scattering*, illustrated in the top left part of the image of fig. 2. (AKENINE-MLLER; HAINES; HOFFMAN, 2018)

The mathematical abstractions that model the interaction of the surface with light are called the Bidirectional Reflectance Distribution Function (BRDF) and Bidirectional Transmittance Distribution Function (BTDF). For the sake of simplicity, we can combine these two abstractions as a single Bidirectional Scattering Distribution Function (BSDF) which is denoted as $f(\mathbf{x}, \omega_i, \omega_o, \lambda)$ in this work. As the name suggests, the function is bidirectional, which means it depends on the directions of incoming light (ω_i) and outgoing view (ω_o). It also can vary based on the position ($\mathbf{x}$) on the surface and the wavelength of the light (λ). Phenomena like fluorescence and phosphorescence are rarely used in rendering, thus we can assume that incoming light of some wavelength is reflected or transmitted at the same wavelength (PHARR; JAKOB; HUMPHREYS, 2016). Fig



(a)



(b)

Figure 3 – Figure (a) illustrates the three common types of BSDF's, Diffuse, Glossy, and Specular, showing both reflection and transmission effects. Figure (b) shows the rendered result using path tracing of each of the BSDF's types illustrated in figure (a), in the same order.

Source: Materials created by me in Blender using BlenderDiplom Cycles Material Test Scene objects.

3a illustrates the behavior of the incoming light rays interacting with different types of BSDF's, while fig. 3b shows the result of rendering objects with each of these BSDF's respectively.

To be physically correct, a Bidirectional Reflectance Distribution Function (BRDF) must have two important properties: *Reciprocity* and *Energy Conservation.*

Reciprocity means that for every possible pair of direction ω_i and ω_o:

$$f_r(\mathbf{x}, \omega_i, \omega_o) = f_r(\mathbf{x}, \omega_o, \omega_i)$$

The *Energy Conservation* property means that the outgoing light reflected or transmitted by the surface is not greater than the total incoming light. Based on the rendering equation (equation 2.1):

$$\int_\Omega f(\mathbf{x}, \omega_i, \omega_o, \lambda, t)(\omega_i.\mathbf{n})d\omega_i \leq 1$$

Note that although a Bidirectional Transmittance Distribution Function (BTDF) must follow the *Energy Conservation* property to be physically correct, it does not need to follow the *Reciprocity* because of the light direction distortion caused by the transmission.

2.1.2 Ray Tracing

The ray tracing technique was first introduced by Appel (1968) to determine the surface visibility on screen, i.e. the closest surface to the screen plane in a pixel. He noted

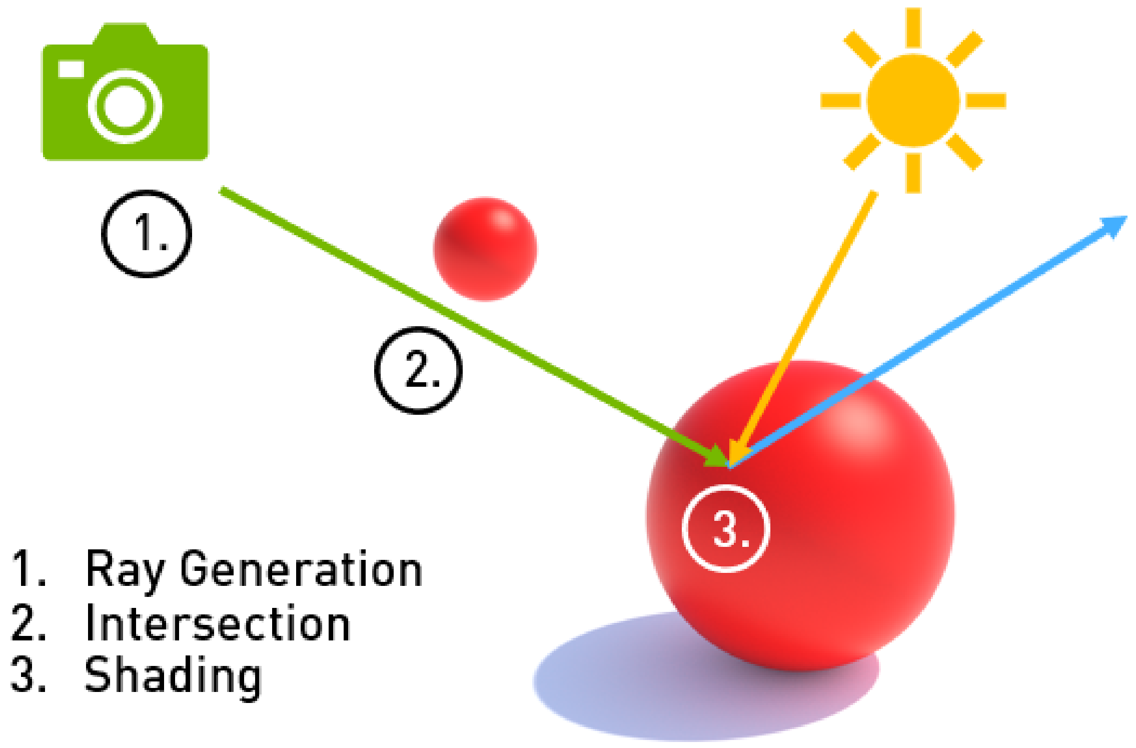


Figure 4 – Illustration of the ray tracing algorithm. First, the camera generate rays in the direction of each pixel, then cast these rays in the scene, testing for the closest ray intersection. When the hit point is found, a ray called shadow ray is cast in the direction of the light, testing if the light is occluded by another surface. For recursive ray tracing, it can generate secondary rays for reflective or refractive surfaces.

Source: NVIDIA Developer blog article: "NVIDIA OptiX Ray Tracing Powered by RTX"

that it could also determine if a surface is visible by a light in order to draw shadows. As illustrated in fig. 4, Appel's algorithm initial steps consists in generating rays exiting from the camera for each pixel in the image plane, which are called primary rays. The ray tracing step tests the ray for intersection with all the scene geometries, calculating the closest intersection. After calculating the hit point, it could cast a new ray in the light direction to test if the point is lit or shadowed. But with the limited hardware of that time and the lack of ray tracing acceleration algorithms, the technique demanded a very high computation time.

Sometime later, Whitted (1980) presented the first recursive ray tracing architecture. It follows the first steps of Appel's algorithm, but for reflective and refractive surfaces, secondary rays are generated in the reflection or refraction direction. It could also simulate diffuse surfaces using direct illumination, like Blinn-Phong shading (BLINN, 1977), but it was not capable of simulating global illumination and diffuse interreflection. At that time, this novel algorithm provided results that were not possible with the previous methods, marking a major advance in physically based rendering techniques.

2.1.3 Microfacets

As previously presented, the first physically based rendering methods started to be developed in the 1980 decade. Whitted (1980) introduced the pre-existing ray tracing algorithm to compute lighting effects, but had not used any advanced physical model for calculating light-surface interactions. Two years later, Cook e Torrance (1982) introduced the microfacet theory (TORRANCE; SPARROW, 1967) into computer graphics, accurately modeling many types of materials such as metals, dielectrics, and diffuse surfaces. The microfacet theory suggests that a macroscopic piece of surface with normal n is actually made up of microscopic flat surfaces with varying normals, being n the average of those microfacets normal vectors. Given a view direction v and light direction l, the halfway vector h is the normal of the microsurfaces that reflects light coming from l into the view direction v, as shown in fig. 5.

Cook e Torrance (1982) proposed a statistical model to formulate the BRDF of physically based surfaces. Although not explained in detail in order to maintain the scope of this work, the BRDF of the Cook-Torrance model is defined as:

$$f(l,v) = \frac{F(l,h)G(l,v,h)D(h)}{4(n.l)(n.v)}$$

Where $F(l,h)$ is the Fresnel function, which computes the amount of reflected light according to the angle between l and h. $G(l,v,h)$ is the Geometry function, which accounts for the proportion of microfacets with normal h that may be occluded by other microfacets, and thus don't contribute to the reflection. Lastly, $D(h)$ represents the Distribution function, which defines the amount of microfacets with normal h present in the surface.

2.1.4 The Rendering Equation

The proposal of several different physically based rendering methods in the early 1980s lead to the work of Kajiya (1986), "The Rendering Equation". In that work, Kajiya presented an integral equation that generalizes previous rendering techniques by modeling the global light transport in a scene.

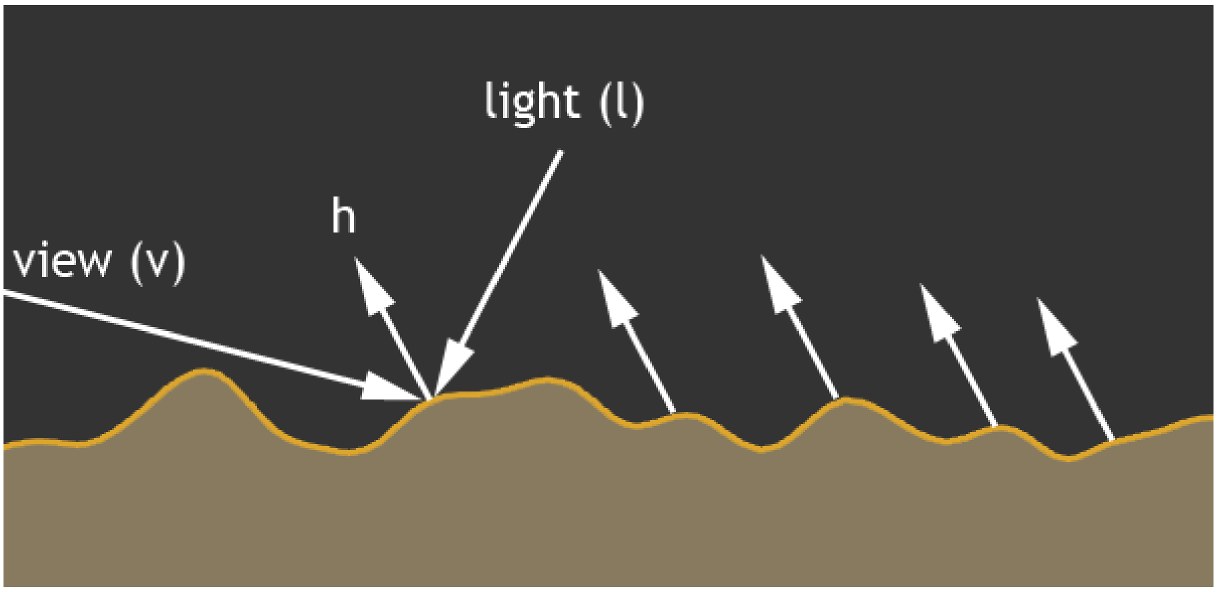


Figure 5 – The figure illustrates the microscopic level of a surface, being made of many micro surfaces with varying normals, called microfacets. As shown in the image, microfacets with normal h are responsible for reflecting light coming from l into the view direction v. These surface properties are statistically formulated in the Cook-Torrance BRDF model.

Source: Kevin George "The BRDF and Microfacet Theory" blog post.

Before introducing the equation, it is essential to understand the concept of **radiance**. Radiance is a physical unit that measures the amount of power arriving or exiting at a given point on a surface or optical system per unit solid angle per unit projected area, with its SI unit being *Watt* per *steradian* per *square meter* ($W.sr^{-1}.m^{-2}$). It can be roughly described as the amount of arriving light energy per time at an infinitesimal area from a certain direction. This unit is the basis of ray tracing algorithms as each ray contains a given radiance.

The Rendering Equation, also known as the Light Transport Equation, is presented below:

$$L_o(\mathbf{x}, \omega_o, \lambda, t) = L_e(\mathbf{x}, \omega_o, \lambda, t) + \int_\Omega f(\mathbf{x}, \omega_i, \omega_o, \lambda, t) L_i(\mathbf{x}, \omega_i, \lambda, t)(\omega_i.\mathbf{n})\, d\omega_i \tag{2.1}$$

We can read the equation as follows: given a point $\mathbf{x}$ in a certain surface $\mathbf{S}$, an outgoing direction ω_o, a wavelength λ, and a time $\mathbf{t}$, the total exitant radiance L_o coming out along the direction ω_o is equal to the emitted radiance L_e of the surface at point $\mathbf{x}$ at direction ω_o plus the integral of the incoming radiance L_i in direction ω_i multiplied by the corresponding BSDF f of the surface, multiplied by the weakening factor $(\omega_i.\mathbf{n})$, for

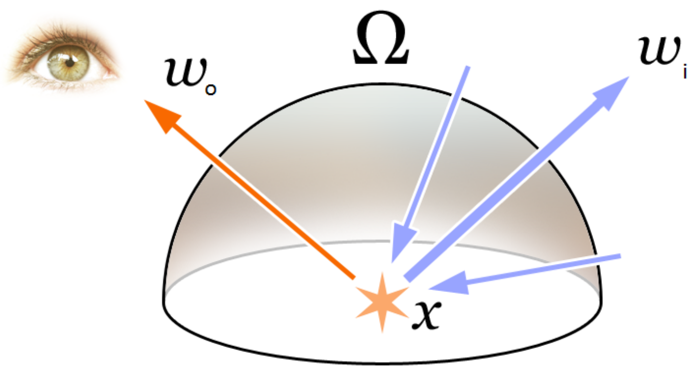


Figure 6 – The radiance arriving at the eye from direction ω_o is given by the surface emitted light plus the amount of light arriving at point $\mathbf{x}$ from all directions in the hemisphere Ω scaled by the surface BSDF.

