RAY TRACING IN VULKAN: WRITING A HARDWARE ACCELERATED REAL-TIME RAY TRACER FOR GAMES USING VULKAN RAY TRACING EXTENSIONS

by

Sergei Grigorev

May 2025

Introduction

Rasterization with programmable vertex and fragment shaders has been the dominant way of rendering graphics in video games for over two decades. In contrast, the film and animation industries have long relied on ray tracing to achieve stylized or photorealistic imagery using offline renderers. Until recently, the computational cost of ray tracing made it impractical for real-time use in games. However, advancements in GPU hardware and the introduction of dedicated ray tracing cores have made real-time ray tracing increasingly viable.

For the past few years, games have increasingly relied on real time ray tracing to render effects such as reflections, shadows, and global illumination. Notably, Indiana Jones and the Great Circle (MachineGames, 2024) is one of the first high-budget commercial games to require GPU hardware ray tracing support to be playable. Ray tracing is likely to become a standard feature in modern rendering pipelines, as researchers and hardware vendors work on making the technology more performant.

Given this context and my interest in graphics programming, I decided to explore ray tracing for my graduation project. During the specialization course I built a game engine with Vulkan and therefore had some familiarity with the API and a foundation to build upon. Since Vulkan also offers ray tracing extensions, it was a natural choice for me to continue using it. Vulkan is a modern API with growing adoption, which makes me confident that the knowledge and skills developed during this project will remain relevant well into the future.

The scope of the project was defined in an 8-week plan. My primary goal was to write a Whitter-style ray tracer, sometimes referred to as conventional or classical ray tracing. This ray tracing algorithm is based on a seminal paper by Turner Whitted (Whitted, 1979). An optional stretch goal was to implement a path tracer; however, I instead decided to focus on polishing the base ray tracer and added a few extra features such as SSAA, alpha clipping and skybox support.

This report documents the process, challenges and results of writing a hardware accelerated real-time ray tracer using Vulkan ray tracing extensions.

This report is organized as follows:

- The **Background** section introduces key concepts, including Whitted-style ray tracing and acceleration structures.
- The **Results** section showcases the final results of the project.
- The **Reflection** section looks at the project plan and scope retrospectively and documents the challenges encountered during development.
- The **Conclusion** summarizes the knowledge and skills gained, as well as the project's relevance to my future career in the games industry.

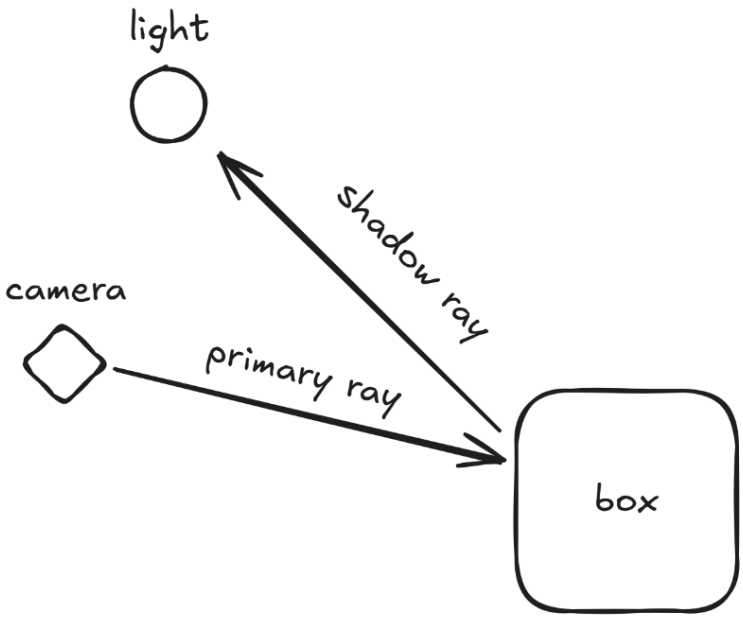
Background

In this section, I briefly explain Whitted-style ray tracing and the concept of acceleration structures, to provide necessary background for the reader.

