



UNIVERSIDAD DE GRANADA

TRABAJO FIN DE MÁSTER

Ray-Tracing on GPU: Light Sampling Algorithms

Autor

Dmitry Ivanov

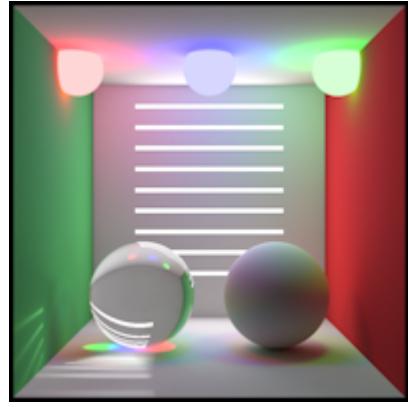
Director

Carlos Ureña Almagro



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE
TELECOMUNICACIÓN

Granada, 20 de septiembre de 2024



Ray-Tracing on GPU: Light Sampling Algorithms

Autor
Dmitry Ivanov

Director
Carlos Ureña Almagro

Ray-Tracing on GPU: Light Sampling Algorithms

Dmitry Ivanov

Palabras clave: ray tracing, path tracing, gpu, realistic rendering, light sampling

Resumen

El trazado de rayos es una tecnología de visualización muy madura que ha evolucionado a lo largo de los años y continúa incorporando nuevos instrumentos que cambian las reglas del juego cada año. El trazado de rayos permite la simulación de comportamientos de la luz y crea imágenes muy realistas de entornos preestablecidos. Durante mucho tiempo, el mayor obstáculo de la tecnología era el rendimiento en tiempo, que limitaba significativamente el alcance de la aplicación y no permitía aplicaciones en tiempo real. Para generar una imagen decente, se requerían horas y días de tiempo computacional.

La falta de trazado de rayos compatible con hardware no permitió realizar avances que permitieran aplicar el trazado de rayos en todas partes. Los intentos de implementar el trazado de rayos basado en hardware se hicieron en *Mitsubishi Electric* con las tarjetas *FPGA* diseñadas específicamente, o en *Caustic Graphics* con las tarjetas *Ray-Tracing Unit (RTU)*.

Todo cambió en 2018 cuando las tarjetas gráficas de propósito general obtuvieron núcleos de trazado de rayos y tipos especiales de *shaders* diseñados exclusivamente para el trazado de rayos.

En este trabajo, describimos los beneficios del trazado de rayos en la *GPU* comparando las implementaciones de algunos algoritmos y métodos de trazado de rayos en la *CPU* y la *GPU*. El resultado de este trabajo son dos plataformas de renderizado de software, un trazador de rayos de *CPU* y un trazador de rayos de *GPU*, que se basan en los principios de trazado de rayos descritos por *Peter Shirley* en su serie *Ray-Tracing in One Weekend* [1][2][3].

El código fuente de las plataformas de trabajo de Ray-Tracing en *CPU* y *GPU* está disponible en el repositorio público de GitHub y publicado bajo la licencia *Open Source MIT* en la siguiente *URL*:

- <https://github.com/d-k-ivanov/ray-tracing>

Ray-Tracing on GPULight Sampling Algorithms

Dmitry Ivanov

Keywords: ray tracing, path tracing, gpu, realistic rendering, light sampling

Abstract

Ray-Tracing is a very mature visualisation technology that has evolved over the years and continues to get new game-changing instruments each year. Ray-tracing allows the simulation of light behaviours and creates very realistic images of preset environments. For a very long time, the biggest bottleneck of the technology was time performance which was significantly limiting the application scope and didn't allow real-time applications. To render a decent picture, it was required to spend hours and days of computational time.

The lack of hardware-supported Ray-Tracing didn't allow to make break-outs which would allow applying Ray-Tracing everywhere. The attempts to implement hardware-based Ray-Tracing were made at Mitsubishi Electric with the specifically designed FPGA cards, or at Caustic Graphics with the Ray-Tracing Unit (RTU) cards.

Everything changed in 2018 when general-purpose graphics cards got Ray-Tracing cores and special types of shaders that are solely designed for Ray-Tracing.

In this work, we describe the benefits of ray tracing on the GPU by comparing the implementations of some Ray-Tracing algorithms and methods on the CPU and GPU. The result of this work is two software rendering platforms, CPU ray-tracer and GPU ray-tracer, that are based on the Ray-Tracing principles described by Peter Shirley in his Ray-Tracing in One Weekend series [1][2][3].

The source code of the Ray-Tracing frameworks on CPU and GPU is available on the public GitHub repository and published under the Open Source MIT Licence at the following URL:

- <https://github.com/d-k-ivanov/ray-tracing>

Acknowledgments

I want to thank every person I met on my way to the Master's Degree:

To **my wife and daughters** whose tremendous support helped me on the way to make this work happen.

To my tutor, **Dr. Carlos Ureña Almagro**, who took me under his wing and whose profound knowledge and success motivated me to start and finish this project.

To my professor, **Dr. Alejandro José León Salas**, whose moral support and encouragement allowed me to proceed despite every obstacle I met.

To **my professors** who motivate everyone to put in additional efforts to reach goals and provide valuable and encouraging feedback.

To **my classmates**, who helped me to stay when my strength and motivation faded. Together, we went toward the next levels of our lives.

Quiero agradecer a cada persona que conocí en mi camino hacia la Maestría:

A **mi esposa e hijas** cuyo tremendo apoyo me ayudó en el camino para hacer realidad este trabajo.

A mi tutor, el **Dr. Carlos Ureña Almagro**, quien me tomó bajo su protección y cuyo profundo conocimiento y éxito me motivaron a iniciar y terminar este proyecto.

A mi profesor, **Dr. Alejandro José León Salas**, cuyo apoyo moral y estímulo me permitieron seguir adelante a pesar de todos los obstáculos que encontré.

A **mis profesores** que motivan a todos a poner más esfuerzas para alcanzar las metas y dan retroalimentación valiosa y alentadora.

A **mis compañeros**, que me ayudaron a quedarme cuando mis fuerzas y motivación se desvanecieron. Juntos, avanzamos hacia los siguientes niveles de nuestras vidas.

Contents

1	Introduction	1
1.1	Motivation	2
1.2	Objectives	3
1.3	Structure	3
2	State of Art	5
2.1	Ray-Tracing	5
2.2	Ray-Tracing Acceleration	10
2.3	Monte-Carlo Methods	12
2.4	Stratified Sampling	13
2.5	Moving Ray Tracing to GPU	16
3	Methodology and Planning	19
3.1	Research	19
3.2	Project Management	21
3.2.1	Software Development Methodology	21
3.2.2	Milestone Schedule	22
3.3	Version Control and GitHub platfporrm	22
3.4	Kanban Board	23
4	Development	25
4.1	Third-Party Libraries	25
4.2	Platform	26
4.3	Ray-Tracing on CPU	26
4.3.1	Class Diagram	27
4.3.2	Single-core and Multi-core CPU implementations . . .	28
4.3.3	Performance	29
4.3.4	Conclusion	30
4.4	Ray-Tracing on GPU	31
4.4.1	Five new shader types	32
4.4.2	Mapping CPU implementation to new shader types .	33
4.4.3	Vulkan Ray Tracing Pipeline	34
4.5	Light Sampling	35

5 Results and Discussion	39
5.1 CPU vs. GPU Ray-Tracing Image Quality	40
5.2 CPU vs. GPU Ray-Tracing Performance	43
5.3 Project Implementation and Availability	44
6 Conclusions and future work	47
6.1 Conclusions	47
6.2 Future Work	48
Bibliography	51
A Compilation	53
A.1 Compilation on Windows	53
A.2 Compilation on Linux	54
B Usage Instructions	55
B.1 Ray-Tracing Framework on CPU	55
B.2 Ray-Tracing Framework on GPU	57
C License	61

List of Figures

1.1	RTX I/O Off	2
1.2	RTX I/O On	2
2.1	Parametric Line	6
2.2	Ray Vector	6
2.3	Ray-Sphere intersections	7
2.4	Ray-Plane intersection	8
2.5	Jordan curve theorem	8
2.6	Geometrical illustration of Möller-Trumbore intersection algorithm	9
2.7	Implicit surface and demonstration of the difficulty of sampling an implicit surface	9
2.8	Classification of acceleration techniques	10
2.9	BVH 100 objects	10
2.10	BVH 1000 objects	10
2.11	BVH 5000 objects	10
2.12	BVH 10000 objects	10
2.13	BVH 20000 objects	10
2.14	Efficient Ray-Tracing for Bezier and B-spline surfaces	11
2.15	GARP - Ray-Patch intersections	11
2.16	GARP performance	12
2.17	Regular and stratified sampling evolution	14
2.18	Stratified sampling papers interconnection in Ray-Tracing	14
2.19	Uniform and stratified sampling of spherical triangles	15
2.20	Uniform and stratified sampling of spherical rectangles	15
2.21	Wavefront Kernels overview	16
2.22	Turing Ray Tracing with RT Cores	17
3.1	Connected papers of Stratified sampling of spherical triangles	20
3.2	Connected papers of An area-preserving parametrization for spherical rectangles	20
3.3	Connected papers of Stratified sampling of projected spherical caps	21

3.4	GitHub Metrics	23
3.5	Ray-Tracing on GPU Roadmap	23
3.6	Kanban Board	24
4.1	Ray Tracer on CPU Class Diagram	27
4.2	CPU 10 Samples Per Pixel	29
4.3	CPU 100 Samples Per Pixel	29
4.4	CPU 250 Samples Per Pixel	29
4.5	CPU 500 Samples Per Pixel	29
4.6	Dependency of rendering time on the number of SPP	30
4.7	CPU 200'000 Samples Per Pixel	31
4.8	Five new Ray-Tracing shader types	33
4.9	Mapped Ray Tracer on CPU Class Diagram	34
4.10	Scene with random scattering on a hemisphere without PDF	37
4.11	Scene with random scattering on a hemisphere with PDF	37
5.1	Cornell Box with multiple light sources demo scene	39
5.2	Dependency of rendering time on the number of SPP	44
B.1	Ray-Tracing framework for CPU	55
B.2	Ray-Tracing framework for GPU	58
B.3	Heatmaps for Cornell Box with Lights	59

List of Tables

3.1 Work plan	22
5.1 Ray Traced Image Quality 1	40
5.2 Ray Traced Image Quality 2	41
5.3 Ray Traced Image Quality 3	42
5.4 Ray Tracing performance comparison	43

Chapter 1

Introduction

The idea of Ray-Tracing (a rendering technique that can produce incredibly realistic images and shading effects) isn't new but well-defined and reachable. Since 1968, when Arthur Appel described [4] Ray-Casting techniques and Turner Whitted presented «An Improved Illumination Model for Shaded Display» using Ray-Tracing [5].

The only problem which seemed to be unsolvable was performance. Even on a modern *Central Processing Unit* (CPU), generating a high-realistic 1600×900 pixels image by Ray-Tracing takes a significant amount of time to reach decent quality and definitely can't reach real-time performance. This makes practical application limited and accessible only for large projects with large computation clusters dedicated to image rendering.

In recent years, *Graphics Processing Units* (GPUs) have been significantly improved. The standardized Ray-Tracing API and new hardware acceleration features [6] started a boom in using these devices for general-purpose calculation and image synthesis by ray tracing with NVIDIA RTX™ and Radeon™Rays technologies. The modern RTX technology is capable of including dedicated ray tracing acceleration hardware, using an advanced acceleration structure and implementing an entirely new GPU rendering pipeline to enable ray tracing algorithms to work in real-time.

The data passing overhead, which used to be a significant blocker in the past, is now mitigated by RTX technology. The figure 1.1 and figure 1.2 show the diagram of the I/O improvements on NVIDIA GPU platform.

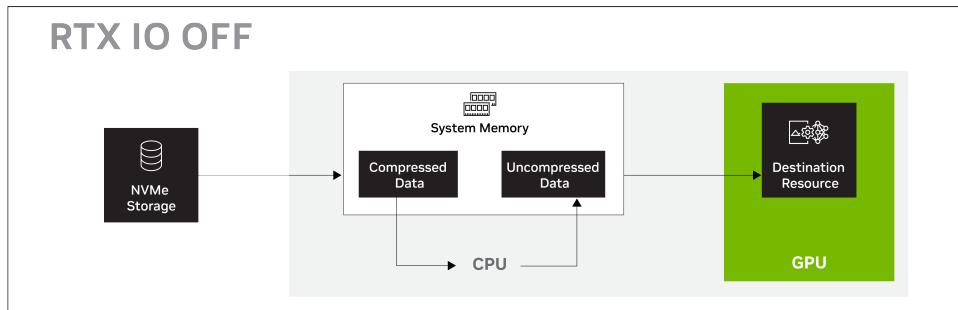


Figure 1.1: Traditional Input/Output throughput.
(Source:nvidia.com)

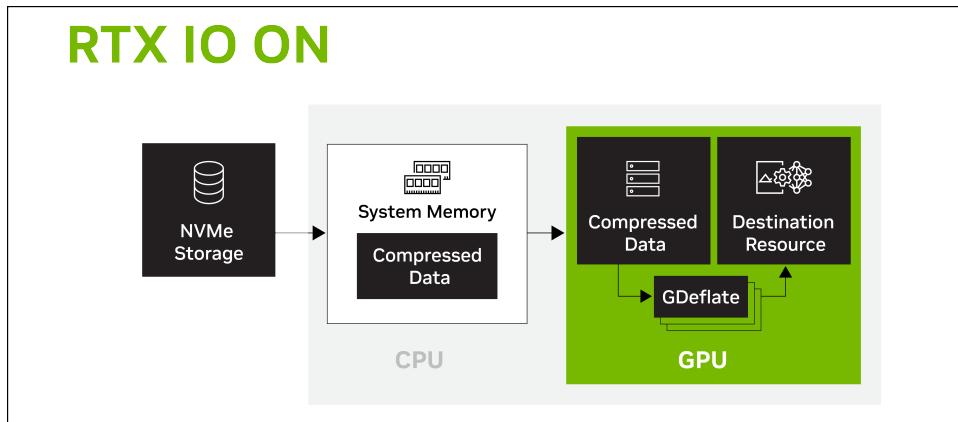


Figure 1.2: GPU accelerated Input/Output throughput.
(Source:nvidia.com)

1.1 Motivation

During our work with the Ray-Tracing algorithms, we experienced significant delays in trying new ideas, mainly due to time-consuming limitations put by the nature of Ray-Tracing technologies and the inability to use mass parallelization easily.

With the new, hardware-supported, Ray-Tracing visualization technology (RTX), ray-racing has become today's technology instead of tomorrow, as it was before the RTX appeared. Numerous technologies and frameworks appear every day, allowing us to make outstanding and realistic visualizations. Some Ray-Tracing technologies are still niche, but production-grade Ray-Tracing support for customer-ranged devices, inspiring further development. Each new visualization framework polishes Ray-Tracing ideas and improves the field.

1.2 Objectives

This work aims to design and implement a software system for testing advanced low-variance light source sampling algorithms used in Ray-Tracing and Path-Tracing Monte-Carlo rendering systems, and also aims to compare the low-efficient CPU-only ray tracing implementation with a time-efficient GPU implementation using modern ray tracing technologies: Vulkan Ray-Tracing Extensions.

The expected result of this work will include simple proof-of-concept 3D interactive scene visualizers, for CPU and GPU, that allow changing the visualization settings and then measuring those changes' effect on the rendering performance and realism while visualizing complex 3D scenes.

From the computational perspective, it's expected that the synthesis of the scenes with a large amount of light sources would be significantly faster using GPU than CPU, allowing testing new ideas on the fly and creating Ray-Tracing solutions for real-world problems.

The goal of this work is to implement the Ray-Tracing framework derived from the framework described by Peter Shirley [1][2][3] on the CPU. Then implement the GPU ray tracing framework using Vulkan Ray-Tracing Extensions and compare the performance of both implementations. We describe the implementation details on both platforms and define the strategy for porting CPU Ray-Tracing technologies from CPU to GPU.

Initially, another objective was to analyse and implement additional advanced algorithms for direct light sampling, derived by Arvo [7] and Ureña et al. [8][9]. We have carried out the analysis, but this target couldn't be fully achieved because we underestimated the time needed for the implementation and limited ourselves to the basic light sampling algorithms and their implementation using Ray-Tracing shaders.

1.3 Structure

In the chapter [2 \(State of Art\)](#) the works related to the Ray-Tracing will be described. We start with the introduction to Ray-Tracing, different intersection approaches of Monte-Carlo methods, and various acceleration methods, and then we describe the stratified sampling methods. We will close this chapter with a description of the state-of-art in the field of Ray-Tracing implementation attempts on GPU.

The chapter [4 \(Development\)](#) will describe the development of the Ray-Tracing frameworks. We will start with the description of the implementation of the CPU-based Ray-Tracing framework in the section [Ray-Tracing](#)

on CPU. Then we will describe the implementation of the GPU-based Ray-Tracing framework in the section [Ray-Tracing on GPU](#). Finally, we will provide the basic implementation of the light sampling and the usage of the *probability density functions* (PDF) in the section [Light Sampling](#) with examples of custom light sampling.

In the chapter [5 \(Results and Discussion\)](#) we will describe the results of our work and provide a comparison of the CPU and GPU-based Ray-Tracing frameworks. We describe the performance of the CPU and GPU-based Ray-Tracing frameworks and provide a comparison of the quality of rendered images.

In the chapter [6 \(Conclusions and future work\)](#) we will provide the general conclusions of our work and describe the future work that can be done using our framework in the field of Ray-Tracing. We will provide future work plans to improve the Ray-Tracing frameworks with additional functionality and more robust Ray-Tracing optimizations.