Source: Wikipedia, Rendering Equation article.

every incoming direction ω_i in the hemisphere Ω around the surface normal $\mathbf{n}$. Fig. 6 provides a more intuitive and non-mathematical explanation for the equation.

The equation is clearly recursive since the outgoing radiance depends on the radiance reaching the surface from all directions. Note that the whole equation also depends on the wavelength and time. This means that the surface light interaction can vary in time, in a scene with an animated light, for example, and in wavelength, like a green surface that reflects green light waves but absorbs red and blue light waves. The trichromatic Red-Green-Blue (RGB) color model is widely used by most production rendering applications to describe the response of the surfaces to each wavelength, although more specialized applications may use more precise spectrum representations.

2.1.5 Monte Carlo Integration

Monte Carlo Integration is a numerical integration technique based on the Monte Carlo Method that uses non-deterministic random numbers to compute a definite integral. The idea behind the algorithm is to compute an approximation to the desired definite integral by sampling the function using random sampling. Given a number N of samples and an integral equation I where x can be a multidimensional variable

$$I = \int_{\Omega} f(x)dx$$

We can describe the Monte Carlo Estimator as:

$$Q_N = \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{p(x_i)} \tag{2.2}$$

Where $p(x_i)$ is the probability distribution function for obtaining the variable x_i.

While not mathematically shown in this work, Monte Carlo Integration error reduces at a rate proportional to $\frac{1}{\sqrt{N}}$, which is quite slow and means that you need to use four times more samples to reduce the error in half. This results in Monte Carlo methods requiring a very high number of samples to compute its result with low variance. While other numerical integration methods that usually use deterministic quadrature approaches converge faster for one-dimensional integrals, the performance of these techniques degrades exponentially whenever one increases the dimension of the integral. On the other hand, the Monte Carlo integration convergence rate remains constant for any number of dimensions of the integral. This makes the Monte Carlo integration the best choice for multi-dimensional integrals, especially for physical and mathematical problems. (PHARR; JAKOB; HUMPHREYS, 2016) (KAJIYA, 1986)

Reducing the variance of Monte Carlo estimators without greatly increasing the number of samples is a major concern for efficiently solving integrals and is a big area of research. One of the most powerful technique developed in this process is Importance Sampling (KLOEK; DIJK, 1978). The Importance Sampling technique consists in taking samples using a distribution function that resembles the shape of the function $f(x)$, generating samples that are more frequent around the area where the function has higher values, hence the area with greater contribution to the integral value.

As shown in Pharr, Jakob e Humphreys (2016), if we take the samples for a Monte Carlo Estimator using a distribution $p(x) = cf(x)$, then for the probability density function integral to remain equal to 1, we need

$$c = \frac{1}{\int f(x)dx}$$

Substituting the distribution $p(x)$ in the Monte Carlo estimator (Equation 2.2), we have

$$Q_N = \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{\frac{f(x_i)}{\int f(x)dx}} = \frac{1}{N}\sum_{i=1}^{N}\int f(x)dx = \int f(x)dx$$

Which is the integral value we are trying to approximate initially. Obviously that to find such a PDF, we would need to know the integral value beforehand. But if we can find a PDF that approximates the shape of $f(x)$, the estimator converges faster.

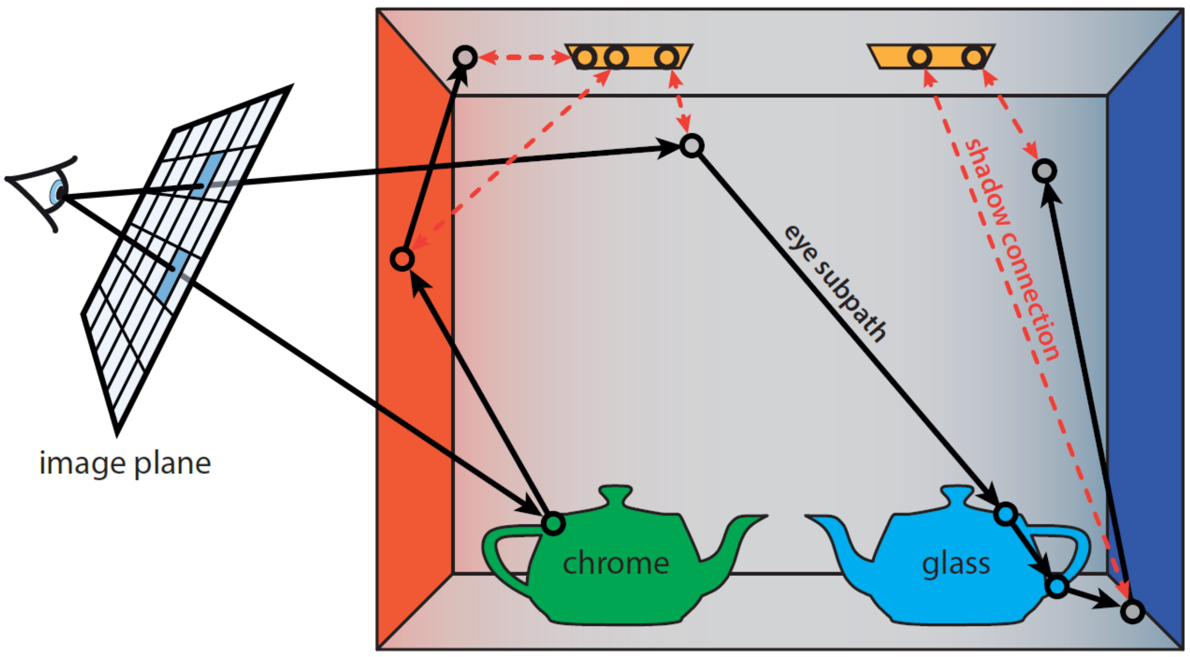


Figure 7 – Illustration of the paths generated by the Path Tracing algorithm. The rays (black arrows) start out of the eye, pass through one pixel in the image plane and hit an object in the scene. When hit in fully specular surfaces like chrome or glass, the path can only continue in one direction. If hit in a diffuse surface, like the walls, it first cast a shadow ray (red arrows) toward a point sampled in the light, testing for the light visibility and applying its direct illumination contribution to the path. Then the path continues in a direction sampled by the surface BSDF.

Source: The Path to Path-Traced Movies (CHRISTENSEN; JAROSZ, 2016).

2.1.6 Path Tracing

Path Tracing is a rendering technique that was proposed by Kajiya (1986) in the same paper that he introduced the Rendering Equation. This technique uses the Monte Carlo integration method (Section 2.1.5) to numerically evaluate the integral of the Rendering Equation (Eq. 2.1), and was the first general-purpose unbiased Monte Carlo light transport algorithm (PHARR; JAKOB; HUMPHREYS, 2016).

The basic algorithm not only accurately reproduces many global illumination and indirect lighting effects that were previously not possible or very difficult to implement to other rendering algorithms, but could also naturally simulate physically based camera lens with effects like motion blur or field depth, area light sources, or caustics.

The core idea of path tracing is to generate a certain amount of samples in the camera lens for each pixel and follow those paths coming out of the camera through a series of light scattering events until it ends in a light source. Many times, the probability of a path reaching a light source is low or even zero, e.g., delta lights like the point light, making the path contribution null and thus wasting computation. To solve that, path tracing implementations sample the direct lighting coming from light sources in the scene for every hit point in the calculated path, checking if the light is visible by casting a ray called 'shadow ray', instead of hoping that a path will end hitting a light source. This optimization is called *Next Event Estimation* and drastically reduces the number of samples necessary to produce a high-quality image. Two paths samples with next event estimation are illustrated in fig. 7.

Note that the use of importance sampling is fundamental for a good performance in path tracing. For example, instead of taking uniform hemisphere samples for any type of BSDF and then evaluating the path even for close to zero contributions, the BSDF samples could be obtained by sampling from distributions that matches the BSDF Probability Distribution Function (PDF). The previously described Next Event Estimation is also an importance sampling technique, which samples points in the light surface instead of randomly choosing a direction and expecting it to hit a light.

Although the path tracing algorithm produced incredibly realistic images never seen before, the computational cost to generate each of such images were absurd, taking about 7 hours to render a 256 x 256 pixels image with 40 samples per pixel on an IBM-4341 (KAJIYA, 1986), a high-end computer at that time that cost between $U\$245.000$ and $U\$275.000$ on launch date. Not only the computers were orders of magnitude less powerful and more expensive at that time, but many ray tracing acceleration techniques and rendering architecture optimizations were developed along the years to allow the recent commercial adoption of the Path Tracing technique.

2.2 Central Processing Units

Modern Central Processing Units (CPUs) are an electronic circuitry implemented on Integrated Circuit (IC) microprocessors responsible for the execution of computer programs through multiple types of instructions and input/output operations. Current CPUs are multi-core processors, physically containing multiple processor cores on a single integrated circuit, allowing it to execute instructions in parallel, enabling programmers to simultaneously execute multiple tasks or a task divided into independent subtasks.

The proper study and review of computer architectures and its features would result in a full work on its own, making an in-depth analysis of the subject impossible in this work. But as previously stated, Path Tracing is a very computational intensive algorithm, requiring that software engineers architect their path tracer implementation to leverage every aspect of the target hardware architecture. This section covers some topics important to understand the bottlenecks of path tracing as well as the numerous developed techniques to overcome the difficulties covered in chapter 3. Almost every technical statements in this section is based in Hennessy e Patterson (2011) computer architecture book.

2.2.1 Memory Latency

Mentioned in chapter 1, Gordon Moore noted in 1965 that the number of transistors in an Integrated Chip doubles every two years. Although he only mentioned it in an interview, this trend continued to happen over the years and is known as *Moore's Law* (INTEL, 2019). This exponential increase of processor speed over the years, as well as the development of multi-core processors, could lead one to think that the only limit to processor performance is when transistors become as small as atoms. Although this is partially true, there is one main problem that slows down CPU performance: Dynamic Random Access Memory (DRAM) latency. As shown in fig. 8, the gap between processor speed and DRAM memory speed is getting wider every year. While CPUs can execute a single arithmetic instruction in a few clock cycles, it may need to wait several hundred clock cycles for the necessary data for the instruction to arrive at the CPU from the DRAM. The instructions the CPU execute are also stored in the memory, which can be a problem especially for large codes. (HENNESSY; PATTERSON, 2011) (NYSTROM, 2014)

2.2.2 Cache Hierarchy

Computer programmers and users want both large and fast memories for their softwares to run smoothly and with large memory space available. The DRAM is a reasonably large memory, but the considerable latency described in section 2.2.1 is a

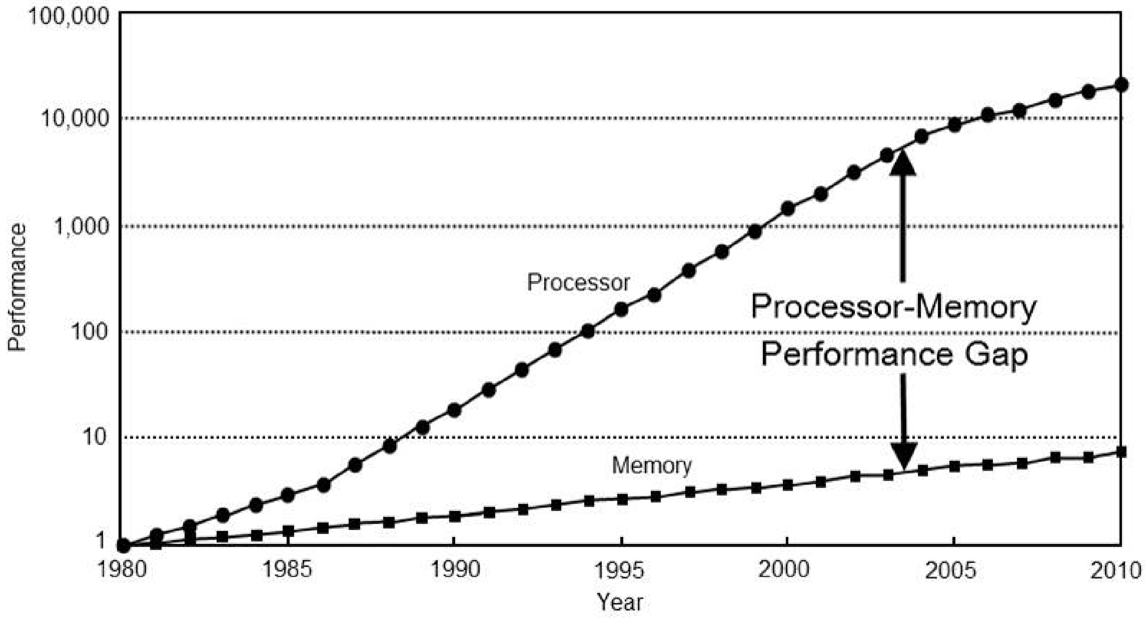


Figure 8 – Using the CPU and memory performance of 1980 as a base for comparison, in 2010, processors speeds has improved to over 10.000 times the performance of 1980. For DRAM access, in 2010 the performance has an improvement below 10 times the baseline. This gap in performance is widening over time, even though processor's speed improvements are diminishing.