Whitted-style ray tracing

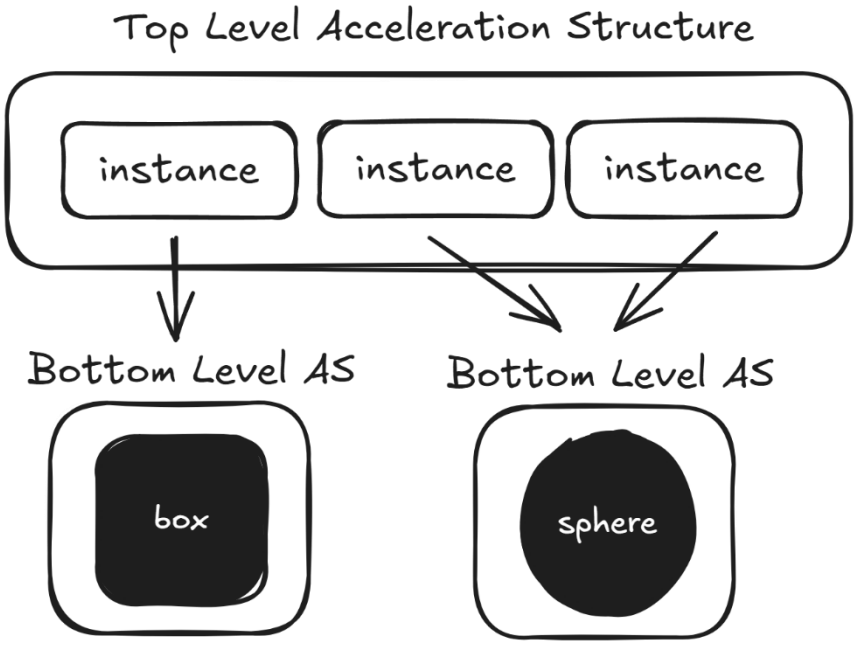
The Whitted-style ray tracing algorithm works as follows. For every pixel of the image, a ray is shot from the camera into the scene. If the ray intersects an object, we can calculate the color of the pixel at the point of intersection. If the material is reflective or refractive, additional rays can be recursively spawned in the reflection or refraction directions (not shown on the diagram). To compute shadows, a ray is cast from the hit point in the direction of the light source. If the ray intersects another object on the way, the point is considered in shadow. All contributions are added up and the final color is passed to the viewer.

This method produces accurate hard shadows and perfect reflections. It's also cheap and can produce a stable and clean, albeit aliased, image with just 1 sample per pixel. The limitations of the method are essentially the opposite of its strengths. It lacks soft shadows and glossy reflection and does not compute global illumination.



