**YEDITEPE UNIVERSITY**
**FACULTY OF ENGINEERING**

# A Real Time Ray Tracer Using Microsoft's DirectX12's DXR API

Utku GÖKALP

**GRADUATION PROJECT REPORT**
Department of Electrical and Electronics Engineering

**Supervisor**
Prof. Dr. Cem ÜNSALAN

ISTANBUL, 2025


# YEDITEPE UNIVERSITY
## FACULTY OF ENGINEERING

**A Real Time Ray Tracer Using Microsoft's DirectX12's DXR API**

**by**

**Utku GÖKALP**

**July 15, 2025, Istanbul**

**DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING**

**YEDITEPE UNIVERSITY**

Signature of Author(s) ……

Department of Electrical and Electronics Engineering

Certified By ……

Project Supervisor, Department of Electrical and Electronics Engineering

Accepted By ……

Head of the Department of Electrical and Electronics Engineering

# ACKNOWLEDGEMENTS

I would like to sincerely thank İmpalko Kimyevi Maddeler Metal İthalat Irhaa Sağlık ve Danışmalık Hizmetleri Limited Şirketi for sponsoring this project and providing the RTX 4060 Ti GPU, which made it possible to develop and test a real-time ray tracing system on modern hardware. Their support enabled the practical application of advanced rendering techniques and ensured that the project could meet its performance goals.

I would also like to thank the open-source community, particularly the maintainers of the DirectX-Graphics-Samples GitHub repository, which served as a foundational reference for DX12 and DXR integration. The complete source code for this project is publicly available at [1].

Special thanks go to my supervisor, whose feedback helped guide the scope and direction of the project.

Finally, I would like to thank the developers of tools and libraries such as ImGui, GLM, and nv_helpers_dx12, which greatly simplified implementation and debugging during development.

**July, 2025**

**Utku GÖKALP**

# Table of Contents

## ABSTRACT

This study presents the design and implementation of a real-time ray tracer using DirectX 12 (DX12) and DirectX Raytracing (DXR), targeting modern GPU architectures. The goal of the project is not to introduce new algorithms but to explore how existing real-time ray tracing technologies could be used to build a functional and extensible rendering system. The implementation includes features such as loading .obj model files in runtime, interacting with the renderer through a UI, and applying a simplified physically based rendering (PBR) material system.

The project was developed in C++17 using Visual Studio 2022, and tested on RTX-capable hardware (NVIDIA RTX 4060 Ti). Results show that the renderer achieved comparable output to Microsoft's DXR benchmark samples [2], while maintaining real-time performance. The system supports dynamic camera control, basic lighting and shadows, and real-time performance profiling via a built-in UI. Although the PBR implementation was functional, certain parameter edge cases revealed limitations in the shading accuracy. Nevertheless, the project successfully demonstrates how DX12 and DXR API along with compatible GPU hardware can be used to achieve realistic, interactive graphics through ray tracing.

Real-time ray tracing is a valuable technique used in various domains such as optical simulation, electromagnetic modeling, CAD/CAE visualization, and sensor system design. For electrical and electronics engineers, understanding and developing ray tracing pipelines provides the foundational necessary skills for building software such as visualization tools or optics simulators used in modern engineering applications.

## LIST OF SYMBOLS

**P**: Point along the ray at some distance *t*

**O**: The origin of the ray

**R**: The direction vector of the ray

*t*: Parameter that allows for selection of various points along the ray

*A*: x component of a plane's normal vector.

*B*: y component of a plane's normal vector.

*C*: z component of a plane's normal vector.

D: Offset of a plane from the origin

**N**: Normal vector of a plane.

$v_0$: First vertex of a triangle.

$v_1$: Second vertex of a triangle.

$v_2$: Third vertex of a triangle.

$\boldsymbol{E_0}$: First edge of a triangle

$\boldsymbol{P_0}$: Direction from first vertex of the triangle to ray-plane intersection point

$\boldsymbol{C_0}$: Cross product of $\boldsymbol{E_0}$ and $\boldsymbol{P_0}$

## ABBREVIATIONS

**DX12:** DirectX 12

**DXR:** DirectX Ray Tracing

**CAD:** Computer Aided Design

**CAE:** Computer Aided Engineering

**PBR:** Physically Based Rendering

**BLAS:** Bottom-Level Acceleration Structure

**TLAS:** Top-Level Acceleration Structure

**SBT:** Shader Binding Table

**FPS:** Frames Per Second

# LIST OF FIGURES

# LIST OF TABLES

# 1. INTRODUCTION

Computer graphics is a mature yet continuously evolving field with extensive real-world applications. It makes (extensive) use of both computer hardware and software to create visuals for entertainment, education, scientific research and engineering. Among its practical uses are data visualization, computer-aided design (CAD), virtual reality, art and interactive media [3].

A fundamental concept in computer graphics is rendering - the process of generating images from 3D model data. This typically involves defining geometry using what's called the vertex and index buffers, where a vertex buffer describes points in a 3D space (and potentially more) and an index buffer defines how these points are connected to form shapes and surfaces. Two dominant rendering techniques are used today: rasterization and ray tracing.

Rasterization has long been the industry standard for real-time rendering. It works by projecting 3D geometry onto a 2D screen and determining which pixels correspond to the rendered objects. While this method is fast and well-suited for interactive applications such as video games, it's not physically accurate. Achieving photorealism with rasterization often requires additional approximations and hacks such as normal mapping, screen-space reflections or ambient occlusion and these hacks often have their noticeable shortcomings.

On the other hand, ray tracing simulates the behavior of light in a more physically accurate manner. It casts rays from a virtual camera into the scene and calculates how these rays interact with surfaces - including reflection, refraction and shadowing, as illustrated in Figure 1.1. This method can produce highly realistic images but until recently it has been too computationally expensive for real-time use, especially in scenes with high geometric or lighting complexity.

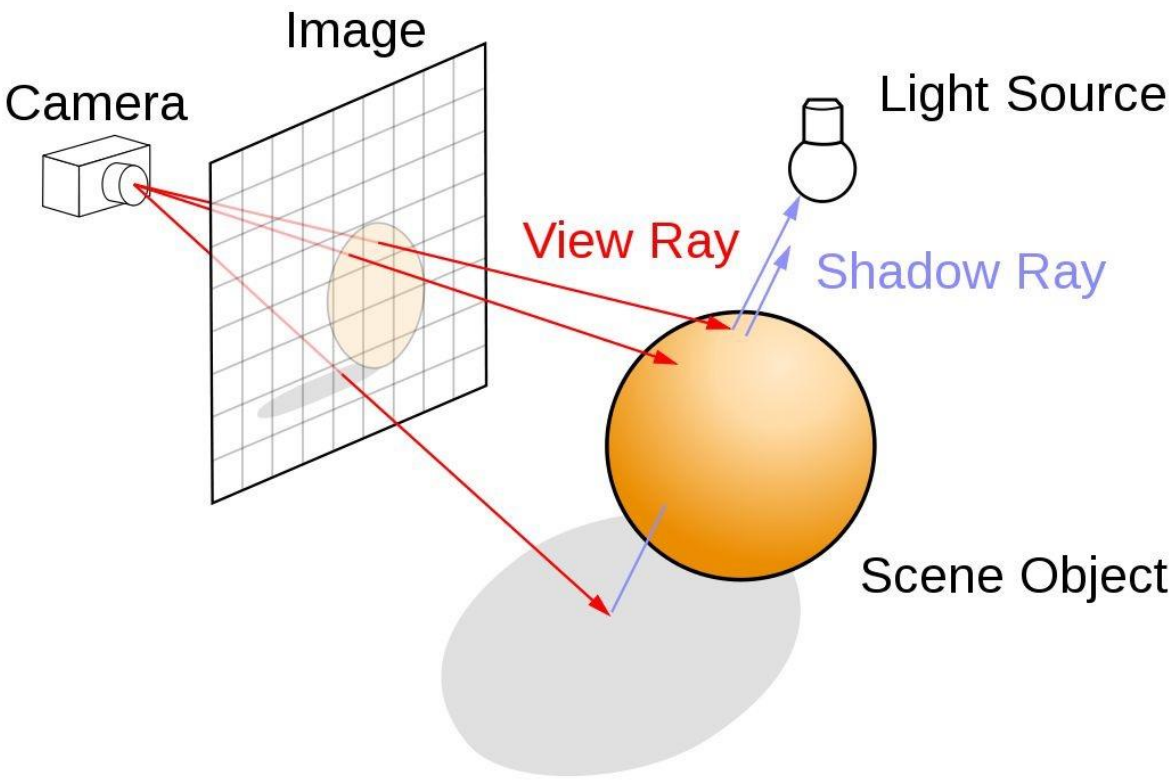


*Figure 1.1: Basic illustration of ray tracing. Source:* [4]