Several annexes will be provided to describe the usage details of the Ray-Tracing frameworks. The source code will be provided in the form of C++ applications, published on the GitHub platform. We will describe how the applications can be built and executed in the local environment with the Vulkan Ray-Tracing Extensions support. Moreover, we will describe licensing information and usage instructions manuals for the Ray-Tracing frameworks on CPU and GPU, with a detailed description of the user interface and the scene parameters.

Chapter 2

State of Art

In modern computer graphics and ray tracing, numerous techniques and algorithms have been developed to play with Ray-Tracing algorithms and methods. This chapter will present the state of the art in ray tracing and its performance acceleration, Monte Carlo methods, stratified sampling, and the state of attempts to move it to the GPU. The most comprehensive list of available tools and plugins from different developers and for various purposes, on the state of 2019, is presented by Jon Peddie in the book «Ray Tracing: A Tool for All» [10]. We're going to concentrate on our implementation of the ray tracing frameworks using the framework developed by Peter Shirley [1] as the starting point.

2.1 Ray-Tracing

When we build the ray tracing environment, we work with Ray as the geometrical object. In Ray-Tracing, we apply different ray behaviours interacting with different parts of the scene we created. Everything on the scene is created with the help of computational geometry, describing geometrical objects of many kinds, and we need to understand how they intersect with each other. The objects are described with the help of mathematical equations, and we need to understand how to apply ray behaviour to them and where to put Ray-Tracing algorithms to simulate the realistic behaviour of light.

Even though ray tracing technology utilizes Ray as the basic geometric term, ray tracing engines work with a parametric line. Peter Shirley et. al. describe [11] the Ray parametric line as a weighted average of points A and B :

$$P(t) = (1 - t)A + tB \quad (2.1)$$

For the full line, the parameter can take any real value, i.e., $t \in [-\infty, +\infty]$, and the point P moves continuously along the line as t changes, as shown in Figure 2.1:

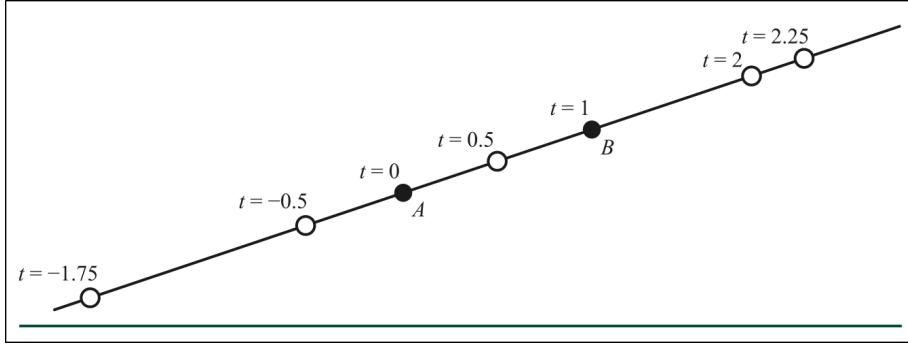


Figure 2.1: How changing values of t gives different points on the line.
(Source: [11])

Instead of two points, choosing a point and a direction is better. We can define the normalized vector between points B and A as \hat{d} (*direction*) and point A as O (*origin*). For various computation reasons, like computing cosines with dot products, it's preferable to use a normalized vector as a direction:

$$P(t) = O + t\hat{d} \quad (2.2)$$

We may select any values of t and the point P moves continuously along the line. When we use a normalized vector as a direction, the value of t represents the signed distance from the origin.

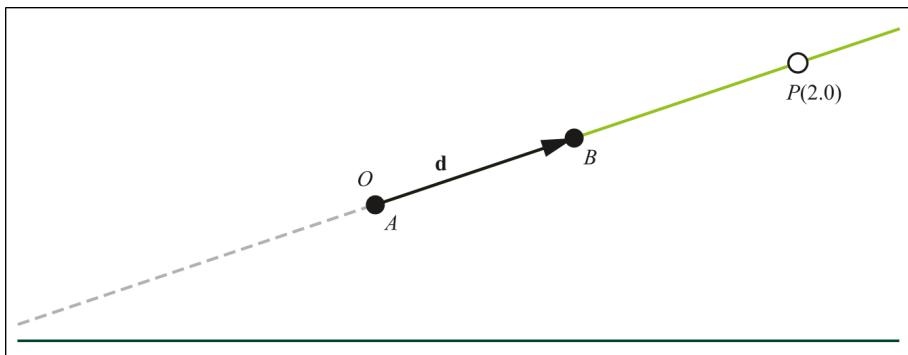


Figure 2.2: A Ray, described by an origin O and direction vector \hat{d} . The points are in front of the origins, i.e. $t > 0$. The dashed line represents points behind the origin. (Source: [11])

Once we generate a ray, we need to compute its intersections with various geometrical objects. Different geometries need different ray-object intersection algorithms, depending on the shape or material of an object. Historically, each ray training engine starts with ray-sphere interactions [12]:

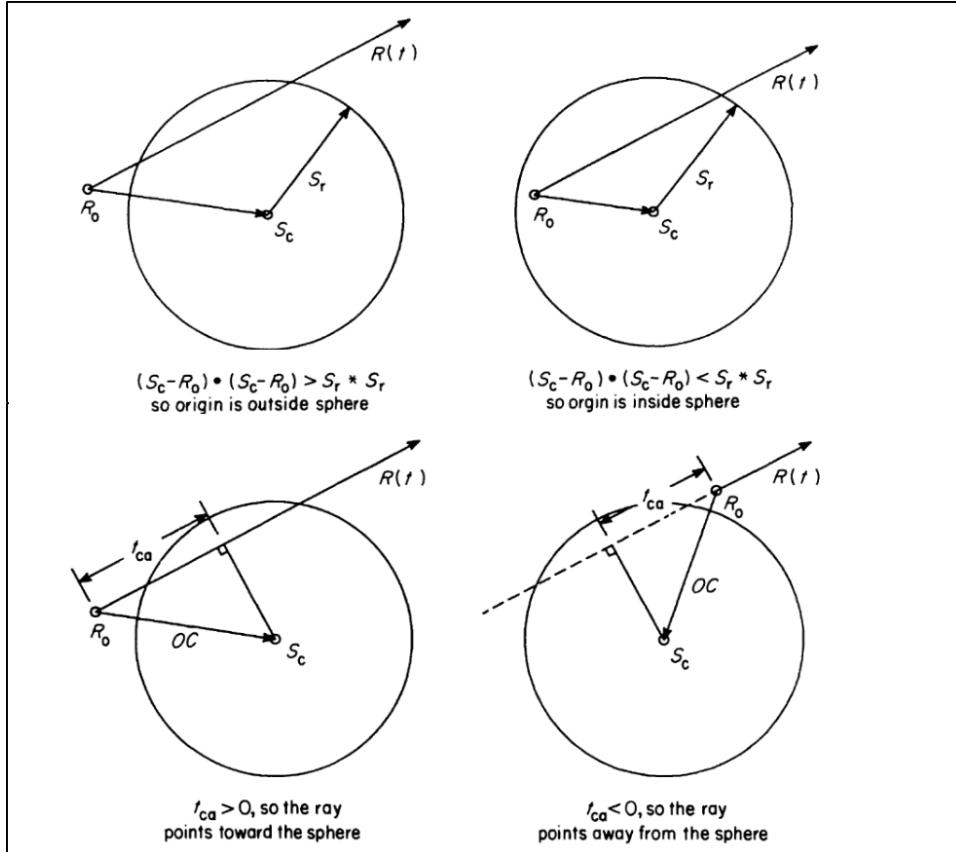


Figure 2.3: The ray origin with respect to sphere location. (Source: [12])

After defining intersections with spherical geometries, the ray tracing engine should get ray-plane intersections for constructing quadrilaterals and further usage in ray-polygon intersection algorithms.

We know a point is on the surface of a plane if it satisfies the plane equation:

$$Ax + Bx + Cz + D = 0 \quad (2.3)$$

where : $A^2 + B^2 + C^2 = 1$

A point with coordinates (x, y, z) is in the plane only when the values x, y and z make the equation hold true. If no point along the ray satisfies the plane equation, the ray and plane do not intersect [13]:

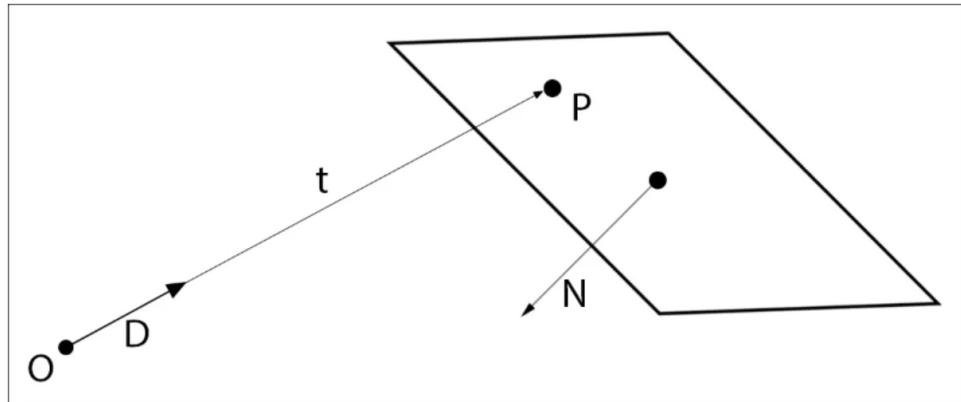


Figure 2.4: Ray-Plane intersection diagram. (Source: [13])

Once the ray-plane intersection is defined, the ray-polygon intersection can be performed. Eric Haines presented [12] one of many methods for testing the location of points (inside or outside), known as the Jordan curve theorem (figure 2.5):

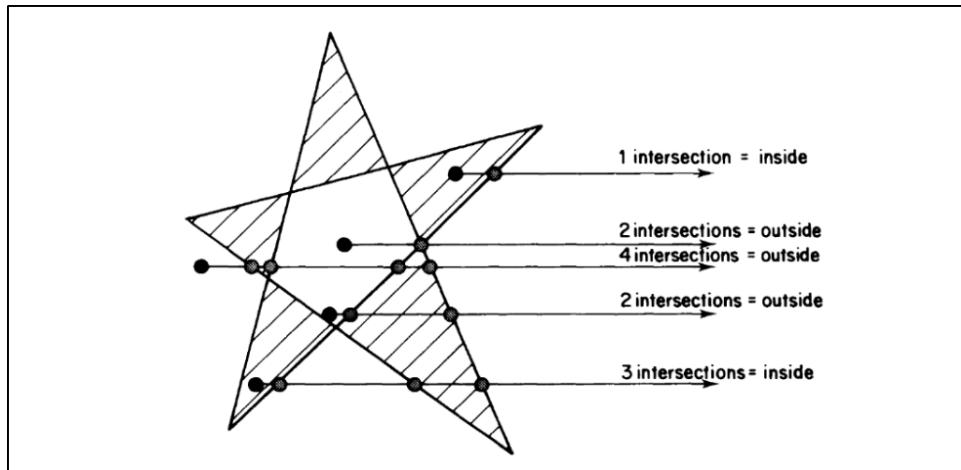


Figure 2.5: Jordan curve theorem. (Source: [12])

This algorithm works by shooting rays in an arbitrary direction and counting the number of intersections with the polygon. If the number is odd, the point is inside the polygon; otherwise, it is outside. The Jordan curve theorem is a fundamental concept in computational geometry. Once the ray-polygon intersection achieved, the ray tracing engine can construct more complex geometries like triangles, quadrilaterals, and other polygons.

Möller and Trumbore presented [14] the faster and simpler ray-triangle intersection algorithm. The algorithm translates the origin of the ray to

triangle-specific barycentric coordinates, then changes the base to yield a vector containing the distance t and the coordinates (u, v) of the intersection, as illustrated on figure 2.6:

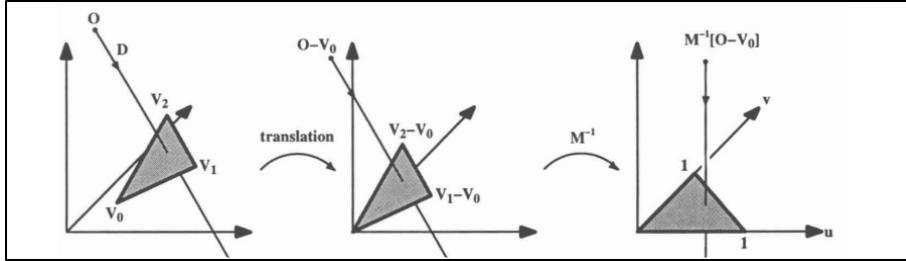


Figure 2.6: Geometrical illustration of Möller-Trumbore intersection algorithm. (Source: [14])

It is shown that the Möller-Trumbore algorithm is comparable in speed to previous methods while significantly reducing memory storage costs, by avoiding storing triangle plane equations. Baldwin and Weber presented [15] a faster ray-triangle intersection calculation at the expense of pre-computing and storing a small amount of extra information for each triangle. The Baldwin-Weber algorithm is 1-6% faster than the Möller-Trumbore algorithm.

Implicit functions can be used to represent interesting geometries, but the task of finding intersections with them is more complicated. It is difficult to create a universal algorithm, based only on the evaluation of an implicit function, that would guarantee a correctly detected intersection. Some functions may introduce spikes that are not detected by the sampling algorithms. Kalra and Barr presented [16] a method for finding guaranteed ray intersections with implicit surfaces. Figure 2.7 represents the problem of finding the intersection of a ray with an implicit surface and the example of implicit surface $f(x, y, z) = 0$:

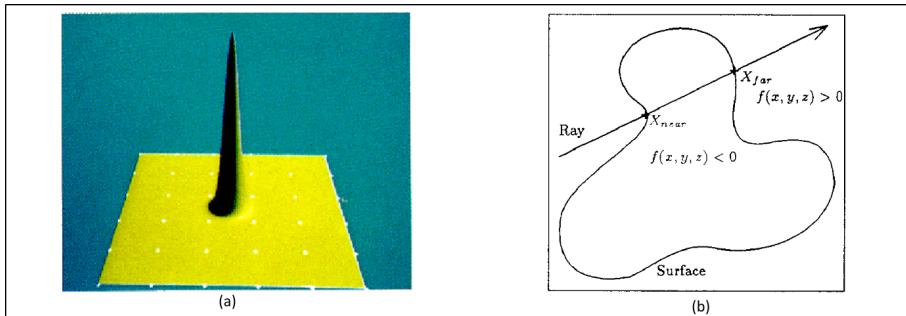


Figure 2.7: Implicit surface (b) and demonstration of the difficulty of sampling an implicit surface (a). (Source: [15])

2.2 Ray-Tracing Acceleration

The weakest point of Ray-Tracing is its performance. There are numerous performance optimizations and acceleration techniques. Using different geometry structures allows for the reduction of complicated computations, increasing the rendering time of a scene. The figure 2.8 presents the broad classification of various optimization approaches described by Arvo and Kirk [17].

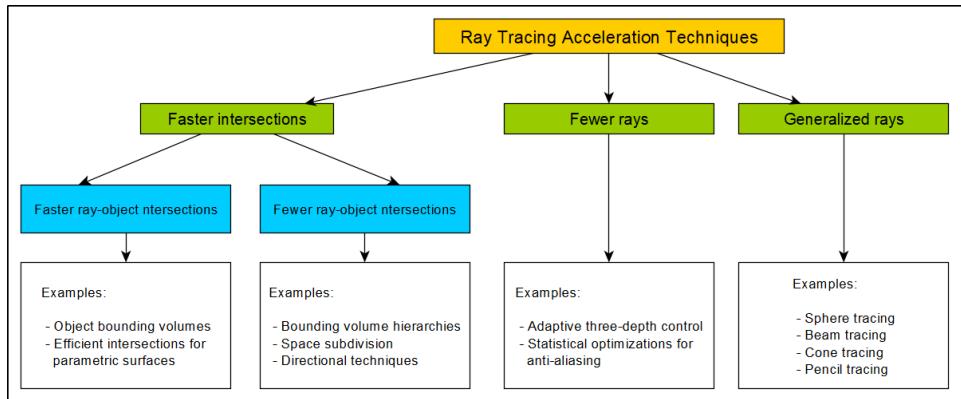


Figure 2.8: A broad classification of acceleration techniques. (*Source: [17]*)

Bounding Volumes Hierarchy (BVH) trees are the most important optimization technology in ray tracing. A scene can consist of thousands of objects. BVH structure makes it possible to avoid computing the intersections between a single ray and each object in a large group of objects when the bounding box of the group does not intersect the ray. BVH tree construction and bounding box hit-checking algorithms help with the time of initial scene construction and Ray-Tracing computations. The figures 2.9, 2.10, 2.11, 2.12, 2.13 show the increasing number of objects on the ray-traced scene. Each sphere is wrapped into a bounding box, and the whole scene exists as one BVH construction.



Figure 2.9:
100 objects
Time: 5s

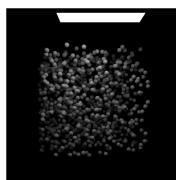


Figure 2.10:
1000 objects
Time: 27s

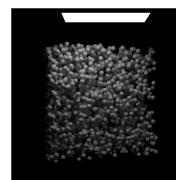


Figure 2.11:
5000 objects
Time: 106s

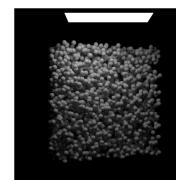


Figure 2.12:
10000
objects
Time: 112s

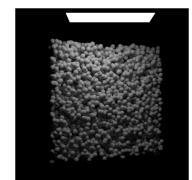


Figure 2.13:
20000
objects
Time: 149s

As was shown in the figures above, the computation time for 100 samples per pixel doesn't increase too much when the number of spheres is quite large, because of the usage of a BVH. The application code is available on GitHub [18].

