Danmarks Tekniske Universitet

DTU

02562 - Rendering
GPU accelerated ray tracing with Metal

Authors

Naglis Naslenas - s223312

December 18, 2024

Contents

1 Introduction

In this section i will introduce the project and what it aims to achieve. I will also explain the motivation behind the the project.

This project similarily to proposed project of *GPU Ray Tracing Using OptiX* aims to accelerate ray tracing using GPU hardware. However, due to not having Nvidia GPU, it was chosen to explore Metal API instead to explore if we can leverage the recently introduced hardware accelerated ray tracing capabilities (in 2023 with M3 series). The idea is to have a similar scene rendered using Metal as was done throughout the course with WebGPU and investigate the steps needed to set up a ray tracing pipeline in Metal.

Aditionally, this project aims to observe the possibilities regarding the tools for profiling and debugging the ray tracing applications for the sake of interest in being potentially a good tool to develop shader and/or ray contribution calculation logic in with less of a hassle when compared to WebGPU where the shader and it's individual pixels cannot be directly debugged.

While writing my own ray tracer using swift/objetive-c from scratch would be out of scope for this project,the project uses an example from Apple's official documentation for metal API.

I have looked at example projcts supplied by Apple [1] [2] [3] to disect the code and understand how to Metal is set up for ray tracing.

2 Methodology

This section will describe the tooles used for this minor project. It will also describe the approach used and the reasoning behind the choices made.

2.1 Tools

Being an Apple-only API, Metal API is only available on Apple devices. This means that the project requires a Mac computer to run the code. For IDE Xcode was used, as it is the place where debugging and profiling tools are available.

Xcode allows to quickly set up a project with Metal API support, for example by starting a new project as a game project. This automatically takes care that the project has a view that can be used for rendering and therefore will not be covered in this report.

Newly made projects are allowed to be set up to have code written in both in Swift and Objective-C. This project uses Objective-C as the language of choice, as the example code uses Objective-C. Swift comes with a more user-friendly syntax and is significantly easier to read and write, howerer, would require to re-write the project code to Swift. While it is great that different programmming languages can be used to achieve the same results, not being familiar withe either of the languages, it makes it significantly more difficult to understand the provided example code and explanations, when some examples use Objective-C and some use Swift.

2.2 Ray Tracing in Metal

This subsection will delve into the details of how ray tracing applications are set up with Metal.

2.2.1 Building a scene

Geometry in a scene can be defined as trianges, bounding boxes with a custom intersection test, or as curves. These different types of geometries come with built-in intersection function. Focusing on a triangles, vertex buffer, index buffer and triangle count is used to make an acceleration structure. Acceleration structure is used for ray tracing application to efficiently query the geometry and find itersections with the rays by partitioning the geometry.

Metal builds this acceleration structure based on the geometry provided using GPU which is then used in subsequent ray tracing passes to find the intersections with the rays. Large scenes can be optimized by combining primitive acceleration structures to instanced acceleration structures transformed to different positions and sizes to represent the entire scene. This allows for a memory efficient way to represent a scene with many objects. Instance acceleration structures can further contain other instance acceleration structures to represent a hierarchy of objects in the scene.

Debugging option of caputing GPU load on Xcode gives access to acceleration structure viewer to see the hierarchy of the acceleration structures and the geometry contained within them. This can be used to move around within the scene, click on to select and verify that the acceleration structure is built correctly, highlight its bounding box and see and that the geometry is correctly partitioned.