This changed with the release of NVIDIA's RTX series GPUs in 2018 [5], which introduced dedicated Ray Tracing Cores (RT cores) that speed up ray-object intersection calculations [4]. Also, deep learning technologies such as NVIDIA's DLSS (Deep Learning Super Sampling), have been developed to reduce the computational cost by upscaling lower-resolution renders. Together, these innovations have made real-time ray tracing possible on consumer hardware.

However, to make use of RT cores, developers must use APIs that allow them to access these features. One such API is DirectX Raytracing (DXR), a part of DirectX12 (DX12), developed by Microsoft. This project explores how DXR can be used to implement a basic real-time ray tracer by utilizing RTX hardware capabilities.

Ray tracing has growing importance in engineering disciplines. It plays a key role in the development of optical simulation tools, sensor design systems, and visual debugging platforms used across various engineering fields. For example, ray-based simulation is essential in designing LIDAR sensors, camera systems, or analyzing electromagnetic wave propagation in structured environments. Real-time ray tracing also enhances CAD/CAE applications by offering physically accurate previews of materials, lighting, and shadows, which are important for engineers working in fields such as product design or robotics.

# 2. RESEARCH OBJECTIVE

## 2.1 Specifics of the Project

The main goal of this project is to understand how DXR and RTX GPUs can be used together to implement a real-time ray tracer. The aim isn't to come up with new algorithms or techniques, but to learn and apply the existing technology to build a working system.

The breakdown of the project's objectives are as follows:

- Build a basic real-time ray tracer using DX12 and DXR. This includes:
    - Setting up the renderer by configuring various low-level parameters
    - Creating the scene by defining vertices and indices
    - Building what's called the acceleration structures. DXR uses these to trace the rays efficiently
    - Writing the DXR shaders to do the computation on the GPU
    - Setting up the memory layout so that the shaders can the necessary data from the CPU side
- Adding realistic effects:
    - Accurate reflections and shadows
    - A camera system to move around and see the scene from different angles
    - A simple UI to inspect the performance and control some of the renderer's parameters
    - A basic material system to control how surfaces look
    - A PBR (Physically Based Rendering) model to make lighting more realistic
    - The ability to load models from files on the computer at runtime

## 2.2 Mathematics Behind Ray Tracing

The core theory of ray tracing is simple: a ray is fired into the scene and whether it intersects with anything needs to be figured out. Mathematically, this just means solving a couple of equations simultaneously.

The reason ray tracing didn't use to be feasible in real time is the number of equations that need to be solved per second. At least one ray needs to be sent into the scene per pixel. This means that, on a resolution of 1920x1080, there needs to be at least 2,073,600 rays that need to

be computed, ideally at least 60 times per second. This means that at 60 FPS, over 124 million equations need to be solved every second.

Almost every 3D model that you see in computer graphics is made up of triangles. This is because triangles are the simplest shape that always lie on a plane, therefore the least error prone and easiest shape to work with. This means, if supplied with a way to calculate the ray-triangle intersection, the renderer can render almost any model.

This is also how modern RTX GPUs are believed to handle ray tracing. The word believed is because NVIDIA's actual implementation is not open source, however, the general structure and math are well known.

The following is the explanation of one way to detect a ray-triangle intersection, based on the geometric method described in [6].

Unfortunately, there's no single triangle equation. But, as mentioned before, every triangle lies on a plane. So, in order to find if a ray and a triangle intersect:

1. Find if the ray and the triangle's plane intersect.
2. If they do, check to see if the intersection point is within the triangle's boundaries.
3. If it is, the ray and the triangle intersect.

Figure 2.1 below shows the interaction between a ray, a plane and a triangle on that plane.

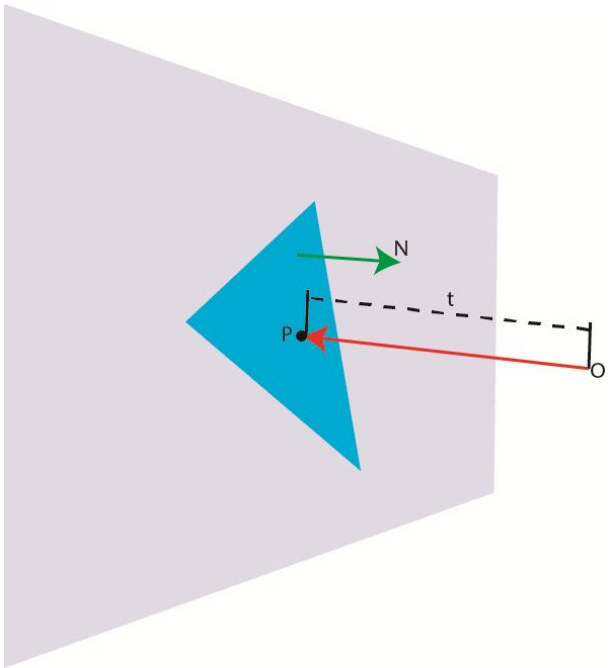


*Figure 2.1: The interaction between a ray, a plane and a triangle on that plane.*

### 2.2.1 The Ray Equation

A ray in 3D space can be represented by (2.1).

$$\boldsymbol{P} = \boldsymbol{O} + t\boldsymbol{R} \tag{2.1}$$

**P**: Point along the ray at some distance *t*

**O**: The origin of the ray

**R**: The direction vector of the ray

Any point on the ray can be obtained by plugging in different values of *t*. This will be useful later on when finding the intersection points.

### 2.2.2 The Plane Equation

(2.2) is one way of representing a 3D plane.

$$Ax + By + Cz + D = 0 \tag{2.2}$$

(*A*, *B*, *C*): x, y, z components of the plane's normal vector **N**, respectively.

*D*: Plane's offset from the origin.

This representation is used because it includes the normal of the plane, therefore the normal of the triangle. The normal of a triangle is used extensively in computer graphics, mainly for lighting calculations. Therefore, the normal is either already known or needs to be calculated in the program anyway, which reduces our unknowns to represent the plane from three to just one – the scalar offset *D*.

If the normal isn't already known it can be calculated using the corners (vertices) of the triangle of interest. Let $v_0$, $v_1$ and $v_2$ be the three vertices of the triangle. The normal can be calculated using (2.3).

$$\boldsymbol{N} = (v_1 - v_0) \times (v_2 - v_0) \tag{2.3}$$

Once the normal is known, (2.2) can be rearranged to find that the scalar *D* can be calculated using the normal and any point on the plane. Any one of the vertices of the triangle can be used as a point on the plane. (2.4) demonstrates the calculation by using $v_0$ of the triangle.

$$D = -\boldsymbol{N} \cdot v_0 \tag{2.4}$$

The simultaneous solution of the ray and plane equations yield the parameter *t* to be (2.5).

$$t = \frac{-(\boldsymbol{N} \cdot \boldsymbol{O} + D)}{\boldsymbol{N} \cdot \boldsymbol{R}} \tag{2.5}$$

Once *t* is known, it can be plugged into (2.1) to get the intersection point of the ray and the plane.

### 2.2.3 Special Cases

Before moving onto checking if the intersection point is within the triangle, there are two special cases worth mentioning.

