CSS 535: High Performance Computing

Ray Tracing Renderer System by CUDA

Xiu Zhou, Qingran Shao, Wentao Gao

March 14th, 2024

University of Washington (Bothell)

Table of Contents

1. Introduction

In industries like film production and video game development, renderers are essential tools for creating visually stunning content. However, rendering demands substantial computational resources. One famous example is that Pixar used 117 machines and completed rendering in 1084 days (about 3 years). Each frame took anywhere from 45 minutes up to 30 hours to render. For the movie industry in general, rendering a single frame can require an excessive amount of time. For video games, the rendering methods can severely impact performance, affecting the smoothness and responsiveness that are critical for an enjoyable gaming experience. Ray tracing render offers great realism and visual quality at a cost of complexity compared to traditional rasterization rendering. These industry issues underscore the need for advancements in rendering technologies, specifically in optimizing ray tracing.

The goal for this project is to achieve the elevated levels of visual realism that ray tracing provides, but in a more efficient manner. By enhancing the efficiency of ray tracing renderer, the film industry could significantly reduce rendering times, making the production process faster and more cost-effective. Similarly, in the video game sector, improved ray tracing techniques could lead to better performance, allowing games to run smoothly without sacrificing the quality of graphics. Achieving these improvements would not only enhance the visual experience for audiences and gamers but also push the boundaries of what is currently possible in digital content creation.

For our research project, we have written a Ray Tracing System from scratch by using a custom data type Vec3 as a basic data to support constructing camera and ray, and to represent a location point in space and color. By calculating the point of contact between the ray and an object, we can get the result of the surface normal. From there we used a model called Phong shading model to do the coloring for every pixel of the surface and get a snapshot. The x, y, z of snapshots' points are transformed into PPM format and is stored as a PPM file for viewing. Each PPM file is a frame corresponding to a snapshot. Besides, we have rendered multiple frames to simulate an animated scene, including the light surrounding on the surface of 3 spheres and the cube moving along Y axis. The rendering process has been accelerated on GPU by CUDA, allowing massive parallelism in the rendering process and ultimately cutting down on the long rendering time.

2. Background

2.1 PPM file

PPM is short for portable Pixmap Format. Although these files are rare these days, you can spot one by looking out for the .ppm extension. The PPM format emerged in the late 1980s to make sharing images between different platforms easier. Each PPM file uses a text format to store information about a particular image. Within each file, every pixel has a specific number (from 0 to 65536) and information about the height and width of an image, plus whitespace data.

Information can store color information in a PPM file. Each PPM usually follows the same format, starting with a “magic number,” which helps identify the file type before moving on to the dimensions and color matrix of the image. The advantages of PPM files are straightforward to write programs capable of processing the PPM format. Also, PPM files have proven effective as an intermediary format to help transfer image information. Finally, PPM can easily transform to other image file types.

2.2 Ray Tracing

In exploring the fundamental principles of ray tracing [1], this essay delves into the simulation of light and its behavior within a virtual environment. Ray tracing, at its core, is a method that imitates the natural paths light rays take as they interact with various objects within a scene. This process initiates from a virtual camera, projecting rays of light into the three-dimensional space. These rays, originating from the camera, penetrate through pixels on a screen and extend into the scene until they encounter a surface. Throughout their journey, these rays engage with objects, either by hitting or missing them, based on the visibility determined by the line segment connecting two points in space and whether it intersects any obstacles.

Upon contact with an object, a ray can reflect, refract, or be absorbed. Reflection causes the light to bounce off surfaces, like a mirror's effect, while refraction allows light to pass through transparent materials, such as glass. Absorption means the light ceases to propagate further within the scene. These interactions enable the collection of detailed information about the object's appearance, including color, texture, and the effect of lighting. The resultant color at the point of intersection is a composite of numerous factors, such as the intensity and direction of light at the object's surface, the properties of the surface itself, and the observer's position, which in this context is the camera.

This technique, by processing every pixel to simulate these light interactions, allows for the creation of highly realistic images. However, the sophistication of ray tracing comes with significant computational demands, especially noticeable in high-resolution gaming, where rendering a scene in 4K requires the calculation of over eight million pixels.

When compared to traditional rendering techniques, such as rasterization, ray tracing stands out for its physical accuracy. Rasterization focuses on projecting 3D objects onto a 2D plane and determining the color of pixels based on their coverage by these objects, using lighting, shading, and textures for color calculation. Although effective for creating visual content, rasterization relies on simplifications and approximations that can limit the realism of the images produced. In contrast, ray tracing, by adhering closely to the physical properties of light, can yield far more lifelike results, albeit at the cost of increased hardware requirements, often necessitating GPUs with specialized optimization for ray tracing.

2.3 Phong Shading Model

The Phong Shading Model [2] is an empirical model of the local illumination of points on a surface designed by the computer graphics researcher Bui Tuong Phong [3]. It describes the way a surface reflects light as a combination of the diffuse reflection of rough surfaces with the specular reflection of shiny surfaces. It is based on Phong's informal observation that shiny surfaces have small intense specular highlights, while dull surfaces have large highlights that fall off more gradually. The model also includes an ambient term to account of the small amount of light that is scattered about the entire scene.

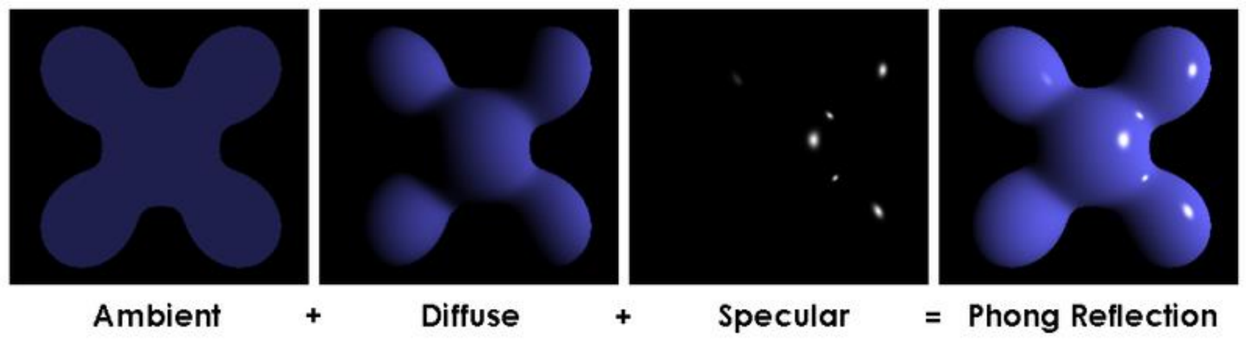


Figure 1Illustration of the Phong Shading Model. Created by the uploaded, Brad Smith, August 7, 2006[3]

In every scene, light sources have two components: the specular intensity i_s and the diffuse intensity i_d, typically expressed in RGB values. These components define the brightness of the specular and diffuse light, respectively. Additionally, there is a single value for ambient lighting i_a, often calculated as the cumulative effect from all light sources.