Source: Computer Architecture: A Quantitative Approach (HENNESSY; PATTERSON, 2011).

large obstacle for the CPU performance. Another problem is that the price for storing a byte of fast memory is far more expensive than using a large and slow memory. This led computer engineers to develop a multi-level memory hierarchy as the solution to overcome this obstacle and avoid the CPU to spend most of its time waiting for data. (HENNESSY; PATTERSON, 2011)

Current commercial processors comprise a memory hierarchy that usually contains 4 levels, each smaller and faster than the level above. The topmost level is the previously described DRAM, which has a large capacity but very slow access latency. In most computer architectures, the DRAM is located outside the processor's chip. The next three levels is known as the cache hierarchy, and they are located inside the processor chip. They are comprised of L3, L2 and L1 cache levels. Both L3 and L2 cache levels are used to store instructions and data, and are shared between all the processor cores. The L1 cache is the smallest and fastest of the hierarchy, and each processor core has its own private L1 cache. Most architectures splits the L1 cache into two different memories: an instruction cache and a data cache. The described hierarchy is illustrated in fig. 9.

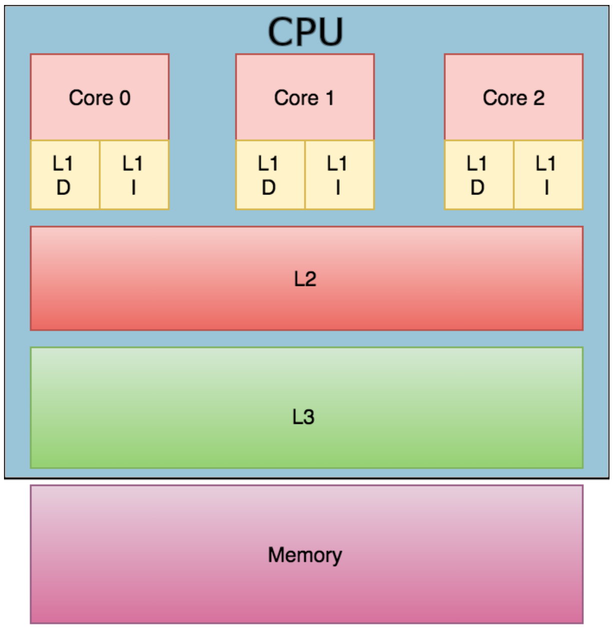


Figure 9 – Illustration of a memory hierarchy with 3 cache levels and a DRAM memory outside the CPU chip. While L3 and L2 cache levels are shared between cores and store both instructions and data, the L1 cache is private for each core and split between instruction and data cache.

Source: Cache Hierarchy Wikipedia article.

Although computer programs basic types usually have from 4 to 8 bytes, the cache memories store their data in larger blocks of bytes, also referenced as cache lines. Using a cache line instead of caching single bytes takes advantage of *temporal locality*, since we are likely to need that data again in the near future, and *spatial locality*, since we will probably use data near to the one we requested soon. Current processor architectures typically use a cache line size of 64 or 128 bytes.

When a processor core requests data from memory, its Memory Management Unit (MMU) will forward the request to the L1 cache, checking if the cache line containing the data is already present there. If the line is not found, a cache miss occurs, and the request is sent to the next level of the hierarchy, reaching the DRAM in case the data is not present in the processor cache hierarchy. When the data cache line is found at some level of the memory hierarchy, it is copied to lower cache levels.

While programmers have control of how data is organized in the storage disk and the DRAM, the cache memory hierarchy is abstracted and fully controlled by the processor. Because of the high level of abstractions provided by modern computer architectures and programming languages, programmers tend to ignore many inner aspects of the hardware that their software will run, and focus on optimizing their algorithms. While ignoring hardware specifics is acceptable for less computationally expensive programs, ignoring how to leverage the full potential of the processor can be crucial to enable algorithms such as path tracing to be used in industry, and not just in academia.

As shown in (HENNESSY; PATTERSON, 2011), there are four ways that a memory access can trigger a cache miss:

- **Compulsory** - When a cache line is requested for the first time, so the line has to be brought to the cache.
- **Capacity** - When a program needs more data than can fit in the cache, a miss will occur when some data already recycled by the cache is later requested again.
- **Conflict** - Occurs when a program references more than one line of data that maps to the same cache line, causing the current line to be recycled to give room to a new line. the conflict miss occurs when the recycled line is requested again.
- **Coherency** - When data shared between cores is written by one of the cores, the cache line is invalidated in the other core caches, causing a cache miss when some data in that cache line is referenced again. Note that it is not the single data that is invalidated, but the whole cache line. This lead to a situation called false-sharing, where different variables owned by different cores are present in the same cache line, and each write operation in one core variable causes caches misses in the other cores variables.

To summarize, it is important for a programmer craving for high performance on the CPU to organize its code around the data that will be processed. While compulsory and capacity cache misses are unavoidable, conflict and parallel coherency can be reduced by how the programmer organize the data stored in memory as well as how that data is accessed. For example, the best case to leverage *space locality* in large collections of data that needs constant processing is to store the collection contiguously in memory, and process each data sequentially in a single loop. The worst case is when the data in the collection is accessed in random order, and/or when the whole collection is scattered in different places of the memory. For data shared between processor cores, programmers can organize the data access pattern such that the cache line it is contained is not concurrently accessed by more than 1 core.

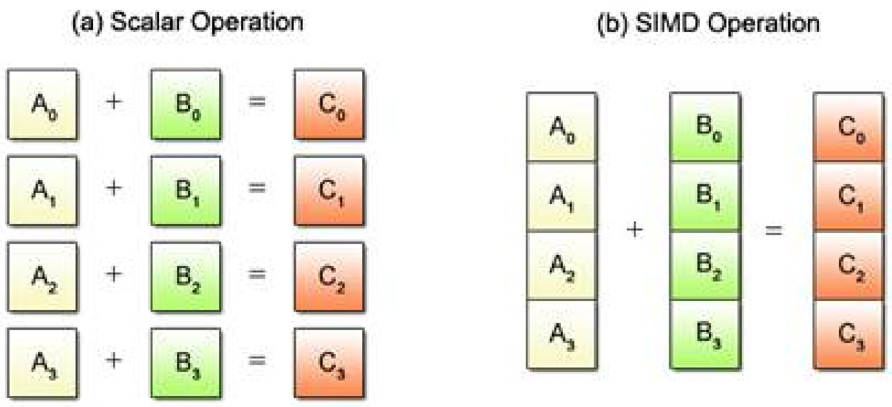


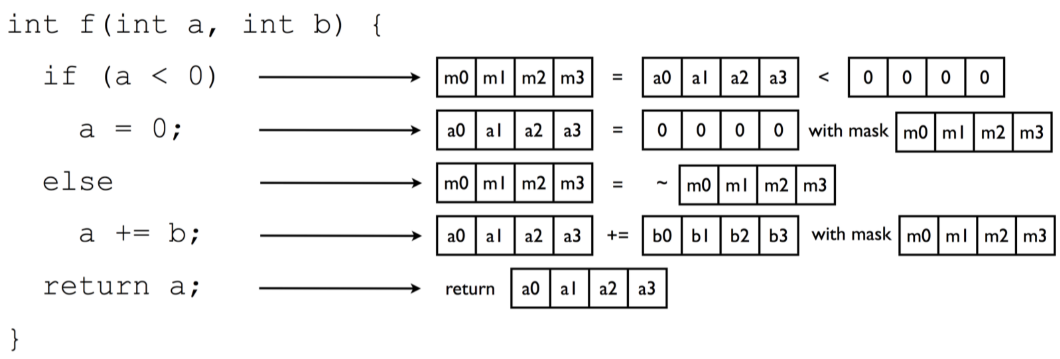
Figure 10 – Figure (a) shows four individual scalar operations between elements of 2 arrays. Figure (b) shows the same operation with four elements of the 2 arrays made with one single SIMD operation.

Source: "Basics of SIMD Programming" article at Czech Technical University website.

2.2.3 Single Instruction Multiple Data

Using multiple cores inside a CPU is a way of Task-Level Parallelism (TLP). Another form of parallelism provided by modern CPUs are Single Instruction Multiple Data (SIMD) instructions. A single SIMD instruction operates on multiple data values at the same time, a form of Data-Level Parallelism (DLP). As an example, fig. 10 shows the illustrates how 4 scalar operations can turn into one single SIMD operation by processing the values inside vectors. The x86 processor architecture currently comes with a number of SIMD instructions extensions that were introduced in the architecture along time. The Streaming SIMD Extensions (SSE) provides 128 bits wide instructions, enabling operations on sixteen 8-bit values, eight 16-bit values or four 32-bit values. The extension was later updated to support two 64-bits operations. Later, the Advanced Vector Extensions (AVX) introduced operations with 256 bits wide registers, and its next version, AVX2, introduced 512 bits wide registers. Although this work focus on the x86 architecture SIMD extensions, other major architectures such as ARM also implements SIMD extensions.

Note that it is very difficult for compilers to predict when a code that was written in a scalar form can be automatically optimized into vector operations, and although some compilers can detect some cases and optimize, it is no always predictable what optimizations the compiler will generate. The SIMD load instructions also requires that the first element of a vector is stored in a position aligned with the vector size. That is,



```
int f(int a, int b) {
  if (a < 0)
    a = 0;
  else
    a += b;
  return a;
}
```

m0 m1 m2 m3 = a0 a1 a2 a3 < 0 0 0 0
a0 a1 a2 a3 = 0 0 0 0 with mask m0 m1 m2 m3
m0 m1 m2 m3 = ~ m0 m1 m2 m3
a0 a1 a2 a3 += b0 b1 b2 b3 with mask m0 m1 m2 m3
return a0 a1 a2 a3

Figure 11 – In ISPC, an integer variable is actually a vector of integers. In this example, the vectors consists of 4 values. For the values of *a* that are below 0, their value is set to zero. For the values greater or equal than 0, their values are summed with the respective values of *b*.

Source: Presentation of the paper "ISPC: A SPMD Compiler for High-Performance CPU Programming" (PHARR; MARK, 2012).

for a vector of 512 bits (64 bytes), the position of the first byte in memory must be a multiple of 64. The best way for a compiler to detect a possible SIMD optimization is to process aligned arrays in a sequential and simple loop, that performs the same operation for each element of the array.

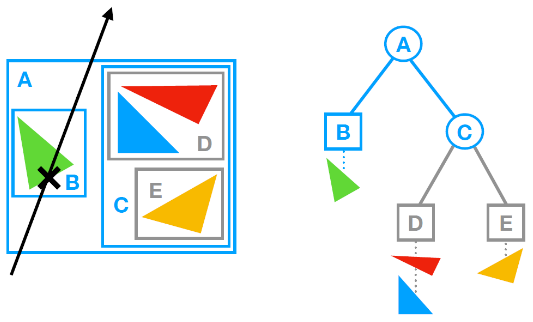
In order for programmers to enforce the use of SIMD operations in their code without depending on the compiler and without directly using assembly instructions, they either need to explicitly use intrinsics libraries provided by the compiler or to use especial programming languages that generate SIMD code, such as Intel's Implicit SIMD Program Compiler (ISPC) (PHARR; MARK, 2012). ISPC is a programming language based on C and C++ that allow programmers to write code similar to as if they were writing scalar code, but when compiled, the code runs in SIMD fashion. ISPC is also easily integrated into C and C++ programs, making inter-operation between scalar and vectorized code much easier. It also automatically handles control-flow by masking operations based on the condition value, as shown in fig. 11.