1. The ray might be parallel to the plane. This means that the direction vector of the ray and the normal vector of the plane are perpendicular. This will cause the dot product in the denominator of (2.5) to be zero – or very close. Such cases should be discarded as they will either crash the program or produce nonsensical results.
2. The intersection point might be behind the origin of the ray. Although conceptually the ray only goes forward in the context of ray tracing, mathematically, there is nothing preventing the ray from intersecting with a plane behind its origin. This however, is quite easy to detect. Should this case occur, the value of *t* will be negative. Therefore, negative *t* values can simply be discarded.

This logic is already handled internally by the RTX hardware, so the renderer doesn't need handle these cases explicitly. But it is important to understand what's going on underneath.

### 2.2.4 Triangle Boundaries

In order to check if the intersection point lies within the triangle, some cross product operations will need to be made. Since cross product isn't commutative ($\boldsymbol{A} \times \boldsymbol{B} \neq \boldsymbol{B} \times \boldsymbol{A}$), a decision needs to be made: will the winding order be clock-wise or counter clock-wise? This decision isn't just mathematical, it will also impact which winding order to use in the renderer. The renderer should use the same winding order that is used in the mathematics. Since DX12 expects a clock-wise winding order by default, the explanation below will also assume a clock-wise winding order. How this clock-wise winding order effects the renderer will be described when the usage of vertex and index buffers are explained.

Now that the intersection point of the ray and the plane are known, the final thing to do is to check if the point is inside the triangle or not. This is done on a per-edge basis.

Let's assume that in order to get the clockwise order, the vertices of the triangle need to be ordered as $v_0 \rightarrow v_1 \rightarrow v_2$. First, one of the vertices is chosen ($v_0$ in this case) and the one of the edge vectors of that vertex is computed that corner using (2.6).

$$\boldsymbol{E_0} = v_1 - v_0 \tag{2.6}$$

Then, the vector from the chosen vertex to the ray-plane intersection point is calculated using (2.7).

$$\boldsymbol{P_0} = \boldsymbol{P} - v_0 \tag{2.7}$$

Next, cross product of $\boldsymbol{E_0}$ and $\boldsymbol{P_0}$ taken as shown in (2.8).

$$\boldsymbol{C_0} = \boldsymbol{E_0} \times \boldsymbol{P_0} \tag{2.8}$$

The intermediate vector $\boldsymbol{C_0}$ obtained from (2.8) has a special property: when taken the dot product with the triangle's normal ***N***, the result will be positive if the point is on the correct side of the edge (i.e. on the side in which the triangle lies) and it will be negative otherwise.

This process can be done for all three vertices of the triangle. Second step will be from $v_1 \rightarrow v_2$ and the third step will be from $v_2 \rightarrow v_0$. If the final dot product is positive for each step, the point is inside the triangle. Figure 2.2 below demonstrates this operation.

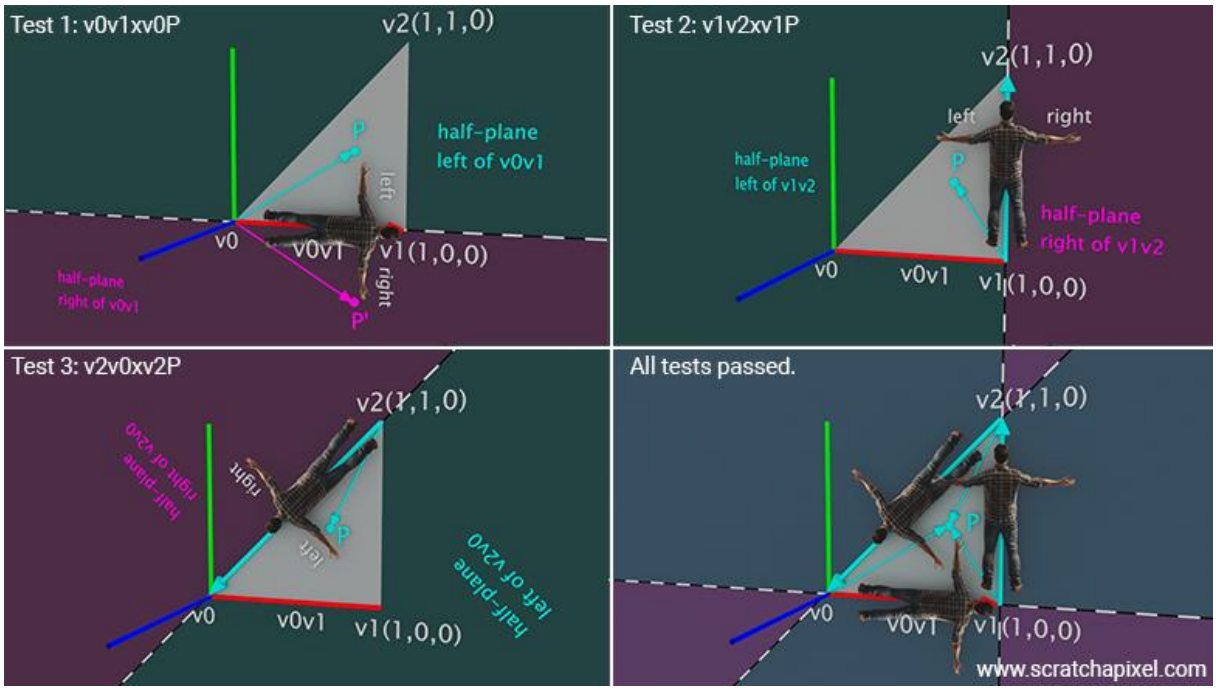


*Figure 2.2: Finding out if a point is inside the triangle. Source:* [6]

Appendix A shows a sample code that implements this algorithm. The code is taken from [6] and is modified so that it will compile as a C++ source file.

# 3. LITERATURE REVIEW

One widely known and accessible set of resources is the “Ray Tracing in XXX” trilogy: Ray Tracing in One Weekend, Ray Tracing the Next Week and Ray Tracing the Rest of Your Life. These books take a practical, step-by-step approach to building a ray tracer from scratch in C++, introducing core concepts one by one. However, it is important to note that the ray tracer developed in these three books aren’t real time.

Ray Tracing in One Weekend starts with basic ray casting and builds up to topics such as camera systems, hit detection, shading, antialiasing, and a simple material system using physically based rendering (PBR) models. Ray Tracing the Next Week builds upon this foundation and introduces more advanced topics such as motion blur, bounding volume hierarchies (BVHs), texture mapping, Perlin noise, lighting models, instancing, and volumetric rendering. Ray Tracing the Rest of Your Life explores Monte Carlo methods for light transport, including importance sampling, probabilistic density functions, orthonormal bases, and random direction generation — concepts that form the foundation for global illumination and photorealism.

Although the material in the latter two books goes well beyond the scope of this project, they are valuable for understanding how a basic ray tracer can be expanded to support more advanced features. That said, the discussion of BVH structures in Ray Tracing the Next Week is relevant, as this technique is used internally by both the DXR API and RTX hardware to accelerate ray-object intersection testing. While the BVH implementation is handled by the API, the developer must explicitly construct and manage these structures. All three books are available in print and are also published freely online via the project’s official GitHub repository [7].

Another important reference is Physically Based Rendering: From Theory to Implementation by Pharr et al. [8], which takes a more formal and detailed approach to the same core topics. It introduces the mathematical foundations of physically based lighting models, Monte Carlo integration, and BRDF definitions, and it serves as a comprehensive guide to rendering theory in both academic and industry contexts.

A more recent publication that covers DXR specifically is Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs, edited by Eric Haines and Tomas Akenine-Möller [9]. This book collects articles and contributions from NVIDIA engineers and

other experts, and it includes practical guidance on using DXR in DX12. While many chapters go beyond the scope of this project, the third chapter in particular provides insight into building a real-time ray tracer using the DXR API. The book is a valuable resource for anyone trying to understand ray tracing in more depth or extend their implementation with more advanced techniques.

The official DXR documentation provided by Microsoft is also directly relevant to this project. Rather than a traditional manual, the documentation is offered as a complete Visual Studio solution that includes several sample projects, ranging from basic rendering setups to fully-featured rendering engines. These can be found at [10].