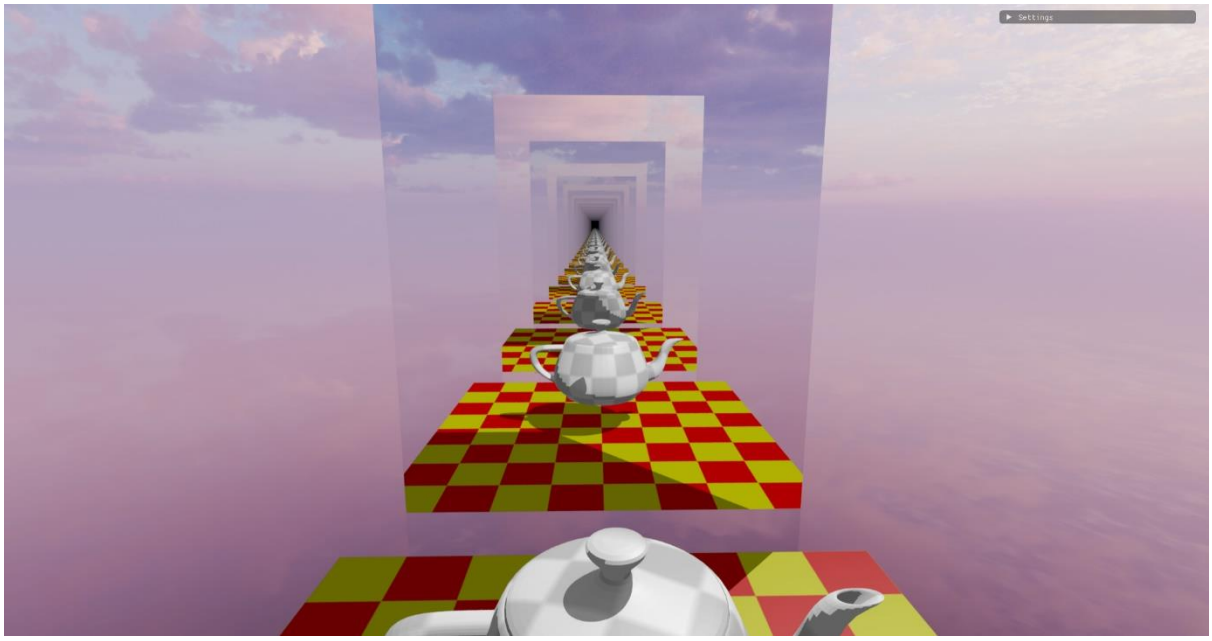
Acceleration structures

Hardware accelerated ray tracing currently uses a form of spatial partitional known as the bounding volume hierarchy (BVH) to make ray tracing more efficient by reducing the number of ray-triangle intersection tests during rendering. Ray tracing extensions, therefore, require you to organize the geometry into an acceleration structure to be submitted for rendering. Two types of acceleration structures are exposed to the user on the API level: bottom level acceleration structures (BLAS) and top level acceleration structures (TLAS). A BLAS typically contains an individual 3D model, whereas a TLAS represents an entire scene and contains one or multiple instances. An instance contains a reference to a BLAS and a transformation matrix. A TLAS can be updated with new transformation matrices, allowing dynamic objects to move without paying the full cost of rebuilding the TLAS.



Results

The results of 8 weeks of development is a fully functinal real-time ray tracer. The final renderer features hard shadows, mirror-like reflections, and Blinn-Phong shading. Additional features include supersampling anti-aliasing (SSAA), alpha-testing, and skybox support. I have also implemented a GUI that allows tweaking scene variables like the direction of the main light, camera parameters, samples per pixel and changing position, rotation and scale of every instance in the scene. The following images showcase the final visual output of the ray tracer developed during this project.



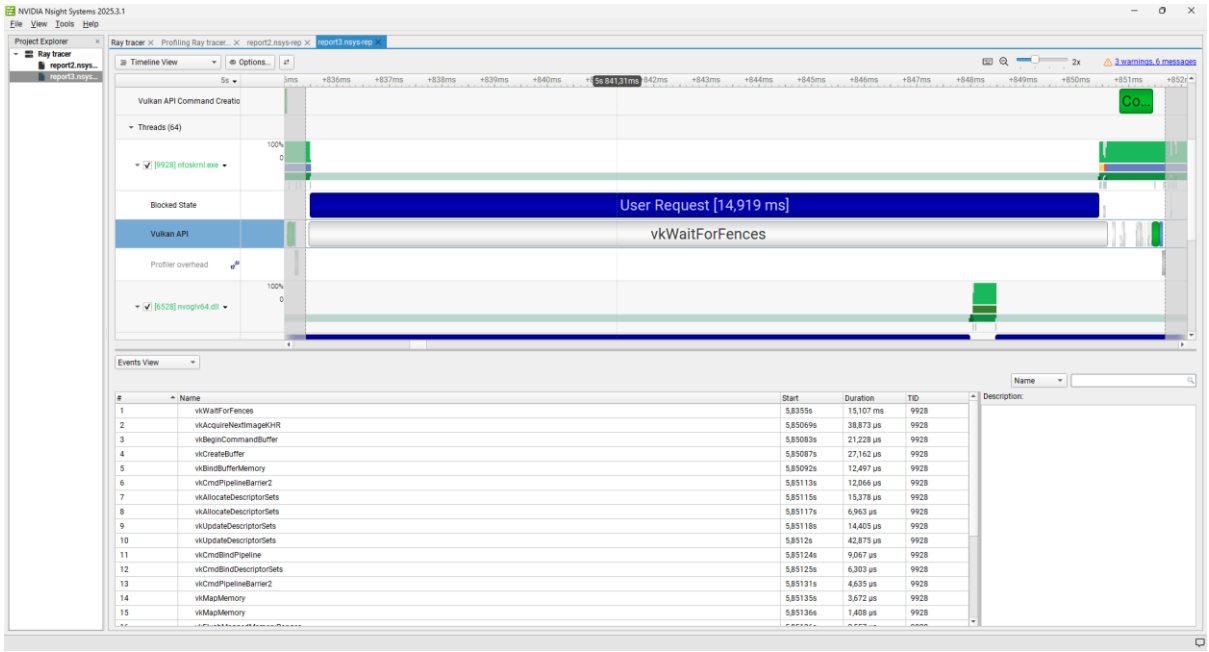
The first image shows the model of Sponza Palace’s atrium, commonly used in graphics programming as a test scene.