3 Path Tracing Acceleration

Since the introduction of ray tracing as a rendering method, part of the computer graphics research was done to optimize ray tracing algorithms in several ways. As stated before, ray tracing is a very computationally expensive algorithm, and if done naively, the algorithm would need to test a ray with all defined surfaces in the scene. In order to avoid this brute force approach, Rubin e Whitted (1980) introduced the Bounding Volume Hierarchy (BVH) (CLARK, 1976) for accelerating ray tracing intersection tests. The BVH starts with a root node, in which the bounding volume encompasses the whole scene. The levels below comprise bounding volumes that contain the lower levels bounds. At last, the leaf nodes of the hierarchy contain the polygons that actually compose the scene.

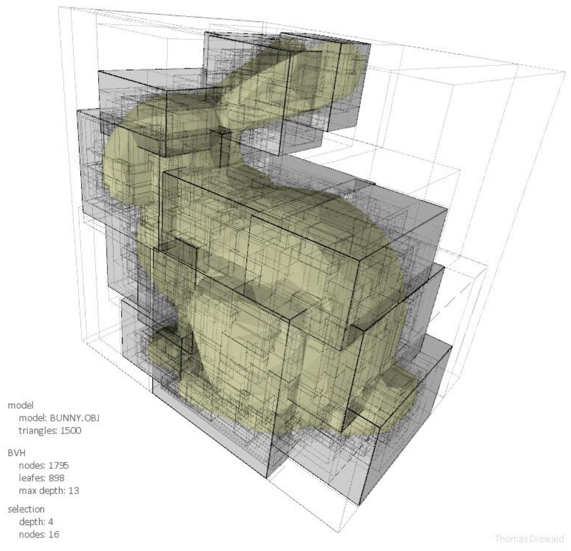
When a ray test is done, the ray is first tested for intersection with the root node volume. If the volume is not intersected, the ray does not hit any object in the scene. If the ray collides with the root volume, the test is performed with the root child nodes, repeating the process until the ray reaches the leaf nodes containing polygons. The polygons in each leaf node collided by the ray are tested for collision against the ray, and the ray-scene intersection test returns the closer hit point information along with its surface properties. A 2D and simplified illustration of the tree traversal algorithm is shown in fig. 12a.

The performance impact of testing a ray with every polygon is not much for simple scenes with few polygons, like the one in fig. 12a. But for big, complex scenes with huge amount of polygons and different types of surfaces, testing ray intersection for each primitive in the scene may turn the algorithm unfeasible. While requiring additional memory and execute more intersection tests, the BVH potentially prevents the ray from testing intersection with a huge number of polygons early on on the tree, saving a lot of computation time. The most common model of BVH is a binary tree of Axis-Aligned Bounding Boxes (AABB), but the hierarchy can have higher branching factors and different volume primitives, such as spheres. There are also other types of hierarchical acceleration structures, like Octrees and kd-trees. But to maintain the scope of this work and due



(a)

Source: A Survey on Bounding Volume Hierarchies for Ray Tracing. (MEISTER et al., 2021)



(b)

Source: Ray Tracing Introduction webpage from UC San Diego

Figure 12 – Figure (a) illustrates a ray intersecting a simple BVH with 4 triangles. Since the ray hit the root node A, it proceeds to test intersection with nodes B and C. Since only node B is intersected, the only ray-triangle intersection is done with the green triangle. Figure (b) shows an automatically constructed BVH for a bunny composed of 1500 triangles.

to the fact that the BVH is currently the de facto standard acceleration data structure (MEISTER et al., 2021), they will not be addressed.

Weghorst, Hooper e Greenberg (1984) described procedures to reduce the complexity necessary to compute a ray-traced image. First, it describes a bounding volume selection method, which selects, for each object, a bounding volume between three available shapes: a sphere, a box and a cylinder. The method to select the volume shape tries to minimize the product of the shape intersection test complexity and the void area inside the shape. At that time, no automatic BVH construction method was used, and the hierarchy of objects had to be previously defined by the user. In order to create more tight bounding volumes, the user had to take care not to join sparse objects in the same bounding volume or parent bounding volume. The last proposed technique was a visible-surface preprocess step, in which the scene is previously rendered in an *item buffer*. The *item buffer* contains the scene objects visible at each pixel, and when the ray-tracing step starts, it can begin testing for the intersection with that object.

As shown in section 2.2, a random access to the memory may cause a lot of cache misses and recycling, introducing a great overhead in CPU-intensive applications. For such applications, the best approach is to perform every necessary computation with a single cache line before requesting another line from the memory. The problem is that is virtually impossible with path tracing, since when a ray hits a surface, the algorithm generates rays in random directions, which will be tested for intersection and hit random surfaces throughout the scene, generating new rays in random directions and repeating the process until reaching a stop condition. All these random rays may end up intersecting very different areas of the scene, causing random accesses not only to geometry memory, but also to textures and other data that may define the different surfaces on the scene, causing a lot of cache misses.

The collection of rays that have no correlated direction and position is called incoherent rays. In contrast, a group of coherent rays has a high chance of intersecting the same surface or region of the scene. One of the main tasks for optimizing recursive ray tracing and global illumination algorithms, such as path tracing, is designing systems that excel in exploiting ray coherence, since incoherent rays cause incoherent memory access and drastically increase the number of cache misses. Note that the primary rays, i.e. the rays coming out of the camera, are coherent rays, since they have the same origin and points to the same region in the scene.

To tackle the ray incoherence problem, Hanrahan (1986) proposed a new ray tracing architecture that combined two previous methods, *beam-tracing* (SPEER; DEROSE; BARSKY, 1985) and *coherent ray-tracing* (SPEER; DEROSE; BARSKY, 1985). The method consists of using a software caching system that stores a ray-tree corresponding to the previous ray intersection computation and that guides the intersection tests of the following rays. The cache stores the last intersected surface and a list of potential surfaces close to the path of the last ray that might be in front of the current ray. When a ray is tested, it is first tested with that last intersected surface and the potential surfaces in the list. If the last intersected surface is hit and none of the potential blocking surfaces is hit, then there is a cache hit and the computation proceeds to the next ray. If the ray does not hit the last intersected surfaces or hit any of the potential blocking surfaces, then a cache miss occurs, the traditional ray intersection test is done and a new ray-tree is built and cached. Also, in order to perform all the computations of a region of coherent rays, the ray tracing is executed in breadth-first search order. This means that instead of following each ray path recursively, the algorithm computes a series of rays intersections in one depth and stores a list of rays generated from those intersections. After dealing with all the rays in the current region, the algorithm consumes the rays from the list and repeats the process.

Another challenge for the adoption of global illumination methods in the industry was memory requirements. Scanline algorithms previously used by the film industry processed one object at a time, accessing the data coherently and requiring only that object information to be in the DRAM memory. On the other hand, scanline algorithms did not provide realistic lighting effects such as global illumination or diffuse interreflection. In contrast, path tracing is a global illumination algorithm, and as such, needs to compute lighting contributions coming from every part of the scene. In order to compute the lighting contributions, rays with random directions need to be cast in the scene, and since there is no previous knowledge of what surface or area of the scene the ray will hit, every geometry present in the scene must be available in the system DRAM memory for computing ray intersection tests. Note that geometry data is not the only major consumer of memory. Production-quality films often use scenes that contain tens or hundreds of gigabytes of texture data, and in a path tracer, this data is also accessed incoherently. In some production scenes with a high number of highly detailed textures, the shading stage may become the slowest part of a path tracer, and thus addressing coherent access to memory is also one very important part of optimizing the path tracing algorithm.

This challenge was handled by Pharr et al. (1997), which proposed using a software defined geometry cache and texture cache, as well as reordering the rendering computations to improve coherent access to the software caches, consequently improving coherent access to the CPU cache memory. The texture cache (PEACHEY, 1990) divides the all the textures used in the scene into tiles that are stored in the disk. When a texture tile is accessed by the shading function but is not present in the texture cache, i.e., the main memory, the cache loads the tile into the main memory. If the cache is already full, the loaded tile replaces the least recently used tile in the cache. The geometry cache is based on the same idea as the texture cache, loading blocks on demand and with a least recently used replacement heuristic. To simplify the memory management and reduce the variation in the size of geometric data, the geometry cache only supports triangles as

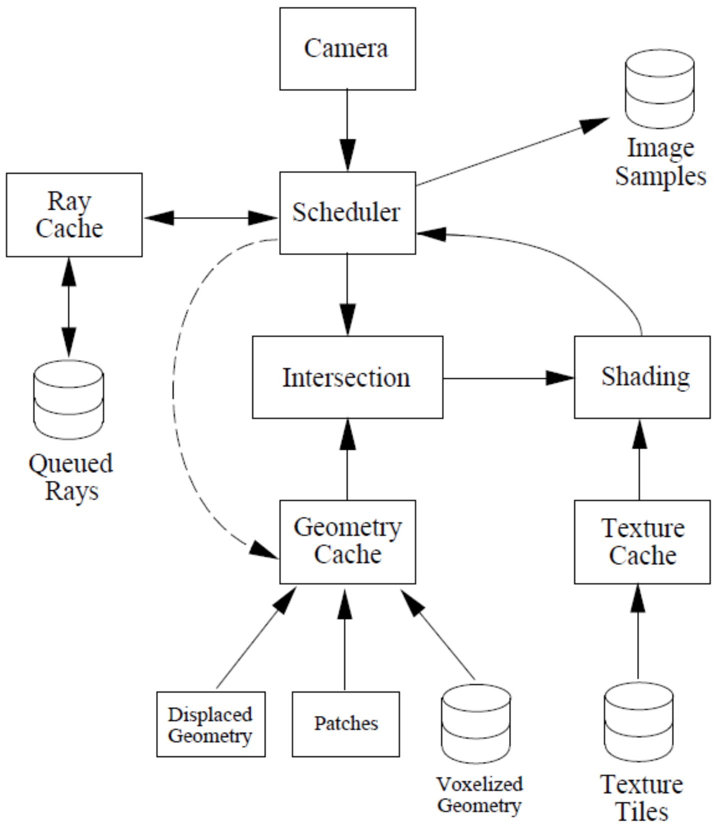


Figure 13 – A block diagram of the proposed cached path tracer. The intersection test system consumes data from the Geometry Cache, responsible for managing geometric data in and out of the main memory depending on their usage. The shading step consumes memory from the Texture Cache, which stores the recently used texture blocks inside the main memory. The scheduler is responsible for selecting the rays that will be processed, and it consumes memory from the Ray Cache.

Source: Block diagram of the system proposed in "Rendering Complex Scenes with Memory-Coherent Ray Tracing" (PHARR et al., 1997)

primitives. Other types of primitives, like subdivision surfaces, displacement mapping and procedurally generated geometry, can be tessellated into triangles when they are loaded into the cache. A block diagram summarizing the overall systems behavior and relations is shown in fig. 13.

In order for the texture and geometry caches to perform well, the shading and ray intersection computations must be reordered. The computation reordering must leverage coherent access to the caches, and must prioritize computations that use geometry and texture data that are already present in the cache. For that, they divided the scene into a set of voxels called *scheduling grid*, where each voxel has a queue of rays and information on the geometry inside it. When a scheduling voxel is selected for processing, all the queued rays are tested for intersection. If an intersection is found, the shading functions are computed, new rays are generated and appended to the queue. Otherwise, the ray is added to the queue of the next scheduling voxel it enters. The heuristic to select a schedling voxel for processing uses the combination of a benefit and a cost function. The benefit function seeks to predict the voxel contribution to the completion of the rendering by calculating the number of rays with high weight queued in the voxel. The cost function is based on the amount of geometry inside the voxel that is already inside the geometry cache. If all the geometry inside the voxel is not in the cache, the cost will be higher.

Another important effort to optimize ray tracing was the implementation of a ray tracing engine capable of running interactive applications (WALD et al., 2001). Although that paper has no focus on path tracing or physically based rendering, it was the first to propose the use of ray packets with SIMD operations to compute intersection tests, shading calculations and BVH traversal in a data-parallel manner. The implemented ray tracing engine only supports triangles as geometric primitives and makes use of few conditionals and tight inner loops, making it easier for compiler optimizations. Since their application does not focus on complex illumination techniques like diffuse reflections, the majority of the rays calculated rays are coherent, thus not using ray reordering techniques like the previously described paper. Their work shows that on a single 800MHz Pentium-III with a L1 cache level of 31 KBytes and a L2 cache level of 256 KBytes, the ray tracing algorithms are bound by memory bandwidth rather than the CPU speed, with triangle intersection tests performing at least 60% slower when data is accessed in the main memory instead of the CPU cache. As such, they organized their data contiguously in memory to make the most out of the CPU's cache lines.