The main tutorial followed during development was NVIDIA's DXR tutorial series [11]. The tutorial is a step-by-step guide to building a ray tracer using DX12 and DXR. The tutorial begins with project setup and the creation of a minimal renderer, and later sections expand on this base to create a more complete ray tracing system. This tutorial served as the foundation for the implementation presented in this project.

Finally, the physically based rendering techniques used in this project were initially learned from the LearnOpenGL PBR tutorials [12] [13]. While not as in-depth as academic books, these tutorials provide a clean and accessible introduction to lighting theory and implementation, making them ideal for developers seeking a working PBR pipeline without overwhelming complexity.

# 4. DESIGN

## 4.1. Realistic Constraints and Conditions

**Economy:** This project relies entirely on consumer-grade hardware and freely available SDKs. The core development was done on an NVIDIA RTX-compatible GPU (RTX 4060 Ti 8GB), using Microsoft's DirectX 12 Ultimate SDK and the DXR API — all of which are free to use for development. No specialized or custom hardware was used.

This project aligns with the increasing demand for real-time graphics across gaming, simulation, and virtual production sectors. Global virtual production market relies heavily on real-time rendering technologies. This sector was valued at 2.97 billion USD in 2023 and is expected to grow to 10.07 billion USD by 2032 [14]. Also, over 150 games and applications in the gaming industry make use of NVIDIA's RTX technology for more realistic graphics [15].

**Cost Analysis:** The main costs of the project are given in Table 4.1 below.

*Table 4.1: Main costs of the project*

| **Item** | **Description** | **Cost (TL)** |
|---|---|---|
| GPU | NVIDIA RTX 4060 Ti 8GB | 20,882 TL (VAT included) |
| Development tools | Visual Studio, Windows SDK, DXR API | Free |
| Labor | Development time (~400 hrs) | N/A |
| Electricity | Estimated power usage | 1,250 TL |

This makes the total cost of the project 22,132 TL. If a real time ray tracing compatible card is already at hand, this cost can be minimized.

**Environmental Issues:** As a software-only project, the environmental footprint is minimal. However, energy consumption of the computer during development does contribute to overall electricity use. Optimizations such as more efficient shader code or optimized acceleration structure updates could reduce the number of necessary computations and thus the power draw and the carbon footprint.

**Sustainability:** By releasing the code on GitHub under an MIT license, the project promotes long-term sustainability through collaboration with other developers.

**Manufacturability:** Manufacturability in this context refers to software deployment and portability. The application is built using widely supported APIs (DX12 and DXR) and compiles on Windows versions above Windows 10 version 1803.

**Ethical:** There are no major ethical concerns involved in this project. It does not involve collection or processing of user data, AI training, surveillance, or any use case with privacy implications. All proprietary or licensed SDKs were used in compliance with their licenses.

**Health:** There are no health-related risks associated with this project, as it involves only software development.

**Safety:** No physical safety risks were present in this project. All development was conducted in a standard software development environment without interaction with high-voltage electronics, batteries, or otherwise hazardous materials.

**Social and Political:** Ray tracing technology has growing relevance in recent technologies, such as autonomous vehicle vision systems [16] and advanced scientific visualization [17]. The skills developed here are aligned with global technological trends.

## 4.2. Cost of the Design

More details about the costs of the project can be found in Table 4.2 below.

*Table 4.2: Costs of the project in more detail*

| Component | Description | Cost |
|---|---|---|
| GPU | NVIDIA RTX 4060 Ti | 20,882 TL (VAT included) |
| IDE | Visual Studio | Free |
| SDKs | DX12 and DXR | Free |
| Miscellaneous Utilities | GPU drivers, GitHub etc. | Free |
| Electricity | Power consumption price | 1,250 TL |
| | Total | 22,132 TL |

## 4.3. Engineering Standards

Both formal and informal standards used in this project are as follows:

- C++17 Standard: Ensures compatibility with modern tools and compilers.
- Microsoft DX12 Ultimate API: Official documentation provided by Microsoft.
- HLSL Shader Model 6.3: Used for compatibility with DXR API.
- A GPU that supports real time ray tracing. This means any of the following GPUs:
    - NVIDIA's RTX 2000 Series or later
    - AMD's RX 6000 Series or later
    - Intel's Arc Series
- A version of Windows that supports DX12 and DXR. This means Windows 10 version 1803 or later.

## 4.4. Details of the Design

### 4.4.1 CPU Side

This project is based on Microsoft's D3D12HelloWorld sample project, available at Microsoft's official DirectX GitHub repository [18]. The version used here is a simplified variant with the .appxmanifest and .png files removed, as provided in the first part of NVIDIA's DXR tutorial. This sample handles DX12 initialization and creates a window using the Windows API. Since setting up even a basic DX12 renderer requires hundreds of lines of low-level configuration, building on this foundation allows the project to focus directly on implementing the ray tracing pipeline.

The first step in the pipeline is to verify that the computer's GPU supports hardware-accelerated ray tracing. This is done using DX12's built-in feature-checking methods. If the required capabilities are not available, the application will terminate early to avoid undefined behavior.

Following the hardware check, the project loads a 3D model from a file into memory. This involves reading the model's vertex and index data, which is then used to build the acceleration structures required for ray tracing.

To explain this in more detail: as mentioned before, nearly all 3D models in computer graphics are constructed from triangles, which have three corners — each defined by a vertex. At its simplest, a vertex is a point in 3D space, usually defined by its (x, y, z) coordinates. However, for more advanced renderers, vertices often carry additional attributes. For example, this project makes use of vertex normals, which is an extra attribute each vertex carries.

All of the vertices are stored in a vertex buffer, and a separate index buffer tells the renderer how to connect them into triangles. Values in the index buffer are indices into the vertex buffer. Each set of three consecutive indices describe which vertices in the vertex buffer need to be connected to construct a single triangle. Figure 4.1 below shows vertex-index buffer relationship.

