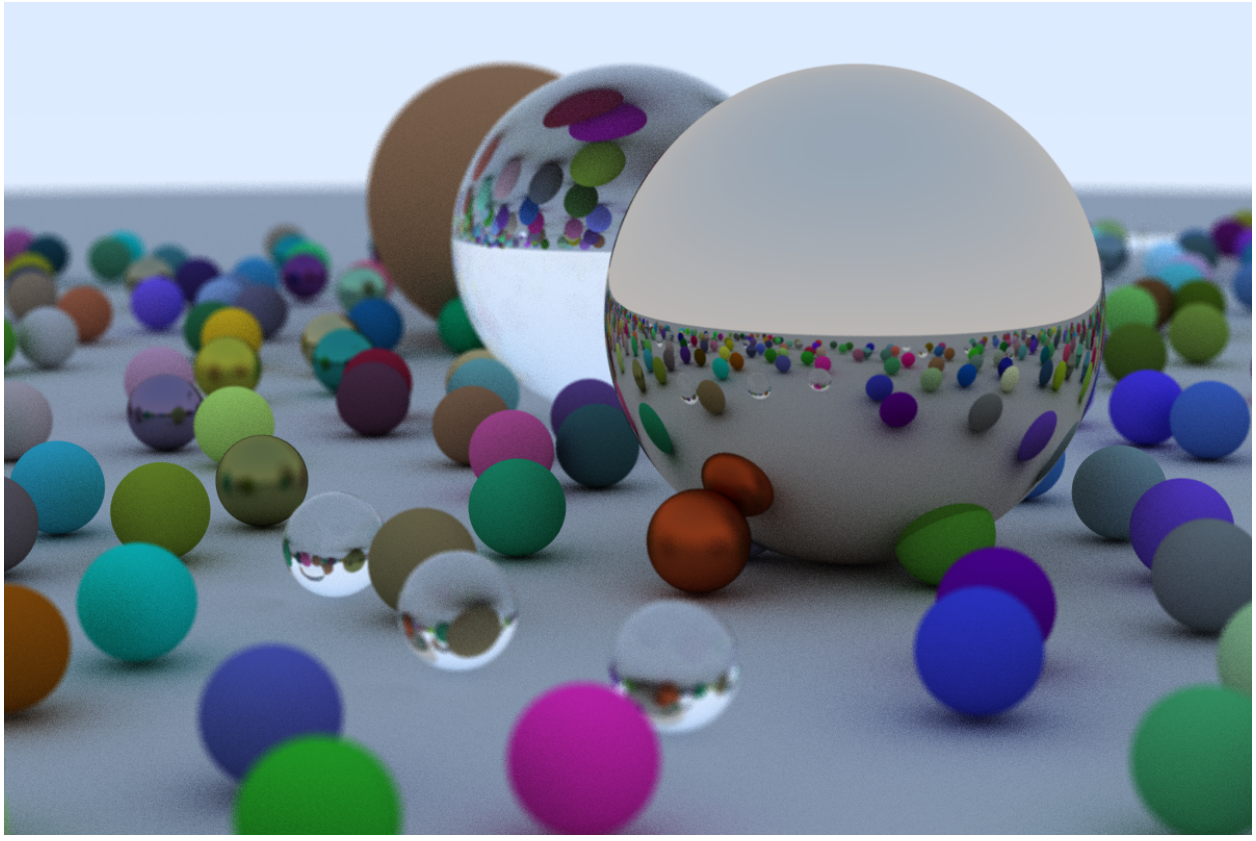
COMP410 Project Report
GPU-Accelerated Ray Tracer

Ameer Taweel (0077340) / Omar Al Asaad (0075155)



Introduction

Ray tracing is a powerful technique used in computer graphics to generate realistic images by simulating the behavior of light in a scene. With its ability to produce highly detailed images with realistic lighting and shadows, ray tracing has become an increasingly popular tool for creating visual effects in movies, video games, and other digital media.

Brief Overview

In this project, we explored the fundamentals of ray tracing and its applications in computer graphics. And we developed a GPU-accelerated ray tracer that can generate high-quality images using OpenGL's compute shaders.

We started by learning the underlying principles of ray tracing, including the physics of light and optics. After that, we implemented the ideas using C++, OpenGL, and compute shaders.