Materials in the scene are characterized by specific parameters:

- K_s, the specular reflection constant, determines how much specular light (shiny reflections) is reflected.
- $K_{d,}$ the diffuse reflection constant, affects the amount of diffuse light reflected, based on Lambertian reflectance.
- $K_{a,}$ the ambient reflection constant, controls the reflection of ambient light that is evenly distributed in the scene.
- a, the shininess constant, indicates the material's glossiness. Higher values make surfaces appear smoother and more mirror-like, concentrating on specular highlights.

To compute lighting effects like Phong and Blinn-Phong shading, several vectors are used:

A set of light sources (lights)

$\hat{L}m$ the direction vector from the surface point towards each light source (where m specifies the light source),

$\hat{N}$ the normal vector at the surface point.

$\hat{R}_m,$ the direction a perfectly reflected ray takes from the surface point.

$\hat{V},$ the direction pointing towards the viewer.

The Phong Shading model calculates the illumination (I_p) of a surface point as follows [4].

$$I_p = k_a i_a + \sum_{m \in \text{lights}} \left(k_d\left(\widehat{L_m} \cdot \hat{N}\right) i_{m,d} + k_s\left(\widehat{R_m} \cdot \hat{V}\right)^{\alpha} i_{m,s}\right).$$

Here, the reflected direction vector $\hat{R}_m$ is computed as the reflection of $\hat{L}_m$ off the surface (using the normal $\hat{N}$):

$$\widehat{R_m} = 2\left(\widehat{L_m} \cdot \hat{N}\right)\hat{N} - \widehat{L_m}.$$

Note that vectors are normalized. The specular highlight's visibility depends on the viewer's angle, being pronounced when $\hat{V}$ aligns with $\hat{R}_m$. This alignment is quantified by the "a" power of their dot product's cosine. High "a" value, indicative of a near-mirror finish, result in smaller specular highlights due to rapid falloff when the viewing angle diverges from the reflection direction.

The inclusion of each term in the model depends on the positivity of their respective dot products. Moreover, when color is represented in RGB, the equation is modeled separately for R, G, and B intensities, allowing for channel-specific reflection constants k_a, k_d, and k_s.

2.4 Color in Picture

CMYK, Pantone, RGB, and RAL represent four distinct color systems, each with its specialized application within the spectrum of color management. The CMYK system is primarily utilized in the printing industry, leveraging a subtractive color model to blend Cyan, Magenta, Yellow, and Key (black) to produce a wide range of colors. However, the RGB system is designed for digital displays, such as websites and video screens, employing an additive color approach where red, green, and blue light are combined in varying intensities to create many colors. The Pantone system is renowned for its precision and consistency, offering a standardized color-matching system that is indispensable for design professionals seeking accurate color calibration and reproduction. Meanwhile, the RAL color system finds its application in the industrial domain, commonly used for powder coating and plastics, providing a robust palette for material finishes.

In the context of this project, the RGB color model is the chosen framework for representing the colors within each frame of a video or GIF. This model operates on an additive principle, where the primary colors of light—Red, Green, and Blue—are mixed in different proportions to achieve an extensive color spectrum. Each color channel can vary in intensity from 0 to 255, allowing for the precise rendering of a wide array of colors. This method not only facilitates a rich visual experience but also ensures compatibility and vibrancy across digital platforms, making it an ideal choice for the dynamic and detailed presentation of video and animated content.

2.5 CUDA

Developed by NVIDIA, CUDA represents a groundbreaking parallel computing platform and programming model that has revolutionized the way developers harness GPU accelerators to enhance application performance. To date, it has been downloaded over 20 million times, highlighting its significant impact on speeding up a diverse range of applications.

CUDA's influence extends beyond the realm of high-performance computing (HPC) and academic research, finding its place in various sectors of the consumer and industrial markets. It is utilized in the pharmaceutical industry to accelerate the discovery of new treatments, in automotive technology to improve autonomous driving capabilities, and in retail — both physical and online — to refine customer purchase analysis and targeted advertising through data analytics.

What exactly is CUDA? While some might mistakenly categorize it merely as a programming language or an API, CUDA, which debuted in 2006, encompasses much more. With over 150 specialized libraries, development kits, and tools for profiling and optimization, CUDA offers a comprehensive ecosystem for developers.

NVIDIA's commitment to innovation is evident in the thousands of GPU-accelerated applications built on CUDA, which remains the preferred platform for exploring and implementing innovative deep learning and parallel computing algorithms. CUDA's design not only facilitates access to the latest GPU architectural advancements, as exemplified by NVIDIA's recent Ampere architecture, but also ensures developers can efficiently leverage these technologies to push the boundaries of what is possible.

