Computer System Architecture Project Report

Serhii Ivanov*, Sviatoslav Lushnei*, Oleh Omelchuk*, Ivan-Vitalii Petrychko*,
* *Faculty of Applied Sciences of Ukrainian Catholic University, Lviv, Ukraine*

***Abstract*—The Real-Time Raytracer Project aims to create a high-performance, interactive rendering system capable of generating realistic and immersive graphics in real time. This project report outlines the key objectives, methodologies, and outcomes of our endeavor to develop a state-of-the-art raytracer capable of rendering complex scenes with accurate lighting, shadows, reflections, and refractions.**

I. Introduction

A. What is Raytracing

Raytracing is a rendering technique used in computer graphics to generate realistic images by simulating the behavior of light. It traces the path of light rays as they interact with objects in a virtual scene, calculating how the rays are reflected, refracted, and absorbed by various surfaces.

The basic idea behind raytracing is to simulate the physical behavior of light by following the path of individual rays. These rays are typically emitted from a virtual camera or eye position and travel through the scene, interacting with objects they encounter along the way. By tracing the paths of these rays, the technique can determine the color and intensity of light arriving at each pixel on the image plane.

B. General algorithm overview

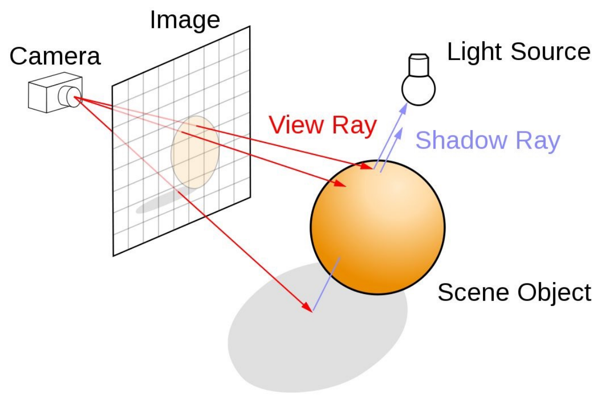


Fig. 1. How it works.

The process of raytracing involves several steps. First, the primary rays are generated from the camera position through each pixel on the image plane. These rays are then traced into the scene, and intersection tests are performed to determine if they hit any objects or surfaces.

When an intersection occurs, secondary rays are generated from the intersection point and traced further into the scene. These rays can be used to simulate effects such as reflections, refractions, and shadows. By recursively tracing secondary rays and considering the contributions of each ray to the final pixel color, raytracing produces accurate lighting and shading effects.

Finally, computed colors are combined and mapped to 2D screen coordinates for the final image. As long as the calculation of individual pixel results is unrelated, they can be effectively calculated in parallel.

C. Related Works

Raytracers have a wide range of applications in various fields, including entertainment, architecture, engineering, and scientific visualization. Over the years, they have made significant progress in computational performance and visual quality. There were some important works for various approaches in this field.

One notable technique is the Whitted ray tracing algorithm introduced by Whitted in 1980. This algorithm enables the simulation of global illumination effects like reflections and refractions through recursive ray tracing. By tracing rays recursively, the algorithm accurately models how light interacts with different surfaces in a scene, resulting in more realistic visual effects.

Another significant technique is path tracing, first introduced by Kajiya in 1986. Path tracing uses Monte Carlo sampling to simulate the paths of light rays in a scene. Path tracing approximates global illumination effects by randomly sampling different light paths, improving renderings' accuracy and visual appeal. Path tracing is particularly effective in generating soft shadows.

When talking about rendering, we can not omit Rasterization, a widely adopted rendering technique introduced in the seminal paper ”A Comprehensive Theory of Rasterization” by Foley, Van Dam, et al. in 1982. It was developed as a means to efficiently render graphics on limited hardware, revolutionizing the real-time rendering of complex scenes

II. Implementation Setup

A. Used libraries

The main code is written in C++ and GLSL(OpenGL Shading Language). Almost all algorithms and optimizations were calculated and written manually by us. However, we incorporated certain libraries to assist with input/output and other general functionalities:

- **SDL** – cross-platform library to efficiently manage windows and input/output – used for drawing pixels on the screen, reading keyboard and mouse input, and other technical stuff

- **GLM** – (OpenGL Mathematics) library – used as a main math library. We switched from our own math library to this one because of the performance difference
- **STB** – a library that provides a simple and efficient interface for loading images – used for reading images into memory for further applying them as textures
- **OpenGL** – an open-source, cross-platform API for rendering 2D and 3D graphics – used for GPU support

B. Scene

The scene refers to everything we can see in a virtual environment, such as objects, lights, and the surrounding environment.