Our implementation achieves significant speedups compared to our reference implementation, which works on the CPU.

Technical Details Overview

- Ray Tracing Type: Path Tracing
- Features:
    - Parallel Processing via Compute Shaders
    - Different Material Types:
        - Diffuse Materials
        - Metal
        - Dielectrics
    - Uses Antialiasing
    - Positionable Camera
    - Defocus Blur
- Platform:
    - C++
    - OpenGL

Platform Choice

We couldn't use the standard OpenGL graphics pipeline for our project because OpenGL uses local illumination rather than global illumination. We needed to leverage the power of the GPU, but outside of the typical graphics pipeline. We tried many GPGPU (General-purpose computing on graphics processing units) platforms, such as OpenCL, CUDA, ArrayFire, HIP, and Kokkos. However, we encountered technical challenges with all of them.

We decided that OpenGL's compute shaders provided a viable solution for our needs as we explored different options. By utilizing compute shaders within the familiar OpenGL framework, we leveraged the GPU's parallel processing capabilities and performed the required computations efficiently without being limited by the graphics pipeline. Despite the initial hurdles we faced with other platforms, opting for OpenGL's compute shaders allowed us to overcome those challenges and complete our project.

Implementation Details

Our ray tracer starts with a scene description on the CPU, which includes objects, positions, materials, camera position, and other parameters. For now, we only support sphere objects. We can describe a sphere by specifying its center, radius, and material. Possible materials are diffuse (Lambertian), metal, and dielectric.

The CPU sends the scene data to the GPU using a Shader Storage Buffer Object (SSBO). SSBOs can store a lot of data, so they are suitable for storing scene data, even for large scenes. The OpenGL spec guarantees that SSBOs can hold 128MB at least. However, most implementations allow arbitrary allocation up to the GPU memory limit.

The CPU creates a 3D texture with the following dimensions:

- Width (W): Image Width
- Height (H): Image Height
- Depth (D): Samples-per-Pixel

Then, the CPU dispatches $W \times H \times D$ compute shader invocations, one per 3D pixel.

On the GPU, the compute shader starts by generating a ray of light. The rays originate from the camera position and pass through the corresponding pixel. After that, the shader tests the ray for intersections with the objects in the scene.

For each intersection point, we calculate the contribution to the ray color. The color contribution depends on the material type and properties of the object. For example, metallic materials have a fuzziness property. Reflections on less fuzzy metals are clear (like mirrors). However, they are vague on more fuzzy metals. Moreover, dielectric materials do not contribute to the ray color; they only change its direction.