For surfaces defined by its parametric equations (like Bézier and B-splines), the usage of binary trees with small parts of the surfaces enclosed by parallelepipeds (as it's shown in the figure 2.14) and testing these enclosures which part of the surface may be hit by the ray is a good idea, as proposed by Barth and Stürzlinger [19].

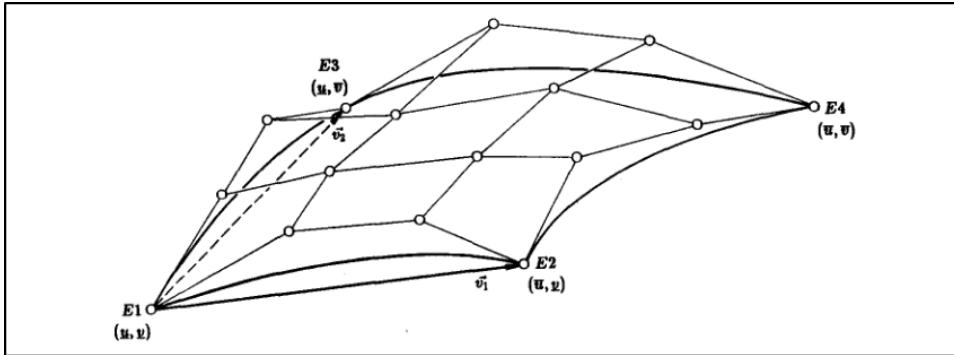


Figure 2.14: Efficient Ray-Tracing for Bezier and B-spline surfaces. A part of the surface. (Source: [19])

Another interesting optimization approach proposed by Alexander Reshetov [20]. Their GARP method (Geometric Approach to Ray/bilinear Patch intersections) is trying to find a balance between the simplicity of triangles and the richness of such smooth shapes as subdivision surfaces, NURBS, and Bézier patches.

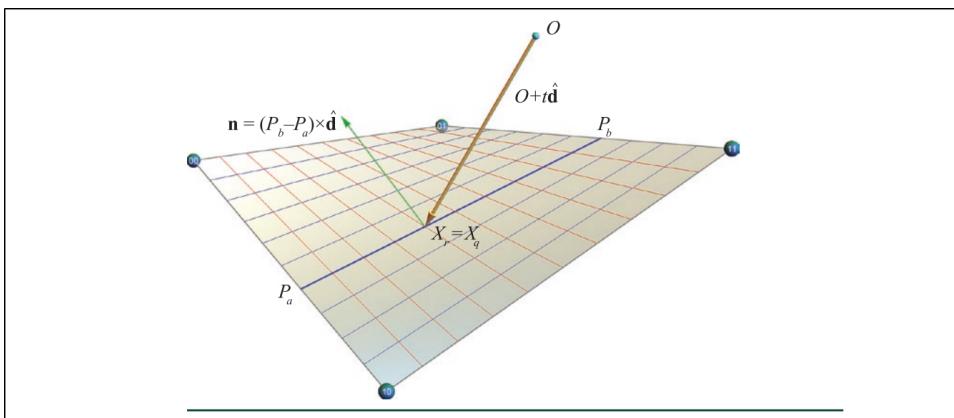


Figure 2.15: Finding Ray-Patch intersections. (Source: [20])

The intersection point could be computed as either $X_r = R(t)$ or as $X_q = Q(u, v)$ using the found parameters t , u , and v . The two-step GARP process dynamically reduces a possible error in each step. In the first step, we find the best estimation for u . On the second step, using the found $u - aim$, minimizing the total error.

The figure 2.16 shows the performance measurements performed by the author by counting the total number of rays processed per second.

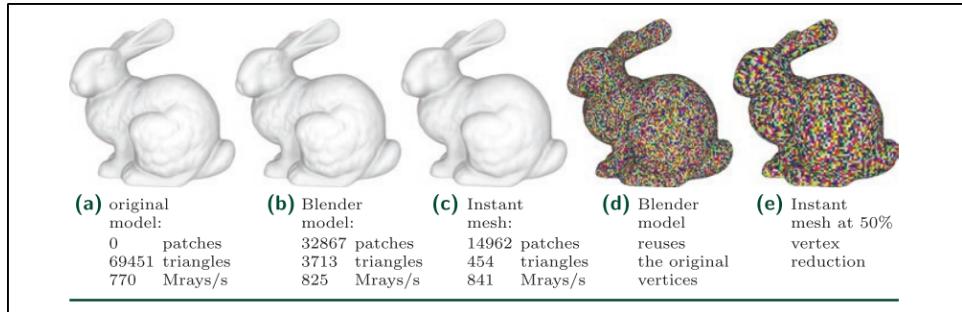


Figure 2.16: Ray-Patch intersection performance. (Source: [20])

In this work, we would like to explore another approach to accelerate Ray-Tracing computations. We will use the Vulkan Ray Tracing extension to offload the computations to the GPU. The chapter 4 (Development) will describe the implementation details of the sample Ray-Tracing framework using the Vulkan Ray Tracing extension.

2.3 Monte-Carlo Methods

Computing how light is reflected and scattered in arbitrary and complex 3D scenes is a challenging computational effort, which may require a vast amount of time and memory when using traditional finite-element methods. However, it has been proved that Monte-Carlo methods are far more appropriate for this kind of computation as their time complexity does not increase so much when the complexity of the scene does, as compared to finite-element methods. Monte-Carlo methods are more favourable in ray tracing when it's possible to task many independent colour samples from a single point.

Kajiya presented the rendering equation[21], the formula that gives an expression for the radiance leaving a point x in a direction $\hat{\omega}_o$, and thus models how light is globally reflected in a 3D scene, which can be solved using the Monte-Carlo methods to approximate numerical integration for calculations of ray intersections that employ a collection of random samples.

$$L_o(x, \hat{\omega}_o) = L_e(x, \hat{\omega}_o) + \int_{S^2} L_i(x, \hat{\omega}_i) f_r(x, \hat{\omega}_i, \hat{\omega}_o) |\hat{\omega}_i \cdot \hat{n}| d\hat{\omega}_i$$

The diagram illustrates the rendering equation. It shows the calculation of outgoing light (L_o) as the sum of emitted light (L_e) and reflected light. Reflected light is calculated by integrating incoming light (L_i) over all incoming directions (S^2) multiplied by the reflection function (f_r) and the cosine of the angle between the incoming direction and the surface normal ($|\hat{\omega}_i \cdot \hat{n}|$). Labels include: Outgoing Light (orange), Incoming light (blue), All Incoming Directions (S^2), Reflected Light (green), Material Reflection (red), Incoming Direction (blue), and Outward Irradiance (purple).

Eric Veach and Leonidas Guibas from Stanford presented [22] a way to construct Monte-Carlo estimators by combining samples from several ray distributions. The experiments indicated that combining sampling techniques can reduce the variance of Monte-Carlo rendering calculation. The authors think that better sampling distributions have great potential for practical applications.

The Monte-Carlo methods have been used in ray tracing for a long time and seems to be the first candidate for porting it to the GPU. Shirley et al. [23] presented Monte-Carlo techniques for direct light sample calculations. Combining them with the Martinsen et al. work [24] on accelerating Monte-Carlo simulations using the NVIDIA CUDA toolkit. We may reach significant results in porting the Monte-Carlo-based algorithms to the GPU, using modern implementations of CUDA on top of RTX. Martinsen et al. showed [24] that the GPU-based implementation of the Monte-Carlo methods for photon tracing is 70 times faster than a single-threaded CPU implementation.

2.4 Stratified Sampling

The Monte-Carlo method has a downside, which can be described by the law of Diminishing Returns from economics, where each sample helps less than the last because, when using Monte-Carlo, every new sample requires a fixed amount of computational effort but its contribution to lower the error decreases with the number of samples already computed. Shirley pointed out [25]) and [3], that the diminishing return can be mitigated by stratifying the random samples. Instead of taking random samples, the grid can be taken and then one sample within each cell of the grid. However, the rendering result of the usage of this technique becomes more noisy.

In the figure 2.17 we can see the evolution of the regular and stratified sampling. In the picture (a), pixel sampling produces the same answer for each pixel. No matter how many samples are taken, the result is the same because we approximate the true average. The picture (b) shows the stratified sampling, the samples are taken from the bins, and the result is more noisy, but the error is lower. Alternatively, we might use random

sampling within the pixel, but, in this case, Moiré patterns can arise, and these artefacts can be turned into unpleasant noise.

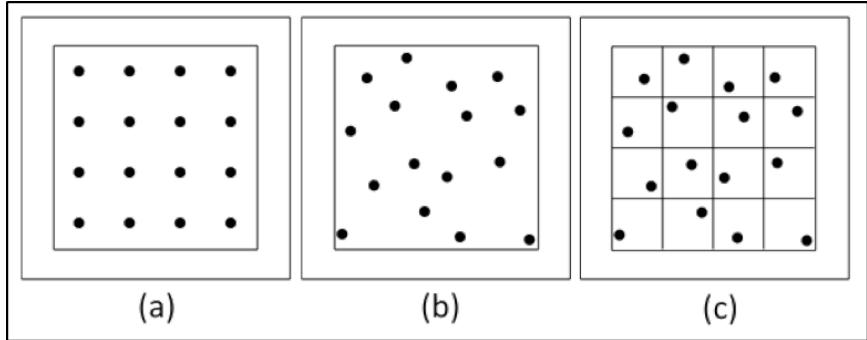


Figure 2.17: Sixteen regular samples for a single pixel (a). Sixteen stratified (jittered) samples for a single pixel shown with and without the bins highlighted (b). There is exactly one random sample taken within each bin (c). (*Source: [25]*)

Stratified sampling is a well-known technique for reducing variance in statistics, here we focus on its application in computer graphics. Specifically, we are interested in applying stratified sampling to the problem of sampling light sources in ray tracing. In the figure 2.17 we can see the distribution of the works related to stratified sampling in Ray-Tracing.

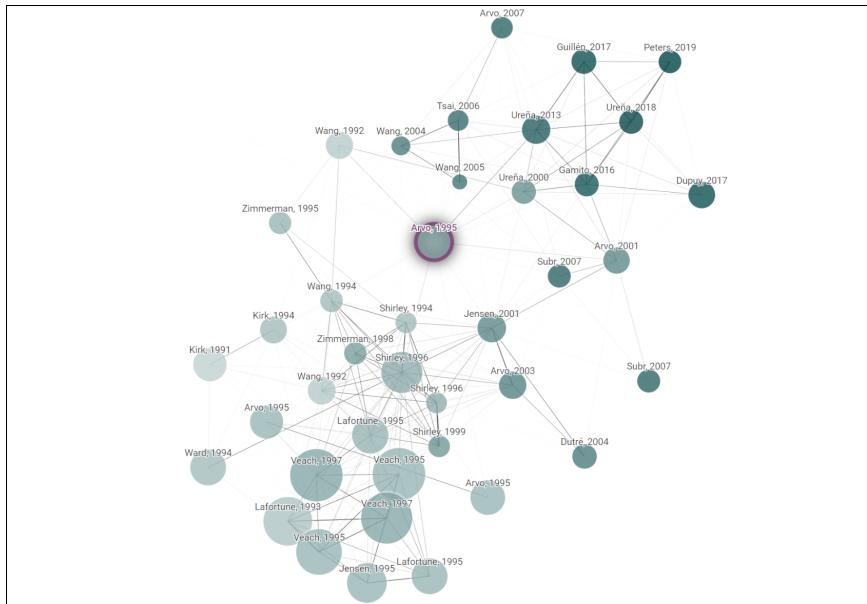


Figure 2.18: Stratified sampling papers interconnection in Ray-Tracing concentrated around. [7]

James Arvo applied stratified sampling to light source sampling and derived [7] the algorithm of stratified sampling of spherical triangles to improve the distribution of random samples and indicated that stratified sampling can be performed on each triangle component of the sphere independently.

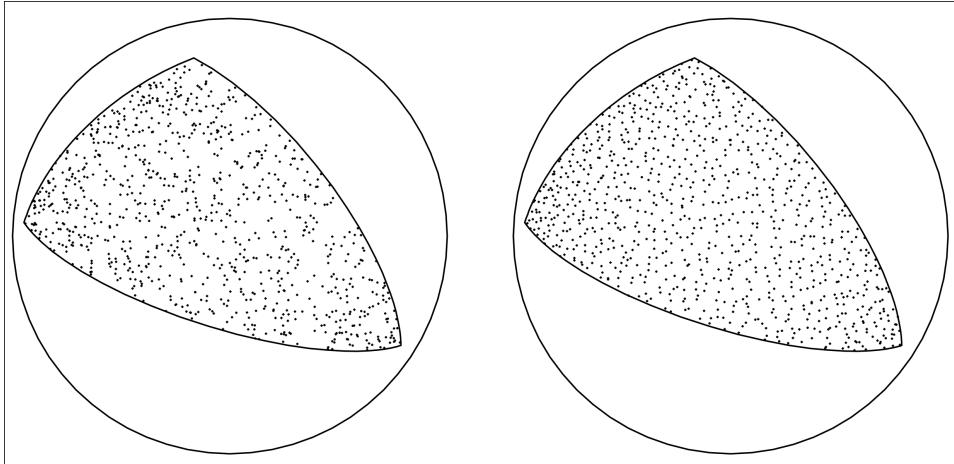


Figure 2.19: Uniform and stratified sampling of Spherical Triangles.
(Source: [7])

Ureña et al. extended stratified sampling algorithms to cover spherical rectangles [8] and spherical caps [9]. The figure 2.20 shows the distribution improvements.

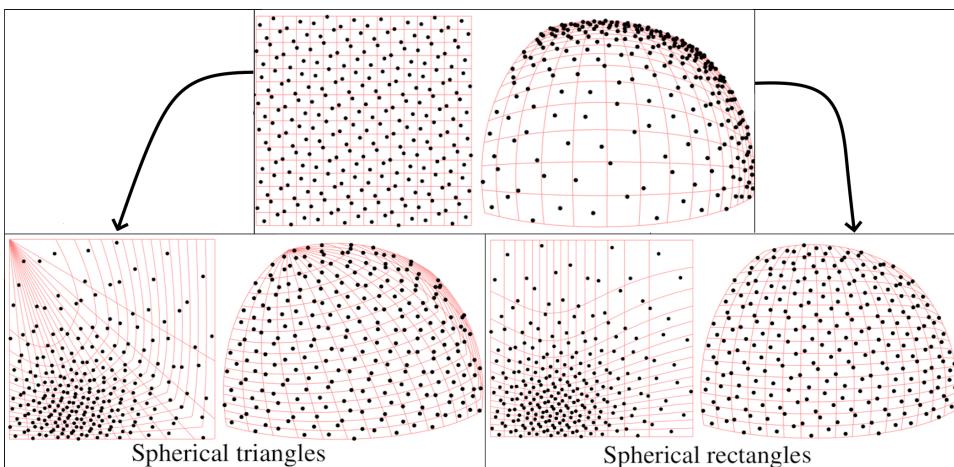


Figure 2.20: Uniform and stratified sampling of Spherical Rectangles
(Source: [8])

2.5 Moving Ray Tracing to GPU

The randomized nature of ray generation makes it natural to use GPUs in ray tracing, but previous implementations faced hardware limitations and were not very performant.

Martin Christen [26] has attempted to use OpenGL and Direct3D shaders. He concluded that GPU ray tracing is feasible, but the GPU-based implementation was not faster than the CPU implementation using the 2005-year hardware.

Parker et al. [27] described how the NVIDIA OptiX engine with a programmable ray tracing pipeline can be used to implement ray tracing algorithms. OptiX uses a small set of programmable operations, which is similar to the approach in using programmable rasterization pipelines employed by OpenGL and Direct3D.

Antwerpen in his Master's thesis evaluated [28] streaming implementation of a BiDirectional Path Tracer (BDPT) and found that the streaming implementation of BDPT required only storage for a single light and eye vertex in memory at any time during sample evaluation, making the memory footprint independent of the path length.

Laine et al. presented [29] the implementation of a *wavefront* path tracer. A wavefront ray tracer separates the main operations into separate kernels instead of using a single kernel for the entire ray tracing process, which is harmful to the ray tracing performance. The per-process operation queues are used for each primitive operation request: new ray generation, shading, material evaluation, and extension ray casting. In figure 2.21 we can see the overview of the wavefront kernels.