- **The Camera** defines the point from which the virtual scene is rendered, and the screen represents the image plane onto which the scene is projected. The camera has the following properties: position, rotation, and field of view. They define the point and behavior of the rays.
- **The Screen** represents the image plane onto which the virtual scene is projected. It is a 2D plane that corresponds to the final image dimensions and behaves like a window through which we view the virtual scene. The screen has resolution and aspect ratio properties, which also affect ray behavior.
- **Movement** Using the mouse and keyboard, we can navigate our virtual world, move around, and rotate our viewpoint. It is implemented by changing the Camera's position and its rotation angles. Input is read and tracked using SDl functionality. To make smooth movements, we measure the elapsed time between frames and, based on that, calculate the distance to move.

C. Objects

1) Overview: The scene contains multiple objects of different types. All of them have their position and rotation parameters. Then, based on the specialization, additional attributes are specified, such as a triangles list, points coordinated, direction vectors, or other size parameters.

2) Graphical Object: The graphical object is a special type of object that has intersect methods and some attributes related to its size. Special Types of Graphical Objects

- **Plane** – defined by its normal
- **Sphere** – with additional characteristic as its radius
- **PolyMesh** – an object that consists of a collection of triangles

Triangle is a basic geometric primitive used to represent flat surfaces in a 3D scene. It consists of three vertices, each with their normal connected by edges. Triangles are widely used in rendering because they are simple to work with and can approximate complex surfaces. Each triangle is related to some MashObject and is given in its local coordinates (related to the object's position and rotation).

3) Matherial: Materials define visual properties such as color, reflectivity, and shininess, influencing how light interacts with object surfaces. Different material models, like the Lambertian or Phong model(which we used), dictate how light is reflected or absorbed by objects.

4) Texture: Textures provide colors and surface properties to objects. They enable the mapping of 2D images onto 3D surfaces. Texture mapping involves associating texture coordinates with object vertices and using these coordinates to sample colors or other attributes from texture images.

D. Ray Intersection

1) Ray: Ray represents the path along which light travels. It originates from the camera's position and extends into the scene, intersecting with triangles and objects. Rays are essential for determining the color of the screen pixels, computing reflections, and generating shadows. They are defined by their origin (the camera's position) and a direction vector. We trace rays through each pixel on our screen and calculate its intersections and reflections. By doing this, we can simulate the behavior of light and generate realistic images.

2) Sphere Intersection: The idea behind solving the ray-sphere intersection test is that spheres can be defined using an algebraic form. if we consider that x, y, and z are the coordinates of point $P = (O + tD - C)$, we can write:

$$(O + tD - C)^2 = R^2$$

Where C is the center of the sphere in 3D space. O is the point of the ray, and D is its direction vector It is easy to see that it is a quadratic equation related to the variable t. Solving it, we can find an intersect point(the closest one), or find out that ray misses the sphere.

3) Plane Intersection: For planes, it is also easy to find intersection points. Having point P_0 on the plane and its normal n, we can write that, for an intersection point $P = O + tD$:

$$(O + tD - P_0) \cdot n = 0$$

From that equation, we can get t and the intersection point or state that the ray misses the plane.

4) Ray-triangle Intersection: To find the ray intersection point with the triangle, we decided to use The Möller-Trumbore algorithm. In this algorithm, a transformation is constructed and applied to the ray's origin. The transformation yields a vector containing distance t to the intersection and the barycentric coordinates u and v of the intersection. The point on the triangle is given by:

$$T(u, v) = (1 - v - u)V_0 + uV_1 + vV_2$$

where u and v are the barycentric coordinates. Intersection equation yields:

$$O + tD = (1 - u - v)V_0 + uV_1 + vV_2$$

The above can be thought of geometrically as translating the triangle to the origin and transforming it to a unit triangle in y, z coordinates with the ray direction lightened with x:

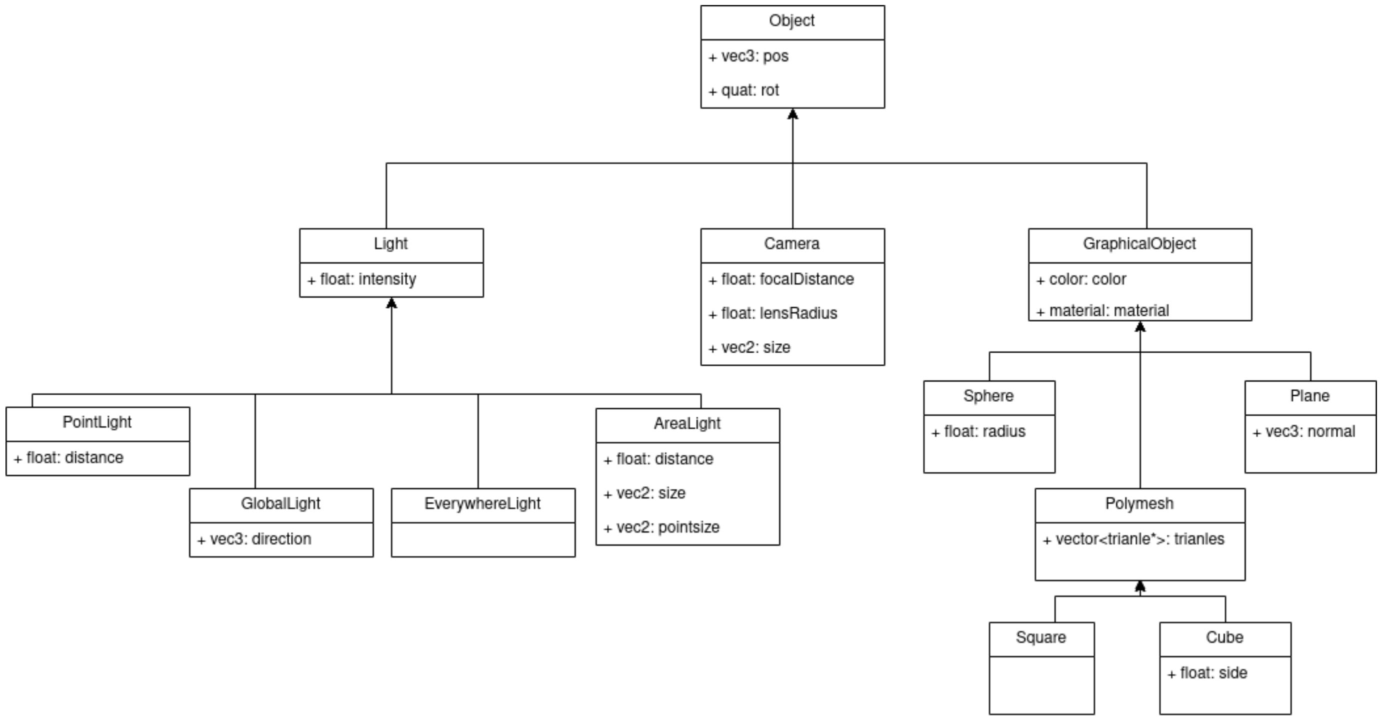


Fig. 2. Objects Hierarchy.

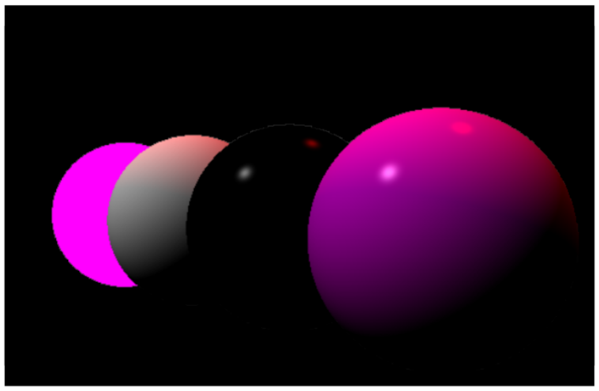


Fig. 3. Textures.

E. Lights

Lights simulate the sources of illumination in a scene. They contribute to the shading and coloring of objects and affect their appearance. They are characterized by their position, intensity, color, and other attributes. Lights can have different types:

- **Point Light** – emits light from a specific position in all directions
- **Directional Light** – parallel to a specific direction, simulates light from infinitely away sources
- **Area Light** – represent light sources that have a defined shape and size, providing soft and diffused lighting effects
- **Specular Light** – light reflecting off a smooth surface in a concentrated manner. It is reached by considering the angle between the surface normal and the viewer's line of sight,

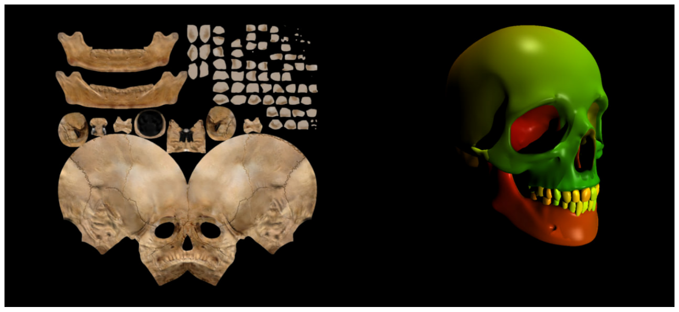


Fig. 4. Texture and UV Coordinates representation.