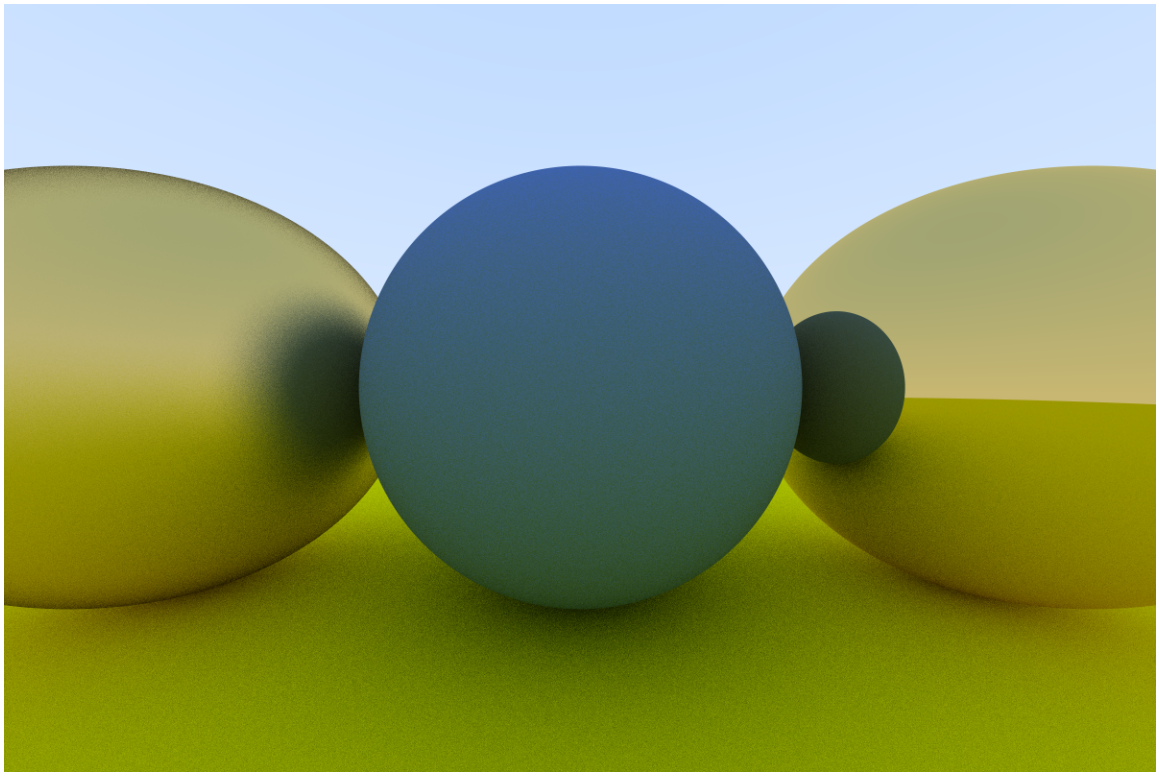


Figure 1: From left to right: Fuzzy Metal, Diffuse Material, Non-Fuzzy Metal

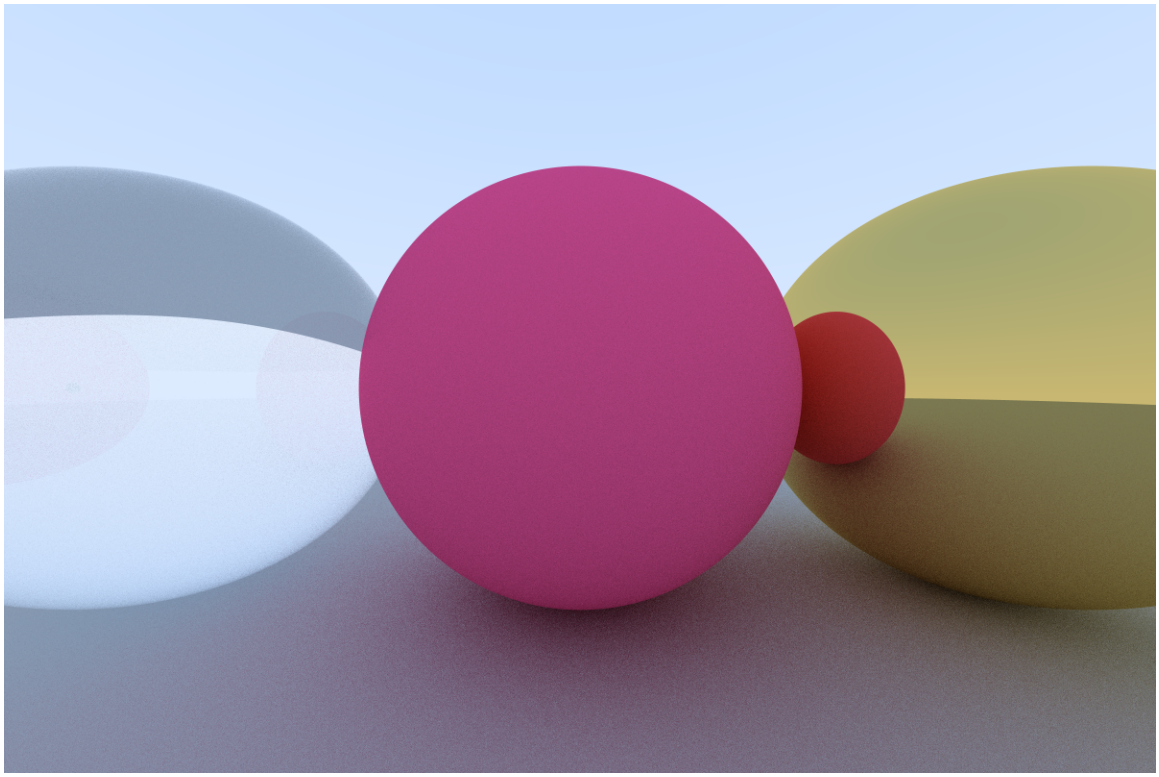


Figure 2: From left to right: Glass (Dielectric), Diffuse Material, Non-Fuzzy Metal

If the ray intersects with an object, it changes direction based on the material's reflective or refractive properties (we don't cast additional rays). Diffuse surfaces reflect the light in a random direction. However, metallic surfaces are more specular. Dielectric materials can refract light in addition to reflecting it, and that's why they look somewhat transparent.