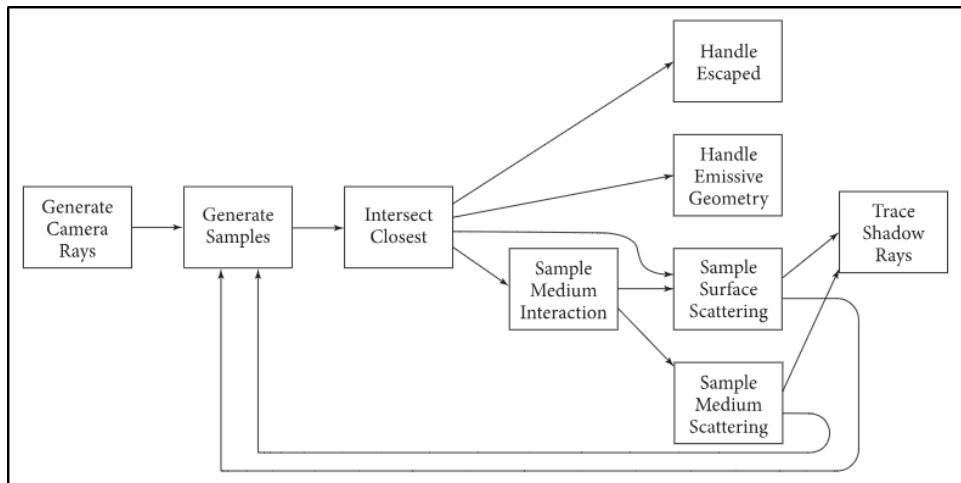


Figure 2.21: Wavefront Kernels overview (*Source: [30]*)

We believe that Laine et al. work [29] has greatly influenced Nvidia Ray-Tracing hardware design. Better implementations of Ray-Tracing algorithms became available when NVIDIA released the RTX technology with RT Cores in 2018 and support of dedicated Ray-Tracing shaders for the major Ray-Tracing stages. The RTX technology became available in the Nvidia Turing GPU architecture. The release of RTX technology led to the development of a Real-Time Ray tracing section in DirectX, NVIDIA OptiX and Vulkan.

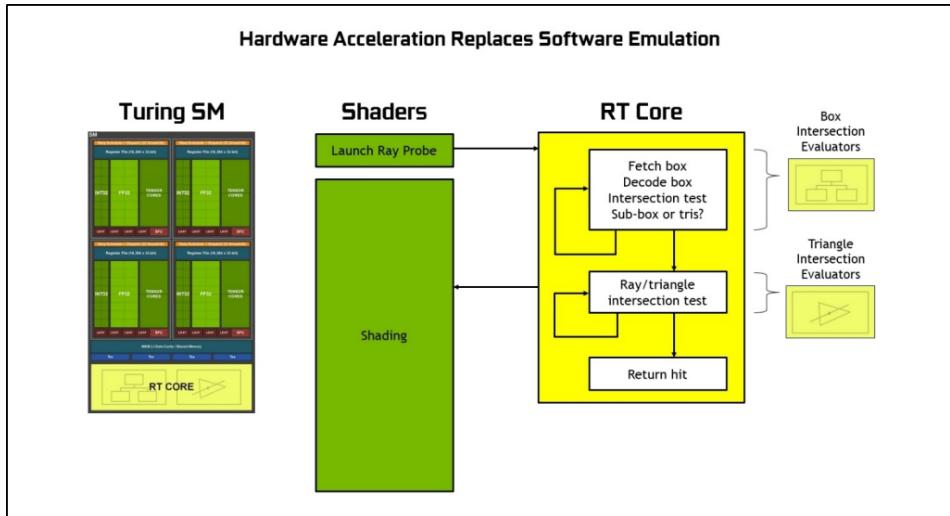


Figure 2.22: Turing Ray Tracing with RT Cores (*Source: nvidia.com*)

Knoll et al. [31] provided a method to leverage RTX GPUs with RT Cores for efficient rendering of large particle data. Daniel Meister, Jakub Boksan-sky, Michael Guthe, and Jiri Bittner [32] compared nine methods for ray reordering using RTX-based technologies.

Chapter 3

Methodology and Planning

In this chapter, we will describe the planning, methodology and project management tools we use to develop the project. We start with the description of the research strategy and knowledge management toolbox. We continue with the project management tools and the version control system we use to develop the project.

3.1 Research

Our development has been started from the analysis of the initial papers we selected:

- Arvo, J. Stratified sampling of spherical triangles [7]
- Ureña, C. P. and Fajardo, M. and King, A. An area-preserving parametrization for spherical rectangles [8]
- Ureña, C. P. and Georgiev, I. Stratified sampling of projected spherical caps [9]

A connected papers graph is used to make a comprehensive literature analysis and extend or reduce the scope of the work if necessary. The figures 3.1, 3.2, and 3.3 show the graph for the initial paper. These papers greatly contributed to the development of the Ray-Tracing project. They allowed us to summarise the state of the art in the field and the requirements for the Ray-Tracing framework we will develop to evaluate Ray-Tracing algorithms.

The main deduction we made from the initial paper analysis, is that before starting working on the precise Ray-Tracing algorithms, we need to develop the Ray-Tracing framework. The Ray-Tracing framework should be able to evaluate the Ray-Tracing algorithms and provide the results in a visual

form. The Ray-Tracing framework should be able to run on the CPU and GPU with the RTX technology, if available.

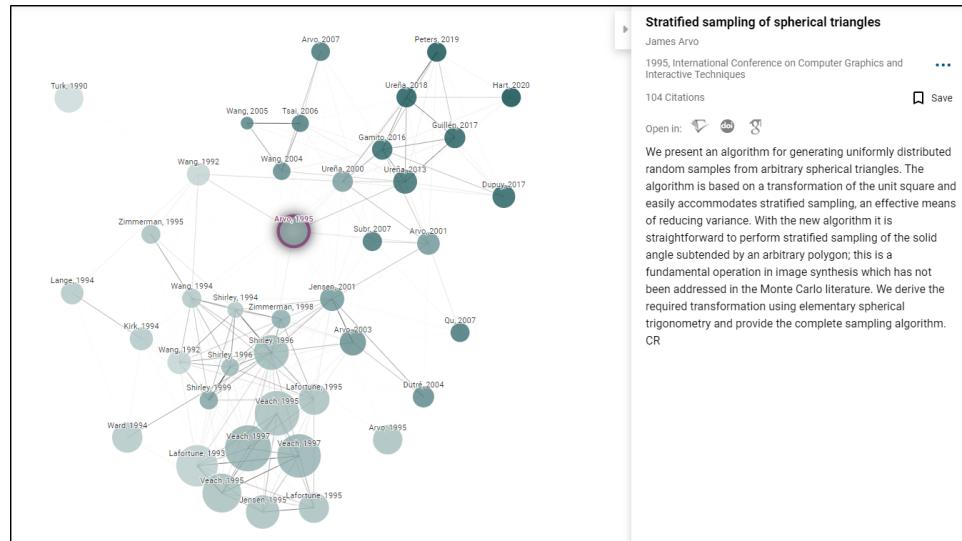


Figure 3.1: Connected papers of Stratified sampling of spherical triangles
[7]

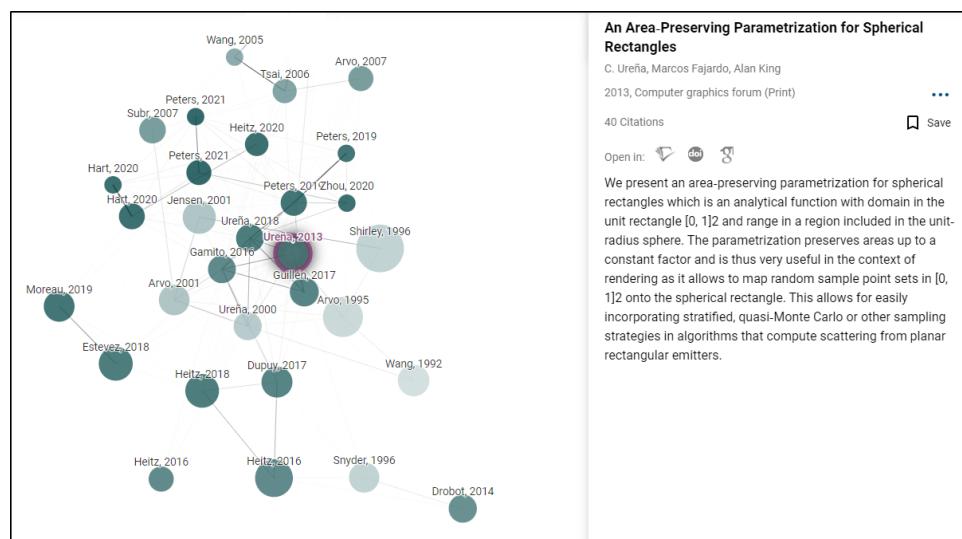


Figure 3.2: Connected papers of An area-preserving parametrization for spherical rectangles [8]

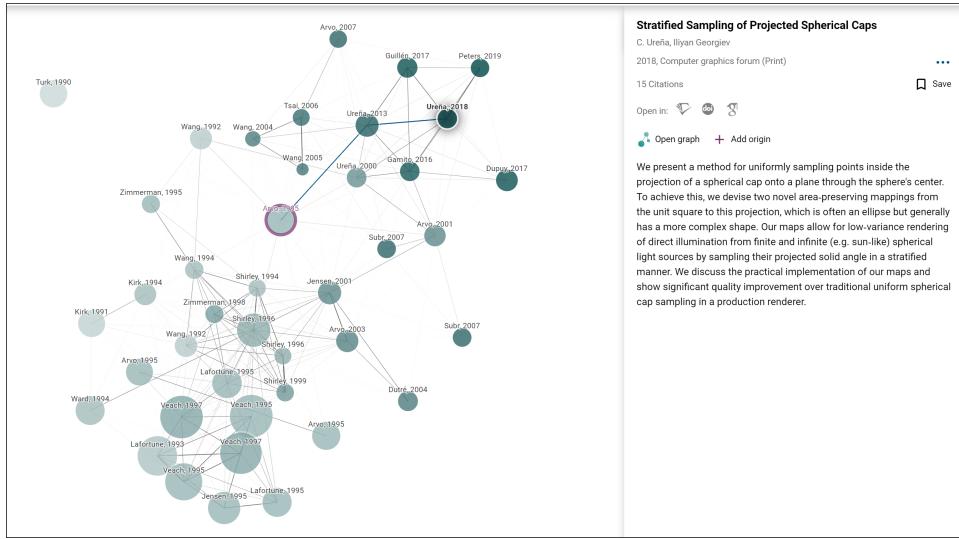


Figure 3.3: Connected papers of Stratified sampling of projected spherical caps [9]

All related to the work papers collected in the Zotero database and are available for further analysis. The Zotero database is used to store the papers, the notes, and the tags. The tags are used to categorize the documents and to make the search easier. The notes are used to summarize the papers and to extract the main ideas. The Zotero database may be synchronized with the cloud storage or moved manually, so the papers are available from any device.

3.2 Project Management

3.2.1 Software Development Methodology

For repetitive development, the aim is to follow an agile development methodology, developing tasks of small size and grouping them to meet an important objective. The tasks are assigned to a sprint of a specific duration, thus allowing small work units to form more complex milestones. The basic implementations are verified with quick checks using expected results as a driver for further development. Once the outputs of the initial implementation have been validated with satisfying results, the development moves on to the next iteration. It is essential to keep the tests quick and simple enough to progress faster, fulfilling the main and improvement backlog. The improvement backlog is the reserved task list within each milestone to fulfil the development of unforeseen events of low priority. The main backlog is the reserved task list within each milestone to fulfil the development of

unforeseen events of high priority.

For simplicity and due to a lack of developers, we've chosen the semi-waterfall methodology for the project. The Waterfall methodology is a linear project management approach, where the project is divided into sequential phases. Each phase must be completed before the next phase begins.

3.2.2 Milestone Schedule

The development sections (also known as sprints) are combined into milestones. Each milestone represents the self-sufficient stage of the working plan and should be completed on time. The Table 3.1 shows the project methodology plan.

Milestone	Duration	Start	End
Study and analysis of relevant literature	4 weeks	01/01/2024	31/01/2024
Analysis of existing solutions	8 weeks	01/02/2024	31/03/2024
Design and implement the Ray-Tracing evaluation framework on CPU	8 weeks	01/04/2024	31/05/2024
Design and implement the Ray-Tracing evaluation framework on GPU	8 weeks	01/06/2024	31/07/2024
Writing the TFM report document	4 weeks	01/08/2024	31/08/2024
Writing the TFM presentation	2 weeks	01/09/2024	14/09/2024

Ray-Tracing on GPU project work schedule.

3.3 Version Control and GitHub platform

The project is developed on the GitHub collaboration platform: <https://github.com/d-k-ivanov/ray-tracing>. GitHub offers version control, collaboration, project management and development automation tools. Each milestone is divided into tasks. Each task could be developed in a separate branch. The branches are merged into the main branch when the task is completed. The main branch is used to build and release the project. The GitHub metrics allow tracking of the project progress, such as the number of commits, the number of branches, the number of pull requests, and the number of issues.



Figure 3.4: GitHub Metrics.

Each project item, such as Milestone, Epic, Story, or Task, is visualized on the Kanban board. The Review state section is reserved for item milestone validation. The milestone roadmap can be described as a Waterfall diagram, which fits the best for time scheduling. The initial roadmap for the Ray-Tracing on GPU project is shown in Figure 3.5.

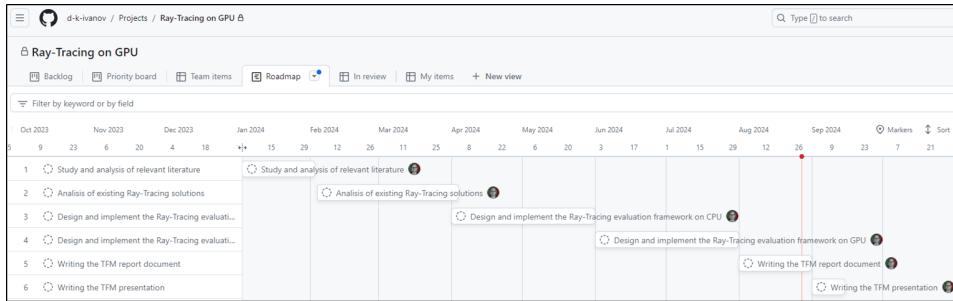


Figure 3.5: Ray-Tracing on GPU Roadmap on GitHub. The original interactive version of the roadmap is available at
<https://github.com/users/d-k-ivanov/projects/2/views/4>.

3.4 Kanban Board

For better visualization of the project's progress, we use the Kanban board. The board is divided into columns: Backlog, Ready (to take in the development), In Progress, Review, and Done. Each task is represented as a card on the board. The card contains the task description, the assignee, the milestone, and the due date. The board is used to track the project's progress and to identify the bottlenecks when we set the maximum number of work

items in the specific state. The board is updated daily to reflect the current state of the project. The board is used to plan the work for the next day and to identify the tasks that need to be completed.

Figure 3.6 shows the Kanban board for the Ray-Tracing on GPU TFM project.

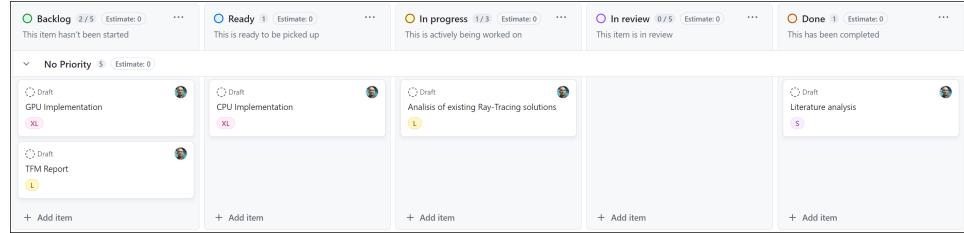


Figure 3.6: Kanban Board. The original interactive version of the board is available at

<https://github.com/users/d-k-ivanov/projects/2/views/1>.

Chapter 4

Development

The project is written in C++ and uses CMake as the build system. All libraries used in this project are open-source and available on GitHub. The project is divided into two main parts: the CPU and GPU Ray-Tracing implementations. The CPU implementation is based on the Vulkan API, while the GPU implementation is based on the Vulkan Ray Tracing extension.

4.1 Third-Party Libraries

We use VCPKG to manage the third-party libraries in the project. It supports all popular platforms: Windows, Linux, and macOS. The libraries used in the project are:

GLFW is an Open Source, multi-platform library for OpenGL, OpenGL ES and Vulkan development on the desktop. It provides a simple API for creating windows, contexts and surfaces, as well as receiving input and events. We use it for window creation and input handling.

GLM is a header-only C++ mathematics library for graphics software based on the GLSL specifications. It is used for the vector and matrix operations in the project. We use it for the math operations in the project and to get the general math classes that are compatible with GLSL.

Dear ImGui with Docking is a bloat-free graphical user interface library for C++. It outputs vertex buffers that you can render in your 3D-pipeline enabled application. It is fast, portable, renderer agnostic and self-contained (no external dependencies).

spdlog is a fast C++ logging library. It is header only and we use it in the project for logging.

stb is a single-file public domain image loader for C/C++. We use it for storing images and visualising them in Dear ImGui using a Vulkan wrapper.

tinyobjloader is a tiny but powerful single-file object loader written in C++. We use it to load wavefront object files into the scene to get triangulated surfaces.

Vulkan SDK is the official Vulkan SDK from LunarG. It includes the Vulkan headers, libraries, tools, and documentation. It is the main library used in the project for Vulkan API calls.

4.2 Platform

To develop the Ray Tracing using Vulkan, we need to have an RTX-compatible GPU and driver. The project is developed on a desktop computer with the following specifications:

- **System:** XMG Neo 17 (Laptop)
- **OS:** Windows 10
- **CPU:** Intel i9-14900 HX
- **RAM:** 64 GB
- **GPU:** NVIDIA GeForce RTX 4070 (with Ray Tracing cores)
- **GPU Ram:** 8 GB
- **Compiler:** MSVC 19.41 (Visual Studio 2022)
- **CMake:** 3.30.2
- **DirectX Support:** 12.1

4.3 Ray-Tracing on CPU

We started our project with the development of a simple Ray-Tracing framework on the CPU. We use the implementation of the ray-tracer described by Peter Shirley in his book series *Ray Tracing in One Weekend* [1] [2] [3].