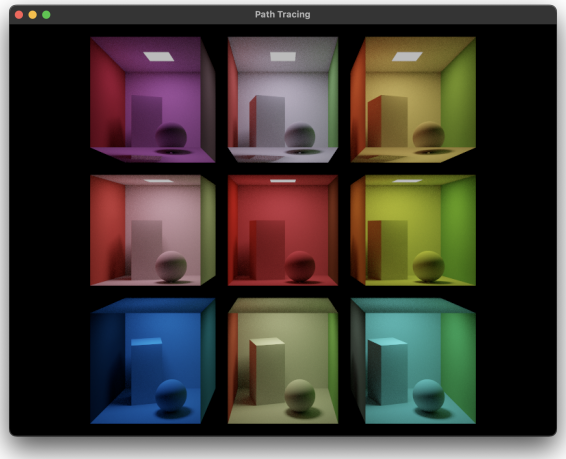

Figure 1: Rendered scene

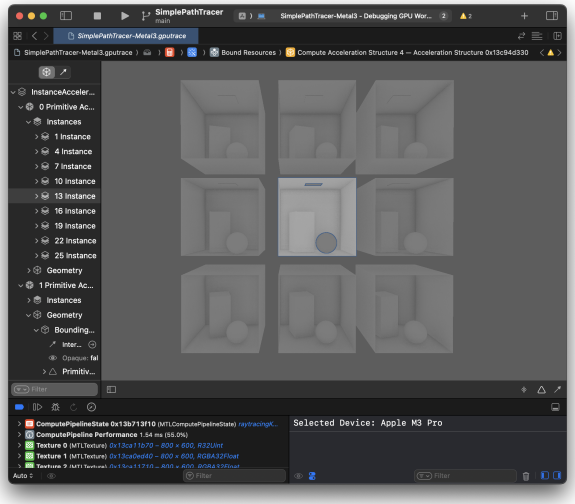

Figure 2: Acceleration structure viewer

2.2.2 Initialization

Much like with WebGPU, Metal API needs set up to be able to render the scene. This includes setting up the device, configuring buffers, creating acceleration structures from the scene. Then, the pipeline is set up assigning vertex and fragment shaders to the application. Additionally, a Metal compute pipeline is created that uses ray tracing kernel to calculate the color of the pixels in the scene. Essentially, a group of threads (in this case 8x8) is dispatched to calculate the color of the pixels in the scene. each thread calculates the color of a single pixel of an output image.

2.2.3 Ray tracing kernel

Ray tracing kernel uses unique thread ids to reference a pixel within the bound of an output image. A random offset is added using a random texture for an anti-aliasing effect. Disregarding the differences between how the scene was described (acceleration structure vs BSP tree in WebGPU), the ray tracing kernel follows same logic as the fragment shader we've implemented for WebGPU, generating a ray from the camera to the pixel and calculating the color of the pixel based on the intersections with the geometry and the lights in the scene. Shading calculations are also within the kernel, writing the color of the pixel to a texture.

Listing 1: Ray tracing kernel

```
    // Intersect rays with a primitive acceleration structure

[[kernel]]
void trace_rays(acceleration_structure<> as, /* ... */) {
  intersector<triangle_data> i;

  ray r(origin, direction);

  intersection_result<triangle_data> result = i.intersect(r, as);

  if (result.type == intersection_type::triangle) {
    float distance = result.distance;
    float2 coords = result.triangle_barycentric_coord;

    // shade triangle...
  }
}
```

Kernel threads are continuously dispatched to render the entire image, at which point it continues to accumulate the color of the pixel based on more samples of the pixel to continuosly refine the value and reduce the noise in the image.

2.2.4 Vertex and Fragment shaders

The vertex shader in the program just used just was used to position a quad that covers the entire screen and to calculate the UV coordinates for the fragment shader.

The fragment shader then just samples the target texture, applies simple tonemapping, and writes the resulting color to the output color buffer.

2.2.5 Draw call

Launches the grid of kernel threads described in 2.2.3 to render the scene, swaps between accumulation and render textures for the next frame, and presents the rendered image to the screen.

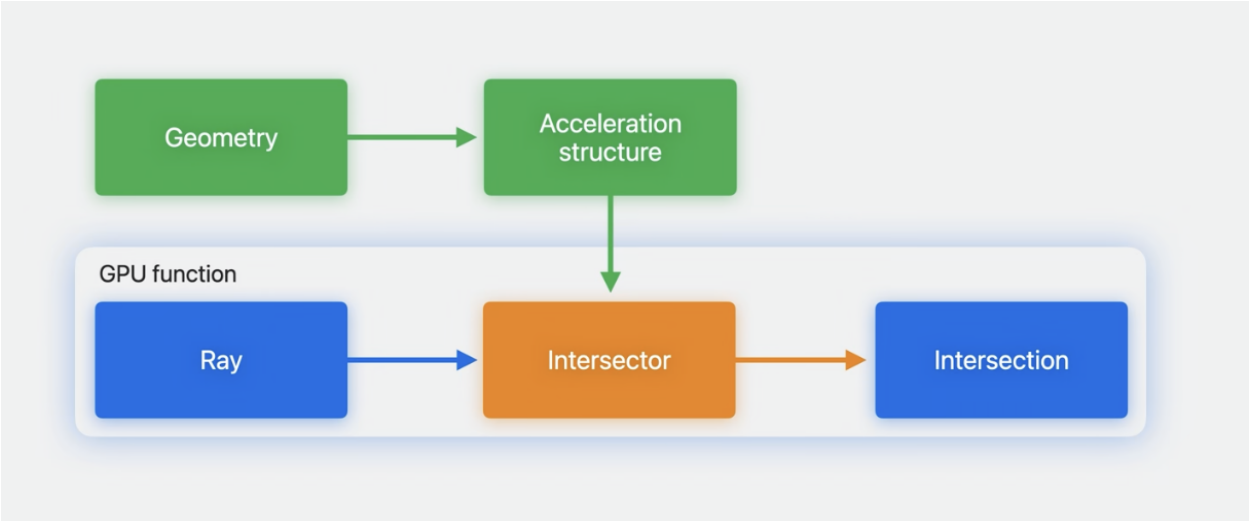

Figure 3: Ray tracing pipeline [4]

3 Implementation

This section will describe the implementation of the project. It will describe the code and the changes made to the example code.