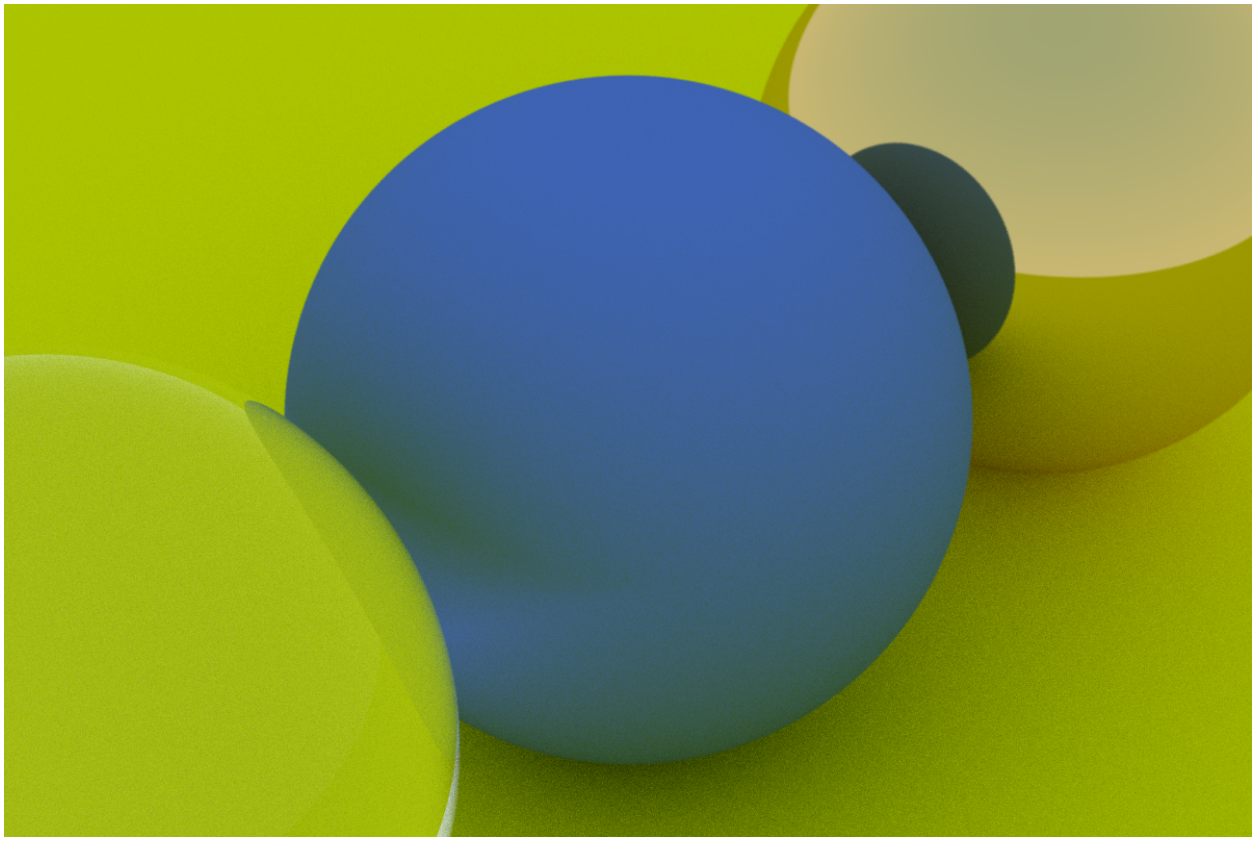


Figure 3: Positionable Camera

When the compute shader computes the colors of all the samples, we call another compute shader to average all the samples for each pixel and generate the final image. Averaging many samples is a technique called antialiasing. Antialiasing makes the generated image smoother and more realistic.