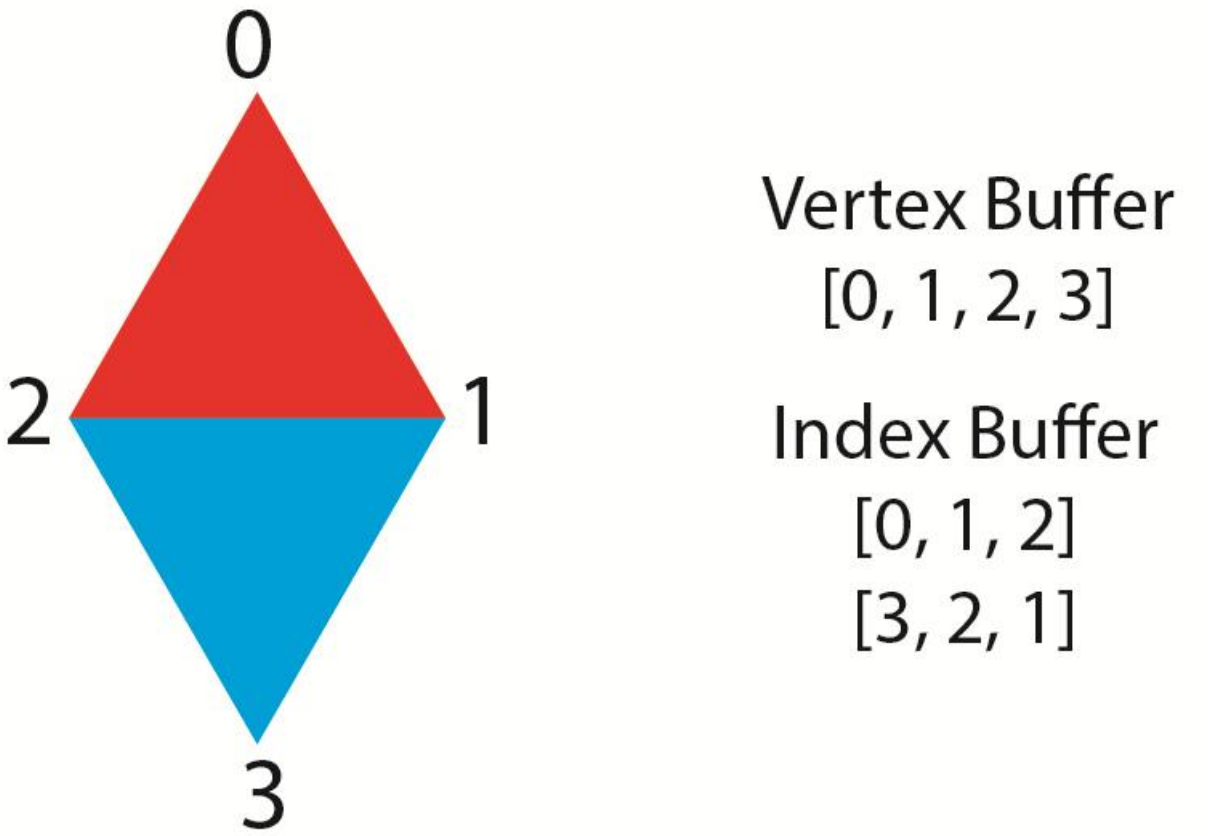


*Figure 4.1: Vertex-Index buffers relationship*

The reason the second triangle in the index buffer is defined as [3, 2, 1] rather than [1, 2, 3] has to do with the winding order, as discussed in Section 2.2.4. In this image, a clockwise winding order is assumed, which is the default in DX12. Winding order determines in which way the vertices are connected. In this case, the vertex order [1, 2, 3] would define a counter-clockwise triangle, which would be considered back-facing and potentially discarded during rendering. Alternatives like [2, 1, 3] or [1, 3, 2] would also be valid, as they represent the same triangle in a clockwise order.

It is essential to maintain the correct winding order when loading vertex and index data from model files. This can be achieved either by manually reordering indices during import or by using model files that have the appropriate winding order.

This project uses the .obj file format to load model data. This format is simple and text-based. Lines starting with the letter "v" define vertex positions, while lines starting with the letter "f" define faces (triangles). Although .obj files can include additional data via different tags, such as vertex normals (tagged with "vn") or texture coordinates (tagged with "vt"). These

fields are ignored in this project. Instead, normals are calculated programmatically based on the triangle geometry.

Once the geometry is in memory, the next step is to build the acceleration structures. DX12 uses a two-level hierarchy: Bottom-Level Acceleration Structure (BLAS) and Top-Level Acceleration Structure (TLAS). BLAS is built directly from the model's vertex and index data, while TLAS is built using one or more instances of BLAS objects. The TLAS includes transformation data such as translation, rotation, and scaling, represented using 4×4 matrices. These transformation matrices can be obtained using DX12's XMMatrixTranslation(), XMMatrixRotation(), and XMMatrixScaling() functions.

To simplify the common and verbose DX12 functionality, such as creating buffers or acceleration structures, NVIDIA's *nv_helpers_dx12* library is used. Available at NVIDIA's DXR helper repository, this library wraps verbose DX12 operations into more manageable functions. For example, while creating a buffer normally takes over a dozen lines of code, the helper function reduces this to a single line.

With the acceleration structures complete, the ray tracing pipeline needs to be set up. This includes compiling shader programs, organizing them into shader libraries, defining shader root signatures, and creating the state object that describes the pipeline layout. This state object defines which shaders to run at different stages (e.g., ray generation, miss, and hit shaders), what resources are available to those shaders, and how communication between shaders is managed. It also specifies the maximum number of recursion calls allowed while tracing a single ray.

Renderers—whether they are traditional rasterizers or real-time ray tracers—commonly use a technique called *double buffering* to prevent flickering and screen tearing. This method involves two renderable textures: while one texture is being displayed on the screen, the GPU renders the next frame into the other. Once rendering is complete, the two textures are swapped. The texture previously used for rendering is now displayed, while the one that was shown is reused to render the next frame. This alternating pattern ensures smooth visuals and avoids the artifacts that occur when writing directly to the display buffer. In the context of ray tracing, however, the renderer does not write directly to these renderable textures. Instead, an output buffer is created, matching the resolution of the render target. The ray tracing result is written into this output buffer, which is then copied into the active render texture that will be presented on screen. An extra intermediate buffer is used because the way GPU works for ray tracing

calculations is fundamentally different than rasterization calculations. Therefore, they require buffers with different properties, which determine different ways of accessing those buffers.

The project also includes a camera buffer, which stores transformation matrices used to generate rays from the camera's point of view. This buffer is updated every frame to reflect camera movement, and is largely managed by NVIDIA's *manipulator* utility. This utility relies on the *glm* mathematics library. All the necessary files can be found in this project's GitHub page. The *glm* library is under its own folder called glm, and *manipulator.h* and *manipulator.cpp* files for the *manipulator* utility can be found under the include and src folders respectively.

A materials buffer is also used, which stores information about the surface properties of the objects. Since materials are controlled via the UI, this buffer is updated every frame and made accessible to the shaders.

All these buffers and data are organized using a shader resource heap. This heap acts as GPU-accessible memory storage for buffers like the TLAS, camera matrices, materials, and the output texture. Each slot in the heap has a descriptor that specifies the type and location of the associated data.

Finally, a Shader Binding Table (SBT) is then created to link shaders with the data in the resource heap. The SBT tells each shader what data it has access to and where that data lives in memory. If the SBT and heap are not correctly aligned and consistent, the renderer can crash during execution.

For the UI, this project uses Dear ImGui, an open-source graphical interface library. It is initialized using platform-specific methods as described in the official ImGui repository's docking branch [19]. The UI provides live access to various settings, including material properties and camera behavior.

With all initialization steps complete, the program begins rendering. This is done using a command list, a structure in DX12 that holds GPU commands to be executed. Once populated, the command list is submitted to the GPU, which processes it and generates the final frame for display.

### 4.4.2 GPU Side

In order to execute code on the GPU, the logic must be written in the form of shaders. Shaders are specialized programs that run on the GPU and are responsible for executing parallel computations. In the case of ray tracing, this computation is simulating light transport through the scene.

DirectX 12's ray tracing framework defines five shader types: the ray generation shader, the miss shader, and three types of hit shaders - closest-hit, any-hit, and intersection shaders.

- The ray generation shader acts as the entry point for ray tracing, similar to *int main()* on the CPU side. It is responsible for calculating and shooting rays into the scene.
- Miss shaders are invoked when a ray doesn't intersect with any geometry.
- Closest-hit shaders are executed at the nearest point a ray intersects a triangle.
- Any-hit shaders are executed on every intersection a ray encounters and are often used for handling transparent materials.
- Intersection shaders are used when working with procedural geometry that need custom intersection logic, rather than relying on the DX12's built-in ray-triangle tests.

In this project, only ray generation, miss, and closest-hit shaders are used. The more advanced any-hit and intersection shaders are not required as the scene doesn't contain transparency or procedural geometry.

The *RayGen()* shader computes the direction of each ray using the camera matrices provided by the CPU. This calculation ensures that rays originate from the camera's position and diverge outward toward their respective pixels, producing a perspective projection. This mimics how humans perceive depth in the real world, with closer objects appearing larger and distant ones appearing smaller. By contrast, sending all rays in parallel (known as orthographic projection) results in a flatter, more two-dimensional appearance, which is less realistic for most scenes. Figure 4.2 and Figure 4.3 demonstrate the difference between orthographic and perspective projection on a 3x3 resolution, respectively.