The next image is a scene with 2 mirrors opposite of each other and a teapot in the middle, reflecting “ad infinitum” (maximum depth of reflection is set to 50).

Both scenes were rendered at 32 samples per pixel.

Optimization

Although there is room for more extensive optimizations, I have implemented BLAS compaction, which reduced the memory footprint of the Sponza scene from 17.8 MB to 15.4 MB. With the help of Nsight Systems, I profiled the renderer and ensured that the TLAS is updated efficiently, which previously blocked the main thread. The following image is a typical frame after optimizations. The frame begins and spends most of the time waiting for the GPU to finish rendering the previous frame. Note that there is just a single vkQueueSubmit at the end of the frame. Before optimizations, the thread was blocked multiple times and there were multiple command buffer submissions.



Reflection

Project Retrospective

I completed the development of a real-time Whitted-style ray tracer, achieving most of the core functionality as planned. Week by week, I largely stayed on schedule, although some tasks took more time than anticipated while others were completed more quickly.

One planned feature I chose not to implement was refraction. Although it produces visually interesting results, its implementation is conceptually similar to reflection. Instead, I added SSAA, alpha testing, and skybox support, which allowed me to explore the ray generation, any hit and miss shaders more.

In the final two weeks, I made a decision to prioritize polish rather than striving to complete the stretch goal of implementing a path tracer. Sticking with Whitted-style ray tracing gave me the opportunity to refine the renderer, work on optimization, and ensure that the final product was functional.

Ultimately, I believe the project's scope held up well because I was conservative during initial planning and set realistic time estimates. For me, this project reaffirmed the importance of balancing ambition with practicality. I undertook a challenging project for my current skill level and feel that, while part of me wanted to produce visually more impressive results, I chose the scope well and didn't take on more than I could manage.

Challenges

Building Acceleration Structures

One of the first big tasks in my plan was to build acceleration structures and it became the first major challenge. The part of the Vulkan API related to acceleration structures is somewhat convoluted and wasn't intuitive to me. Before building an acceleration structure, you must create a sufficiently big scratch buffer. To determine what size is needed you must fill out several structs to query the build size of the acceleration structure. Then after creating the scratch buffer you need to fill out several more structs, partially reusing earlier data during the actual build process. It's not very straightforward. There's also an alignment requirement for the scratch buffer. The NVIDIA samples, while being an otherwise excellent resource, add unnecessary complexity to this part instead of clarifying the workflow. What helped me overcome this challenge

was tracking down the most minimal reference code available. By reading the example thoroughly, I was able to understand the correct sequence of steps.