F. Model Loading

The .obj files contain essential data about the objects, including the coordinates of each vertex position (v-coordinates), vertex normals (vn-coordinates), triangles (f-coordinates), and texture coordinates (vt-coordinates).

Vertex normals are vectors that represent the direction perpendicular to a vertex or point on a 3D mesh or surface. They play a crucial role in shading and determining the behavior of light on the surface of an object. F-coordinates give information about the vertexes of each triangle.

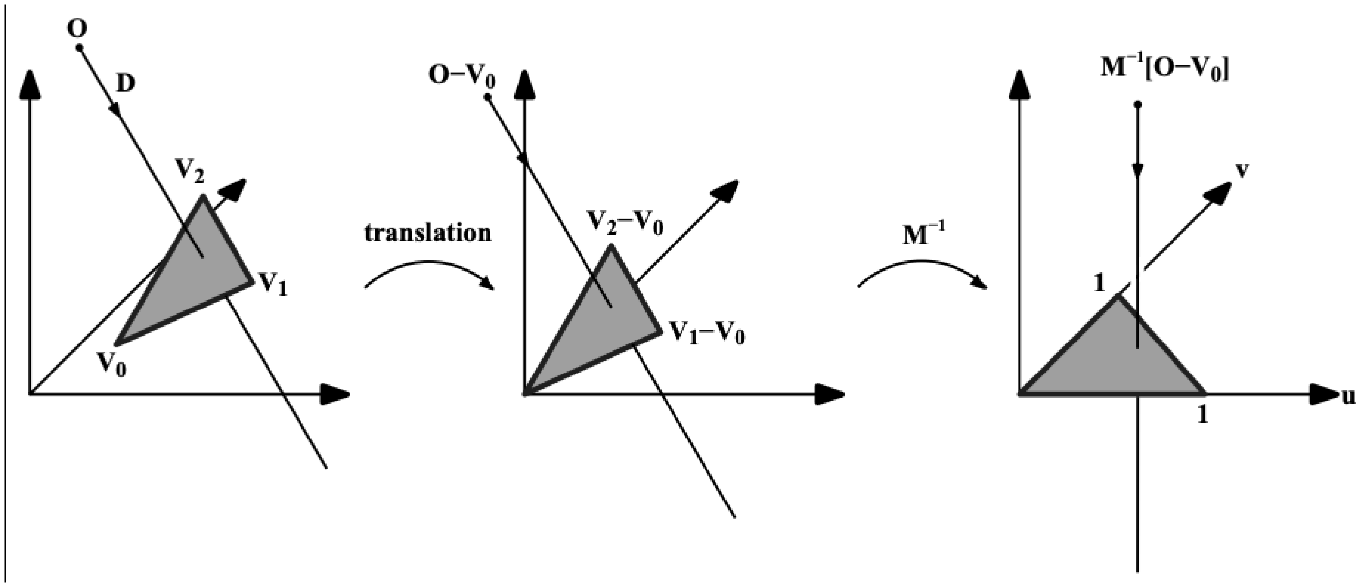


Fig. 5. The Möller-Trumbore algorithm.

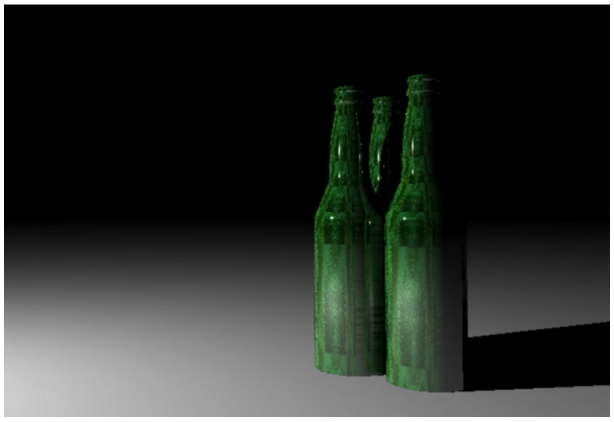
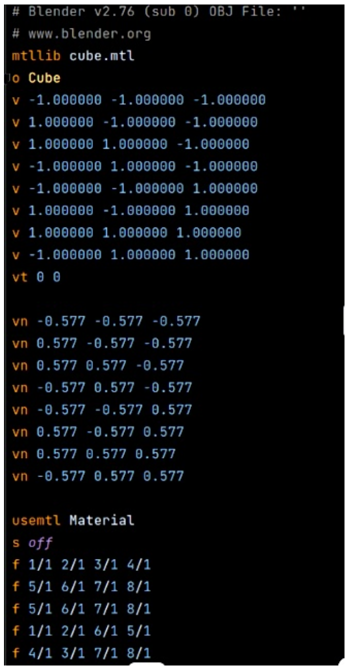