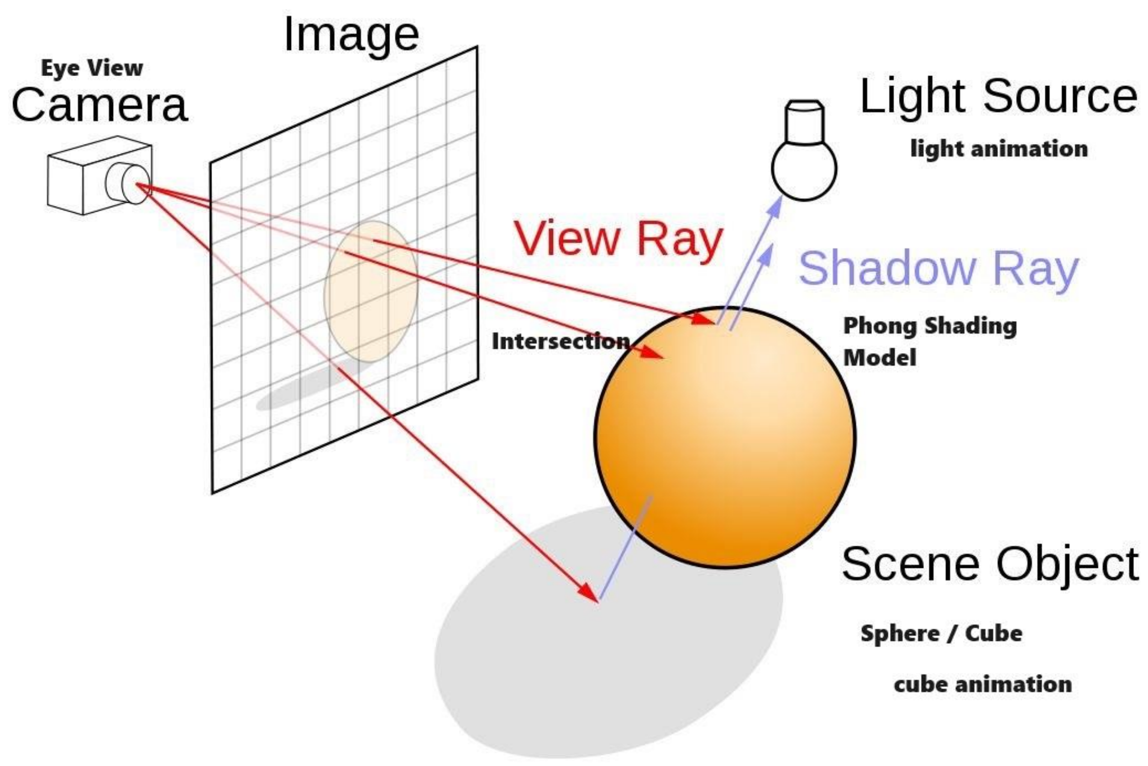


Figure 2 Illustrative Diagram

3. Implementation

3.1 System Program Design (GPU)

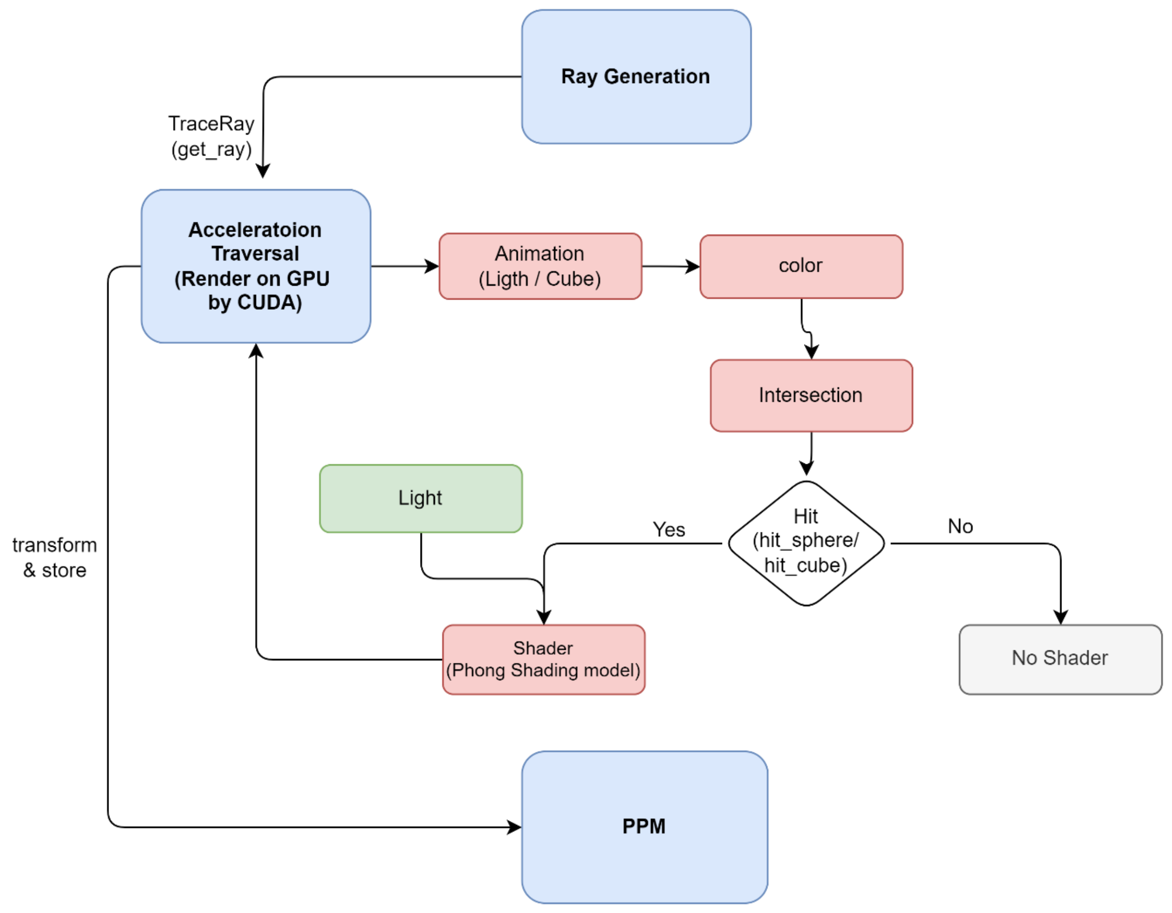


Figure 3 Program Flow Chart

In Fig 3, it shows the overview of our Ray Tracing System Program[5]. The camera generates rays, then intersects with current scene / objects, then color all the pixels of image plane based the interaction results by using Phong Shading Model, finally, transform and store the data as PPM format file. This entire process will be repeated multiple times for capturing new scene/objects snapshots because of the moving of light and cube.

For simplifying compiling and being more readable for instructor and TA, we combine all the classes and functions into one file with full comments.

3.2 Classes

In our project, there are these classes.

1) Vec3: Represents a 3D vector and provides various vector operations.

```
public:
float x, y, z;
//+, -, *, \, dot(), normalize( ) etc.
```

2) Ray Class (Ray):

```
public:
        Vec3 origin, direction;
//Represents a ray in 3D space.
//Defined by an origin point and a direction vector.
//Used for casting rays from the camera into the scene
//to determine what objects are visible
//and how they interact with light.
```

3) Camera Class (Camera):

```
public:
      Vec3 origin, lower_left_corner, horizontal, vertical, light_position;
      float aspect_ratio;
       Ray get_ray(float u, float v) const;
```

- Represents a simple camera in 3D space.
- Defines the position and orientation of the camera relative to the scene.
- Calculates the rays corresponding to each pixel in the viewport of the camera.
- Generates rays that pass through each pixel of the viewport based on the camera's position, orientation, and field of view.

4) The relationship between the Ray and Camera classes in your program is as follows:

- The Camera class utilizes the Ray class to generate rays for each pixel in its viewport. It does so through the get_ray method, which calculates the direction of the ray based on the pixel coordinates and the camera's properties (such as origin, horizontal, and vertical vectors).
- Each ray generated by the Camera class using the get_ray method is used to trace the path of light from the camera's viewpoint into the scene. These rays are then used for tasks such as determining intersections with objects in the scene, calculating lighting effects, and rendering the final image.
- The Camera class serves as the "eye" through which the scene is viewed, while the Ray class enables the simulation of light rays emanating from this viewpoint to interact with objects in the scene, allowing for the creation of realistic renderings.