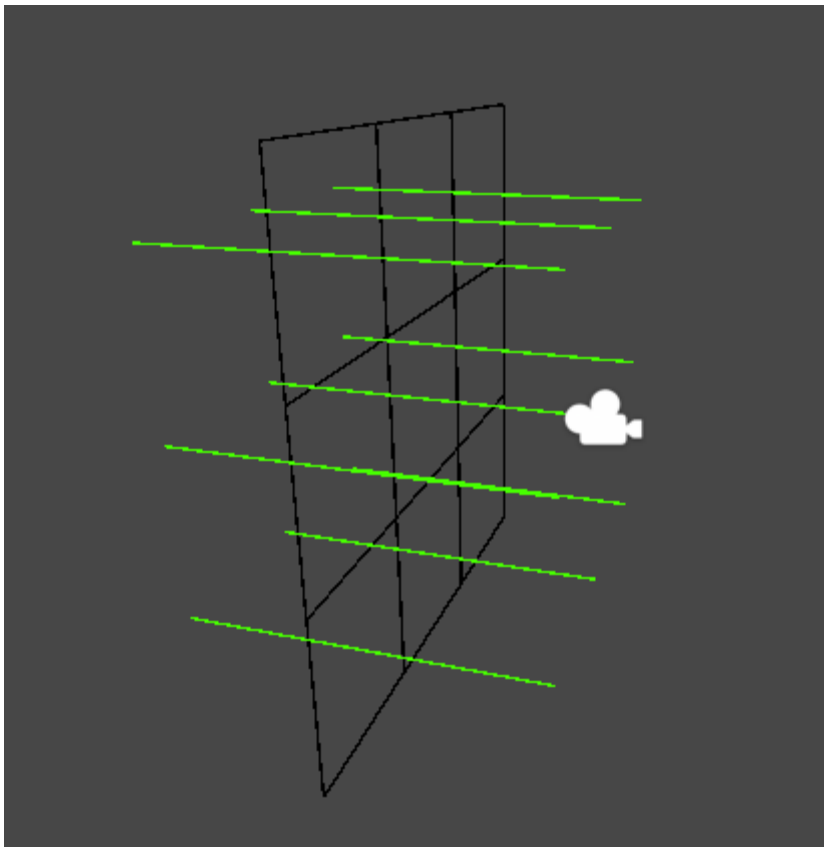

*Figure 4.2: Orthographic projection on a 3x3 resolution*

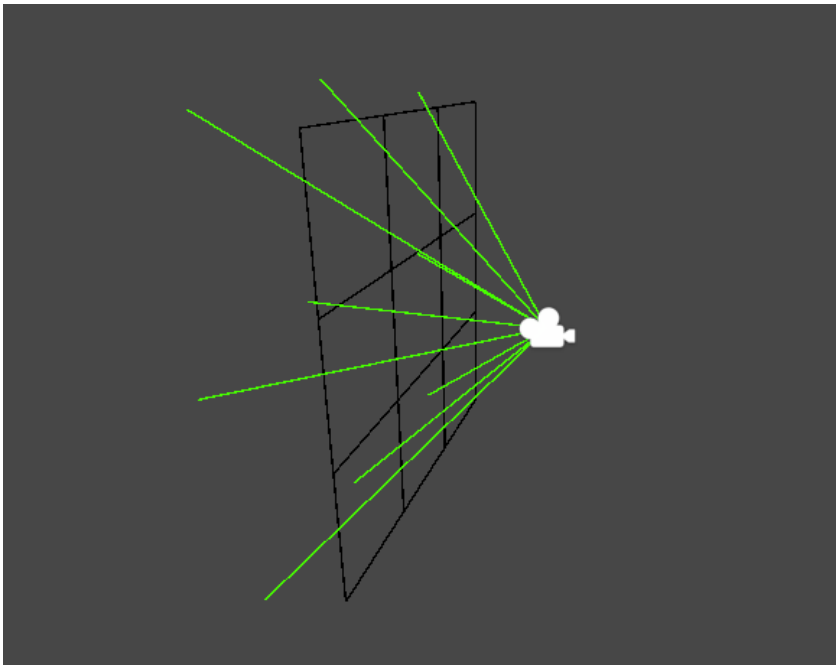

*Figure 4.3: Perspective projection on a 3x3 resolution*

If a ray doesn't intersect with anything in the scene, the *Miss()* shader is executed. This shader generates a background color using a simple gradient that darkens toward the bottom of the screen and brightens toward the top, simulating a sky-like effect.

When a ray hits an object, DX12 invokes the necessary shader. This shader is found using the SBT. For the scene geometry, there are two closest-hit shaders: *ClosestHit()* and *PlaneClosestHit()*.

- *PlaneClosestHit()* is executed when the ground is hit by a ray. It implements a simple lighting model that is less computationally heavy.
- *ClosestHit()* is executed when other geometry is hit. It implements a PBR lighting model for more realistic rendering. The details of this PBR implementation can be seen in [12] and [13].

This separation simplifies implementing shadows and reflections. In ray tracing, shadows can be simulated by firing an additional ray (known as a shadow ray) from the point of intersection toward the light source. If this ray intersects another object before reaching the light, the point is considered to be in shadow and its brightness is reduced accordingly. Figure 1.1 illustrates this algorithm.

Similarly, for reflections, a reflection ray is shot from the intersection point of the initial ray. The color returned from the reflection ray is the color for a perfectly reflective surface. Then, the actual color of the object and the perfectly reflective surface color can be linearly interpolated according to how reflective the object needs to be.

*PlaneClosestHit()* implements logic to compute and apply shadows by tracing a shadow ray. *ClosestHit()*, on the other hand, does not implement this logic and instead implements the reflection logic. This means that the ground cannot be reflective but can be cast shadows onto and the other geometry cannot receive shadows but can be reflective.

There are two additional shaders, *ShadowClosestHit()* and *ShadowMiss()*, to complete the shadow logic. These are used to record whether the shadow ray hit another object or not on its way to the light source.

# 5. METHODS

## 5.1 Software Used

The project was developed using Visual Studio 2022 with the C++17 standard. The Windows SDK version used was 10.0.18362.0, which includes support for DX12 and DXR APIs. Shaders were compiled using Microsoft's *DirectXShaderCompiler* library. Version control was managed through GitHub, integrated directly into the Visual Studio environment to allow for easy tracking of code changes.

## 5.2 Choice of Rendering API

Since the release of NVIDIA's RTX GPUs in 2018 [5], several APIs have been introduced that support real-time ray tracing. Among these, DX12 with DXR was selected for this project. Other options such as Vulkan Ray Tracing and Apple's Metal 3 are valid alternatives, but each comes with trade-offs. Vulkan is cross-platform but it is significantly lower-level than DX12, making it more complex for an individual developer without prior Vulkan experience. Metal is exclusive to Apple hardware, which was not available nor necessary for this project.

Additionally, although general-purpose GPU computing libraries such as CUDA or OptiX can be used to implement ray tracing, they were not picked for various reasons. CUDA is not a rendering-focused API, and OptiX, while designed for ray tracing, only supports NVIDIA GPUs. DXR, by contrast, is supported on any hardware that meets the required ray tracing feature level. Finally, while OpenGL is a high-level graphics API, it currently lacks official support for hardware-accelerated ray tracing and was therefore not considered a viable option.

## 5.3 3D Model Format

For loading 3D geometry, the .obj file format was selected. This format is simple, widely supported and text-based, making it easy to parse without requiring external libraries. In the context of this project, only two types of entries are parsed:

- Lines beginning with "v" define the vertex positions using three floating-point values for the x, y, and z coordinates.
- Lines beginning with "f" define faces (triangles) using three indices, each pointing to a vertex.

This minimal set of features is enough for basic rendering. Although vertex normals can also be read into the memory from the .obj file, finding a .obj file with proper vertex normals proved to be difficult. Therefore, vertex normals are calculated during runtime of the project.

## 5.4 Performance Profiling

As this is a real-time rendering project, maintaining acceptable frame rates is a must. A target of 60 FPS (≈16 ms per frame) is ideal, with a minimum threshold of 30 FPS (≈33 ms per frame) considered acceptable for real-time interaction.

To monitor performance, this project uses C++'s *std::chrono::high_resolution_clock* to measure the time taken to render each frame. From this, the frame time (in milliseconds) is calculated. Frames per second (FPS) is then derived using (5.1) below.

$$FPS = \frac{1000}{Frame\ Time\ In\ Milliseconds} \tag{5.1}$$

These performance metrics are displayed on the UI and provide immediate feedback during runtime.