Fig. 6. Lights.



```
# Blender v2.76 (sub 0) OBJ File: ''
# www.blender.org
mtllib cube.mtl
o Cube
v -1.000000 -1.000000 -1.000000
v 1.000000 -1.000000 -1.000000
v 1.000000 1.000000 -1.000000
v -1.000000 1.000000 -1.000000
v -1.000000 -1.000000 1.000000
v 1.000000 -1.000000 1.000000
v 1.000000 1.000000 1.000000
v -1.000000 1.000000 1.000000
vt 0 0

vn -0.577 -0.577 -0.577
vn 0.577 -0.577 -0.577
vn 0.577 0.577 -0.577
vn -0.577 0.577 -0.577
vn -0.577 -0.577 0.577
vn 0.577 -0.577 0.577
vn 0.577 0.577 0.577
vn -0.577 0.577 0.577

usemtl Material
s off
f 1/1 2/1 3/1 4/1
f 5/1 6/1 7/1 8/1
f 5/1 6/1 7/1 8/1
f 1/1 2/1 6/1 5/1
f 4/1 3/1 7/1 8/1
```

Fig. 7. ".obj" file.

Additionally, these files can provide information about textures associated with the objects. The texture coordinates, representing 2D image-texture coordinates, are matched to the 3D vertices.

Moreover, we have implemented scene saving and loading from our custom .json files. That means that we can reload pre-defined scenes at a later time.

III. Performaince improvements

A. Bounding Volume Hierarchy

The intersection of rays with triangles and other objects takes about 99% of all computations. The time was linear on a number of triangles, and we could improve that. We implemented a bounding volume hierarchy, which reduces our complexity from linear to logarithmic. The technique is about recursively dividing the objects into bounding boxes up to the point when each box contains only a few triangles/objects. The logic of traversing the three that we got from the algorithm is as such: if a ray intersects a bounding box, we check for intersection with all child boxes(or triangles if it is a leaf node), and if not – we just skip it. Visually, our logic looks like shown in Fig. 8.

B. Optmizations

1) Precalculating Values: Since each triangle in a 3D scene needs to calculate its global coordinates, it is advantageous to precalculate these values and update them only when necessary. By storing the precalculated global coordinates of

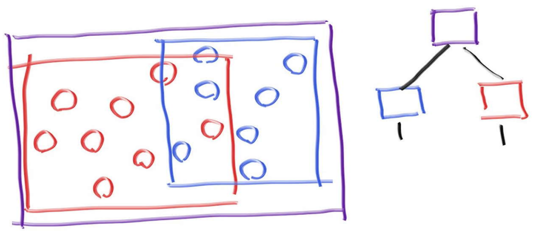


Fig. 8. Bounding Volume Hierarchy.

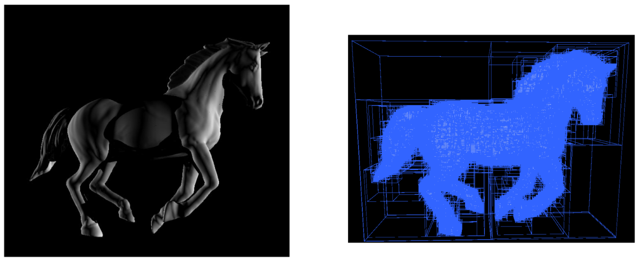


Fig. 9. Bounding Volume Hierarchy.

each triangle, we can minimize the computational overhead and improve efficiency when rendering the scene.

2) Face Culling: We have implemented face culling for the triangles in our rendering system. This technique allows us to selectively render only the front-facing triangles and discard the back-facing ones. By utilizing face culling, we optimize the rendering process by reducing the number of unnecessary calculations and improving the overall performance of our rendering pipeline.