3.3 Program Functions

1) Global and Device Functions

```
// Phong Shading - Calculates shading using the Phong Shading model
__host__ __device__
Vec3 phong_shading(const Vec3& normal, const Vec3& light_direction,
const Vec3& view_direction,  const Vec3& ambient_color, const Vec3&
diffuse_color, const Vec3& specular_color,  float shininess);

// Sphere Intersection - Calculates the intersection of a ray with a sphere
__host__ __device__
float hit_sphere(const Vec3& center, float radius, const Ray& ray);

// Cube Intersection - Calculates the intersection of a ray with a cube
__host__ __device__
float hit_cube(const Vec3& center, float side_length, const Ray& ray);

// Calculate cube normal at a given point
__host__ __device__
Vec3 calculate_cube_normal(const Vec3& point, const Vec3& center,
float side_length);

// Color Calculation - Determines the color at a specific pixel based on ray-
sphere intersections
__host__ __device__
Vec3 color(const Ray& ray, const Vec3& light_position, Vec3& cube_center);

// Save Image to PPM in text format (P3)
void save_image(const std::string& filename, Vec3* framebuffer, size_t
width, size_t height);

// Animation function to move only the cube
__host__ __device__
void animate_cube(float animation_time, Vec3& cube_position,
const Vec3& camera_origin, float min_y_boundary, float max_y_boundary);

// Main animation function to move the light source
__host__ __device__
void animate_scene(float animation_time, Vec3& light_position,
const Vec3& camera_origin);
```

3.4 Kernel Function - Render Kernel

Each thread in the render kernel is responsible for rendering a single pixel in the output image. It calculates the color of that pixel by casting a ray from the camera's viewpoint and tracing its path through the scene to determine how it interacts with objects and light sources. The final color value is then stored in the framebuffer.

```
// Render kernel
__global__ void render(Vec3* framebuffer, size_t width,
size_t height, size_t num_samples, float animation_time,
Vec3 cube_position_local, float min_y_boundary, float max_y_boundary) {
  int i = threadIdx.x + blockIdx.x * blockDim.x;
  int j = threadIdx.y + blockIdx.y * blockDim.y;

  if (i < width && j < height) {
    int pixel_index = j * width + i;
    curandState_t rand_state;
    curand_init(1984 + pixel_index, 0, 0, &rand_state);

    Vec3 col (0, 0, 0);
    for (size_t s = 0; s < num_samples; ++s) {
      float u = (i + curand_uniform(&rand_state)) / width;
      float v = (j + curand_uniform(&rand_state)) / height;

      Camera camera;
      Ray ray = camera.get_ray(u, v);

      Vec3 light_position;

      // Animate only the cube independently
      animate_cube(animation_time, cube_position_local, camera.origin,
min_y_boundary, max_y_boundary);

      // Animate the light source and the rest of the scene
      animate_scene(animation_time, light_position, camera.origin);

      col += color(ray, light_position, cube_position_local);
    }

    col /= num_samples;
    framebuffer[pixel_index] = col;
  }
}
```

3.5 Program Running Process

- **Initialization**:
  - The program starts by initializing various parameters such as frame width, height, number of samples per pixel, and the number of frames to generate.
  - It also sets up the camera and defines the initial positions and properties of the cube.
- **Frame Generation Loop**:
  - The program enters a loop to generate multiple frames. For each frame:
    - It calculates the animation time based on the current frame index.
    - It allocates memory for the current framebuffer.
    - It animates the movement of the light source and the cube based on the animation time.
    - It launches a CUDA kernel (**render**) to generate the frame by performing ray tracing for each pixel.
    - Inside the **render** kernel, each thread calculates the color of a specific pixel by casting rays into the scene and computing intersections with objects.

    - The computed colors are accumulated and averaged over multiple samples per pixel to reduce noise.
    - Once all pixels are processed, the resulting frame is stored in the framebuffer.
    - The framebuffer is saved to a PPM image file.
    - Memory allocated for the current framebuffer is freed.
- **Finalization**:
  - After generating all frames, the program calculates and prints the total time taken for the entire process.
  - It frees any remaining allocated memory and resets the CUDA device.

3.6 System CPU Implementation Declaring

This CPU implementation is just for comparing the execution efficiency with GPU implementation. All the functions implementation logic are the same as the GPU's.

3.7 Compile and Run System Programs

For letting instructor and TA compile and read program code easier, we combined all the classes and functions into one file with fully comments.
To build and execute the program in UWB GPU lab machine, follow these steps:
<u>For the CPU-based version of the program:</u>
Compile the CPU version using g++ by running the bash command:

```
g++ -o cpu_anima animation_CPU.cpp
```

Execute the compiled program with:

```
./cpu_anima
```

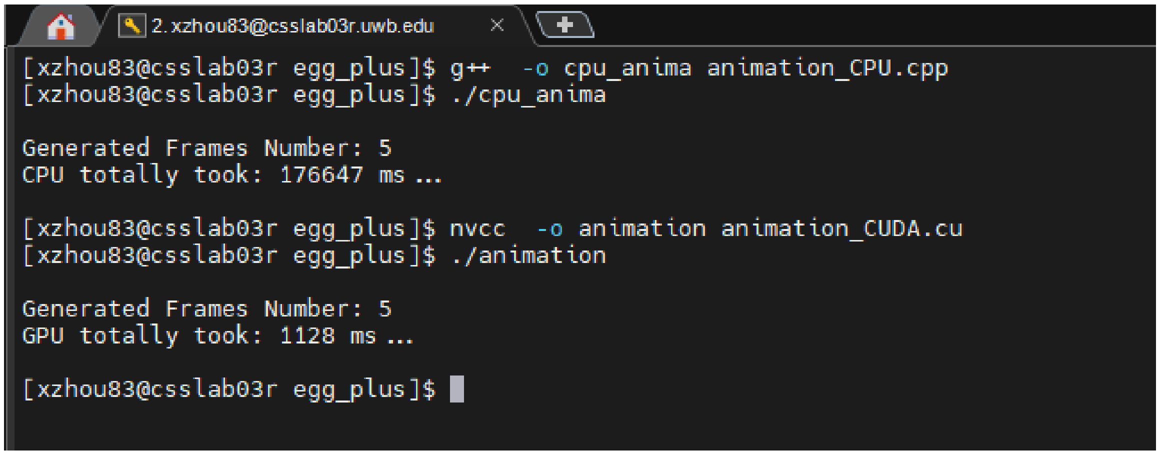
<u>For the GPU-accelerated version of the program:</u>
Compile the GPU version using nvcc by running the bash command:

```
nvcc  -o animation animation_CUDA.cu
```