In the next section, we described the relationship between classes and the most important fields and methods which are used to trace rays and make decisions about the ray behaviour.

The implementation details are published on GitHub:

- Ray-Tracing Frameworks and Visualizers: RayTracingCPU

4.3.1 Class Diagram

On the figure 4.1, we can see the class diagram of the RayTracer on CPU.

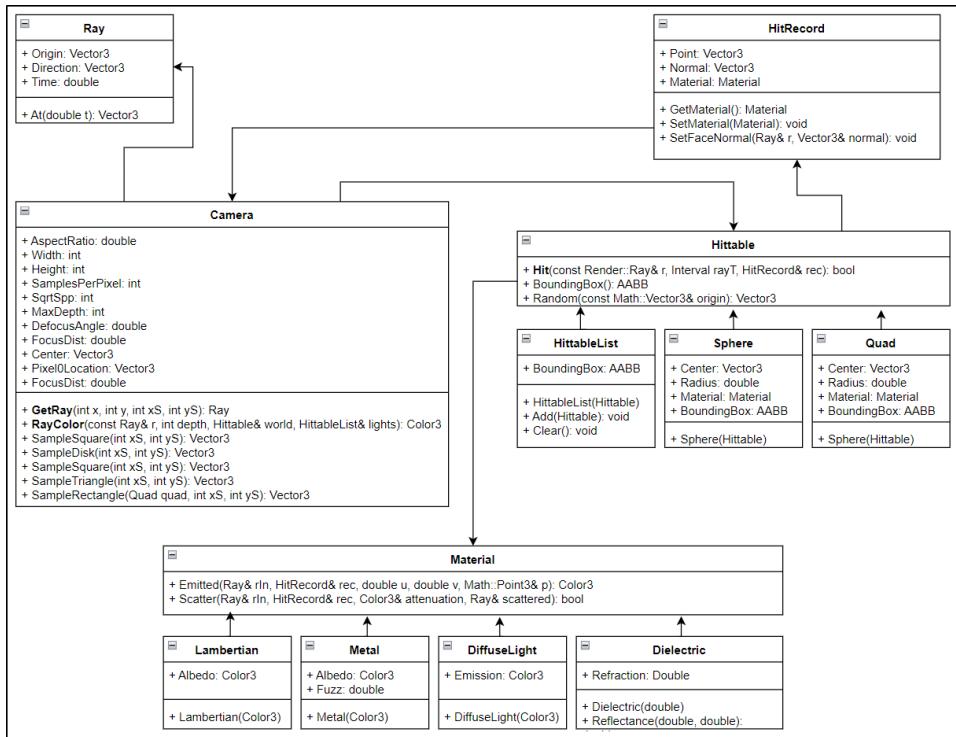


Figure 4.1: Ray Tracer on CPU Class Diagram

- **Camera:** class for the camera in the scene. The camera has a position, a look-at point, and a field of view. The method *GetRay* returns the ray from the camera to the scene, these rays are usually called «primary rays». The method *GetColor* returns the colour of the pixel in the scene, depending on the hit record or background colour if the ray does miss any object.
- **Hittable:** abstract class for all objects in the scene. The main method is *Hit* which returns the hit record and the material of the object.
- **HittableList:** class for the list of objects in the scene. We use it to group some objects. The bounding box of the Hittable List is used to speed up the Ray-Tracing algorithm in high-density scenes.

- **Material:** abstract class for all materials in the scene. The main method is *Scatter* which returns the scattered ray and the attenuation of the ray. Reflected or emitting ray behaviour.
- **Ray:** class for the ray in the scene. The main method is *At* which returns the point on the ray at the given distance.
- **HitRecord:** class for the hit record of the object in the scene. The main fields are the hit point, normal, and material. The method *SetFaceNormal* sets the normal of the hit record depending on the ray direction.

4.3.2 Single-core and Multi-core CPU implementations

The Ray-Tracer on the CPU is implemented in two versions: single-core and multi-core. The single-core version is a simple implementation of the Ray-Tracing algorithm that uses a single for loop to iterate through all pixels in the image and calculate the colour of the pixel. The multi-core version uses the C++ parallel version std::for_each loop to parallelize the rendering process. The parallel execution evaluates each pixel in a separate thread. On the algorithm 1, we can see the main rendering loop of CPU Ray-Tracer implementation. The main difference between the single-core and multi-core implementations is the execution policy of the <> for loop. The single-core version uses the sequential execution policy, while the multi-core version uses the parallel execution policy.

Algorithm 1 Ray-Tracing on CPU main rendering loop

```

1: Depth                                     ▷ The maximum depth of the ray
2: Image                                      ▷ The image data array
3: Lights                                     ▷ The list of lights in the scene
4: NumberOfSamples                            ▷ The number of samples per pixel
5: Objects                                    ▷ The list of objects in the scene
6: ExecutionPolicy   ▷ The execution policy for the "for" for loop: parallel or sequential
7:
8: The iteration through all pixels in the image and calculates the pixel colour. The execution policy is used to parallelize the rendering process if parallelization is available.
9:
10: The colour sampling is executed in the same thread sequentially, regardless of the execution policy.
11:
12: for each y in the ImageHeight do
13:   for each x in the ImageWidth do
14:     for each sample in the NumberOfSamples do
15:       Index = y * ImageWidth + x
16:       Image[Index] = RayColor(Ray(x, y, Depth, Objects, Lights)
17:     end for
18:   end for
19: end for

```

4.3.3 Performance

In the pictures 4.2, 4.3, 4.4, and 4.5, we can see the rendered images with 10, 100, 250, and 500 samples per pixel (SPP) settings respectively. As we can see, the pictures are very noisy, especially with a low number of SPP. We need to significantly increase the number of SPP to render an image of decent quality.

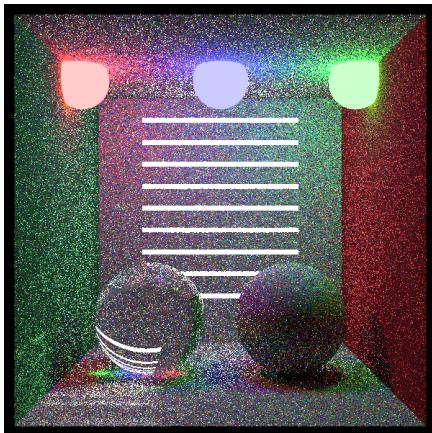


Figure 4.2:
10 Samples Per Pixel

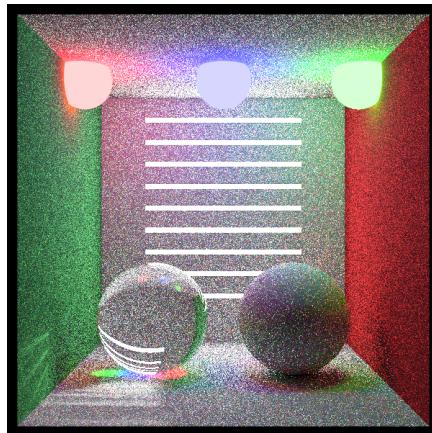


Figure 4.3:
100 Samples Per Pixel

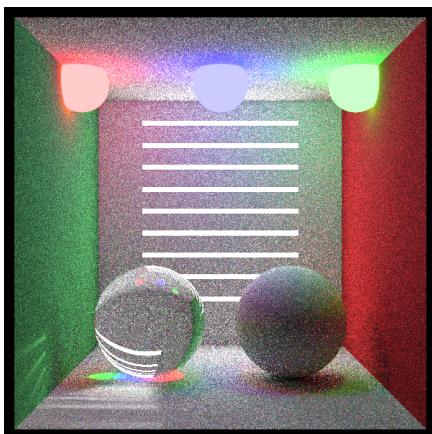


Figure 4.4:
250 Samples Per Pixel

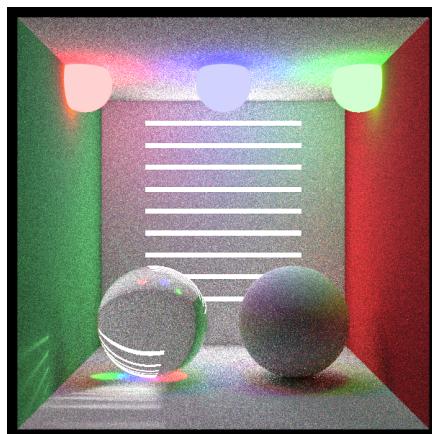


Figure 4.5:
500 Samples Per Pixel

Let's analyse the performance of the Ray Tracer on the CPU with different samples per pixel (SPP) settings. In the graph 4.6, we can see the rendering time of the Ray Tracer on CPU with different SPP settings. The values for GPU are not available yet.

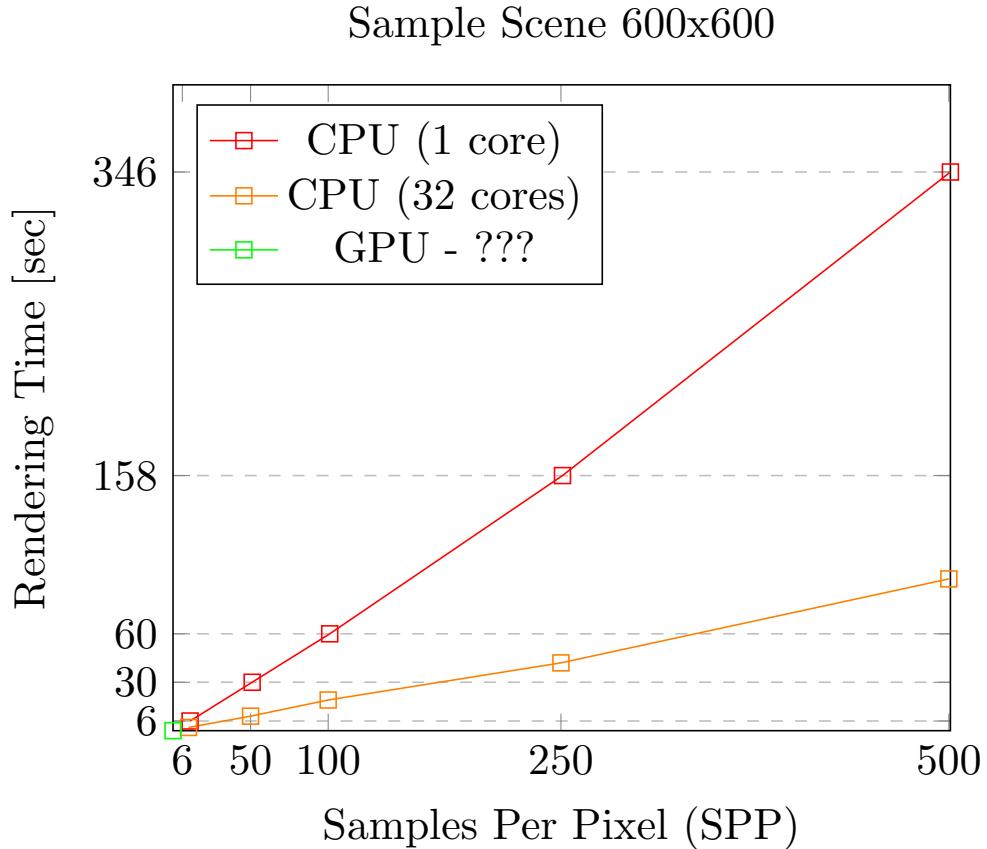


Figure 4.6: Dependency of rendering time on the number of SPP. The results will be different on different CPU hardware.

The graph indicates that the rendering time linearly depends on the number of SPPs. The more SPP, the more time is needed to render the image. The rendering time is also dependent on the number of CPU cores. The more cores, the less time is required to render the image. Since the dependencies are linear, we can conclude that moving the computations to the GPU will significantly reduce the rendering time because the modern RTX GPUs have much more RT (Ray-Tracing) cores specifically designed for Ray-Tracing computations.

4.3.4 Conclusion

The RayTracer on CPU is a simple implementation of the Ray-Tracing algorithm. It is not optimized for performance. The main goal of this implementation is to understand the Ray-Tracing algorithm and the relationship between classes and methods.

Using just the CPU, we can get very realistic images, but the rendering time is very high. To use the Ray-Tracing algorithm in real-time applications, we need to solve the performance issue. In figure 4.7, we can see the rendered image with 200'000 samples per pixel. It took 8 hours 44 minutes 53 seconds to render this image using a multicore (32) CPU renderer on Intel i9-14900HX CPU.

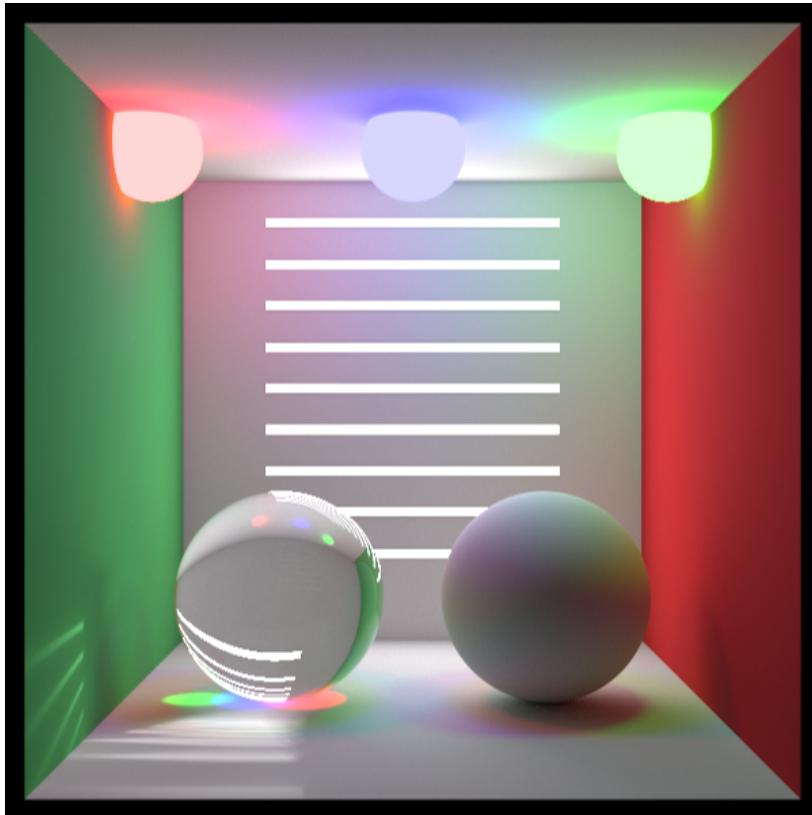


Figure 4.7: 200'000 Samples Per Pixel. Rendering time: 8 hours 44 minutes 53 second. CPU: Intel i9-14900HX (32 cores).

4.4 Ray-Tracing on GPU

After we created the Ray-Tracing framework on the CPU, we can now port it to the GPU. The GPU implementation is based on the Vulkan Ray Tracing extension. The Vulkan Ray Tracing extension is a set of functions and structures that allow us to create and manage Ray-Tracing pipelines on the NVIDIA RTX GPUs.

The implementation details are published on GitHub:

- Ray-Tracing Frameworks and Visualizers: RayTracingGPU

4.4.1 Five new shader types

In 2018, NVIDIA and Microsoft introduced the new DirectX Ray Tracing API (DXR) for ray tracing on the GPU, which was in 2020 adopted by Khronos Group in Vulkan as VK_KHR_ray_tracing extension. At a high level, DXR introduced three new concepts:

- **Ray Tracing Pipeline State Objects**: contain the compiled shader code that gets executed during a ray tracing dispatch.
- **Acceleration Structures**: contain the data structures used to accelerate ray tracing itself, i.e. the search for intersections between rays and scene geometry.
- **Shader Tables**: define the relationship between ray tracing shaders, their resources (textures, constants, etc), and scene geometry.

We're going to concentrate on the shader part. In the figure 4.8 we can see the schematic representation of the RTX execution model.

- **Ray Generation** shader defines how to start ray tracing. It's called the entry point to the ray tracer to generate the initial set of primary rays to be traced. The rays originate at the observer position according to the current camera settings.
- **Any Hit** shader runs once per any intersection found between a ray and an object, regardless of whether this intersection is the closest to the ray origin or not. It's used to determine transparency. It is called when the Intersection Shader finds a hit. The **Any Hit** shader decides if that intersection should be accepted or ignored. We're not going to use it here.
- **Intersection** shader defines how rays intersect geometry. When using non-triangle geometry, an intersection shader is required to compute ray intersections with the custom primitives. This isn't necessary for triangle meshes, as the ray-triangle intersection test is done in hardware. We will use it to detect intersections for procedural spheres in the scene.
- **Miss** shader defines behaviour when rays miss geometry. Usually, it returns the background colour or black colour if we don't have any background.
- **Close Hit** shader runs once per ray to shade the final (closest) hit. It's called with the information about the hit that happened closest to the

viewer. Typically, lighting is done here, or firing off new rays to handle shadows, reflections, and refractions. This is where we calculate the colour of the pixel.