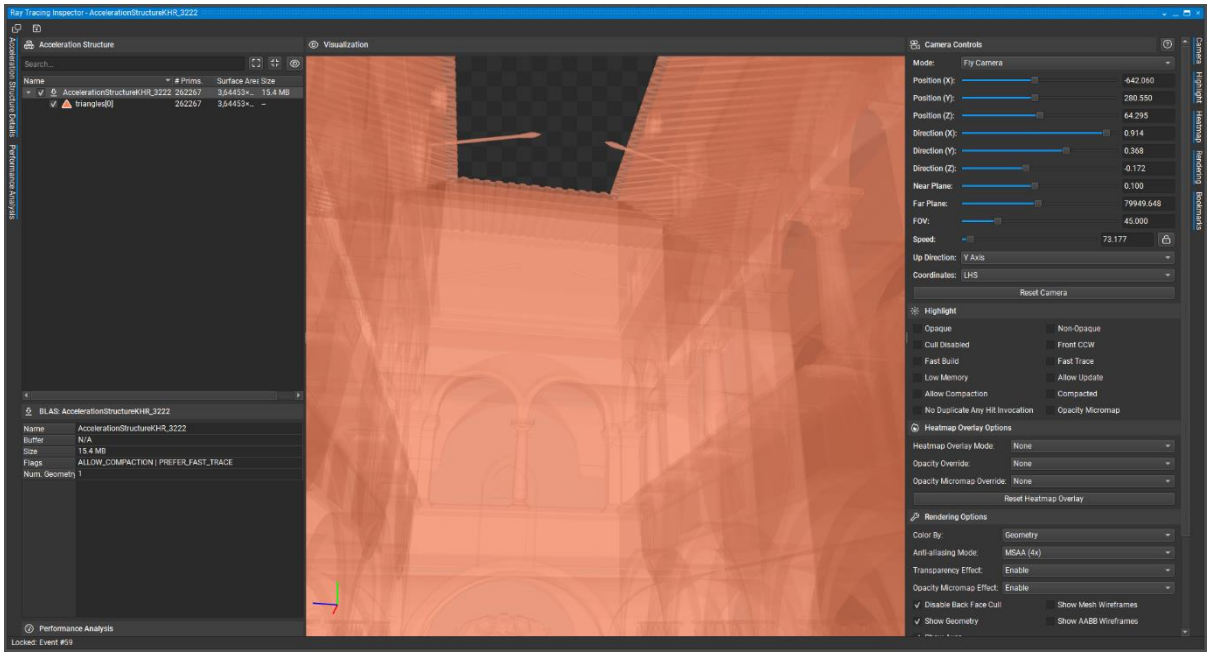
Materials

In traditional render pipelines, objects are often grouped by material. Shaders may only need access to a single material at a time and descriptor set that references a buffer with material properties only needs to be bound when drawing objects of that material. In ray tracing, rays are shot for every pixel rather than for every object and it is impossible to know in advance which object in the scene a ray will hit. Therefore, shaders must have access to all materials in the scene, which requires uploading all material data to the GPU before tracing rays.

Adapting to this paradigm was initially difficult. It required enabling several GLSL extensions and rethinking how my engine handled materials and textures. At the same time, I added support for MTL materials to my OBJ model loader, which further complicated the process. While I had originally allocated a day or two to set up descriptor sets, the complexity of making sure the GPU has access to all the data ended up extending this task to a full week. I overcame this challenge by being patient and making progress in small steps.

Debugging graphics

Debugging graphics applications can be challenging, especially when working at a low level. Vulkan validation layers help, but sometimes they just print out synchronization errors without providing meaningful insight into the root cause. In these cases, other tools can be helpful. For this project I had to learn how to



use vendor-specific software called Nsight Graphics, because RenderDoc, a tool I'm familiar with, doesn't support ray tracing. While "debugging" isn't a challenge that can be solved once and for all, becoming familiar with different tools that are available to a graphics programmer helped me overcome some of the obstacles along the way. For example, I had a bug where only part of a mesh was being rendered on screen. Using Nsight Graphics to inspect the acceleration structure, I discovered that only a portion of the mesh had been uploaded to the GPU. The issue, as it turned out, originated in my model importer.

Conclusion

Over the course of 8 weeks, I have successfully implemented a Whitted-style real-time ray tracer using Vulkan. Working on this project has been a major step in developing my skills in graphics programming and provided hands-on experience with a wide range of industry-relevant tasks. Throughout the development process, I modified the existing codebase to meet new requirements, built a render pipeline, and wrote shaders. I learned the theory behind ray tracing and got practical experience working with Vulkan ray tracing extensions. I also learned to debug graphics applications with Vulkan validation layers and Nsight Graphics, and to profile and optimize with Nsight Systems. In addition to aforementioned hard skills, this project also helped me develop essential soft skills such as effective planning and the ability to adapt and prioritize tasks to meet development goals within time constraints.

The knowledge and skills I gained will be relevant not only in the games industry but other industries as well, for purposes like CAD or data visualization. This project is useful for my future career not only as a path to a graphics programming job but also as a demonstration of my ability to work on highly complex tasks.

In conclusion, this project provided valuable experience in real-time rendering and low-level graphics programming. I have laid a foundation to explore more advanced ray tracing algorithms such as unidirectional or bidirectional path tracing, physically based rendering, denoising and other advanced topics. Moving forward, I am well-equipped to tackle more complex rendering challenges and continue building expertise in graphics programming.

References

Whitted, J. T. (1979). An improved illumination model for shaded display. *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques*, 14–23. https://doi.org/10.1145/800249.807419

MachineGames. (2024). *Indiana Jones and the Great Circle* [Video game]. Bethesda Softworks. https://indianajones.bethesda.net/en-US