Run the compiled GPU program with:

```
./animation
```

3.8 Output Screenshot



```
[xzhou83@csslab03r egg_plus]$ g++  -o cpu_anima animation_CPU.cpp
[xzhou83@csslab03r egg_plus]$ ./cpu_anima

Generated Frames Number: 5
CPU totally took: 176647 ms ...

[xzhou83@csslab03r egg_plus]$ nvcc  -o animation animation_CUDA.cu
[xzhou83@csslab03r egg_plus]$ ./animation

Generated Frames Number: 5
GPU totally took: 1128 ms ...

[xzhou83@csslab03r egg_plus]$
```

Figure 4 Runtime of CPU version and CUDA version

4. Result Presentation and Analysis

4.1 Result Presentation

The program will provide following picture in PPM format as fig.5:

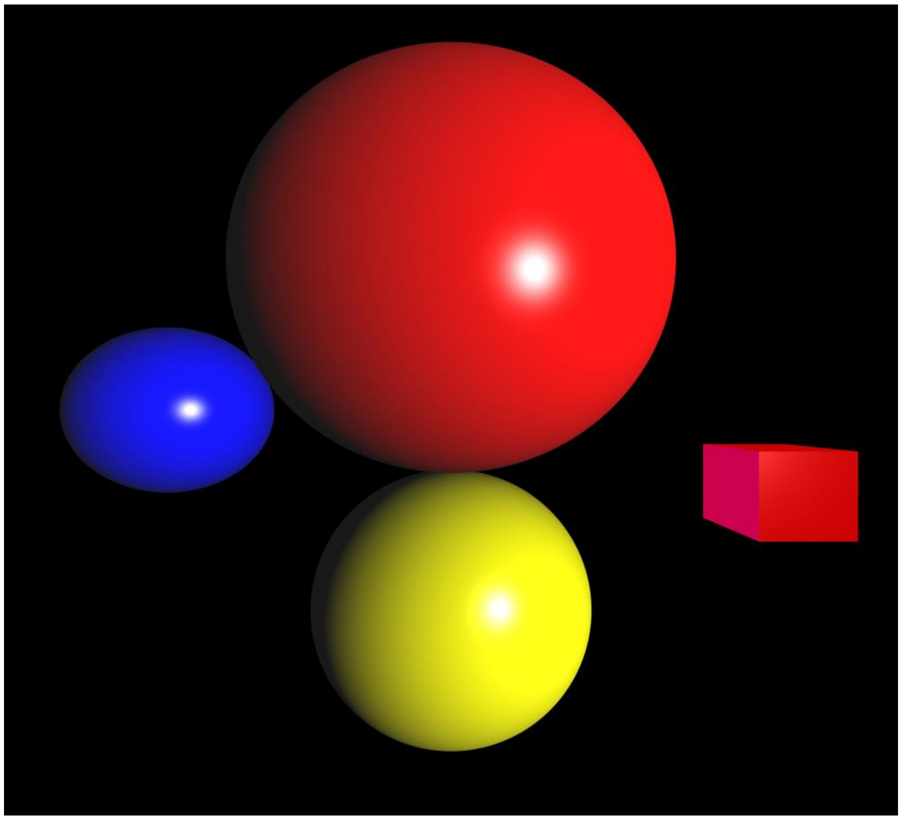


Figure 5 Output PPM result frame 61

4.2 GPU Analysis

The analysis of the GPU technology utilized in this study is presented in the table below, highlighting the specifications of the NVIDIA CUDA® technology:

Catalog	Feature
NVIDIA CUDA® Cores	5888
Standard Memory	8GB GDDR6
Memory Interface Width	256-bit

4.3 CPU Analysis

Similarly, the CPU technology deployed for the environment is detailed in the following table, highlighting the characteristics of the 9th Generation Intel® Core™ i9 processor:

Essentials

Catalog	Feature
Product Collection	9th Generation Intel® Core™ i9 Processors
Code Name	Products formerly known as Coffee Lake
Vertical Segment	Desktop
Processor Number	i9-9900K
Lithography	14 nm
Use Conditions	PC/Client/Tablet

CPU Specifications:

Catalog	Feature
Total Cores	8
Total Threads	16
Max Turbo Frequency	5.00 GHz
Intel® Turbo Boost Technology 2.0 Frequency‡	5.00 GHz
Processor Base Frequency	3.60 GHz
Cache	16 MB Intel® Smart Cache
Bus Speed	8 GT/s
Max Memory Size (dependent on memory type)	128 GB
Memory Types	DDR4-2666
Max # of Memory Channels	2
Max Memory Bandwidth	41.6 GB/s
DirectX* Support	12
OpenGL* Support	4.5

Intel® Quick Sync Video	Yes
Intel® InTru™ 3D Technology	Yes
Intel® Clear Video HD Technology	Yes
Intel® Clear Video Technology	Yes
Catalog	**Feature**
Total Cores	8
Total Threads	16
Max Turbo Frequency	5.00 GHz
Intel® Turbo Boost Technology 2.0 Frequency‡	5.00 GHz

4.4 Operating System

For the software environment, this project was executed on a Linux-based system, leveraging the robust and versatile capabilities of Linux for development and testing purposes. This choice underscores the project's commitment to using open-source and highly customizable platforms to achieve its technical objectives.

In the specified program configuration, the display resolution is set to one thousand by nine hundred pixels. This setting is crucial for defining the visual clarity and detail of the output on the screen. Within the context of CUDA, which stands for Compute Unified Device Architecture and serves as a parallel computing platform leveraging the power of NVIDIA GPUs, the grid and block sizes are meticulously configured to optimize the execution of parallel tasks.

4.5 Configuration

The grid size, defined as 63 by 57 (where 63 equals to 1000 divided by 16 then plus one, and 57 equals to 900 divided by 16 then plus one), represents the division of the computation domain into smaller, more manageable sections, each of which can be processed simultaneously across multiple threads in the GPU. This division is fundamental for achieving high performance in parallel processing tasks, as it allows for the efficient distribution and execution of complex computations across the extensive array of CUDA cores available in modern GPUs.

Similarly, the block size, set to 16 by 16, specifies the number of threads within each block. A block represents a group of threads that can cooperate by sharing data through shared memory and synchronizing their execution to perform collective operations. The choice of block size is critical, as it directly impacts the efficiency of memory usage and the overall performance of the CUDA program. By carefully selecting the block size, developers can ensure that the GPU's resources are utilized optimally, leading to faster execution times and more efficient processing of large datasets or complex algorithms.

This precise configuration of resolution, grid size, and block size in CUDA programming is essential for harnessing the full computational capabilities of GPUs. It allows for the effective parallel processing of tasks, making it possible to tackle computationally intensive applications such as image processing, scientific simulations, and machine learning algorithms with improved performance and efficiency.

Fig 6 shows the duration of producing the first five frames between CPU and GPU based on Fig 4 results.