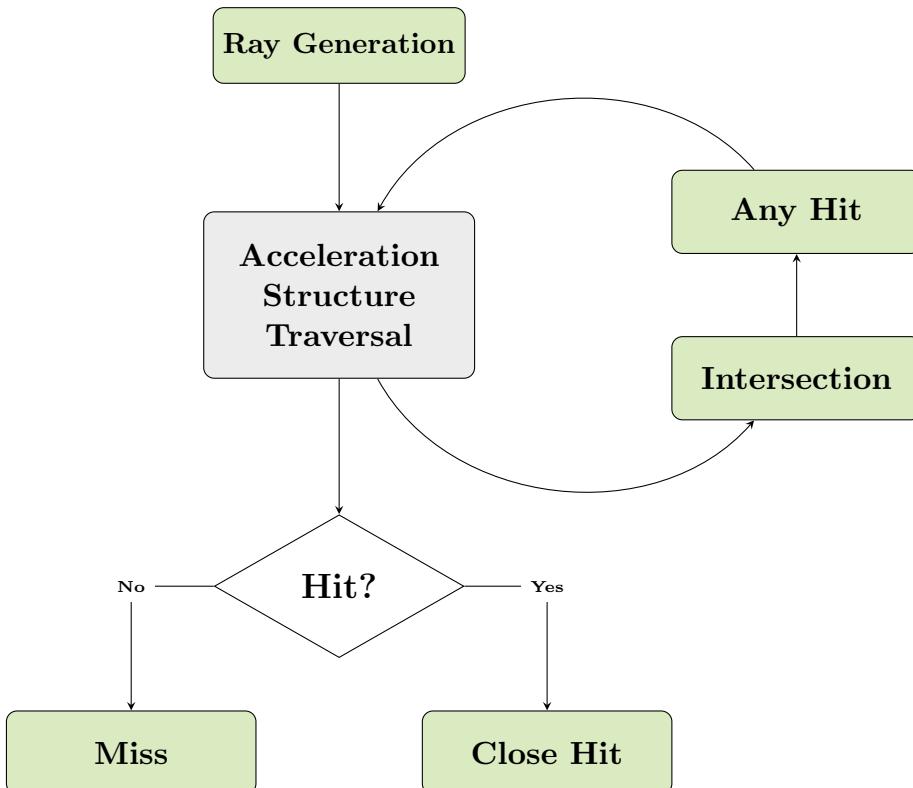


Figure 4.8: Five new Ray-Tracing shader types.

4.4.2 Mapping CPU implementation to new shader types

As we mentioned in the previous section, we're going to use 4 of 5 (we skip any hit because we don't have transparency in our scene) new shader types in our implementation. In the figure 4.9 we marked method which we implement in the ray tracing shaders.

Our camera class has two methods: *GetRay* and *GetColor*. We're going to use *GetRay* in the Ray Generation shader to generate the initial set of rays to be traced. The *GetColor* method will be used in the Close Hit shader to calculate the colour of the pixel, and in the Miss shader to define the behaviour when rays miss geometry.

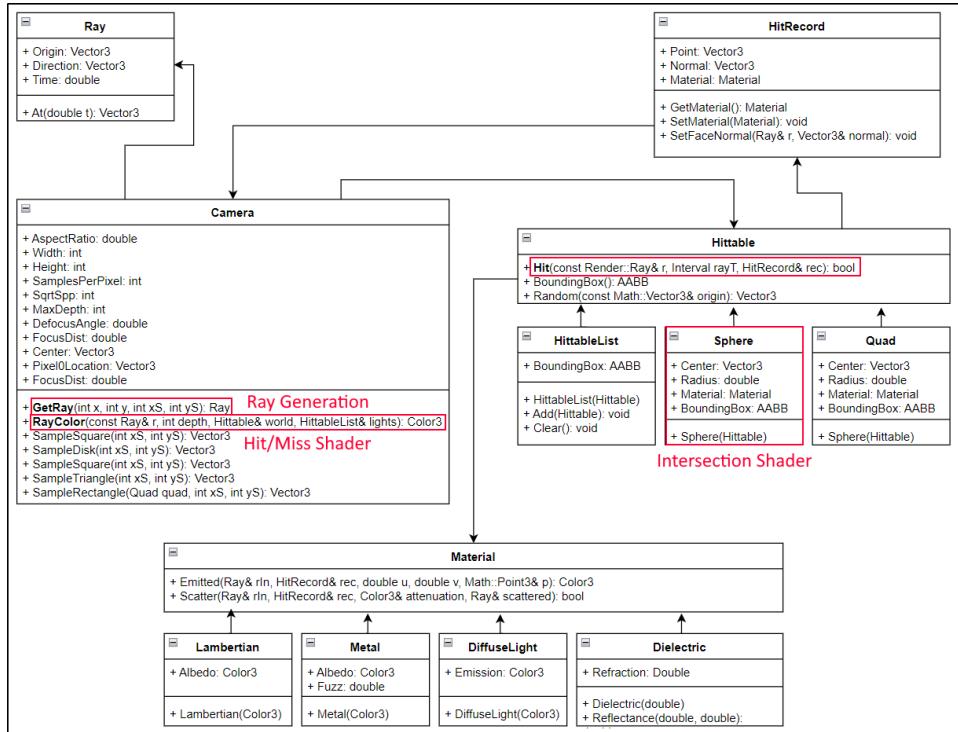


Figure 4.9: Ray Tracer on CPU Class Diagram with marked methods for GPU implementation

4.4.3 Vulkan Ray Tracing Pipeline

To build a ray tracing pipeline in Vulkan, we need to create a ray tracing pipeline object. The ray tracing pipeline object is created by calling the *vkCreateRayTracingPipelinesKHR* function. The function takes a logical device, a pipeline cache, the number of ray tracing pipeline objects to create, an array of ray tracing pipeline create info structures, and a pointer to an array of ray tracing pipeline objects. The ray tracing pipeline creates the info structure that is used to specify the shader groups, shader stages, the maximum recursion depth, and the layout of the pipeline.

We need to create the following shader groups:

- **Ray Generation Group.** The general type of shader group defines the ray generation shader. The ray generation shader is the entry point to the ray tracer to generate the initial set of rays to be traced. The shader stage is *VK_SHADER_STAGE_RAYGEN_BIT_KHR*, in terms of Vulkan bits, the shader file has the name *RayTracing.ugen* in the project. It also declares its access to the ray tracing output buffer image, and the ray tracing acceleration structure *topLevelAS*, bound

as an *accelerationStructureEXT*.

- **Miss Group.** Similar to the ray generation group, the general type of shader group defines the miss shader. The miss shader defines behaviour when rays miss geometry. The shader stage is *VK_SHADER_STAGE_MISS_BIT_KHR*, and the shader file has the name *RayTracing.rmiss* in the project. It's the simplest shader which returns the background colour or black colour if the scene doesn't have any background, in other words, no sky. It will write a constant colour into the ray payload *rayPayloadInEXT*.
- **Close Hit Group for Triangular Data.** The special type of shader group defines the closest hit shader for triangular data. The closest hit shader runs once per ray to shade the final hit. The shader stage is *VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR*, and the shader file has the name *RayTracing.rchit* in the project. It takes the colour properties of the material of the object, physical properties of the object (e.g. scattering or refraction), and calculates the resultant colour of the pixel. This shader is used for triangular data, i.e. loaded meshes. As the miss shader, it takes the ray payload *rayPayloadInEXT*. It also has a second input defining the intersection attributes *hitAttributeEXT* (i.e. the barycentric coordinates).
- **Close Hit Group for Non-Triangular Data.** As the previous group, the special type of shader group defines the closest hit shader for non-triangular data, in other words for objects defined procedurally (mathematically). Similar to the shader for triangular data, it calculates the resultant colour of the pixel and uses the same shader stage *VK_SHADER_STAGE_CLOSEST_HIT_BIT_KHR*. The main difference is that we need to define the intersection shader for objects of specific shape, i.e. sphere, cube, etc. In our case, we're going to use the sphere object. The shader files have the name *RayTracing.Procedural.rchit*, for hit shader, and *RayTracing.Procedural.rint* for intersection shader. The hit shader almost repeats the hit shader for triangular data, with additions for the specifics of the object (radius and centre of the sphere). The intersection shader is used to calculate the intersection of the ray with the object based on the mathematical formula of the object. It also takes the ray payload *rayPayloadInEXT* and has a second input defining the intersection attributes *hitAttributeEXT* in both, *rchit* and *rint*, shaders.

4.5 Light Sampling

After the initial implementation of the Ray-Tracing framework GPU, we can start implementing the light sampling algorithms. The light sampling

is a crucial part of the Ray-Tracing algorithm, as it allows us to calculate the direct light contribution to the scene. The direct light contribution is the light that comes directly from the light sources in the scene. The direct light contribution is the most significant part of the light in the scene, and it's essential to calculate it correctly to get a realistic image.

The first concept we need to introduce is the *probability density functions* (PDF). The PDF is a function that describes the probability of a random variable taking a specific value. In the context of Ray-Tracing, the PDF is used to describe the likelihood of a light ray hitting a specific point on the object's surface. The PDF is used to calculate the direct light contribution to the scene. The PDF is calculated using the light sampling algorithms.

To introduce the light sampling algorithms, we need to improve the following shaders and classes:

- **Scatter functions.** We need to change our scattering functions to support the light sampling. In the lambertian scattering model, we need to calculate the scattering PDF and pass the PDF value from the emitting geometry.
- **Ray Generation** shader. The main loop of the Ray-Tracing algorithm is located in this shader. We need to change it to process PDF values from the scattering functions and geometries.
- **Close Hit** shaders, for procedural and triangle meshes. These shaders are used to calculate the PDF value for a specific emitting light source. In the case of procedural spheres, we need to calculate the PDF value for the spherical surface. In the case of triangle meshes, we need to calculate the PDF value for the triangle or we may use algorithms for rectangle surfaces.

The basic implementation of the light sampling algorithms was taken from the *Ray Tracing: The Rest of Your Life* book by Peter Shirley [3]. We added the same implementations of the light sampling algorithms to the Ray-Tracing framework on the CPU and GPU. Additionally, on the Ray-Tracing for CPU was added the implementation of the light sampling algorithm presented [7] by James Arvo.

In the pictures 4.10 and 4.11 we can see the comparison of the scene rendered random scattering on a hemisphere with and without PDF distribution on the Ray-Tracing framework for GPU with two light sampling algorithms: solid angle of the sphere and stratified sampling of spherical triangles [7] on the same scene. The scene in the picture 4.11 appears lighter in the parts where the light sources are located, and the shadows are less pronounced.

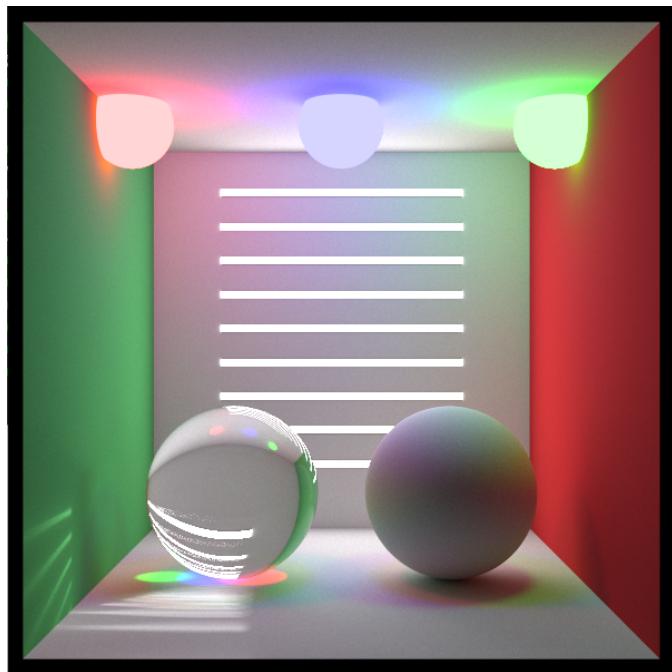


Figure 4.10: Test scene rendered using random scattering without PDF.

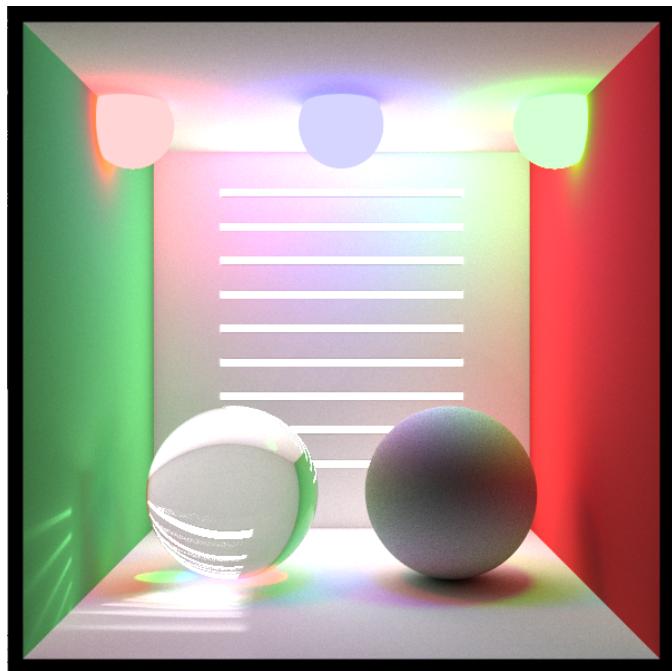


Figure 4.11: Test scene rendered using random scattering with PDF.

Chapter 5

Results and Discussion

In this chapter, we will compare rendering time and quality on CPU and GPU. The section 5.1 shows the quality of the ray-traced images rendered by CPU and GPU. The section 5.2 shows the performance comparison of the Ray-Tracing on CPU and GPU. We used the Cornell Box with multiple light sources as the test scene. The resolution is 600x600 pixels. SPP numbers are 10, 50, 100, 250, 500, 1000, 2000, 5000, 10000, and 20000.

In figure 5.1 the scene parameters are shown with the resultant image rendered by the Ray-Tracing framework using 200'000 sample per pixel.

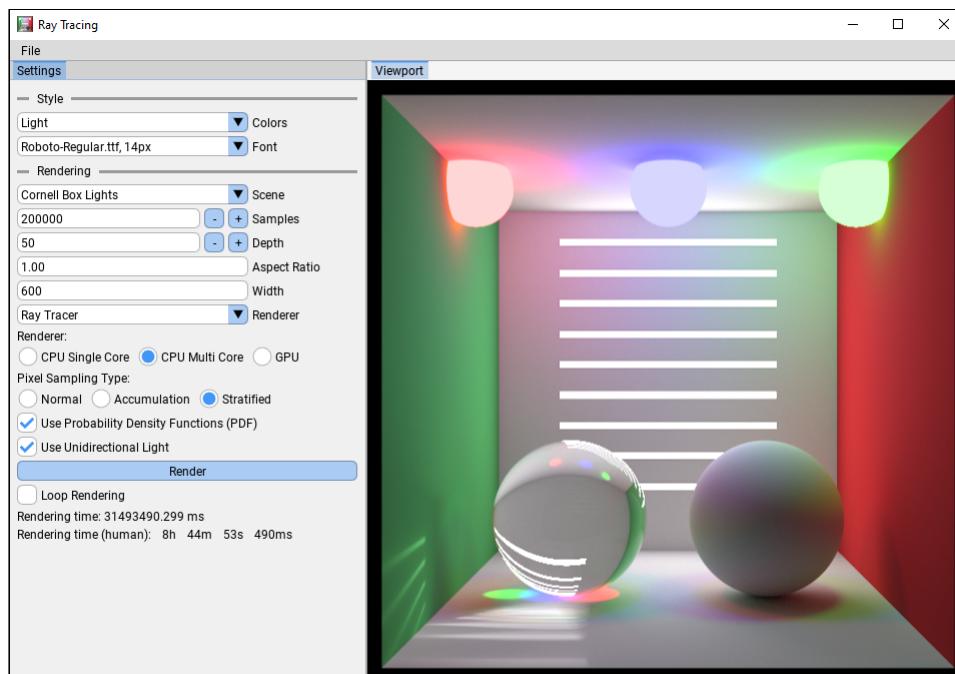


Figure 5.1: Cornell Box with multiple light sources demo scene.

5.1 CPU vs. GPU Ray-Tracing Image Quality

In the tables 5.1, 5.2, and 5.3, we can see the quality of the ray-traced images rendered by CPU and GPU. The quality of the images rendered by the GPU is comparable with the quality of the images rendered by the CPU, so ray tracing on the GPU doesn't affect the quality of the images too much.

There is a noticeable difference in refracted light, but it's because the CPU implementation uses the probability density function (PDF) while the GPU implementation doesn't. This doesn't affect the original goal of the project, which is to compare the generic performance of the ray tracing on the CPU and GPU.

