Project Ray tracing - Computer graphics

Sari Pagurek van Mossel[*] Livia Agenssa Wöller [†]

[1] Uppsala University

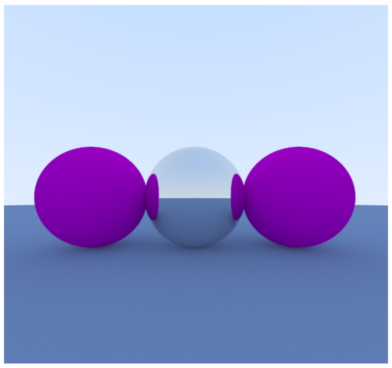

Figure 1: Ray traycing scene

1 Introduction

Ray tracing stands as one of the foundational techniques in computer graphics, offering a sophisticated method for generating visually realistic images. At its core, ray tracing simulates the behavior of light as it interacts with objects within a scene, enabling the creation of highly detailed and lifelike renderings. This technique tackles the fundamental challenge of rendering: accurately depicting the interaction of light with surfaces and materials to produce images that closely resemble real-world scenes.

In traditional rendering pipelines, techniques such as rasterization focus primarily on determining the visibility of objects and their appearance from the viewpoint of the camera. While rasterization excels at rendering real-time graphics, it often simplifies the lighting and shading calculations, leading to less physically accurate results. Ray tracing, on the other hand, adopts a more comprehensive approach, tracing rays of light as they propagate through a scene and interact with objects.

The ray tracing process begins with casting rays from the virtual camera into the scene, simulating the path that light would take as it travels from the observer's viewpoint. These primary rays intersect with objects in the scene, initiating a cascade of secondary rays that simulate the interactions of light with surfaces, materials, and volumes. [2]

Despite its computational intensity, ray tracing has witnessed significant advancements in performance and efficiency. Modern implementations of ray tracing leverage the power of multi-core CPUs, GPUs, and specialized hardware such as ray tracing accelerators to achieve real-time or near-real-time rendering speeds.

The aim of this project is to implement CPU-based ray tracing for global illumination. A pre-built code skeleton, following the structure outlined in the book *Ray Tracing in One Weekend* [1], was utilized for the implementation.

[*]e-mail: sari.pagurek-van-mossel.3621@student.uu.se
[†]e-mail: livia-agnessa.woller.9786@student.uu.se

2 Our Approach

At first, we implemented anti-aliasing to improve our image quality and reduce aliasing artifacts. To achieve this, we introduced a random_double() function. This function returns a canonical random number in the range $0 \leq n < 1$. We utilized this function to calculate the u and v coordinates for sampling around each pixel. By doing so, we were able to smooth out the edges.

We implemented rendering of diffuse surfaces with ray tracing and multiple bounces by defining a diffuse material model and setting up a ray tracing framework. When primary rays hit diffuse surfaces, secondary rays are generated to simulate multiple light bounces. Random directions for these secondary rays are generated using rejection sampling. The color of each bounced ray is calculated based on surface properties and surrounding illumination. A termination criteria was implemented to control recursion depth.

Next, we expanded our scene by adding additional spheres. To enhance the visual variety, we implemented the metallic (reflective) material for these spheres. This was achieved by creating classes for different materials. Specifically, we created a Material class to define how rays interact with the surface. When a ray hits a surface, such as a sphere, the material pointer in the hit_record is set to point at the material assigned to the sphere. Then, in the color() routine, the material pointer is used to determine how the ray scatters. Additionally, we developed a Lambertian class for diffuse materials and a Metal class for reflective materials. We also modified the Sphere class to include a material pointer. These classes are initialized and bound to the spheres in our scene during setup.

We added several controls to the ImGui window for managing aliasing, adjusting material properties of the spheres in our scene, and controlling the number of bounces. Additionally, we included options to display surface normals and toggle gamma correction on/off.

For the final step, we added a low-polygon mesh (a rabbit) to our scene. To expedite the rendering process, we calculated and stored a bounding box around the mesh. Then, during the tracing step, we performed a bounding volume test. This optimization accelerated the rendering process.

The result for the ray traying you can see in Figure 1.

References

[1] Peter Shirley, Trevor David Black, and Steve Hollasch. *Ray Tracing in One Weekend*. https://raytracing.github.io/books/RayTracingInOneWeekend.html. Aug. 2023. URL: https://raytracing.github.io/books/RayTracingInOneWeekend.html.

[2] J. Turner Whitted. *Ray-Tracing Pioneer Explains How He Stumbled into Global Illumination*. en-US. Aug. 2018. URL: https://blogs.nvidia.com/blog/ray-tracing-global-illumination-turner-whitted/ (visited on 03/14/2024).