As mentioned, the project is based on the example code available in Apple's documentation on Metal API. This provides a codebase that renders Cornell Box scene instanced 9 times with differently colored lights to showcase the setup for ray tracing pipeline and its available features as well as the optimization techniques to improve performance.

While the example was already set up to render a predefined scene, the big challenge was to understand exactly what code is executed and at what time and what it is used for. As previously stated, code being in Objective-C made it significantly harder to disect it and understand what is being executed.

Firstly, the template code already had a a custom intersection function to be called when the ray intersects with a defined bounding box for a sphere. I did a cleanup of the code to

only include the necessary parts for the ray tracing pipeline needed to render a triangle mesh making it easier to understand the code while also getting a better understanding of the flow.

3.0.1 Importing OBJ files

To be fully in control of the scene and its calculations i have initially removed the instancing loaded objects in the scene and instead added functionality to import an OBJ file as a triangular mesh. While i wasn't as successful in using Model I/O framework to import the OBJ file as mesh, OBJ file can be read as a string and parsed to create vertex, index buffers and triangles required to build the acceleration structure. This was done by adding a new function to the project that reads the strings from the OBJ file and parses them to create said buffers that are then used to build the acceleration structure (see Appendix 6.1). The imported geometry was verified by using the acceleration structure viewer, the results of which can be seen in Figure 4. Image showcases multiple instances of bunny and a plane object and acceleration structure viewer verifies that to be the case.

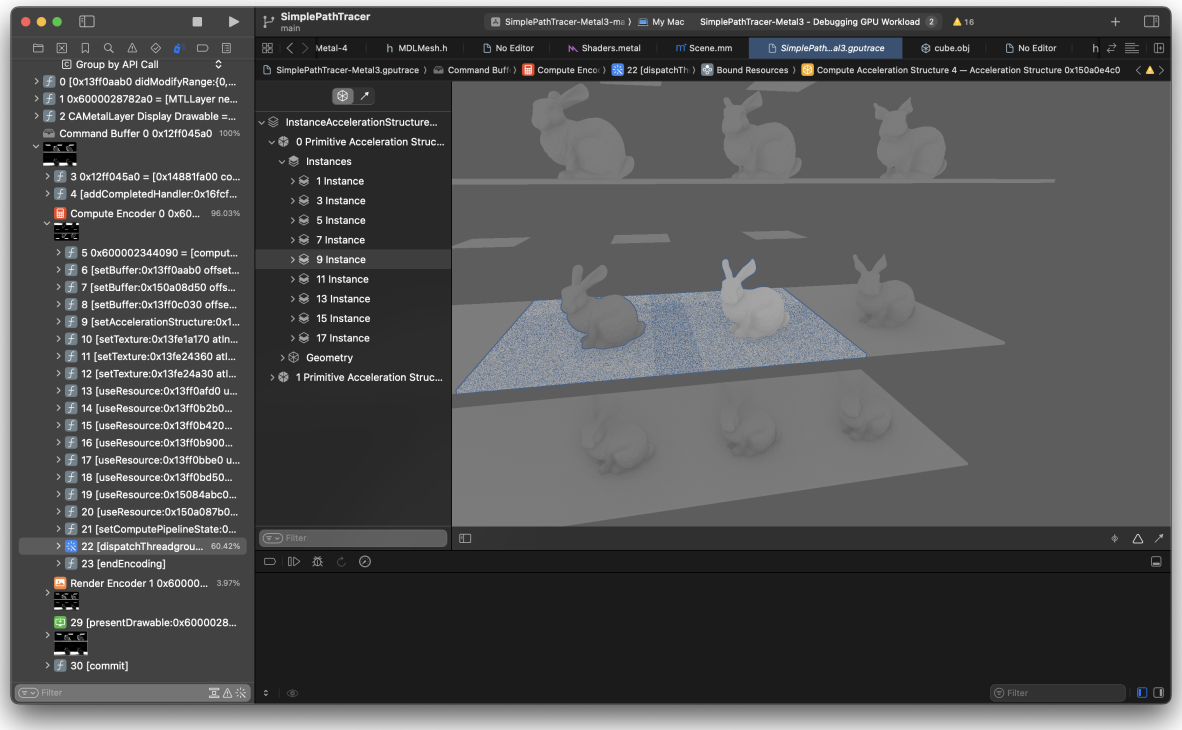

Figure 4: Acceleration structure viewer with imported OBJ file instanced 9 times

Having my own scene to work with, gives me a better understanding of the code behind the setup stage of the application and purpose of used data buffers and lets me then experiment with the ray tracing kernel to see how the scene is rendered and how the shading calculations are done.