Through the use of SIMD operations for the traversing, intersecting and shading computations, Wald et al. (2001) reduce memory bandwidth by requesting data only once per packet. This means that, instead of traversing the whole scene, performing intersection tests and compute shading functions with one single ray, and then repeating the process for the next ray, their system will perform those operations with a packet of 4 rays in parallel. In a perfect scenario, such an approach would reduce the memory bandwidth to a fourth, but the gains are reduced due to code divergence, i.e. rays in the same packet that hit different triangles or traverse through different BVH nodes. When code divergence happens, the computations are still done for every ray in the SIMD vector, but the result is masked to store only in the rays that followed that code path.

On the other hand, since most computations are done with coherent primary rays, code divergence is minimized, making the gains in performance still very meaningful. Their SIMD intersection calculations resulted in a speedup of 3.5 to 3.7 times the single ray implementation, while their SIMD shading computations gave a speedup of 2 to 2.5 times the single ray implementation.

After the SIMD interactive ray tracer from Wald et al. (2001), most interactive ray tracing applications took advantage of the high coherence of rays coming out of the camera, and adopted coherent ray packets and SIMD operations to decrease computational and memory bandwidth costs compared to single ray implementations. But global illumination renderers, like path tracers, were still based on single ray implementations, especially because global illumination algorithms generate a lot of incoherent rays for computing shadow and reflections, nullifying performance gains with ray packets and SIMD operations. The solution to this problem was to recover packet and SIMD coherence for secondary ray distributions through the reordering of rays into more coherent ray packets (BOULOS; WALD; BENTHIN, 2008). Although the reordering operation add some overhead, it is outweighed by the improved coherence in the calculations. Also, contrary to interactive ray tracing where calculating primary rays intersections are the main workload, global illumination renderers often spend most of their time computing secondary rays intersections. Therefore, the main focus of the proposed system is to leverage the coherence of shadow and secondary rays, which can quickly diverge in position and direction.

The implemented system uses packets with arbitrary lengths with a maximum size of 256 rays. The method does not try to automatically generate packets with coherent rays, but the coherence of the rays inside a packet is exploited in their approach. Just like in the SIMD interactive ray tracer (WALD et al., 2001), when a packet is being processed, the rays inside it are processed in parallel through SIMD instructions, where the size of the vector depends on the processor architecture. Also, using contributions from the deferred ray queuing path tracer proposed by Pharr et al. (1997), the implemented method reorder the rays inside a packet aiming to increase the SIMD utilization, i.e., decrease the amount of code divergence, and consequently the number of masked rays in a SIMD operation. One of the main changes to the BVH packet traversal is that inactive rays are temporarily removed from the packet when visiting nodes that the ray do not hit, greatly contributing to a higher SIMD utilization and the average packet traversal coherence. When the traversal returns from the node where the ray was removed from the packet, the ray is reinserted in the packet.

Note that the packet reordering operation is not done at every step, but only when the packet utilization, i.e., the number of active rays divided by the number of rays in the packet, drops below a predefined threshold. This avoids the wasteful reordering overhead caused if the packet was reordered when only a small number of rays inside it become inactive. Another optimization included in their system is to fall back to a specific, optimized, single-ray traversal code when the number of active rays inside a packet is only one. Otherwise, the ray intersection tests are done with masked SIMD instructions.

While Boulos, Wald e Benthin (2008) were focused in optimizing SIMD ray traversal and intersection tests, it also proposed the reordering of shading computations. The reordering technique join rays that hit the same material type inside the same shading packets, e.g., diffuse materials in a diffuse shading packet, glossy materials in a glossy shading packet. Then, the shading points are processed in a single linear pass for each of the shading packets, copying the shading results back to the original ray packet at the end of the process.

The results achieved with the reordering method showed better results compared to single ray tracing and incoherent SIMD ray tracing, specially in completely diffuse scenes which generate rays with little coherence in general. In the most complex scene, the reordering method achieved a speedup of $1.5X$ compared to the single ray tracing and incoherent SIMD ray tracing. Note that, although exploiting the SIMD architecture, the incoherent SIMD ray tracing had the same performance of the single ray tracing in this scene. This happens due to a high amount of incoherent rays in such scene, which cancels out the gains from the SIMD parallelism. The reordering also reduced the average number of ray-box and ray-triangle intersections tests per ray, as well as the number of BVH traversal steps per ray. Reducing the average number of BVH traversal steps per ray and ray-box intersections tests is directly related to a higher SIMD utilization. The work also demonstrated that their method benefits from increases to the processor SIMD vector register width. While doubling the SIMD vector register width does not double the performance, it still gives a significant speedup, especially in reducing the amount of BVH traversal steps per ray.

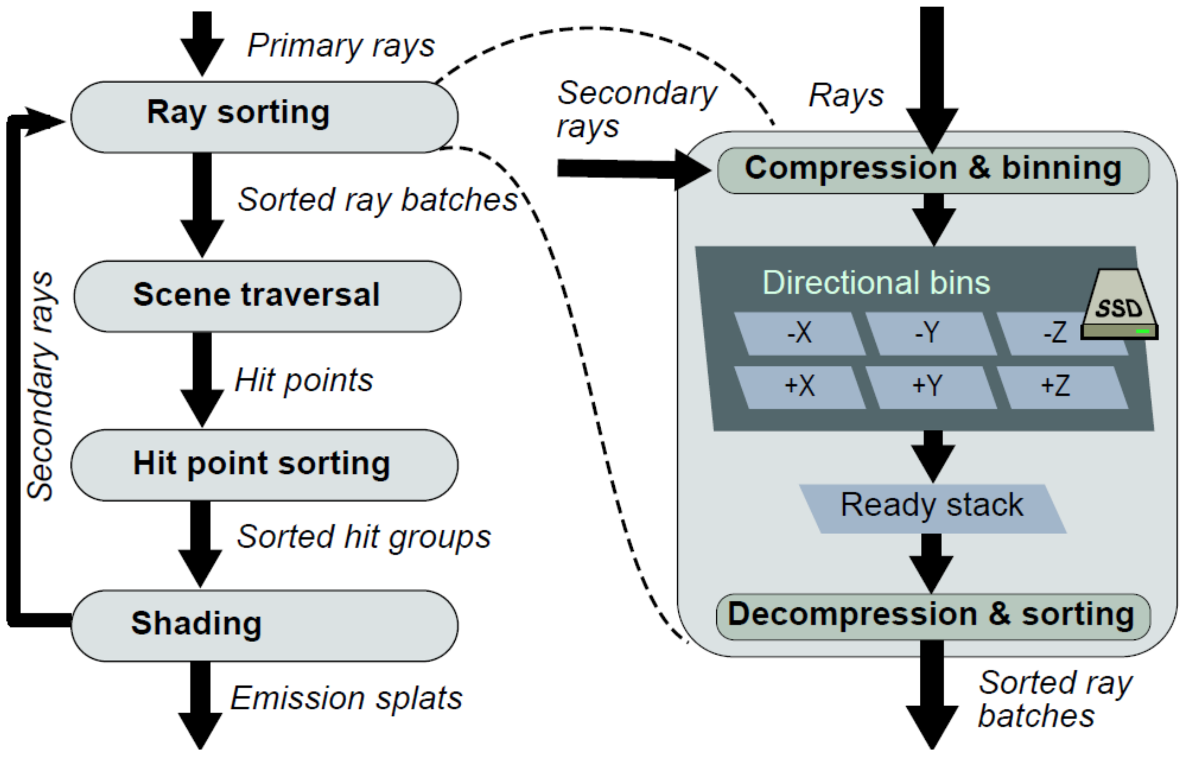


Figure 14 – On the left is shown the overall system scheme with its two sorting stages. The ray sorting step is fed with primary and secondary rays, and generate sorted ray batches. The scene traversal algorithm is performed on the sorted ray batch. The resulting hit points are then passed to the hit point sorting stage, that provides sorted hit groups for the shading step. The shading step not only store its results, but also provides new secondary rays to the ray sorting stage. On the right, the ray sorting stage is further detailed.

Source: System scheme of the paper "Sorted Deferred Shading for Production Path Tracing" (EISENACHER et al., 2013).

All the contributions from the previously described works paved the road to the development of a deferred path tracing method (EISENACHER et al., 2013). Instead of trying to extract coherence from small-sized packets, their work sorts large, potentially out-of-core ray batches of around 30-60 millions rays. As previously shown with the cached path tracer by Pharr et al. (1997), shading computations may become the slowest part of a path tracer in some scenes, especially caused by incoherent access to high definition textures. Thus, to improve shading coherence, the proposed path tracer sorts ray hits for deferred shading with coherent access to out-of-core textures, i.e., highly detailed textures that are stored in the system disk or SSD because of their memory size. The schematic for the proposed system is shown in fig. 14.

The ray batching takes place by binning rays in their major directions. There are 6 cardinal direction bins that are filled in a lock-free manner, where each bin holds a single batch of rays mapped to a file with fixed capacity stored in the system SSD. For each thread, first, they add rays to one of six small local buffers. When the local buffer is full, it atomically increases the bin size and copies the rays into the bin file. When the incremented size exceeds the bin size, the bin file is closed, the file name is added to a stack of ready batches and a new empty bin is initialized. When the system starts or when the current batch finishes the traversal and shading stages, they pop the next batch filename from the ready stack, open the file, decompress it, sort the rays and place the rays in a global active ray buffer. The system also streams the next batch in memory to reduce delay when loading the next batch.

To sort ray batches, the system performs a recursive median partitioning along the longest axis of each step. First, they make the partition based on ray origins until reaching a subset of rays with no more than 4096 rays. Then, the subsets are partitioned based on ray directions until obtaining groups of 64 rays. These groups of 64 rays form coherent ray packets that are used in the scene traversal algorithm. In order to sort hit points, they are first sorted by *mesh ID* using the parallel radix sort algorithm. Afterwards, one shading task is dispatched for each group of hit points to run in a separate thread. At the start of each shading task, they sort the hit points by *face ID*, causing the shading order to exactly match the on-disk order of the textures.

Their results show a vast improvement in the rendering of highly complex scenes, both in terms of detailed geometry and detailed texture maps. All the tests were rendered using 12 threads on a 12-core Xeon 5675 3.46GHz system with 48GB of DRAM while running the Windows 7 operating system. As a comparison, their simplest scene, a house interior with a single texture layer, takes around 900 minutes to render without any kind of sorting method. When sorting only rays, the same scene takes around 200 minutes, granting a great improvement on traversal performance. When sorting only the hit points, the improvement on performance is even higher, taking around 80 minutes to render the same scene. When sorting both rays and hit points, the best performance is achieved, reaching a rendering times of around 40-50 minutes.

The development of many ray tracing acceleration techniques, some of which were covered in this work, provided great results and enabled the use of path tracing as a viable option for production rendering. But *Intel Corporation* researchers noted that most professional rendering applications still did not used the most efficient and up-to-date combination of intersection algorithms, acceleration data structures, and parallel computing techniques. Also, if you consider that different processors supports different Instruction Set Architectures (ISA), optimizing for each specific architecture is not an easy task even for highly experienced programmers. These observations lead the researchers to develop a ray tracing kernel framework aimed at efficient parallel ray tracing on the different x86 computer architectures (WALD et al., 2014).

The Embree ray tracing framework provides a clean and easy-to-use API that hides the internal data structures and procedures from the user, facilitating its integration with existing rendering softwares. It also supports a variety of commonly used ray tracing kernels manually optimized for each different ISA supported vector width, for different workloads like coherent or incoherent ray distributions, for static versus dynamic scenes, and for user-defined options like selecting between maximal performance versus minimal memory usage. Those different combination of kernels are selected at run-time by Embree according to the expected best performance for the user-defined options and the processor architecture used.

All these features were designed with the primary focus of being used on professional rendering applications. To achieve that, Embree is designed to provide highly optimized ray tracing operations only, not dealing with any other part of the rendering pipeline. Focusing on creating an entire rendering system would limit the use cases of the framework and consequently its adoption by pre-existing professional rendering applications.

To leverage the maximum performance of modern processors, Embree makes extensive use of SIMD operations. When a single-ray tracing operation is called by the user, the intersection is calculated in parallel using SIMD operation for 4 or 8 triangles, depending on the ISA vector width. When the intersection is called on a ray packet, Embree computes the intersection of each ray with a single triangle in parallel using SIMD operations. Its BVH uses a branch factor of 4, which performs well both in packet or single ray traversal, where the single ray traversal tests one ray with all the nodes in parallel, and the ray packet test multiple rays on each node in parallel. There are two variations of the BVH: the static BVH, and the dynamic BVH with full support to motion blur. The BVH construction kernels come with two construction algorithms: Binned Surface Area Heuristic (SAH) with optional Spatial Split, and a very fast Morton-Code BVH construction. Embree also supports multi-level BVH, where the system builds a BVH of entire objects, and then a second level BVH for each object containing its geometric primitives.