# 6. RESULTS AND DISCUSSION

The goal of this project was to implement a real-time ray tracer using DX12 and DXR, capable of loading 3D models in runtime, producing realistic lighting using PBR, and supporting basic user interaction. The overall results demonstrate that the objectives outlined in Chapter 1 have been met.

The visual quality of the renderer was evaluated by comparing it to Microsoft's D3D12RaytracingProceduralGeometry sample project [2], which serves as a reference implementation. While the benchmark project uses procedural geometry and a more advanced shader structure, the general output in terms of visuals were closely matched. In particular, the rendered scene in this project was able to achieve:

- Accurate shading and hard shadows
- Object-specific shaders for better visual control
- Real-time interaction with stable frame rates

Additionally, the use of ImGui provided an interface for inspecting the real-time impact of various parameters, which was not present in the benchmark reference.

However, one area where the project needs improvement is in the implementation of the PBR material model. While a simplified PBR model was implemented, it does not always behave as expected under certain parameter values. While this issue does not significantly impact general usage, it does limit the renderer's ability to produce certain types of materials accurately.

In summary, the project successfully implements a real-time ray tracing system using DX12 and DXR API. The key goals were achieved with results comparable to existing reference implementations. Future improvements could focus on refining the shading model for more physically accurate results, especially in edge-case material configurations.

An output frame from the benchmark project, taken from [2] and a screenshot of the output of this project can be seen in Figure 6.1 and Figure 6.2 below.

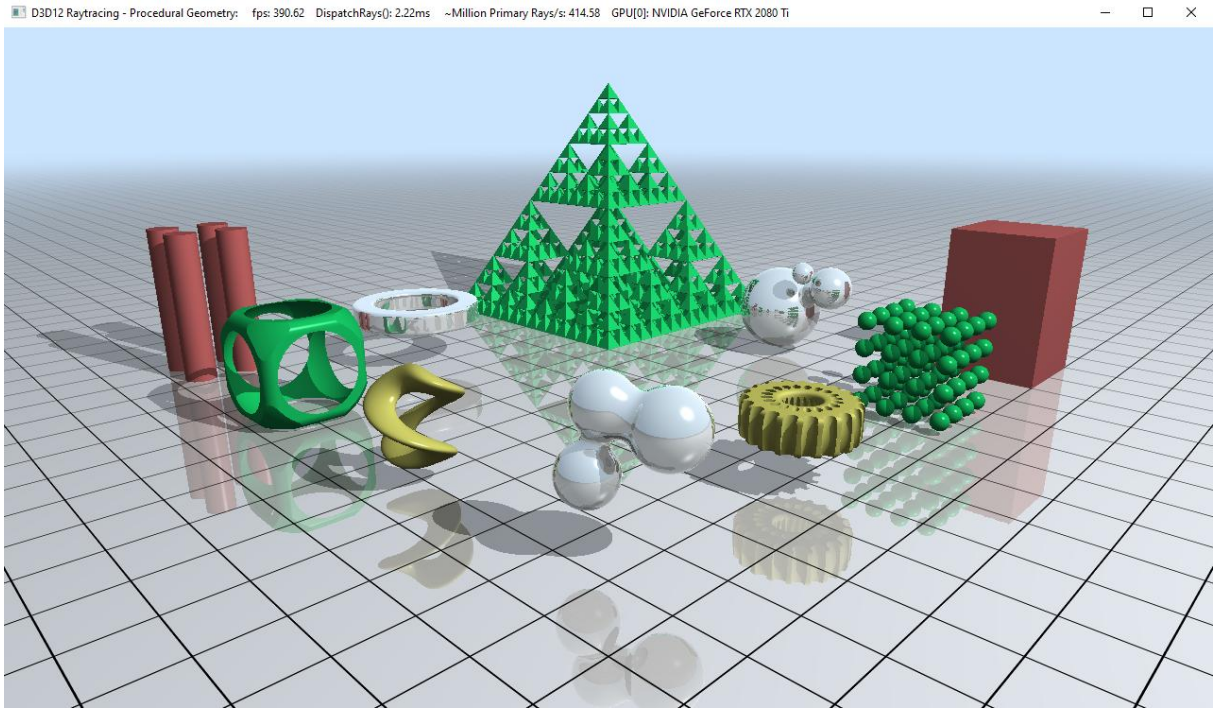


*Figure 6.1: Output of the reference project*

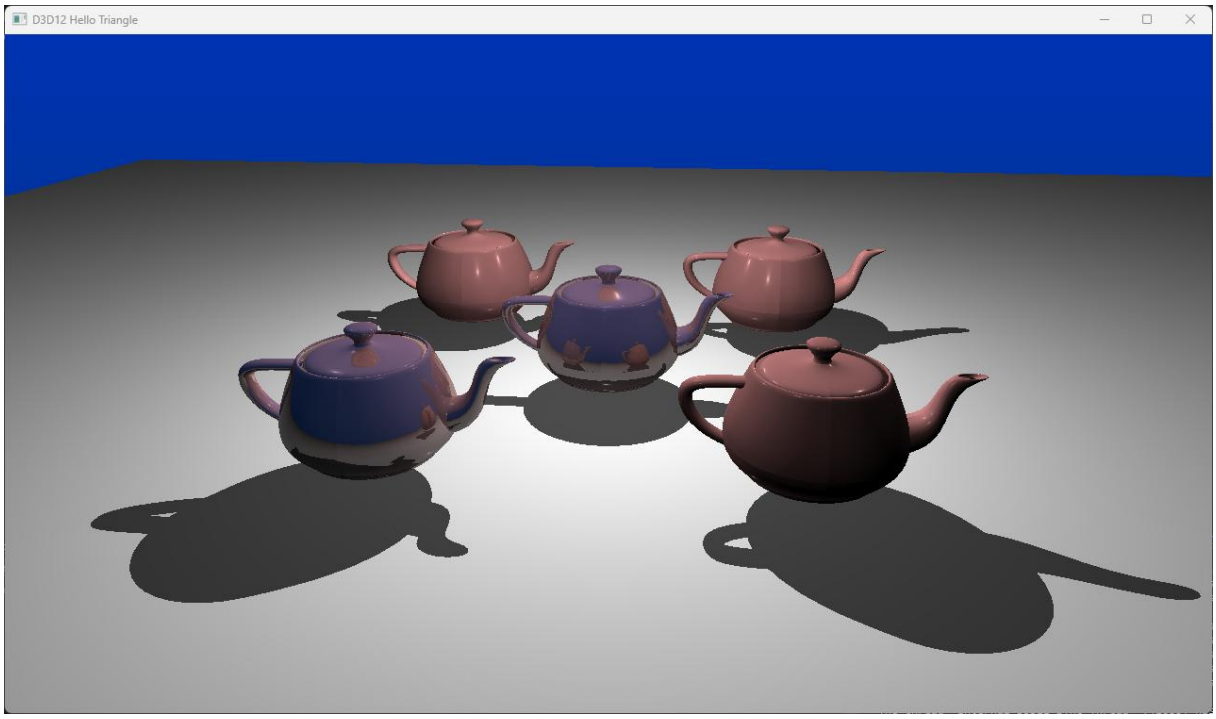


*Figure 6.2: Output of this project*

# 7. CONCLUSION

This study set out to explore and implement a real-time ray tracer using DX12 and DXR, leveraging modern GPU hardware to simulate realistic lighting, shadows, and materials.

The research objectives were defined as understanding and applying existing technologies to develop a working real-time ray tracing system. The scope included loading 3D models, implementing shaders, simulating physically based lighting, and achieving interactive rendering performance.

The literature review highlighted both theoretical and practical resources. These guided the design and helped shape the implementation approach.

In the design and methods section, the technical foundation of the renderer was detailed. Core decisions were explained alongside the role of DX12's pipeline, shader stages, and supporting libraries. Shader logic, particularly ray generation, hit and miss shaders, was also covered in depth.