3) Rotation with quaternionis: As long as we have camera and object rotations, with triangles' global coordinates calculated using the main object position and orientation, it is essential to have precise and fast rotations evaluations. We utilized quaternions due to their compact and efficient representation of orientation changes. Quaternions extend the concept of complex numbers to four dimensions: a scalar part (w) and a vector part (x, y, z). Typically denoted as $q = w+xi+yj+zk$, this representation offers advantages over other rotation representations like Euler angles. Quaternions are well-suited for interpolating and combining rotations, making them ideal for real-time camera rotation and providing smooth and accurate transformations.

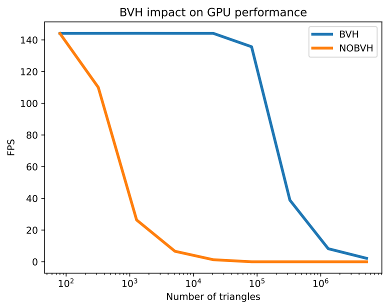


Fig. 10. BVH vs no BVH.

C. GPU

1) Graphics Processing Units: Graphics Processing Units are used to perform rapid and efficient computations required for rendering complex graphics. Unlike the CPU that excel in general-purpose tasks, GPUs are specifically optimized for parallel processing of graphical computations.

2) OpenGL: We used OpenGL, which is a widely adopted and platform-independent graphics API that provides a standardized framework for developing interactive 2D and 3D applications. It offers a set of functions and commands for rendering graphics, manipulating textures, handling user inputs, and communicating with GPUs in general.

3) GLSL: Shaders are programs where all of the GPU logic is written, and we use GLSL language because of its cross-platform support. OpenGL Shading Language (GLSL) is a high-level language designed specifically for programming shaders within the OpenGL framework. Shaders are small programs that run on the GPU and handle different aspects of the rendering pipeline, such as vertex transformations, fragment shading, and geometry manipulation.

4) Problems: However, raytracing is a bit different from what the general usage is designed for. When raytracing, we skip the part where we use Vertex shader to map points to the screen, which is an essential part for rasterization. After the Vertex shader goes to the Fragment shader, where the coloring happens. For each pixel inside of triangles, which came from the Vertex shader, the Fragment shader is applied. It returns a single color for the current pixel. However, when raytracing, we need to trace rays from each pixel in our screen, so our vertex shader accepts from the CPU only 2 triangles, which cover the entire screen. This way, the fragment shader(where all the calculations will happen) will be able to render the whole scene(see fig. Screen triangles).

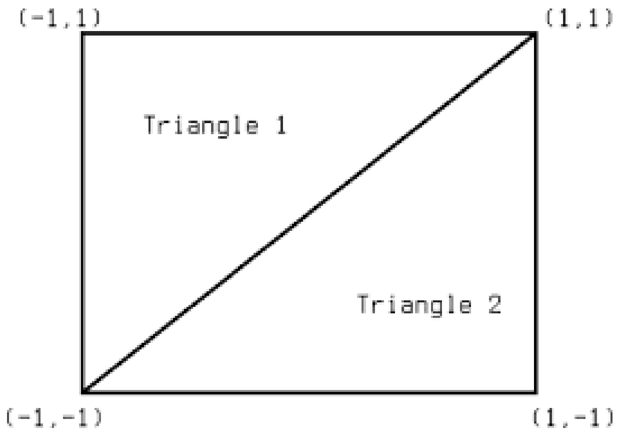


Fig. 11. Screen triangles.

5) Performance: After moving most of our code base to the GPU the performance increased by approximately 100 times with BVH disabled. We've tested the FPS boost with different amounts of triangles and with BVH enabled and here are the results:

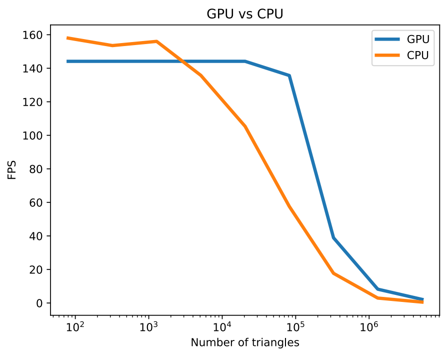


Fig. 12. GPU vs CPU.

IV. Visual effects

A. Blur

Blur (focusing on particular distance,) is achieved by emulating the behavior of a camera lens. By simulating the effects of a lens with a certain radius, rays are emitted from a randomly chosen point within that radius, and their direction is altered to mimic the defocusing of light. This process introduces randomness into the ray direction, resulting in a blurred effect, focusing on objects.