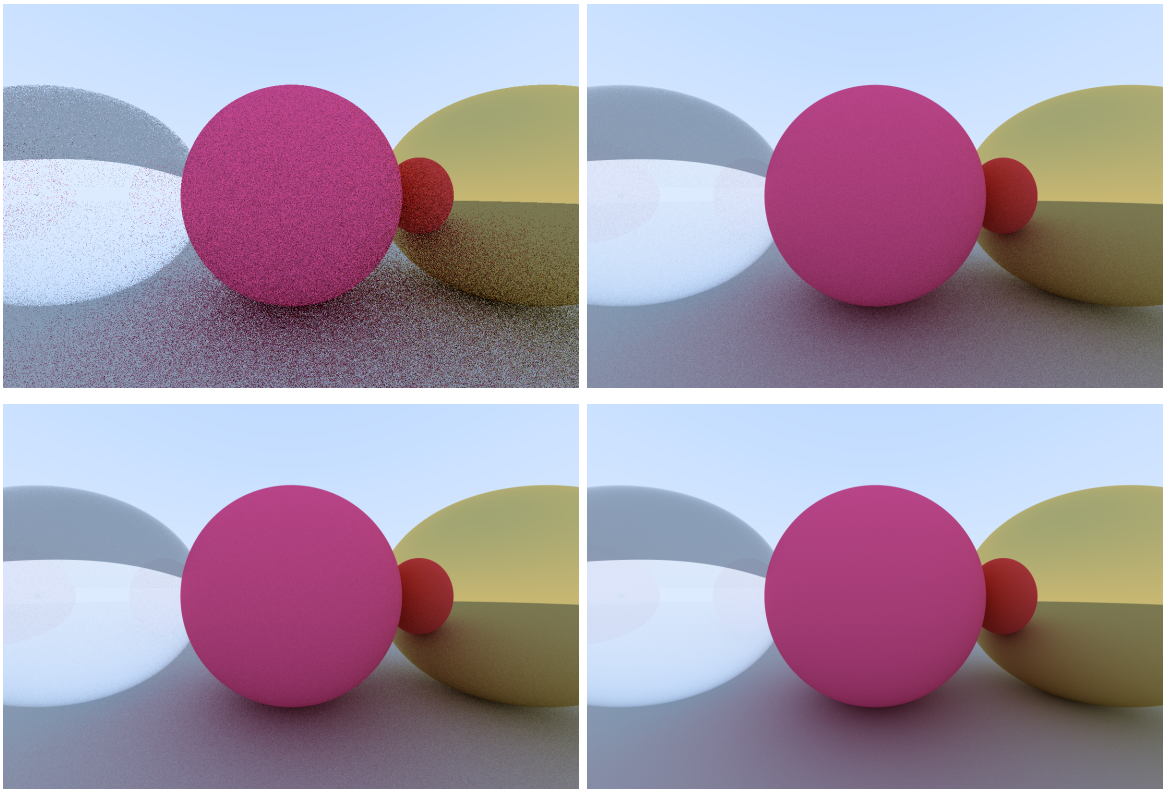


Figure 4: Antialiasing.
Samples-per-pixel from left to right, and top to bottom:
1, 25, 50, 500

We continue tracing the ray through reflections and refractions until we reach a maximum limit.

Performance

Our tracer computes all samples in parallel on the GPU using compute shaders. The GPU should be much faster than using the CPU for this task. Our tracer renders the following scene in about 7 minutes, while the reference CPU implementation needs about 18 minutes on the same hardware and with the same settings.