The results demonstrate that the project successfully met its goals. The renderer achieved similar visual output compared to Microsoft's official DXR benchmark project while maintaining real-time performance. These results confirm the project's validity and the correctness of its core systems.

However, limitations were also observed—most notably, the simplified PBR model occasionally produces incorrect output, especially under extreme parameter values. While this does not significantly affect general rendering behavior, it provides a potential for future refinement in PBR model accuracy.

Overall, this project contributes a functioning real-time ray tracing pipeline, demonstrating the practical application of complex GPU-based rendering techniques using DX12 and DXR. It serves both as a personal learning milestone and a foundation for further experimentation and expansion.

# 8. REFERENCES


[1] U. Gökalp, "Project's Github Page," [Online]. Available: https://github.com/UtkuGokalp/RealTimeRayTracing_GradProject. [Accessed 13 June 2025].

[2] Microsoft, "Microsoft DX12 Ray Tracing Procedural Geometry Sample (Benchmark Project)," Microsoft, [Online]. Available: https://github.com/microsoft/directx-graphics-samples/tree/master/Samples/Desktop/D3D12Raytracing/src/D3D12RaytracingProceduralGeometry. [Accessed 12 June 2025].

[3] D. Hearn and M. P. Baker, Computer Graphics With OpenGL, 3rd Edition, New Jersey: Pearson Prentice Hall, 2004.

[4] NVIDIA, "Ray Tracing, NVIDIA Developer," NVIDIA, [Online]. Available: https://developer.nvidia.com/discover/ray-tracing. [Accessed 15 May 2025].

[5] M. Stich, "Introduction to NVIDIA RTX and DirectX Ray Tracing, NVIDIA Developer," NVIDIA, 19 March 2018. [Online]. Available: https://developer.nvidia.com/blog/introduction-nvidia-rtx-directx-ray-tracing/. [Accessed 15 May 2025].

[6] "Scratchapixel," [Online]. Available: https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/ray-triangle-intersection-geometric-solution.html. [Accessed 19 May 2025].

[7] P. Shirley, T. D. Black and S. Hollasch, "Ray Tracing in One Weekend - The Book Series," [Online]. Available: https://raytracing.github.io/. [Accessed 2025 June 1].

[8] M. Pharr, W. Jakob and G. Humphreys, Physically Based Rendering: From Theory To Implementation, 4th ed., Cambridge: The MIT Press, 2023.

[9] E. Haines and T. Akenine-Möller, Eds., Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs, Berkeley: Apress, 2019.

[10] Microsoft, "Microsoft's DX12 Ray Tracing Samples," [Online]. Available: https://github.com/microsoft/directx-graphics-samples/tree/master/Samples/Desktop/D3D12Raytracing. [Accessed 13 June 2025].

[11] NVIDIA, "DX12 Raytracing tutorial - Part 1," NVIDIA Developer, [Online]. Available: https://developer.nvidia.com/rtx/raytracing/dxr/dx12-raytracing-tutorial-part-1. [Accessed 29 May 2025].

[12] J. d. Vries, "LearnOpenGL - PBR Theory," [Online]. Available: https://learnopengl.com/PBR/Theory. [Accessed 29 May 2025].

[13] J. d. Vries, "LearnOpenGL - PBR Lighting," [Online]. Available: https://learnopengl.com/PBR/Lighting. [Accessed 29 May 2025].

[14] Fortune Business Insights, "Fortune Business Insights: Media & Entertainment," 12 May 2025. [Online]. Available: https://www.fortunebusinessinsights.com/virtual-production-market-107105. [Accessed 1 June 2025].

[15] I. Nnoli and E. Einhorn, "NVIDIA Developer," NVIDIA, 24 March 2022. [Online]. Available: https://developer.nvidia.com/blog/new-ray-tracing-ai-cloud-and-virtual-world-tools-streamline-game-development-at-gdc-2022. [Accessed 1 June 2025].

[16] rFpro, "ADAS & Autonomous Vehicles - Ray Tracing," rFpro, [Online]. Available: https://rfpro.com/applications-autonomous-vehicle-development/adas-autonomous-vehicles/ray-tracing/. [Accessed 2 June 2025].

[17] S. Cielo, L. Iapichino, J. Günther, C. Federrath, E. Mayer and M. Wiedemann, "Cornell University, Computational Physics," 17 October 2019. [Online]. Available: https://arxiv.org/abs/1910.07850. [Accessed 2 June 2025].

[18] Microsoft, "DirectX-Graphics-Samples on GitHub," Microsoft, [Online]. Available: https://github.com/microsoft/DirectX-Graphics-Samples/tree/master/Samples/UWP/D3D12HelloWorld/src/HelloTriangle. [Accessed 11 June 2025].

[19 "ImGUI Official Github Page," [Online]. Available: https://github.com/ocornut/imgui.
] [Accessed 2025 June 12].

# APPENDICES

## Appendix A

### Ray-Triangle Intersection Sample Code

```
#include <cmath>

class Vec3f
{
public:
    float x, y, z;

    Vec3f()
    {
        x = y = z = 0.0f;
    }

    Vec3f(float x, float y, float z)
    {
        this->x = x;
        this->y = y;
        this->z = z;
    }

    Vec3f crossProduct(const Vec3f& other) const
    {
        return Vec3f(
            y * other.z - z * other.y,
            z * other.x - x * other.z,
            x * other.y - y * other.x
        );
    }

    float dotProduct(const Vec3f& other) const
    {
        return x * other.x + y * other.y + z * other.z;
    }

    inline Vec3f operator-(const Vec3f& other) const
    {
        return Vec3f(x - other.x, y - other.y, z - other.z);
    }

    inline Vec3f operator*(const float& scale) const
    {
        return Vec3f(x * scale, y * scale, z * scale);
    }

    inline Vec3f operator+(const Vec3f& other) const
    {
        return Vec3f(x + other.x, y + other.y, z + other.z);
    }
};
```

```
inline Vec3f operator*(const float& scalar, const Vec3f& vec)
{
    return Vec3f(vec.x * scalar, vec.y * scalar, vec.z * scalar);
}

bool rayTriangleIntersect(
    const Vec3f& orig, const Vec3f& dir,
    const Vec3f& v0, const Vec3f& v1, const Vec3f& v2,
    float& t)
{
    // Compute the plane's normal
    Vec3f v0v1 = v1 - v0;
    Vec3f v0v2 = v2 - v0;
    // No need to normalize
    Vec3f N = v0v1.crossProduct(v0v2); // N

    // Step 1: Finding P

    // Check if the ray and plane are parallel
    float NdotRayDirection = N.dotProduct(dir);
    float kEpsilon = 1e-6; //Really small number
    if (fabs(NdotRayDirection) < kEpsilon) // Almost 0
        return false; // They are parallel, so they don't intersect!

    // Compute d parameter using equation 2
    float d = -N.dotProduct(v0);

    // Compute t (equation 3)
    t = -(N.dotProduct(orig) + d) / NdotRayDirection;

    // Check if the triangle is behind the ray
    if (t < 0) return false; // The triangle is behind

    // Compute the intersection point using equation 1
    Vec3f P = orig + t * dir;

    // Step 2: Inside-Outside Test
    Vec3f Ne; // Vector perpendicular to triangle's plane

    // Test sidedness of P w.r.t. edge v0v1
    Vec3f v0p = P - v0;
    Ne = v0v1.crossProduct(v0p);
    if (N.dotProduct(Ne) < 0) return false; // P is on the wrong side

    // Test sidedness of P w.r.t. edge v2v1
    Vec3f v2v1 = v2 - v1;
    Vec3f v1p = P - v1;
    Ne = v2v1.crossProduct(v1p);
    if (N.dotProduct(Ne) < 0) return false; // P is on the wrong side

    // Test sidedness of P w.r.t. edge v2v0
    Vec3f v2v0 = v0 - v2;
    Vec3f v2p = P - v2;
    Ne = v2v0.crossProduct(v2p);
    if (N.dotProduct(Ne) < 0) return false; // P is on the wrong side

    return true; // The ray hits the triangle
}
```