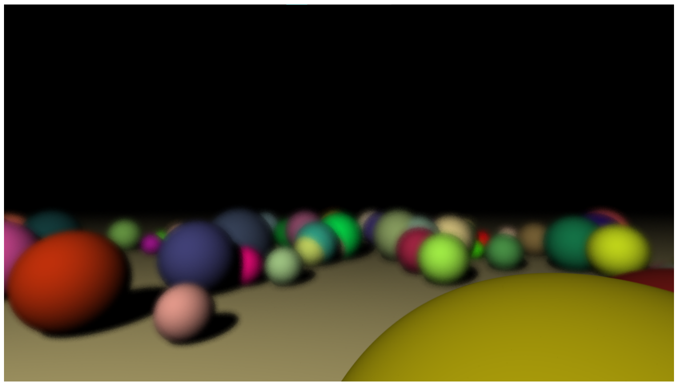


Fig. 13. Blur example.

B. Anti-Aliasing

Aliasing is common in computer graphics that causes jagged edges and visual artifacts, particularly when rendering diagonal lines or curved surfaces. We have implemented two approaches to handle them:

- **MSAA** (Multiple Samples) – samples multiple points within each pixel, calculating an average color value. It gets the best results but is computationally expensive.

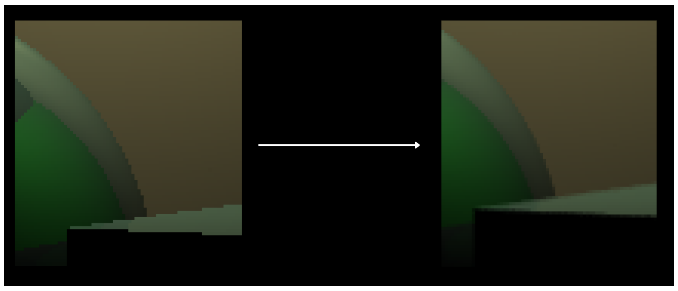


Fig. 14. Anti-aliasing effect.

- **FXAA** – (Fast Approximate) – analyzes the pixel color values and their neighborhood to identify edges and smooth out jagged edges. It reduces aliasing without significant performance impact.

C. Post-processing

After rendering the scene, the resulting image is stored as pixels in a texture object Having saved the computed pixel color results in a texture, we can apply various effects to render an image. GLSL shaders can then be used to perform different functions on these pixels, such as applying a vignette effect(darkening or fading effect towards the edges of an image) or converting the colors. This approach allows flexible post-processing workflows.

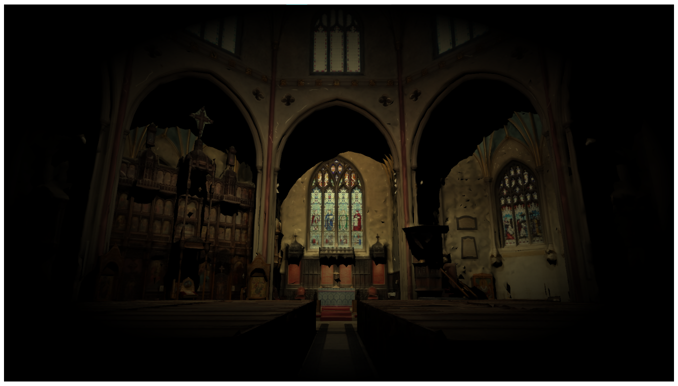


Fig. 15. Vignette effect on church image.

V. Results

Combining most of the listed features above, we managed to get pictures, shown in Fig. 16, 17, 18, 19.

VI. Conclusion

In conclusion, this project successfully implemented a robust rendering technique that accurately simulated the behavior of light rays within a virtual environment. Additionally, we incorporated advanced features such as materials, textures, different types of lights for diverse effects, and visual effects like blur and anti-aliasing. To harness the computational power required for rendering, we utilized OpenGL to interface with the GPU. Through the use of ray casting, shading algorithms, and geometric calculations, the raytracer produced realistic

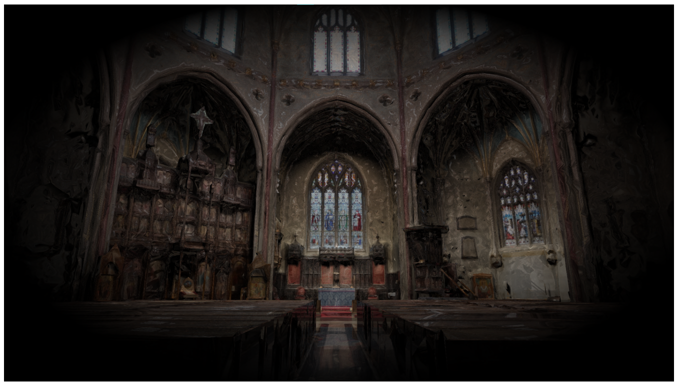


Fig. 16. Final image 1.