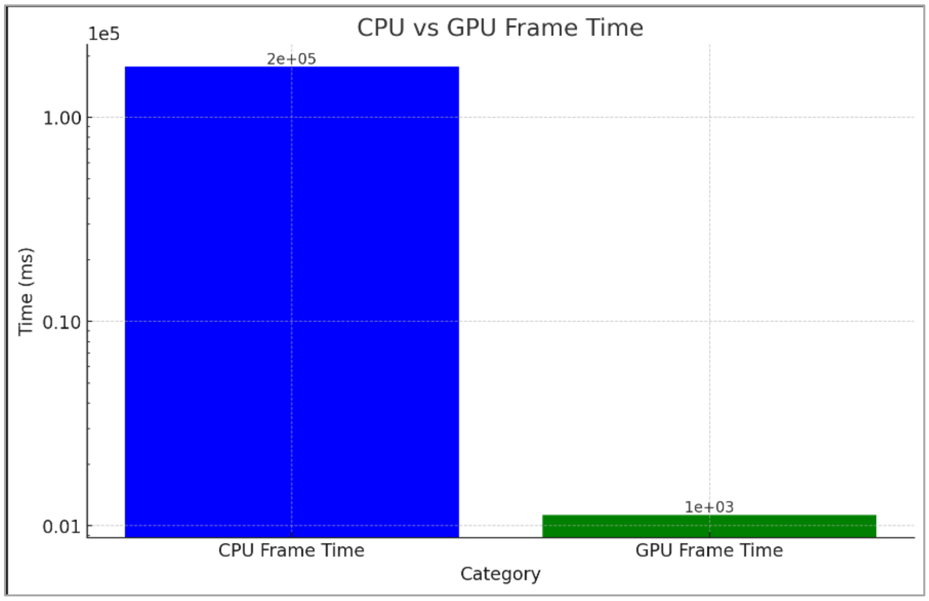


Figure 6 CPU VS GPU Frame Time

Fig 7 shows the time of producing single frame via CUDA.

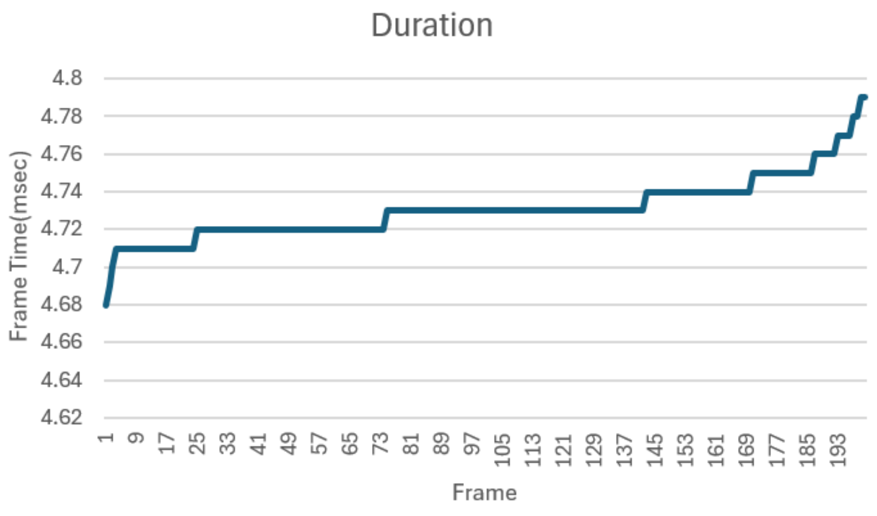
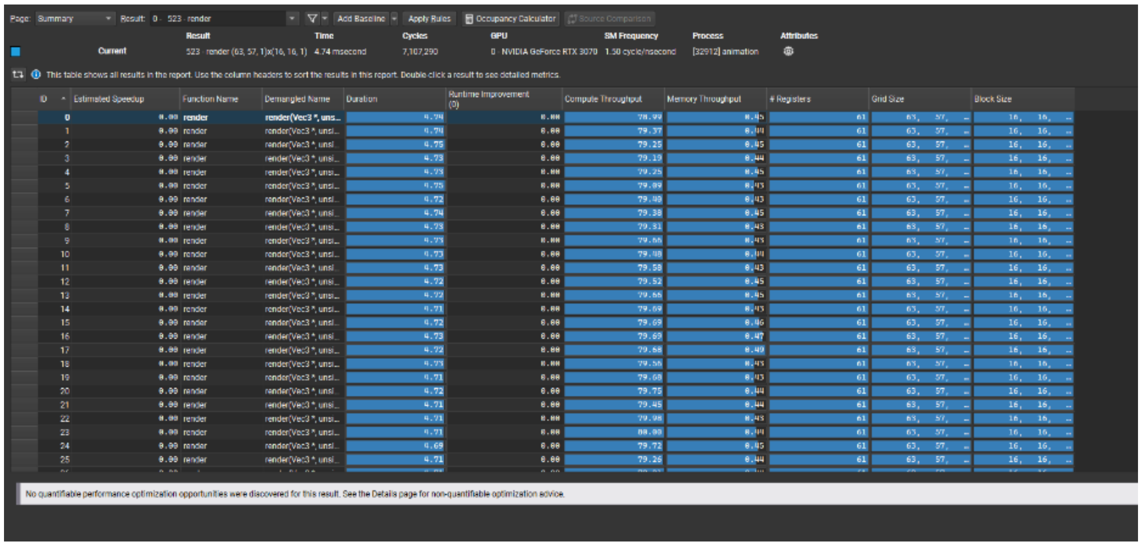


Figure 7 The time of producing single frame.

Fig 8 and Fig 9 present the profiler data from Nsight about CUDA processing.



	Result	Time	Cycles	GPU	SM Frequency	Process	Attributes
Current	523 - render (63, 57, 1)x(16, 16, 1)	4.74 msecond	7,107,290	0 - NVIDIA GeForce RTX 3070	1.50 cycle/nsecond	[32912] animation	

This table shows all results in the report. Use the column headers to sort the results in this report. Double-click a result to see detailed metrics.