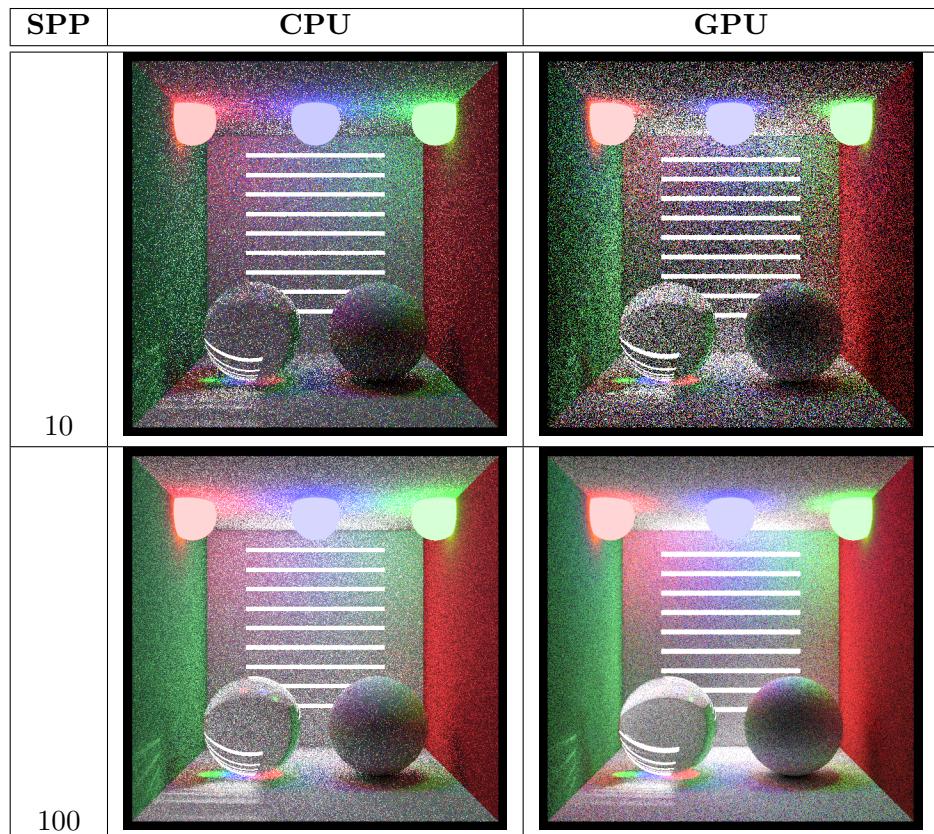


Table 5.1: The quality of the ray traced images rendered by CPU and GPU (part 1).

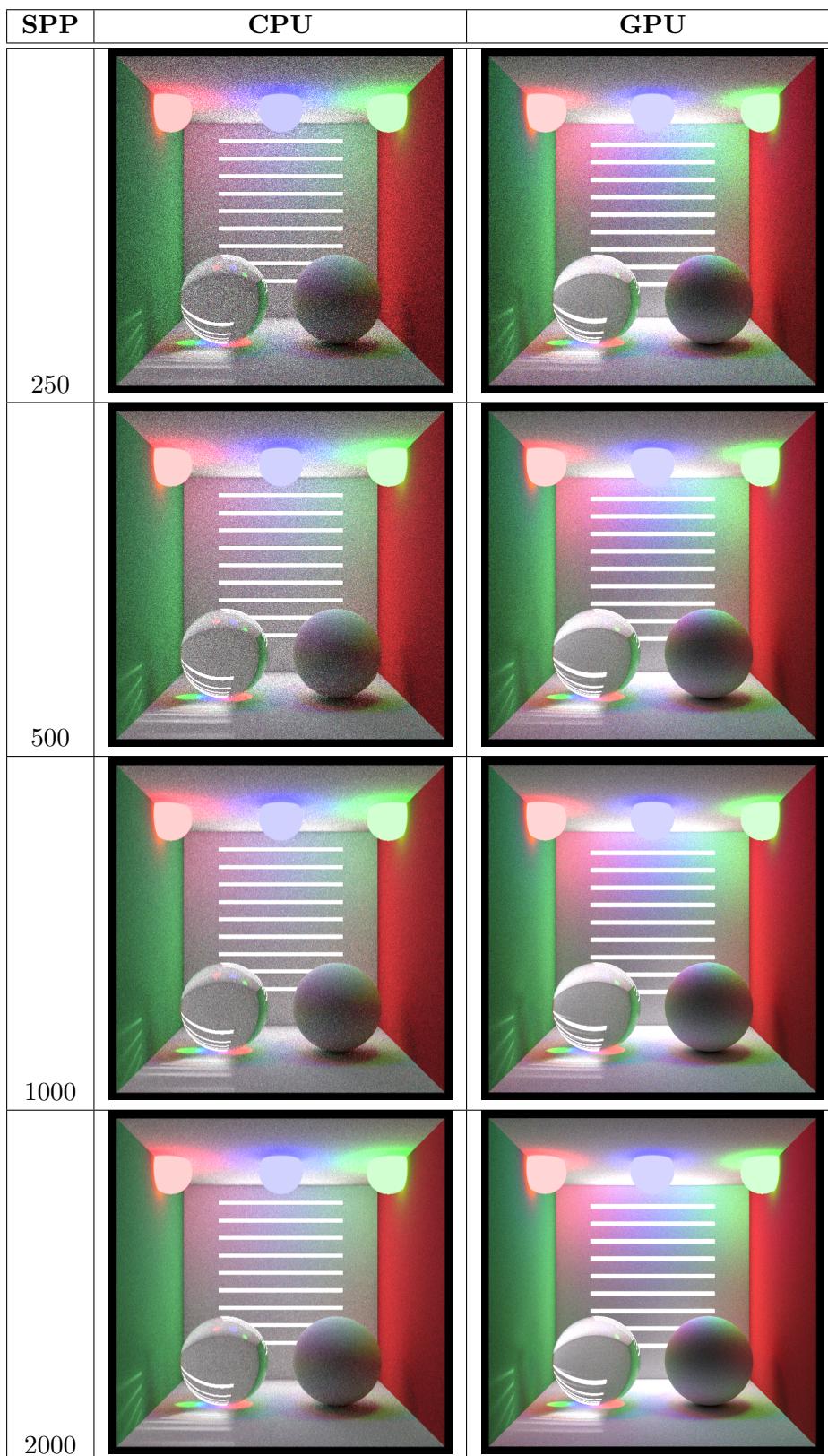


Table 5.2: The quality of the ray traced images rendered by CPU and GPU (part 2).

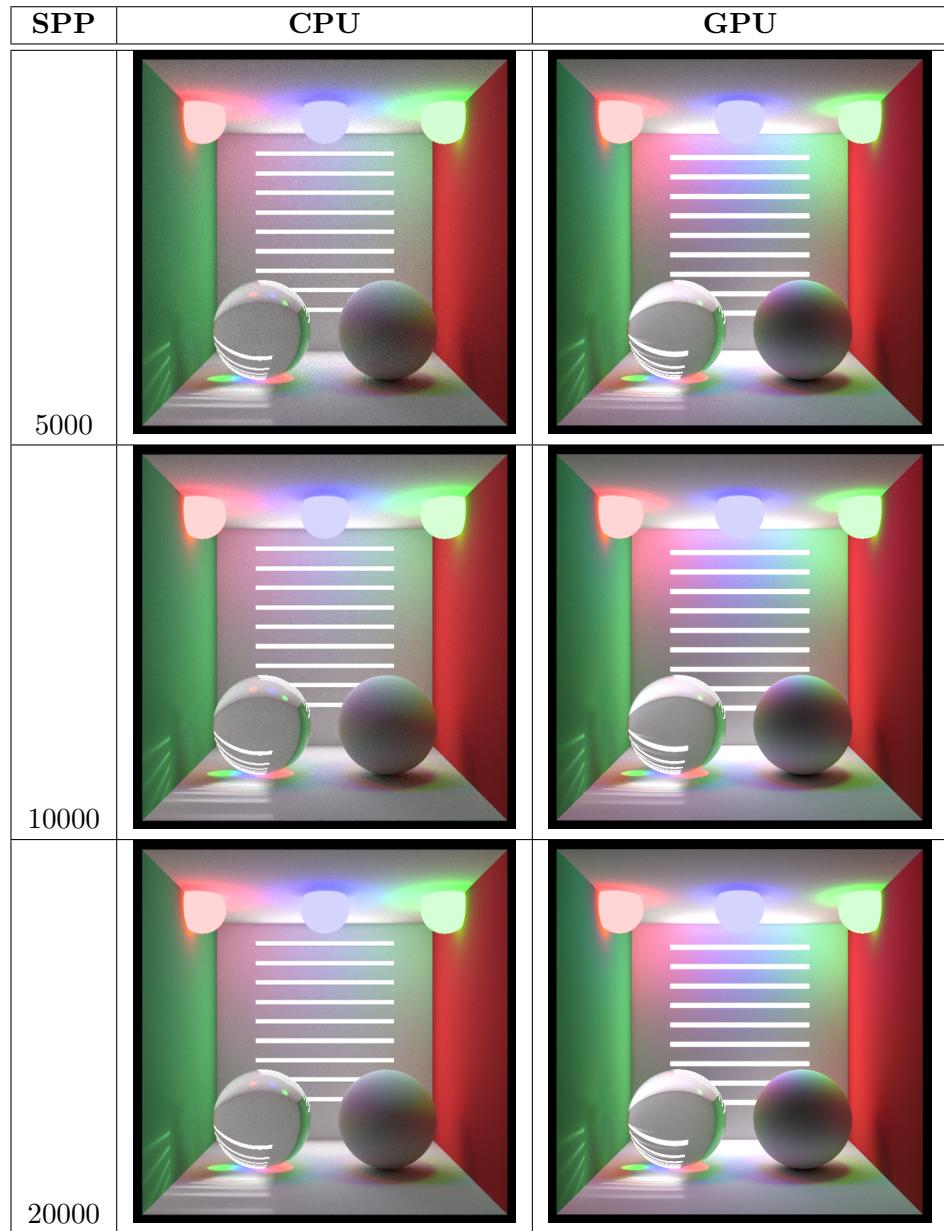


Table 5.3: The quality of the ray traced images rendered by CPU and GPU (part 3).

5.2 CPU vs. GPU Ray-Tracing Performance

The table 5.4 shows the performance comparison of the Ray-Tracing on CPU and GPU. The rendering time of the Ray Tracer on the GPU is significantly lower than the rendering time of the Ray Tracer on the CPU. The GPU is faster than the CPU in all cases and allows us to try theories much faster than ray tracing on the CPU.

Samples per pixel	Rendering Time CPU	Rendering Time GPU
10	2'000 ms	10 ms
50	9'000 ms	44 ms
100	19'000 ms	83 ms
250	42'000 ms	250 ms
500	94'000 ms	546 ms
1000	146'000 ms	1'092 ms
2000	358'000 ms	2'310 ms
5000	888'000 ms	5'730 ms
7500	1'313'000 ms	7'900 ms
10000	1'617'000 ms	11'160 ms
15000	2'370'000 ms	15'800 ms
20000	3'311'000 ms	20'440 ms

Table 5.4: Ray Tracing performance comparison.

According to our evaluation, we can reach a nearly real-time rendering experience, 25 frames per second, using around 25–50 samples per pixel, depending on the scene complexity. But using such a few samples per pixel, we can't produce images of good quality and completely noiseless. Using a larger number of samples per pixel allows us to produce perfect realistic images spending much less time on rendering, but the usability, even on GPU, is still non-real-time. At that point, we need to start using denoising algorithms and sample accumulation. This would allow us to improve the quality of the resultant image without noticeable performance loss, using some quality trade-offs.

The graph 5.2 visualizes the dependency of rendering time on the number of samples per pixel.

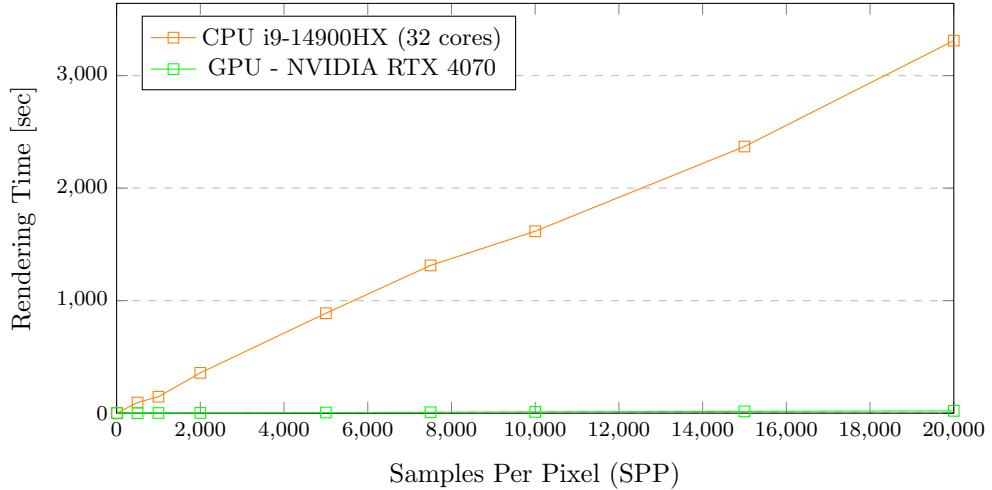


Figure 5.2: Dependency of rendering time on the number of SPP. Sample Scene 600×600 . Ray-Tracing depth 50 hits.

On the graph 5.2 we can see the immediate drop in rendering time on the GPU for the same number of samples per pixel. But still, we can see that the rendering time on the GPU grows linearly with the number of SPP, so for the very high number of SPP, we will still struggle with the rendering time on the GPU. Improvements on the hardware side can help to reduce the rendering time even more, but it is not a "silver bullet" for the high number of SPPs coupled with very complex scenes.

5.3 Project Implementation and Availability

The source code of the Ray-Tracing frameworks on CPU and GPU is available on GitHub under the Open Source MIT Licence. The project is compilable and runnable on Windows, Linux, and macOS using CMake and VCPKG as the package manager. The project is well-organized and follows the implementation of the Ray-Tracing in One Weekend book series by Peter Shirley [1] [2] [3].

The CPU implementation can be run on modern CPU hardware without any limitations. The GPU implementation requires an RTX-compatible GPU and driver. The appendix B ([Usage Instructions](#)) describes the basic features of the Ray-Tracing frameworks and how to run them. The GPU implementation is memory-sensitive, so the resolution and the number of

samples per pixel in one run should be adjusted according to the available GPU memory. The sample accumulation mitigates the issues with memory.

The appendix C provides the full text of the MIT License under which the project is published. Anyone can use the project for any purpose, including commercial purposes, without any limitations. In case of any issues or questions, the author can be contacted via email **d.k.ivanov (at) live.com**.

The source code and the project related artefacts are available at the following links:

- [Ray-Tracing Framework source code](#)
- [Ray-Tracing Framework demonstration video](#)
- [Ray-Tracing on GPU GitHub project](#)
- [Ray-Tracing on GPU project roadmap](#)

Chapter 6

Conclusions and future work

In this last chapter, we will discuss the conclusions and results reached, as well as the potential future improvements that could be made in the projects to make them more mature and usable.

6.1 Conclusions

As a result of our work, we created two independent software systems for testing various Ray-Tracing technologies. The first system is a CPU-based Ray-Tracing framework, and the second system is a GPU-based Ray-Tracing framework using Vulkan Ray-Tracing Extensions. The predictions we made about potential improvements in the ray tracing performance were confirmed by the results of our work. The GPU-based ray tracing framework is significantly faster than the CPU-based ray tracing framework, allowing for real-time rendering of complex 3D scenes.

We figured out that the quality of the ray-traced images rendered by the GPU is comparable with the quality of the images rendered by the CPU. The differences in the current project are noticeable, but the new RTX Ray-Tracing technologies allow porting almost any Ray-Tracing algorithm to the GPU with a significant performance improvement to specifically designed Ray-Tracing shaders.

During our work, we discovered that the GPU-based Ray-Tracing is very mature technology that can be used in real-world light simulations and collision detection systems.

6.2 Future Work

The present work opens ways to further development of the presented Ray-Tracing frameworks and additional features. The following list presents some of the future work that can be done to improve the Ray-Tracing evaluation framework.

Further development plans:

1. Extend the Ray-Tracing framework with more functionality, including UI and better OOP structure.
2. Merge the Ray-Tracing on CPU framework with the Ray-Tracing on GPU framework and refactor architecture of shader functions.
3. Enhance robustness and convergence rates of stratified sampling algorithms.
4. More shapes in sampling algorithms and scene setups.
5. Add ray tracing acceleration structures to support sample grouping in the world scene.
6. Add real-time denoisers support.
7. Fallback to NVIDIA OptiX older structures to support older GPU hardware.
8. Add support for spatio-temporal resampling for real-time ray tracing (RTX Only).
9. Add support for Deep Learning Super Sampling (DLSS) for real-time ray tracing (RTX Only).
10. Split the Ray-Tracing evaluation framework into modules for partial usage in education.
11. Add Material Definition Language (MDL) support and interactive material selection.
12. Add interactive mode to adjust the camera position, viewport configuration, and multiple viewports.
13. Add interactive scene editor to add, remove, and modify the scene objects.
14. Summarize the work done into an educational step-by-step workflow.
15. Integrate concepts from Physically Based Rendering (PBR) [30] into the Ray-Tracing evaluation framework.

Bibliography

- [1] P. Shirley, T. D. Black, and S. Hollasch, “Ray tracing in one weekend,” August 2023. <https://raytracing.github.io/books/RayTracingInOneWeekend.html>.
- [2] P. Shirley, T. D. Black, and S. Hollasch, “Ray tracing: The next week,” August 2023. <https://raytracing.github.io/books/RayTracingTheNextWeek.html>.
- [3] P. Shirley, T. D. Black, and S. Hollasch, “Ray tracing: The rest of your life,” August 2023. <https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html>.
- [4] A. Appel, “Some techniques for shading machine renderings of solids,” *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference on - AFIPS ’68 (Spring)*, 1968.
- [5] T. Whitted, “An improved illumination model for shaded display,” *Communications of the ACM*, vol. 23, pp. 343–349, 1980.
- [6] P. E. Moreau and P. Clarberg, “Importance sampling of many lights on the gpu,” *Ray Tracing Gems*, pp. 255–283, 2019.
- [7] J. Arvo, “Stratified sampling of spherical triangles,” *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH ’95*, 1995.
- [8] C. P. Ureña, M. Fajardo, and A. King, “An area-preserving parametrization for spherical rectangles,” *Computer Graphics Forum*, vol. 32, pp. 59–66, 2013.
- [9] C. P. Ureña and I. Georgiev, “Stratified sampling of projected spherical caps,” *Computer Graphics Forum*, vol. 37, pp. 13–20, 2018.
- [10] J. Peddie, *Ray Tracing: A Tool for All*. Cham: Springer International Publishing, 2019.
- [11] E. Haines and T. Akenine-Möller, eds., *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Berkeley, CA: Apress, 2019.
- [12] E. Haines, *Essential ray tracing algorithms*, pp. 33–77. GBR: Academic Press Ltd., July 1989.