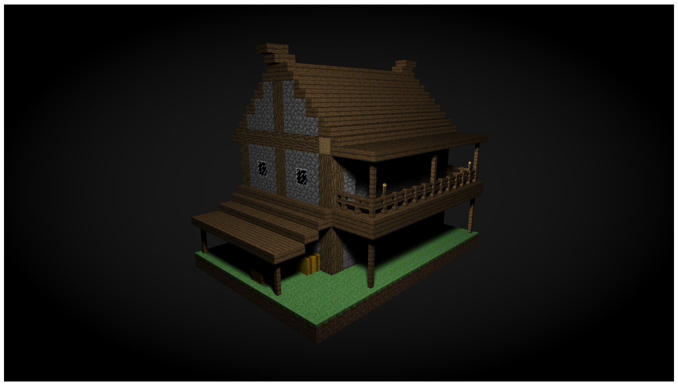


Fig. 17. Final image 2.

images with precise lighting and shadowing. The project underscored the significance of computational efficiency in handling complex scenes and materials, resulting in impressive outcomes. Overall, the raytracer project offered a valuable learning opportunity in computer graphics, fostering a deeper understanding of the principles behind synthesizing realistic images. The code of our project is available at the https://github.com/ucu-computer-science/RealtimeRaytracer.

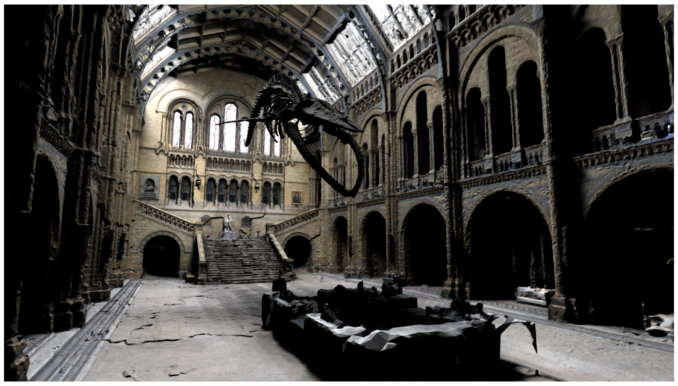


Fig. 18. Final image 3.

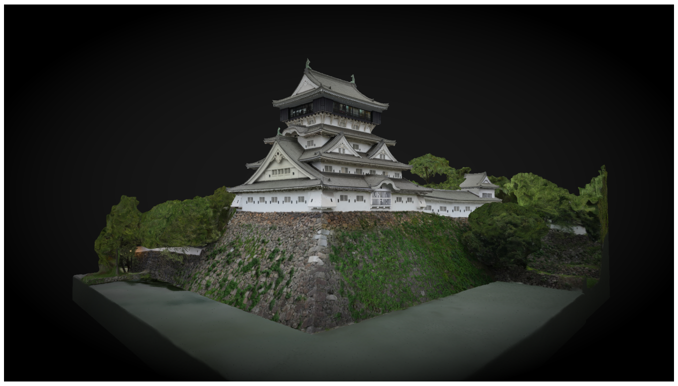


Fig. 19. Final image 4.

REFERENCES

[1] *Modelling a concurrent ray-tracing algorithm using object-oriented Petri-Nets* https://www.researchgate.net/publication/250650966_Modelling_a_Concurrent_Ray-Tracing_Algorithm

[2] *RayTracing in one weekend* https://raytracing.github.io/books/RayTracingInOneWeekend.html

[3] *RayTracing the next week* https://raytracing.github.io/books/RayTracingTheNextWeek.html

[4] *Raytracing:The Rest of Our Life* https://raytracing.github.io/books/RayTracingTheRestOfYourLife.html

[5] *Fast Minimum Storage Ray.Triangle Intersection* https://cadxfem.org/inf/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf

[6] *Implementing a practical rendering system using GLSL* https://cs.uwaterloo.ca/~thachisu/tdf2015.pdf

[7] *3D Rendering for Beginners* https://www.scratchapixel.com/index.html