ID	Estimated Speedup	Function Name	Demangled Name	Duration	Runtime Improvement (0)	Compute Throughput	Memory Throughput	# Registers	Grid Size	Block Size
0	0.00	render	render(Vec3 *, uns...	4.74	0.00	78.99	0.45	61	63, 57, ...	16, 16, ...
1	0.00	render	render(Vec3 *, unsi...	4.74	0.00	79.37	0.44	61	63, 57, ...	16, 16, ...
2	0.00	render	render(Vec3 *, unsi...	4.75	0.00	79.25	0.45	61	63, 57, ...	16, 16, ...
3	0.00	render	render(Vec3 *, unsi...	4.73	0.00	79.19	0.44	61	63, 57, ...	16, 16, ...
4	0.00	render	render(Vec3 *, unsi...	4.73	0.00	79.25	0.45	61	63, 57, ...	16, 16, ...
5	0.00	render	render(Vec3 *, unsi...	4.75	0.00	79.09	0.43	61	63, 57, ...	16, 16, ...
6	0.00	render	render(Vec3 *, unsi...	4.72	0.00	79.40	0.43	61	63, 57, ...	16, 16, ...
7	0.00	render	render(Vec3 *, unsi...	4.74	0.00	79.38	0.45	61	63, 57, ...	16, 16, ...
8	0.00	render	render(Vec3 *, unsi...	4.73	0.00	79.31	0.43	61	63, 57, ...	16, 16, ...
9	0.00	render	render(Vec3 *, unsi...	4.73	0.00	79.66	0.43	61	63, 57, ...	16, 16, ...
10	0.00	render	render(Vec3 *, unsi...	4.73	0.00	79.48	0.44	61	63, 57, ...	16, 16, ...
11	0.00	render	render(Vec3 *, unsi...	4.73	0.00	79.58	0.43	61	63, 57, ...	16, 16, ...
12	0.00	render	render(Vec3 *, unsi...	4.72	0.00	79.52	0.45	61	63, 57, ...	16, 16, ...
13	0.00	render	render(Vec3 *, unsi...	4.72	0.00	79.66	0.45	61	63, 57, ...	16, 16, ...
14	0.00	render	render(Vec3 *, unsi...	4.71	0.00	79.69	0.43	61	63, 57, ...	16, 16, ...
15	0.00	render	render(Vec3 *, unsi...	4.72	0.00	79.69	0.46	61	63, 57, ...	16, 16, ...
16	0.00	render	render(Vec3 *, unsi...	4.73	0.00	79.69	0.47	61	63, 57, ...	16, 16, ...
17	0.00	render	render(Vec3 *, unsi...	4.72	0.00	79.68	0.49	61	63, 57, ...	16, 16, ...
18	0.00	render	render(Vec3 *, unsi...	4.73	0.00	79.56	0.43	61	63, 57, ...	16, 16, ...
19	0.00	render	render(Vec3 *, unsi...	4.71	0.00	79.60	0.43	61	63, 57, ...	16, 16, ...
20	0.00	render	render(Vec3 *, unsi...	4.72	0.00	79.75	0.44	61	63, 57, ...	16, 16, ...
21	0.00	render	render(Vec3 *, unsi...	4.71	0.00	79.45	0.44	61	63, 57, ...	16, 16, ...
22	0.00	render	render(Vec3 *, unsi...	4.71	0.00	79.98	0.43	61	63, 57, ...	16, 16, ...
23	0.00	render	render(Vec3 *, unsi...	4.71	0.00	80.00	0.44	61	63, 57, ...	16, 16, ...
24	0.00	render	render(Vec3 *, unsi...	4.69	0.00	79.72	0.45	61	63, 57, ...	16, 16, ...
25	0.00	render	render(Vec3 *, unsi...	4.71	0.00	79.26	0.44	61	63, 57, ...	16, 16, ...

No quantifiable performance optimization opportunities were discovered for this result. See the Details page for non-quantifiable optimization advice.

Figure 8 Nsight Profile

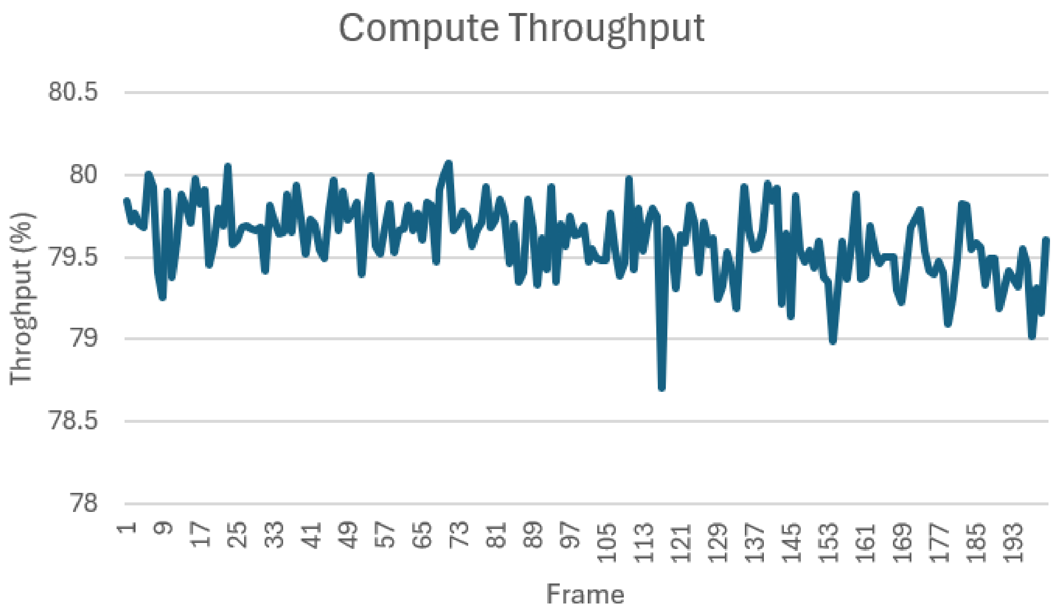


Figure 9 GPU Throughput pre frame.

Fig 10 presents the residual between each frame from CPU and GPU.