The framework API supports several geometry types, including triangle meshes, subdivision surfaces, instances, and user-specified data types. Only the geometric information of the primitives are required by the API, leaving all the specific surface data that might be used in shading, for example, to the application. User-defined geometries must provide callback functions to generate bounding boxes and to calculate ray-primitive intersection.

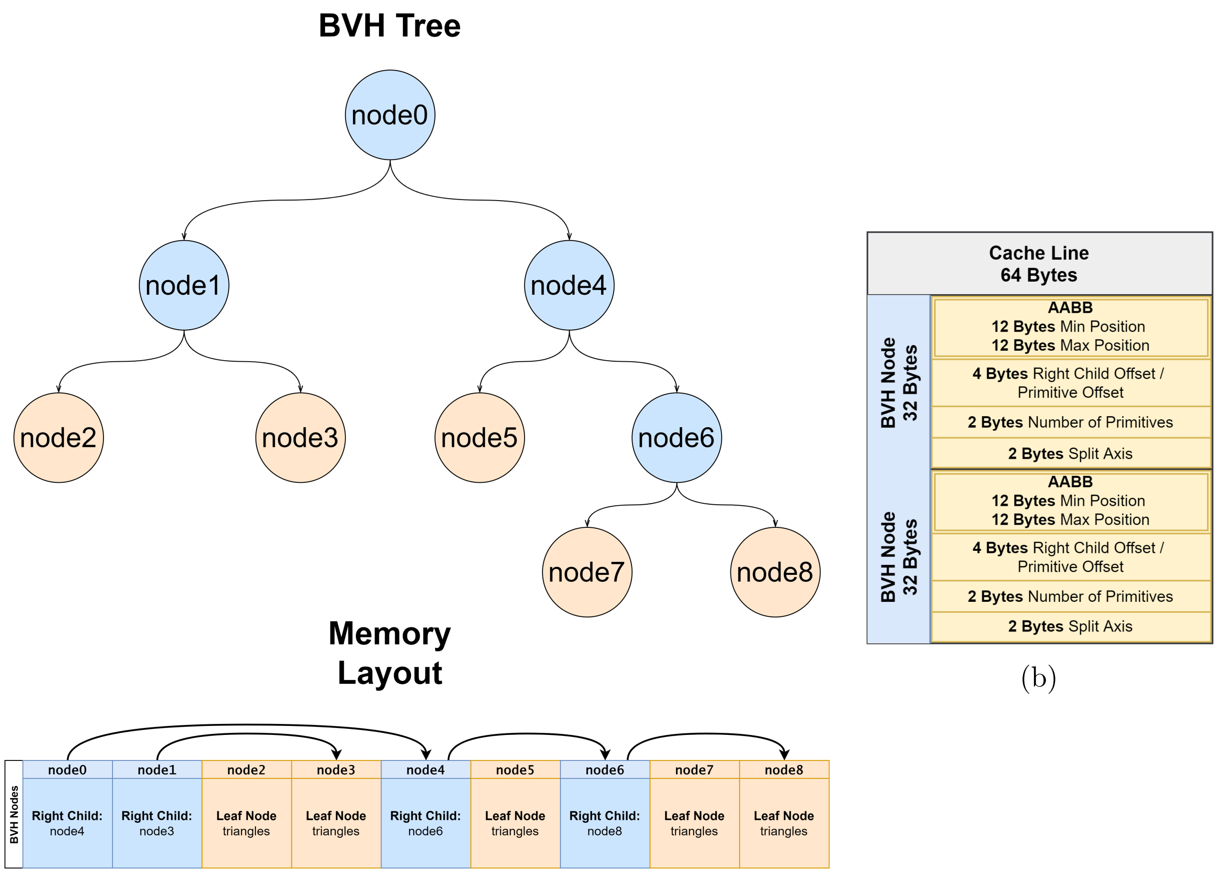
Although an application aiming for the best achievable performance would have to move from scalar code to fully parallel code in every part of the system, like Embree does, the framework does not required an application to be fully parallel to grant extremely good results. Its ease of integration in existing systems also allows programmers to slowly transition their rendering software code from scalar to parallel. Nowadays, Embree is employed by a variety of professional rendering applications, including the fully vectorized production renderer used at *DreamWorks Animation*, *MoonRay* (LEE et al., 2017).

4 Path Tracing Implementation

This chapter describes the development process of a uni-directional path tracer with next event estimation. Most of the theory, ideas, and software architecture behind the development of this project comes from the book *"Physically Based Rendering: From Theory to Implementation"* (PHARR; JAKOB; HUMPHREYS, 2016). The book served as a practical guide for the development of the most recent path tracing systems used in film production. Some examples of film production renderers that used the book as a base architecture guide are: *DreamWorks Animation MoonRay* (LEE et al., 2017), *Pixar's RenderMan* (CHRISTENSEN et al., 2018), and *Solid Angle's Arnold* (GEORGIEV et al., 2018).

Because of the limits to the scope and time available for the development of the project, many compromises had to be made for the project. The implemented path tracer does not use any of the coherency-exploiting techniques described in chapter 3. Instead of using any type of ray queue, this path tracer follows every step of a path until its end before computing any ray from another path. The project also could not extensively use SIMD operations for computing multiple intersection tests in parallel. On the other hand, the linear algebra library used in the project, Eigen (GUENNEBAUD; JACOB et al., 2010), uses SIMD for arithmetic operations on vectors, e.g., the sum of two vectors with 4 floating points is made with only one instruction.

However, the project does make use of the Bounding Volume Hierarchy (BVH) for ray-scene traversal acceleration. The construction algorithm used to build the BVH is the Surface Area Heuristic (SAH) (WALD, 2007). It is a recursive algorithm that receives a node with a given bounding box and a set of geometric primitives, and decide whether to split it and recursively call the algorithm on the children nodes, or to create a leaf node containing the geometric primitives. The construction starts with the root node, containing a volume that encompasses all the scene primitives. Then, recursively split nodes, creating sub-trees until all the geometric primitives are inside some leaf node. The idea behind the SAH construction algorithm for a n-wide BVH is to search the division of the node volume that generates n volumes with the least summed surface area. It is a greedy algorithm with the aim of minimizing the total cost of traversing the tree and


BVH Tree
node0
node1
node4
node2
node3
node5
node6
node7
node8
Memory Layout
BVH Nodes
node0 Right Child: node4
node1 Right Child: node3
node2 Leaf Node triangles
node3 Leaf Node triangles
node4 Right Child: node6
node5 Leaf Node triangles
node6 Right Child: node8
node7 Leaf Node triangles
node8 Leaf Node triangles
Cache Line 64 Bytes
BVH Node 32 Bytes
AABB 12 Bytes Min Position 12 Bytes Max Position
4 Bytes Right Child Offset / Primitive Offset
2 Bytes Number of Primitives
2 Bytes Split Axis
BVH Node 32 Bytes
AABB 12 Bytes Min Position 12 Bytes Max Position
4 Bytes Right Child Offset / Primitive Offset
2 Bytes Number of Primitives
2 Bytes Split Axis

(b)

(a)

Figure 15 – Figure (a) illustrates a BVH Node tree at the top, and the memory layout of the nodes at the bottom. Note that intermediate BVH Nodes only need to store the position of its right child node, since the left child is always the next node in the memory. Figure (b) shows the required memory and the data inside a BVH Node. A single BVH Node has 32 bytes. The first node variable is its Axis-Aligned Bounding Box. The following variable is either the position of the right child node, if it is an intermediate node, or the position of the first geometric primitive, if its a leaf node. The third variable represents the number of primitives inside the node, and is the variable that indicates if it is an intermediate or a leaf node. If it is an intermediate node, then the number of primitives is 0.

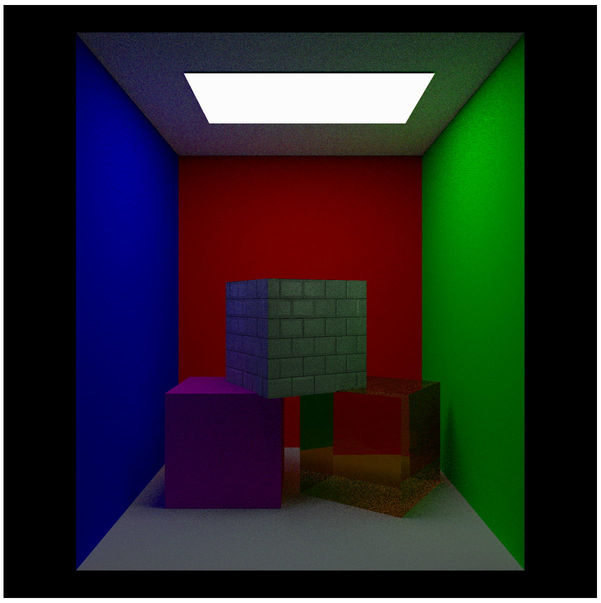
testing for ray intersections. If the algorithm calculates that the total cost of traversing the node is higher than traversing the geometric primitives inside it, it decides to create a leaf node instead of splitting that node. As the rest of the system, the algorithm for computing ray-box intersection (WILLIAMS et al., 2005) only tests a single ray with a single box, but takes advantage SIMD operations to make arithmetic vector operations.

In order to improve coherent memory access when accessing the BVH Node data structure during ray-scene traversal, the nodes from the hierarchy are stored in a contiguous array with a cache-friendly layout. When a node is created in the BVH construction algorithm, the new node is inserted to the end of the BVH Node array. The root node is the first created, and thus, the first node in the array. If the SAH algorithm decides to split the created node, making an intermediate node, the left child node is inserted right after its parent node, and the construction algorithm is recursively called on the left child. The right child node will come right after the left child node sub-tree. If the left child node is a leaf node, then the right child node comes right after the left child node. An example of such memory layout for a BVH tree is shown in fig. 15.

When executing the ray-scene traversal algorithm, this memory layout ensures that the memory will always be accessed in an almost sequential pattern. The left child node is always at the next position relatively to its parent, reducing the chances of a cache miss when accessing it after traversing the parent node. While the right child node position might be distant from its parent position, it is the closer node after the left child node sub-tree. Thus, although some jumps in memory access might be performed, the traversal algorithm never goes back and forth in the node array. This avoids cache misses by conflict, i.e., a cache line that was previously accessed, then recycled to give space to another line, and accessed again later.

Support for other types of geometric primitives can be easily integrated in the system using object oriented inheritance. The current implementation of the path tracer support triangle primitives and triangle meshes. Since most 3D scenes found online or exported from 3D modeling applications comprises only triangles meshes, the decision to only support triangles did not impose any severe limitation to testing the application. The algorithm used to compute ray-triangle intersections is the Möller–Trumbore algorithm (MöLLER; TRUMBORE, 1997). The application also grants support to color textures, automatically loading the necessary textures when reading the scene file from disk. On the other hand, normal textures are not yet supported due to the errors that can be introduced when calculating BSDFs with a fake normal (SCHüSSLER et al., 2017).

Despite not using data-level parallelism to compute multiple ray intersection tests or multiple shading calculations, the path tracer leverages instruction-level parallelism through the use of threads. The code uses the compiler-provided OpenMP (DAGUM; MENON, 1998) implementation for the multi-threading logic. The number of threads created corresponds to the number of cores in the processor. Each thread is initially assigned to a work group of 16 pixels, computing all the samples for all the pixels in the


(a)

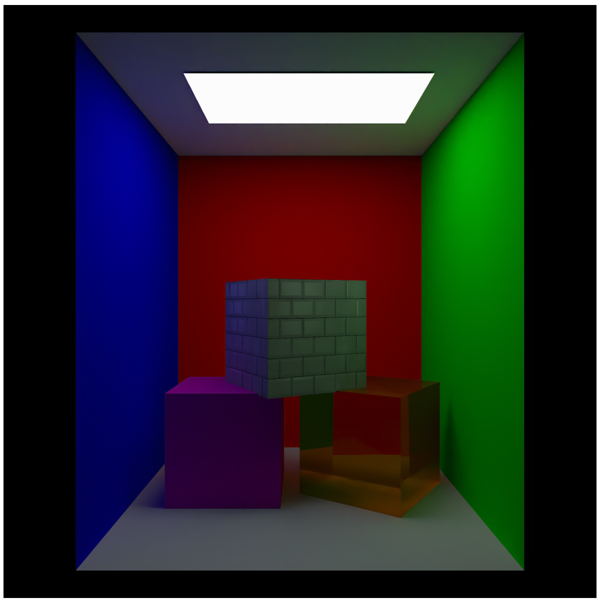

(b)

Figure 16 – Figure (a) shows a custom-made scene, inspired in the famous Cornell Box scene, rendered with only 8 samples per pixel. There is a high amount of noise in the image, especially next to the bottom-right cube with the glass BSDF. Figure (b) shows the same image after being denoised with OIDN library. For input images with very high amount of noise, the denoised output of OIDN may not look realistic. But for images with an acceptable degree of noise, the denoised output is close to what the path tracing algorithm would output for a huge amount of samples per pixel.

group. After a thread finishes its work group, it moves to the next, until the image is fully computed and all the threads ends execution. The path tracing algorithm mostly reads data from memory, where the only memory write operation is to store the calculated pixel color. This characteristic avoids the usage of slow thread synchronization techniques, like mutexes, and enables all the threads to run with almost no synchronization interruptions.