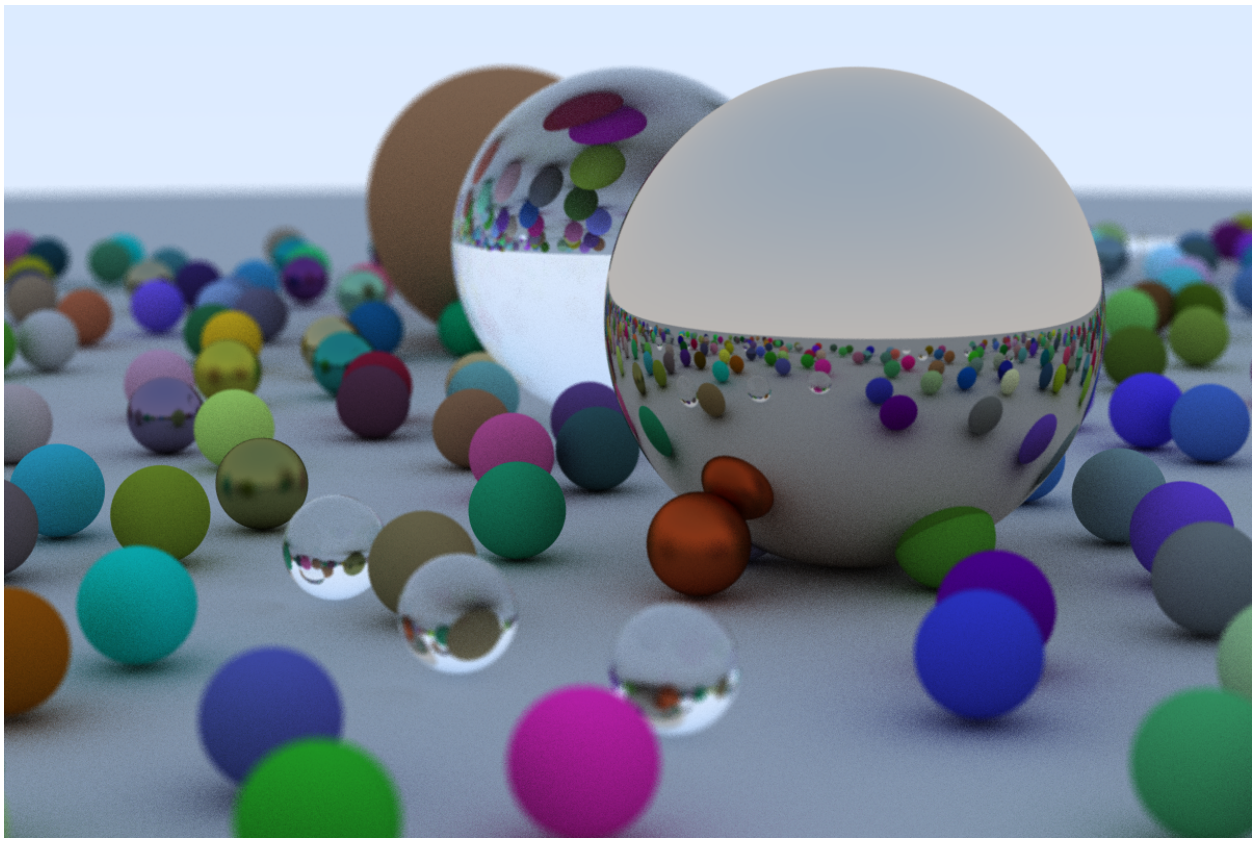


Figure 5: Our tracer renders this scene twice as fast as the CPU implementation

Our tracer is already more than twice as fast as the CPU implementation. However, we believe we can improve the performance even more via profiling and debugging to identify potential performance bottlenecks.

Conclusion

In conclusion, our project focused on developing a GPU-accelerated ray tracer using OpenGL's compute shaders. We explored the principles of ray tracing and implemented a path-tracing algorithm capable of rendering high-quality images with features such as different material types, antialiasing, positionable camera, and defocus blur.

Although we encountered technical challenges with other GPGPU platforms, we found that OpenGL's compute shaders within the familiar OpenGL framework provided a viable solution for leveraging the GPU's parallel processing capabilities.

Our implementation achieved significant speedups compared to the reference CPU implementation, rendering scenes in about half the time. Further performance improvements are possible through profiling and optimization techniques.

Overall, our project demonstrates the potential of GPU acceleration for ray tracing and highlights the benefits of using compute shaders within the OpenGL ecosystem.

References

- Ray Tracing in One Weekend.
  https://raytracing.github.io/books/RayTracingInOneWeekend.html
- OpenGL Wiki
  https://www.khronos.org/opengl/wiki/Main_Page
- Learn OpenGL
  https://learnopengl.com