18

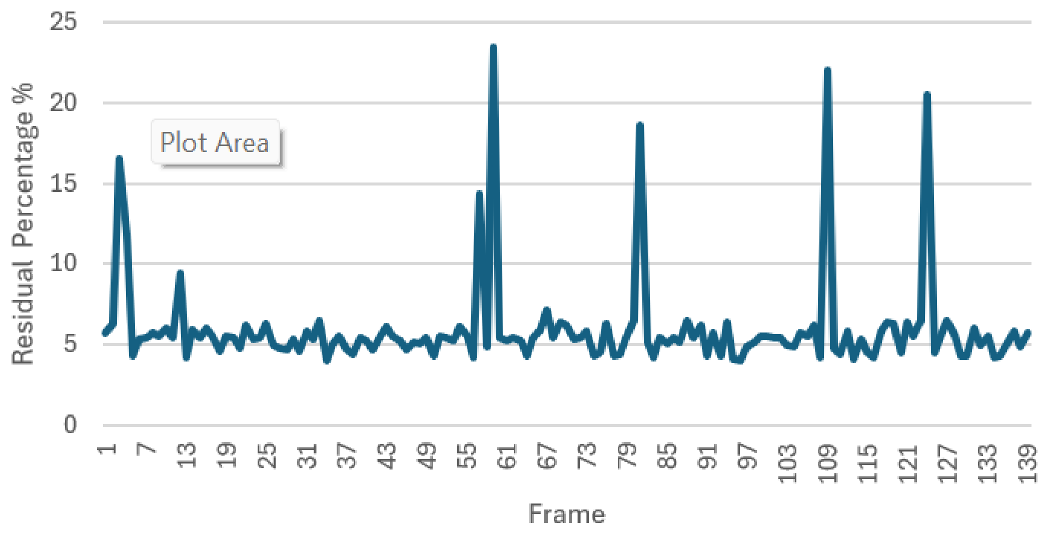


Figure 50 Residual Rate

The calculation method for the residual rate involves comparing the difference between frames generated by the CPU and those produced by the GPU, then expressing this difference as a proportion of the overall canvas. $\boldsymbol{R}(\%) = \left(\frac{\textbf{CPU frame} - \textbf{GPU frame}}{\textbf{resolution}}\right) \times \mathbf{100}$

5. Discussion/Analyzing

The figures referenced illustrate the significant performance disparity between GPU and CPU image processing capabilities. In Figures I and II, we observe that the GPU's frame generation and read-write operations are approximately 225.6 milliseconds, which starkly contrasts with the CPU's much longer duration of 35,329.4 milliseconds (about 35 and a half seconds). From this data, we can infer that the GPU's image processing speed is at least 139 times faster than that of the CPU.

Further examination of Figure III reveals that the frame generation time on the GPU is exceptionally swift, averaging around 4.72 milliseconds. This indicates that, theoretically, an NVIDIA RTX 3070 GPU could generate a video stream at an impressive rate of 12,711 frames per second. However, each frame requires CUDA to output to the host system and then be rendered into video by other software. Consequently, despite CUDA's rapid image generation capabilities, the actual video output rate cannot reach the theoretical maximum of 12,711 frames per second.

It is manifestly clear that the constructive collaboration of GPU acceleration with CUDA drastically outperforms traditional CPU processing in the realm of image handling. The GPU's robust architecture coupled with CUDA's efficient parallel processing framework can significantly reduce the time required for image and video generation tasks, making it an unparalleled choice for high-throughput image processing applications. This advantage is especially prominent in fields that demand real-time processing and high frame rates, such as virtual reality, advanced simulations, and real-time video analytics. The superiority of GPUs, augmented by the computational paradigm offered by CUDA, is indisputable for these computationally intensive tasks.

The analysis of the residual figures reveals that the GPU-generated version exhibits a mere 5% discrepancy when compared to its CPU counterpart, calculated as the ratio of differing pixels to the total number of pixels (residual rate = residual pixels / total pixels). This minimal deviation underscores the GPU's exemplary performance in image generation tasks, highlighting its efficiency and accuracy. Such a low error margin not only attests to the GPU's robust computational capabilities but also positions it as a preferable choice for high-fidelity image processing applications.

6. Relate Work

There are many rendering technologies out there and probably the most well-known is OpenGL [6]. OpenGL (Open Graphics Library) is a cross-platform, open-standard specification for developing 2D and 3D computer graphics. Initially developed by Silicon Graphics in the early 1990s, it provides a set of functions that allow programmers to render graphics and images directly to the screen, regardless of the operating system or hardware.

Comparing OpenGL with our ray tracing method [7], OpenGL primarily uses triangle rasterization method to render graphics. The first step is Vertex Processing. Vertex are points of a geometric primitive. After vertex processing, the vertices are assembled into geometric primitives. Then comes the rasterization process. During rasterization, the assembled primitives are converted into fragments. A fragment represents all the data needed to generate a pixel on the screen. This includes not just color, but potentially depth, stencil, and texture information. The rasterization process determines which pixels on the screen are covered by the primitive. Each fragment generated by rasterization goes through fragment processing, where a fragment shader can modify the fragment's properties before it is written to the screen. This can include applying textures, lighting calculations, and color blending.

7. Conclusions

7.1 Key Takeaway

Rasterization is used extensively in real-time applications like video games. It is faster and more efficient. Ray tracer on the other hand offers a more lifelike scene with better image quality. However, it comes at a cost of much higher complexity, which would cost more resources and time to render. By parallelizing the rendering algorithm, we were able to reduce the rendering time by a dramatic scale.

7.2 Future

Looking ahead, our project holds immense potential for expansion and improvement. Among the enhancements we are contemplating, a significant upgrade is transitioning from the educational but inefficient PPM image format to a more efficient output format in our rendering pipeline. Additionally, we plan to enrich the foundational aspects of our software by integrating basic classes for materials, shaders, textures, and employing parallelized methods to manage these components more effectively. We aim to introduce advanced rendering effects, including dielectrics, total internal reflection, defocus blur, and to parallelize these algorithms for better performance. Expanding the variety of object primitives by adding shapes like triangles and quadrilaterals is also on our agenda. To augment user experience and interaction, we are considering the development of a user interface that allows for visual and interactive modifications and viewing of scene objects. Lastly, incorporating physics interactions between objects, coupled with the acceleration offered by GPU parallelism, stands as a critical upgrade for our project's future developments.

8. References:

[1]“Accelerated Ray Tracing in One Weekend in CUDA,” NVIDIA Technical Blog. Accessed: Mar. 13, 2024. [Online]. Available: https://developer.nvidia.com/blog/accelerated-ray-tracing-cuda/

[2]“Netpbm home page.” Accessed: Mar. 13, 2024. [Online]. Available: https://netpbm.sourceforge.net/

[3]B. T. Phong, “Illumination for computer generated pictures,” Commun. ACM, vol. 18, no. 6, pp. 311–317, Jun. 1975, doi: 10.1145/360825.360839.

[4]B. Tuong, “Phong Shading Reformulation for Hardware Renderer Simplification,” 2006. Accessed: Mar. 13, 2024. [Online]. Available: https://www.semanticscholar.org/paper/Phong-Shading-Reformulation-for-Hardware-Renderer-Tuong/35372839369ca3dac57e2262ccae8cd7a25a3c76

[5]S. Peter, T. David Black, and S. Hollasch, “Ray Tracing in One Weekend.” Accessed: Mar. 13, 2024. [Online]. Available: https://raytracing.github.io/books/RayTracingInOneWeekend.html

[6]J. Kessenich, G. Sellers, and D. Shreiner, OpenGL® Programming Guide: The Official Guide to Learning OpenGL®, Version 4.5 with SPIR-V, 9th ed. Addison-Wesley Professional, 2016.

[7]“OpenGL News Archives.” Accessed: Mar. 13, 2024. [Online]. Available: https://www.opengl.org/Documentation/Specs.html