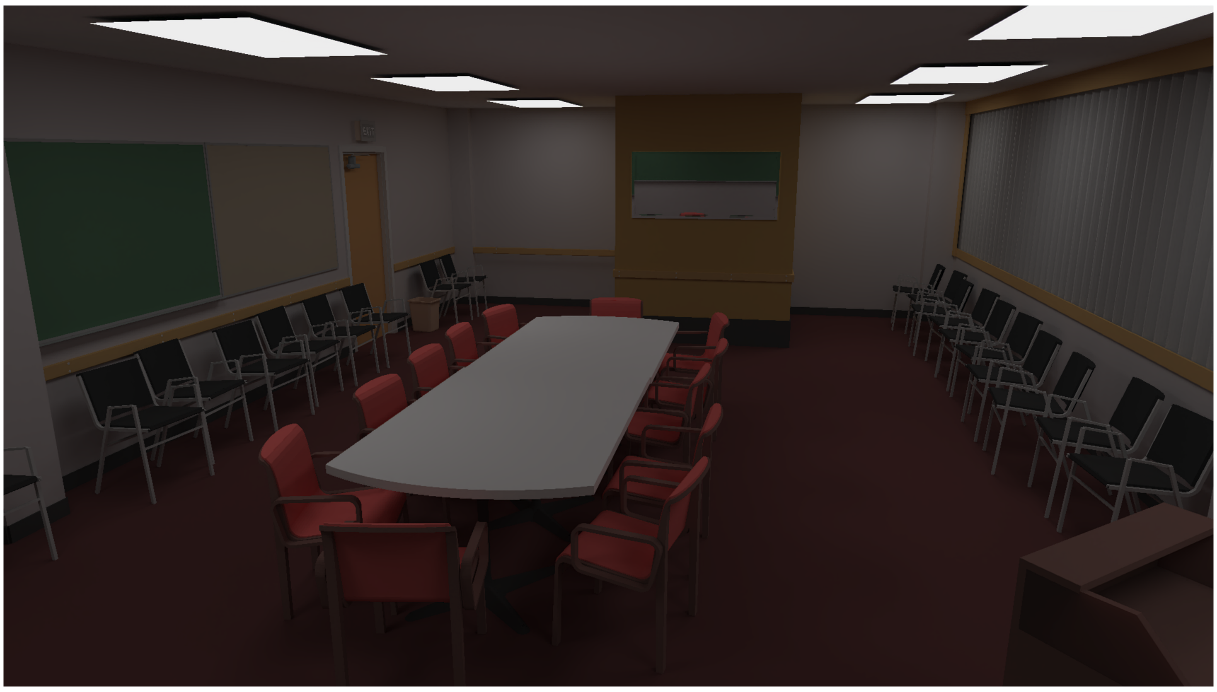
Another feature included in the proposed path tracer is the use of the *Intel's Open Image Denoise* (OIDN) open-source library (INTEL®, 2019). The OIDN C++ library provides a set of denoising filters for images rendered with ray tracing techniques, like path tracing. The use of denoising techniques in rendered images reduces the required amount of samples per pixel to produce a high-quality, noise-free image, as shown in fig. 16. Denoising techniques also had an important role in enabling the usage of path tracing on production movies, since instead of waiting for the path tracing algorithm to converge into a noise-free image by using a huge amount of samples per pixel, it can render a relatively noisy image with fewer samples per pixel, and denoise that image later.

Scenes	**Time**
Sponza	56:30
Conference Room	39:25

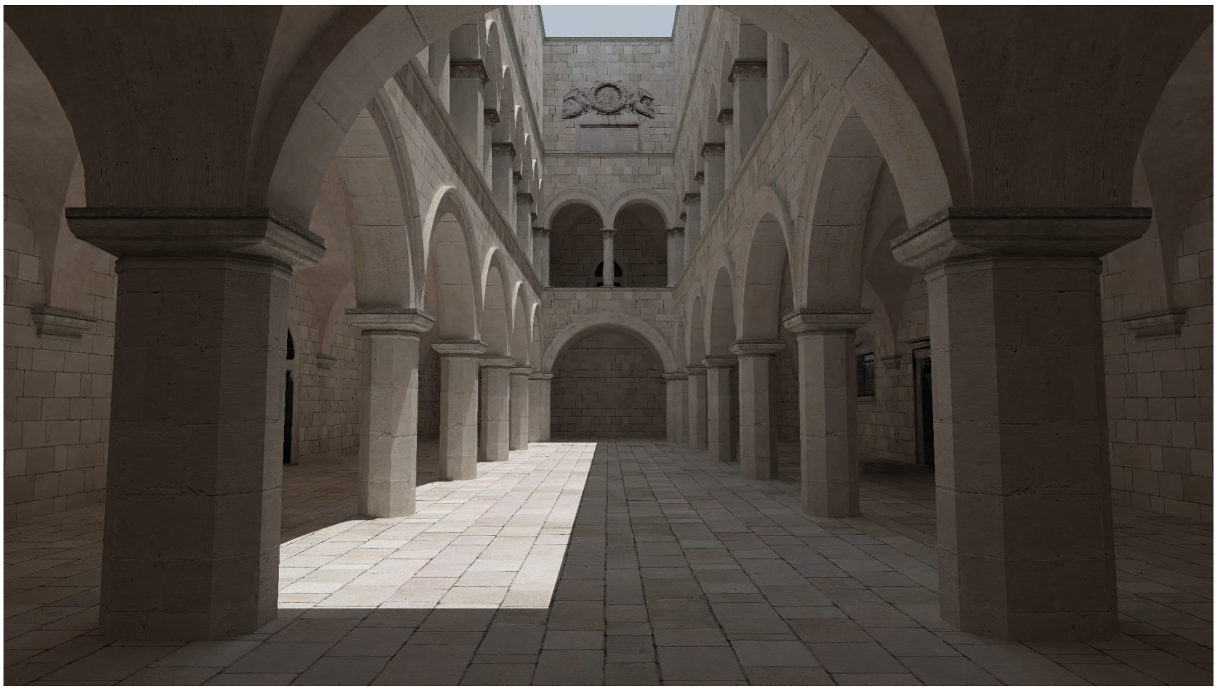
Table 1 – Shows the time taken (min:secs) to render each scene in Full HD (1920 x 1080) with 1024 samples per pixel.

Although the implemented path tracer performance is not competitive with professional renderers while rendering relatively complex scenes in high definition with a high number of samples per pixel, it achieves comparable times rendering simpler scenes in lower resolution. The major factor for improvement of the ray tracing performance was the use of the BVH for ray-scene traversal. Not only it drastically replaces a large amount of costly ray-triangle intersection tests for cheaper ray-box intersections, it also reduces the number of access to geometry memory. Of course, this comes at the cost of more memory space for storing BVH nodes and memory latency for accessing them, but the performance gains outweigh the additional costs. All the source-code of the described project is available in the following GitHub repository: *https://github.com/brayner1/Pathtracer*.

To demonstrate the project, I executed renderings generated with the proposed path tracer, shown in fig. 17. The images were rendered using a computer with 16 GBytes of DRAM with a memory frequency of 3600MHz and an AMD Ryzen 7 3700x CPU, with 3.6GHz and 16 threads. The processor cache memory is a 8-way cache and the hierarchy contains three levels with cache lines of 64 Bytes. The L1 cache level has 512 KBytes, the L2 cache level has 4 MBytes, and the L3 cache level has 32 MBytes. The program was compiled and executed in the Windows 10 operating system using the *MSVC v19.29.30137* C++ compiler. A profiler was executed along the rendering of the *Conference Room* scene (fig. 17a), collecting information about the program execution. The profiler provided a histogram representing the thread usage during the rendering of the conference scene (fig. 17a), shown in fig. 18. The histogram shows that most of the time, only half of the 16 available threads were in use concurrently. This may be caused not only due to thread barriers when writing pixels in the frame buffer, which is the only thread synchronization used during the rendering execution, but to the Operating System scheduling system that manages the threads between each system executing program. The profiler also provided information sampled from each function and instructions in the programe. The table 2 shows the most demanding functions of the program and its main bottlenecks.



(a) Conference Room



(b) Sponza

Figure 17 – The denoised path tracing result of the *Conference Room* and *Sponza* scenes, respectively. Both were rendered in Full HD (1920 x 1080) with 1024 samples per pixel. *Conference Room* scene is provided by Anat Grynberg and Greg Ward. *Sponza* scene is provided by Marko Dabrovic.

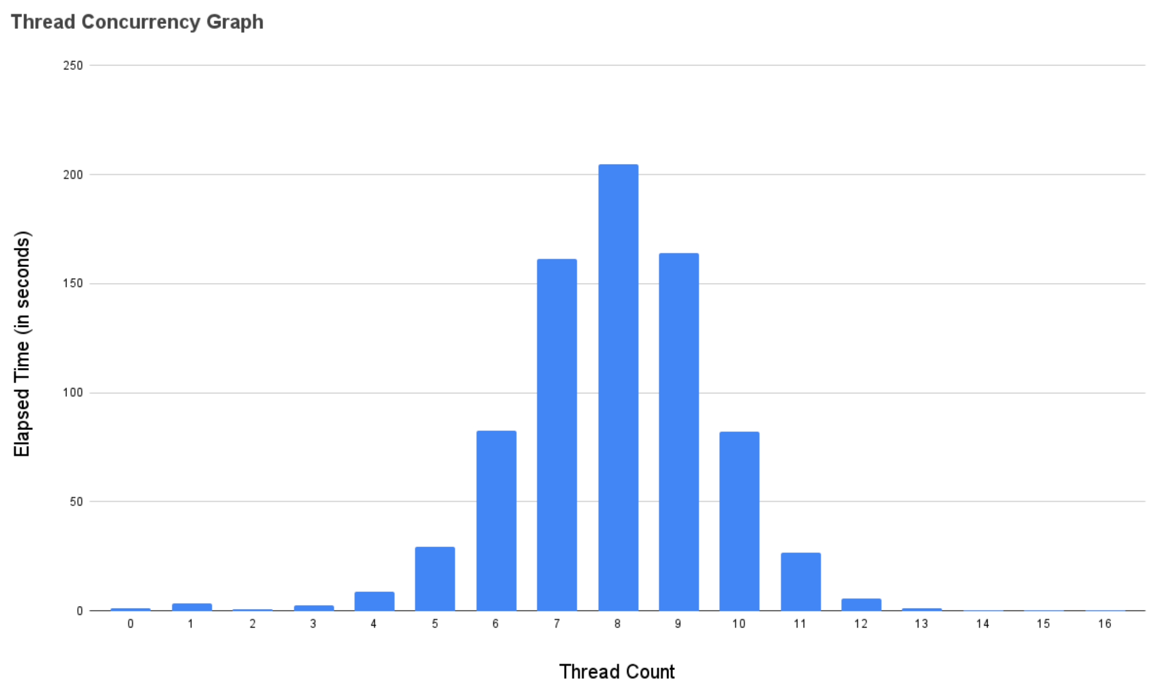


Figure 18 – The histogram shows the processing time spent with a given number of threads active at the same time during the rendering process.

Functions	Sampled Instructions	Data Cache Misses	Data Cache Hits	Misaligned Data Access	Local DRAM Hit
BVHTree::Intersect (Path Rays)	33,38%	26,03%	33,70%	74,18%	29,48%
BVHTree::Intersect (Shadow Rays)	29,44%	24,10%	25,75%	1,85%	19,27%
Mesh::PrimitiveHitByRay (Path Rays)	11,39%	10,05%	13,98%	6,72%	25,34%
Mesh::PrimitiveHitByRay (Shadow Rays)	8,39%	6,79%	7,71%	2,84%	8,40%
Scene::PathTrace	0,86%	3,22%	1,17%	4,02%	0,59%

Table 2 – The table shows the five functions that the program spent more time when rendering the *Conference Room* scene. Each column represents an event sampled by the profiler, and the values indicates the percentage of these events in each function. The first column indicates instructions sampled by the profiler during the rendering. The second column represents data cache misses during the rendering. The third column is for Data Cache Hits. The fourth column indicates the relative amount of misaligned data, i.e., a single data that lies between two cache lines. And the fifth column indicates the relative amount of memory accesses that had to reach the DRAM.