- [13] G. Szauer, *Game Physics Cookbook*. UK: Packt Publishing Ltd., 2017.
- [14] T. Möller and B. Trumbore, “Fast, minimum storage ray-triangle intersection,” *Journal of Graphics Tools*, vol. 2, p. 21–28, Jan. 1997.
- [15] D. Baldwin and M. Weber, “Fast ray-triangle intersections by coordinate transformation,” *Journal of Computer Graphics Techniques (JCGT)*, vol. 5, p. 39–49, Sept. 2016.
- [16] D. Kalra and A. H. Barr, “Guaranteed ray intersections with implicit surfaces,” *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, p. 297–306, July 1989.
- [17] J. Arvo and D. Kirk, *A survey of ray tracing acceleration techniques*, p. 201–262. GBR: Academic Press Ltd., July 1989.
- [18] D. Ivanov, “Ray-tracing framework,” September 2024. <https://github.com/d-k-ivanov/ray-tracing>.
- [19] W. Barth and W. Stürzlinger, “Efficient ray tracing for bezier and b-spline surfaces,” *Computers and Graphics*, vol. 17, p. 423–430, July 1993.
- [20] A. Reshetov, *Cool Patches: A Geometric Approach to Ray/Bilinear Patch Intersections*, p. 95–109. Berkeley, CA: Apress, 2019.
- [21] J. T. Kajiya, “The rendering equation,” *ACM SIGGRAPH Computer Graphics*, vol. 20, pp. 143–150, August 1986.
- [22] E. Veach and L. J. Guibas, “Optimally combining sampling techniques for monte carlo rendering,” *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '95*, 1995.
- [23] P. Shirley, C. Wang, and K. Zimmerman, “Monte carlo techniques for direct lighting calculations,” *ACM Transactions on Graphics*, vol. 15, pp. 1–36, 1996.
- [24] P. Martinsen, J. Blaschke, R. Künnemeyer, and R. B. Jordan, “Accelerating monte carlo simulations with an nvidia® graphics processor,” *Computer Physics Communications*, vol. 180, pp. 1983–1989, 2009.
- [25] P. Shirley, “Ray tracing,” in *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, (New York, NY, USA), pp. 2–es, Association for Computing Machinery, July 2005.
- [26] M. Christen, “Ray tracing on gpu,” *Master’s thesis, Univ. of Applied Sciences Basel (FHBB), Jan*, vol. 19, 2005.
- [27] S. G. Parker, H. Friedrich, D. Luebke, K. Morley, J. Bigler, J. Hoberock, D. McAllister, A. Robison, A. Dietrich, G. Humphreys, M. McGuire, and M. Stich, “Gpu ray tracing,” *Communications of the ACM*, vol. 56, pp. 93–101, May 2013.
- [28] D. V. Antwerpen, “Unbiased physically based rendering on the gpu,” Master’s thesis, June 2011.

- [29] S. Laine, T. Karras, and T. Aila, “Megakernels considered harmful: waveform path tracing on gpus,” in *Proceedings of the 5th High-Performance Graphics Conference*, HPG ’13, (New York, NY, USA), p. 137–143, Association for Computing Machinery, July 2013.
- [30] M. Pharr, W. Jakob, and G. Humphreys, *Physically Based Rendering: From Theory to Implementation*. Cambridge, MA, USA: The MIT Press, 4th ed., 2023.
- [31] A. Knoll, R. K. Morley, I. Wald, N. Leaf, and P. Messmer, *Efficient Particle Volume Splatting in a Ray Tracer*, pp. 533–541. Berkeley, CA: Apress, 2019.
- [32] D. Meister, J. Boksansky, M. Guthe, and J. Bittner, “On ray reordering techniques for faster gpu ray tracing,” in *Symposium on Interactive 3D Graphics and Games*, I3D’20, (New York, NY, USA), pp. 1–9, Association for Computing Machinery, May 2020.

Appendix A

Compilation

A.1 Compilation on Windows

- Clone the project from
<https://github.com/d-k-ivanov/ray-tracing>
- Install Visual Studio from
<https://visualstudio.microsoft.com/downloads/>
- Install Vulkan SDK from
<https://vulkan.lunarg.com/sdk/home>
- Install VCPKG from
<https://vcpkg.io/en/getting-started.html>
- Add VCPKG folder to PATH:
 - Option 1:
Command Line tools

```
1 REM Assuming that VCPKG cloned and bootstrapped in c:\src\vcpkg
2 REM setx for the global environment, set for the local
3 setx PATH c:\src\vcpkg;%PATH%
4 set PATH c:\src\vcpkg;%PATH%
```

- Option 2:
PowerShell

```
1 # Assuming that VCPKG cloned and bootstrapped in c:\src\vcpkg
2 [Environment]::SetEnvironmentVariable("PATH", "c:\src\vcpkg;${PATH}", "Machine")
3 Set-Item -Path Env:PATH -Value "c:\src\vcpkg;${PATH}"
```

- Option 3:
Manually in **System Properties → Environment Variables**

- Navigate to the folder with the project
- Run *build.bat*
- Open *ray-tracing-gpu.sln* in Visual Studio to work with the source code
- Run Ray-Tracin on CPU Framework
build/RayTracingCPU/Release/ray-tracing-cpu.exe
- Run Ray-Tracin on GPU Framework
build/RayTracingGPU/Release/ray-tracing-gpu.exe

A.2 Compilation on Linux

- Clone the project from
<https://github.com/d-k-ivanov/ray-tracing>
- Install Vulkan SDK from
<https://vulkan.lunarg.com/sdk/home>
- Install VCPKG from
<https://vcpkg.io/en/getting-started.html>
- Add VCPKG folder to System PATH

– Option 1:
Temporary local environment

```
1 # Assuming that VCPKG cloned and bootstrapped in ~/vcpkg
2 export PATH="~/vcpkg;${PATH}"
```

– Option 2:
Local environment and Bash profile

```
1 # Assuming that VCPKG cloned and bootstrapped in ~/vcpkg
2 export PATH="~/vcpkg;${PATH}"
3 echo 'export PATH="~/vcpkg;${PATH}"' >> ~/.bashrc
```

- Navigate to the folder with the project
- Run build.sh
- Run Ray-Tracin on CPU Framework
build/RayTracingCPU/Release/ray-tracing-cpu
- Run Ray-Tracin on GPU Framework
build/RayTracingGPU/Release/ray-tracing-gpu

Important Note: the VCPKG requests installation of additional packages.

Appendix B

Usage Instructions

B.1 Ray-Tracing Framework on CPU

The figure B.1 presents the CPU version of the Ray-Tracing framework. The compiled application can be executed in the local environment without additional configuration.

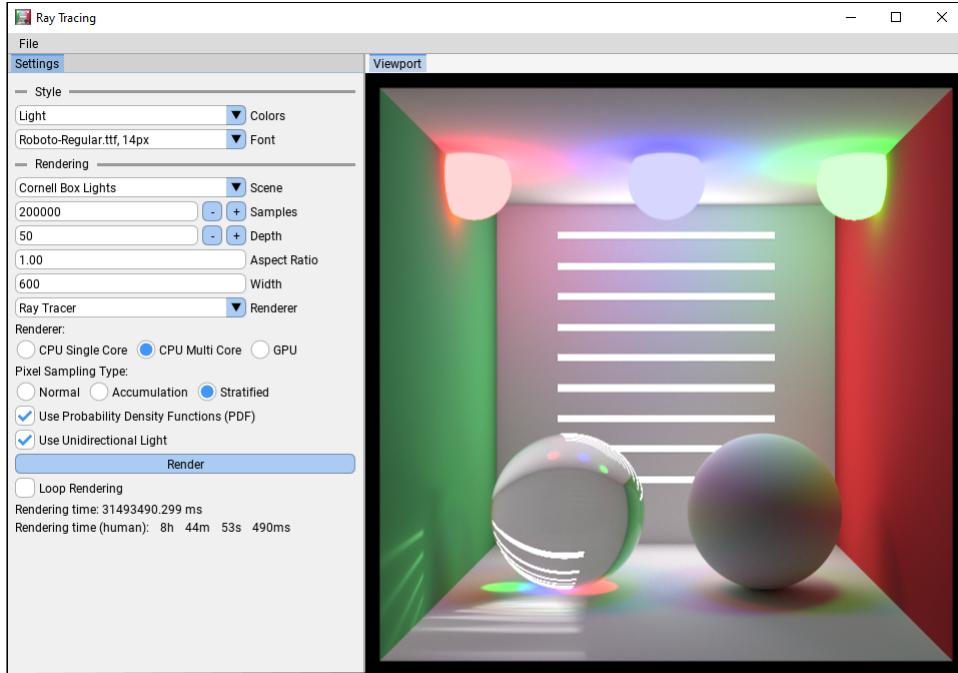


Figure B.1: The initial implementation of the Ray-Tracing framework on CPU.

UI Parameters The application provides controls to change the colour mode (dark or light) and the UI font.

Rendering Parameters is the main control panel for the rendering process. The following parameters can be adjusted:

- **Scene.** Selection of the scene to render. The application provides the following scenes:
 - RTWeekOneDefaultScene: Ray Tracing in one weekend default scene.
 - RTWeekOneTestScene: Ray Tracing in one weekend test scene.
 - RTWeekOneFinalScene: Ray Tracing in one weekend final scene.
 - RTWeekNextDefaultScene: Ray Tracing next week default scene.
 - RTWeekNextRandomSpheresScene: Ray Tracing next week random spheres scene.
 - RTWeekNextTwoSpheresScene: Ray Tracing next week two spheres scene.
 - RTWeekNextEarthScene: Ray Tracing next week earth scene.
 - RTWeekNextTwoPerlinSpheresScene: Ray Tracing next week two perlin spheres scene.
 - RTWeekNextQuadsScene: Ray Tracing next week quads scene.
 - RTWeekNextSimpleLightScene: Ray Tracing next week simple light scene.
 - RTWeekNextCornellBoxScene: Ray Tracing next week Cornell-Box scene.
 - RTWeekNextCornellSmokeScene: Ray Tracing next week Cornell Box with smoke scene.
 - RTWeekNextFinalScene: Ray Tracing next week final scene.
 - RTWeekRestACornellBoxScene: Ray Tracing rest of your life Cornell Box scene.
 - RTWeekRestBCornellBoxMirrorScene: Ray Tracing rest of your life Cornell Box with mirror scene.
 - RTWeekRestCCornellBoxGlassScene: Ray Tracing rest of your life Cornell Box with glass scene.
 - CornellBoxLightsScene: Cornell Box with multiple light sources scene.
 - WhiteSpheresScene: Scene with multiple white spheres to show BVH performance.
- **Number samples per pixel.** Input field to set the number of samples per pixel.

- **Ray-Tracing depth.** Input field to set the Ray-Tracing depth.
- **Aspect ratio.** Input field to set the aspect ratio. The default value is 1.00.
- **Image width.** Input field to set the image width. The default value is 600.
- **Rendering engine.** Selection of the rendering engine. The application provides the following rendering engines:
 - Single Core CPU.
 - Multi Core CPU.
 - GPU. Not working yet, reserved for future implementation.
- **Pixel sampling type.** Selection of the pixel sampling type. The application provides the following pixel sampling types:
 - Normal. The default pixel sampling type without any additional sampling or random distribution improvements.
 - Accumulation. The same as the Normal pixel sampling type, with the accumulation of the samples.
 - Stratified. The pixel sampling type with stratified sampling.
- **Probability density function.** Flag to enable the probability density function (PDF) for the pixel sampling type. The default value is true.
- **Unidirectional light.** Flag to enable the unidirectional light. The default value is true.
- **Loop rendering.** Flag to enable the loop rendering. The default value is false. Useful for accumulation rendering.

Additionally, the application provides the rendering time for the last image.

B.2 Ray-Tracing Framework on GPU

The figure B.2 presents the GPU version of the Ray-Tracing framework. The compiled application can be executed in the local environment without additional configuration. The only requirement is the GPU with Ray-Tracing support (RTX): NVIDIA GeForce RTX 20xx or newer otherwise the application will not start.

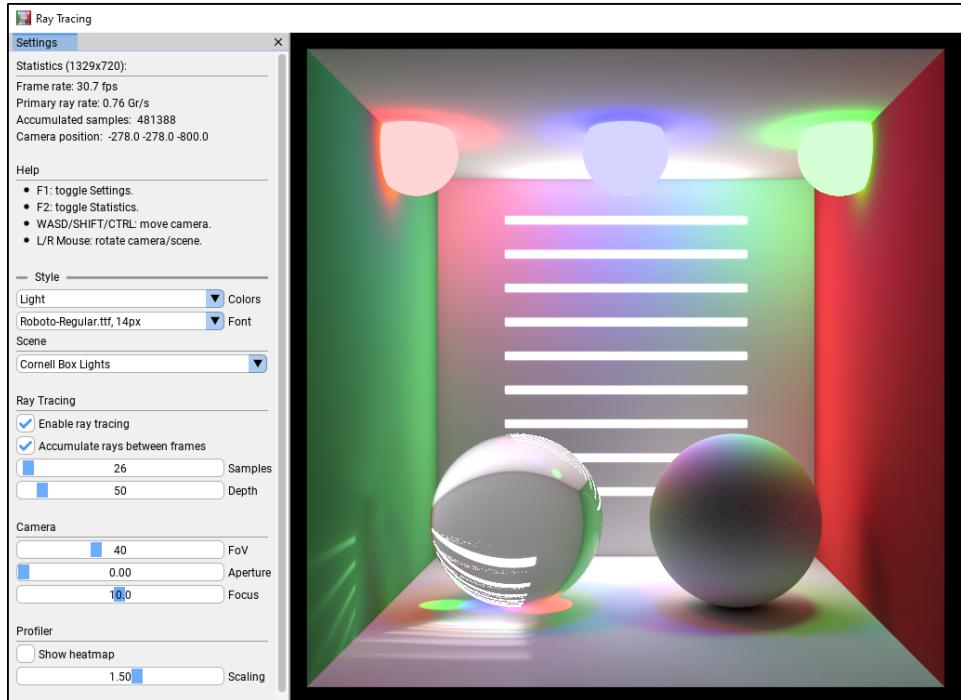


Figure B.2: The initial implementation of the Ray-Tracing framework on GPU.

UI Parameters The application provides controls to change the colour mode (dark or light) and the UI font.

Scene Selection of the scene to render. The application provides the following scenes:

- Cube and Spheres: Scene with a cube and multiple spheres.
- Cornell Box Lights: Cornell Box with multiple light sources scene.
- Ray Tracing in One Weekend: Ray Tracing in One Weekend final scene.
- Planets in One Weekend: Ray Tracing in One Weekend final scene with planetary textures for larger spheres.
- Lucy in One Weekend: Ray Tracing in one weekend final scene with Lucy model instead of larger spheres.
- Cornell Box: Cornell Box scene.
- Cornell Box and Lucy: Cornell Box scene with Lucy model.

Ray Tracing . The application provides the following parameters to adjust the Ray-Tracing process:

- **Enable ray tracing.** Flag to enable the ray tracing. The default value is true. If false, the application will only render models on the scene using the rasterization.
- **Accumulate rays between frames.** Flag to enable the accumulation of samples of rays between frames. The default value is true.
- **Samples.** The number of samples per pixel. The default value is 25.
- **Depth.** The Ray-Tracing depth. The default value is 50.

Camera . The application provides parameters to adjust the camera: **The field of view**, **Aperture**, and **Focus**.

Profiler . The application provides the parameters to adjust and enable heatmaps.

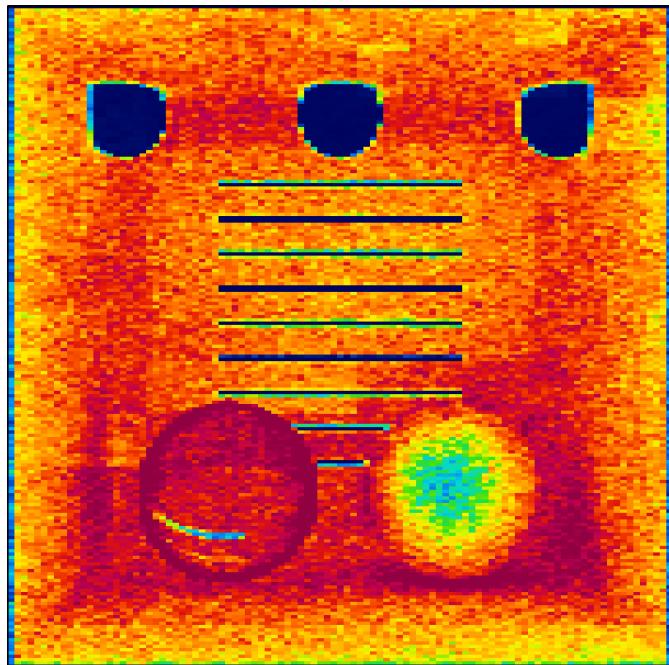


Figure B.3: Heatmaps for Cornell Box with multiple light sources scene.

Appendix C

License

The project is published under the MIT License. Free to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software. The only requirement is to include the license and copyright notice.

MIT License

Copyright (c) 2023 Dmitry Ivanov

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.