The high amount of Data cache misses and local DRAM hits in the intersection tests indicates that the main bottleneck of the program is a high randomness in the data access. In fact, the profiler showed that inside the BVH intersection function, the most expensive instruction was loading from memory the bounding box from the node, while in the primitive hit function, the most expensive instruction was loading from memory the triangle vertices. This shows that, even though the BVH acceleration structure make a major improvement on the ray tracing performance, the usage of coherent ray tracing is fundamental for coherent memory access and consistent gains in performance.

5 Conclusion

This work initially presented a review of the techniques and concepts used to generate photorealistic images with path tracing. To achieve such high quality renderings, the CPU has to deal with a lot of ray tracing operations during scene traversal, as well as many expensive shading computations, usually accessing various high resolution texture maps. This work explained how executing all these computations on the CPU in an unordered, incoherent manner can lead to a bad cache utilization, yielding a low performance.

To handle the bad ray tracing performance caused by incoherent access to memory during ray traversal and shading computations, a set of some of the most important publications was presented in this work. For optimizing ray tracing operations duration ray-scene traversal, the main objective of those works is to extract and take advantage of ray coherence. Coherent rays usually pass through the same areas of the scene, and sometimes hit the same surface. This characteristic is fundamental to improve the cache utilization, since the geometries brought to the cache memory for a previous ray intersection may be used for the next coherent ray. For shading computations, most memory access operations are texture readings. Since complex scenes often handle gigabytes of texture data, sorting the shading computations by textures and material gives a significant improvement in the cache use.

Although chapter 3 provided a consistent review of some of the most important works towards achieving high ray coherence and high SIMD utilization in ray tracing algorithms, there are plenty of very important papers and theses covering these topics that could not be assessed due to the limited time for developing this work. Besides that, it was not possible to do a wider and deeper overview of the Spatial Acceleration Structures techniques, such as the many variants of BVH (MEISTER et al., 2021). Another topic for future works is to analyze and review the techniques and algorithms used in professional production renderers used in the movie industry. The movie industry switch of rasterization-based techniques to path tracing was accompanied by an enormous effort of the graphics engineers of those companies to implement robust and production ready path tracing systems. There is a range of papers describing the development of some of such renderers, like DreamWorks's *MoonRay* (LEE et al., 2017), Pixar's *RenderMan* (CHRISTENSEN et al., 2018) and Solid Angle's *Arnold* (GEORGIEV et al., 2018).

Finally, this work presents the implementation description of a path tracer. The development process of this path tracer was very important for me to learn, in practice, many of the studied subjects for photorealistic rendering. It was also a great opportunity to leverage my knowledge in the C++ programming language, which is currently the most used programming language for high-performance applications. For comparison, the results were accurately matching the renderings of the same scene conditions made within the open source production renderer *Cycles* (BLENDER, 2011). Due to time constraints in the development of this work, the path tracer only supports the perfectly diffuse Lambertian BSDF, the perfectly reflective Specular BSDF, and the Glass BSDF, which reflects or refracts light based on the Fresnel factor. Currently, adding support to more advanced techniques, most of which uses microfacets distributions, is the main focus in the path tracer development.

Along with many of the coherent ray tracing optimizations discussed in chapter 3, there is a lot of room for improvement and optimizations in memory management and data organization. Using SIMD operations, either through the libraries provided by C++ compilers or through the use of standalone SIMD compilers like ISPC (PHARR; MARK, 2012), would also contribute to use the full capacity of current processors. But as shown in chapter 3, to effectively use SIMD operations, the path tracer needs to provide coherent rays for intersection tests. Another major point for improving the time performance would be including support for Embree kernels (WALD et al., 2014) to handle ray tracing operations in the path tracer. Handling the ray tracing operations to an advanced, high-performance library would make much easier to focus on the development for advanced shading techniques, and to work on developing a coherent shading scheduling technique, as the one described in chapter 3 (EISENACHER et al., 2013).

REFERENCES

AKENINE-MLLER, T.; HAINES, E.; HOFFMAN, N. *Real-Time Rendering, Fourth Edition*. 4th. ed. USA: A. K. Peters, Ltd., 2018. ISBN 0134997832.

APPEL, A. Some techniques for shading machine renderings of solids. In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. New York, NY, USA: Association for Computing Machinery, 1968. (AFIPS '68 (Spring)), p. 37–45. ISBN 9781450378970. Disponível em: <https://doi.org/10.1145/1468075.1468082>.

BENEDICT, E. *19 Scientific and Technical Achievements To Be Honored With Academy Awards®*. 2014. Disponível em: <https://www.oscars.org/news/19-scientific-and-technical-achievements-be-honored-academy-awardsr>.

BLENDER. *Cycles Open Source Production Rendering*. 2011. Disponível em: <https://www.cycles-renderer.org/>.

BLINN, J. F. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, Association for Computing Machinery, New York, NY, USA, v. 11, n. 2, p. 192–198, jul 1977. ISSN 0097-8930. Disponível em: <https://doi.org/10.1145/965141.563893>.

BOULOS, S.; WALD, I.; BENTHIN, C. Adaptive ray packet reordering. In: *2008 IEEE Symposium on Interactive Ray Tracing*. [S.l.: s.n.], 2008. p. 131–138.

CHRISTENSEN, P. et al. Renderman: An advanced path-tracing architecture for movie rendering. *ACM Trans. Graph.*, Association for Computing Machinery, New York, NY, USA, v. 37, n. 3, aug 2018. ISSN 0730-0301. Disponível em: <https://doi.org/10.1145/3182162>.

CHRISTENSEN, P. H.; JAROSZ, W. The path to path-traced movies. *Foundations and Trends in Computer Graphics and Vision*, v. 10, n. 2, p. 103–175, out. 2016. ISSN 1572-2740.

CLARK, J. H. Hierarchical geometric models for visible surface algorithms. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 19, n. 10, p. 547–554, oct 1976. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/360349.360354>.

COOK, R. L.; TORRANCE, K. E. A reflectance model for computer graphics. *ACM Trans. Graph.*, Association for Computing Machinery, New York, NY, USA, v. 1, n. 1, p. 7–24, jan 1982. ISSN 0730-0301. Disponível em: <https://doi.org/10.1145/357290.357293>.

DAGUM, L.; MENON, R. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, IEEE, v. 5, n. 1, p. 46–55, 1998.

EISENACHER, C. et al. Sorted deferred shading for production path tracing. In: *Proceedings of the Eurographics Symposium on Rendering*. Goslar, DEU:

Eurographics Association, 2013. (EGSR ’13), p. 125–132. Disponível em: <https://doi.org/10.1111/cgf.12158>.

GEORGIEV, I. et al. Arnold: A brute-force production path tracer. *ACM Trans. Graph.*, Association for Computing Machinery, New York, NY, USA, v. 37, n. 3, aug 2018. ISSN 0730-0301. Disponível em: <https://doi.org/10.1145/3182160>.

GUENNEBAUD, G.; JACOB, B. et al. *Eigen v3*. 2010. Http://eigen.tuxfamily.org.

HANRAHAN, P. Using caching and breadth-first search to speed up ray-tracing. In: *Proceedings on Graphics Interface ’86/Vision Interface ’86*. CAN: Canadian Information Processing Society, 1986. p. 56–61.

HENNESSY, J. L.; PATTERSON, D. A. *Computer architecture: a quantitative approach*. [S.l.]: Elsevier, 2011.

INTEL, C. *Over 50 years of Moore’s law*. 2019. Disponível em: <https://www.intel.com/content/www/us/en/silicon-innovations/moores-law-technology.html>.

INTEL®. Intel® Corporation, 2019. Disponível em: <https://www.openimagedenoise.org/index.html>.

KAJIYA, J. T. The rendering equation. In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: Association for Computing Machinery, 1986. (SIGGRAPH ’86), p. 143–150. ISBN 0897911962. Disponível em: <https://doi.org/10.1145/15922.15902>.

KLOEK, T.; DIJK, H. K. van. Bayesian estimates of equation system parameters: An application of integration by monte carlo. *Econometrica*, [Wiley, Econometric Society], v. 46, n. 1, p. 1–19, 1978. ISSN 00129682, 14680262. Disponível em: <http://www.jstor.org/stable/1913641>.

LEE, M. et al. Vectorized production path tracing. In: *Proceedings of High Performance Graphics*. New York, NY, USA: Association for Computing Machinery, 2017. (HPG ’17). ISBN 9781450351010. Disponível em: <https://doi.org/10.1145/3105762.3105768>.

MEISTER, D. et al. A survey on bounding volume hierarchies for ray tracing. *Computer Graphics Forum*, v. 40, 2021.

MöLLER, T.; TRUMBORE, B. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools*, Taylor & Francis, v. 2, n. 1, p. 21–28, 1997. Disponível em: <https://doi.org/10.1080/10867651.1997.10487468>.

NYSTROM, R. *Game Programming Patterns*. Genever | Benning, 2014. ISBN 9780990582915. Disponível em: <https://books.google.com.br/books?id=9fIwBQAAQBAJ>.

PEACHEY, D. *Texture On Demand*. 1990.

PHARR, M.; JAKOB, W.; HUMPHREYS, G. *Physically Based Rendering: From Theory to Implementation*. 3rd. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016. ISBN 0128006455.

PHARR, M. et al. Rendering complex scenes with memory-coherent ray tracing. In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. USA: ACM Press/Addison-Wesley Publishing Co., 1997. (SIGGRAPH '97), p. 101–108. ISBN 0897918967. Disponível em: <https://doi.org/10.1145/258734.258791>.

PHARR, M.; MARK, W. R. ispc: A spmd compiler for high-performance cpu programming. In: *2012 Innovative Parallel Computing (InPar)*. [S.l.: s.n.], 2012. p. 1–13.

RUBIN, S. M.; WHITTED, T. A 3-dimensional representation for fast rendering of complex scenes. In: *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques*. New York, NY, USA: Association for Computing Machinery, 1980. (SIGGRAPH '80), p. 110–116. ISBN 0897910214. Disponível em: <https://doi.org/10.1145/800250.807479>.

SCHüSSLER, V. et al. Microfacet-based normal mapping for robust monte carlo path tracing. *ACM Trans. Graph.*, Association for Computing Machinery, New York, NY, USA, v. 36, n. 6, nov 2017. ISSN 0730-0301. Disponível em: <https://doi.org/10.1145/3130800.3130806>.

SPEER, L. R.; DEROSE, T. D.; BARSKY, B. A. A theoretical and empirical analysis of coherent ray-tracing. In: *Proceedings of Graphics Interface '85 on Computer-Generated Images: The State of the Art*. Berlin, Heidelberg: Springer-Verlag, 1985. p. 11–25. ISBN 4431700102.

TORRANCE, K. E.; SPARROW, E. M. Theory for off-specular reflection from roughened surfaces. In: . [S.l.: s.n.], 1967.

WALD, I. On fast construction of sah-based bounding volume hierarchies. In: *2007 IEEE Symposium on Interactive Ray Tracing*. [S.l.: s.n.], 2007. p. 33–40.

WALD, I. et al. Interactive rendering with coherent ray tracing. *Computer Graphics Forum*, v. 20, 2001.

WALD, I. et al. Embree: A kernel framework for efficient cpu ray tracing. *ACM Trans. Graph.*, Association for Computing Machinery, New York, NY, USA, v. 33, n. 4, jul 2014. ISSN 0730-0301. Disponível em: <https://doi.org/10.1145/2601097.2601199>.

WEGHORST, H.; HOOPER, G.; GREENBERG, D. P. Improved computational methods for ray tracing. *ACM Trans. Graph.*, Association for Computing Machinery, New York, NY, USA, v. 3, n. 1, p. 52–69, jan 1984. ISSN 0730-0301. Disponível em: <https://doi.org/10.1145/357332.357335>.

WHITTED, T. An improved illumination model for shaded display. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 23, n. 6, p. 343–349, jun 1980. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/358876.358882>.

WILLIAMS, A. et al. An efficient and robust ray-box intersection algorithm. In: *ACM SIGGRAPH 2005 Courses*. New York, NY, USA: Association for Computing Machinery, 2005. (SIGGRAPH '05), p. 9–es. ISBN 9781450378338. Disponível em: <https://doi.org/10.1145/1198555.1198748>.

WILSON, J. *Physically-Based Rendering, And You Can Too!* 2020. Disponível em: <https://marmoset.co/posts/physically-based-rendering-and-you-can-too>.