3.0.2 Shader debugging

As mentioned, the ray tracing kernel closely resembles what we had to work with throughout the course, so just to experiment with the code, i've created a function to sample a directional light and adjusted the shading calculations accordingly to disregard the distance to the light source. This was done to interact with shader debugging tools that are available in Xcode.

As mentioned, one of the points of interest for this project was to see how the shader code can be debugged. While there is a whole suite of tools available for debugging shaders in Metal API, the ones i found useful was to use the GPU capture tool to capture the GPU load, see the acceleration structure viewer to see the hierarchy of the acceleration structures and the geometry contained within them. Also, while not directly related to this project, investigating texture data and buffer data provides good reassurance to the developer that the data is being passed correctly to the shaders.

Moreover, the shader code can be debugged by setting breakpoints in the shader code or in the case of the ray tracing kernel, by selecting a specific pixel on the output texture, i can use these as ID to pinpoint the thread that is responsible for calculating its pixel values allowing me to see the results of the calculations and the values of the variables for the executed code. In this case, i can see how many bounces were perfomed for a ray before it returned a color and all the intermediate calculations. This is an absolutely great feature that can potentially save a lot of time for a new developer without having to spend hours trying to debug the shader setting it up to return desired values using color buffers.

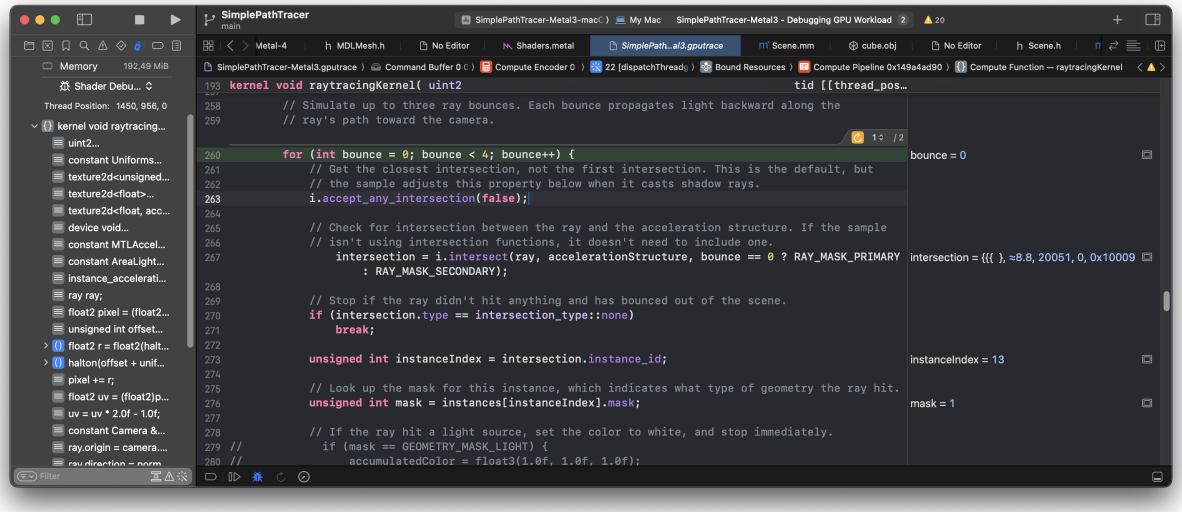

Figure 5: Shader debugging tool in Xcode

3.0.3 Profiling

Looking for sources online, unfortunately i am unable to find any way to bypass hardware acceleration to compare the performance of the ray tracing application with and without

hardware acceleration, because it is relying on usage of acceleration structures which are also used without hardware acceleration. However, i looked around the profiled stats and was able to verify that GPU RT units are being engaged while the application is running.

Furthermore, while not relevant for the purposes of understanding simple ray tracing/rasterization concepts, there is an option to see percentage wise how long certain lines of code take to execute which can be extremely useful when developing more complex shaders to see potential bottlenecks or unnecessary calculations that can perhaps be done on the CPU side and etc.

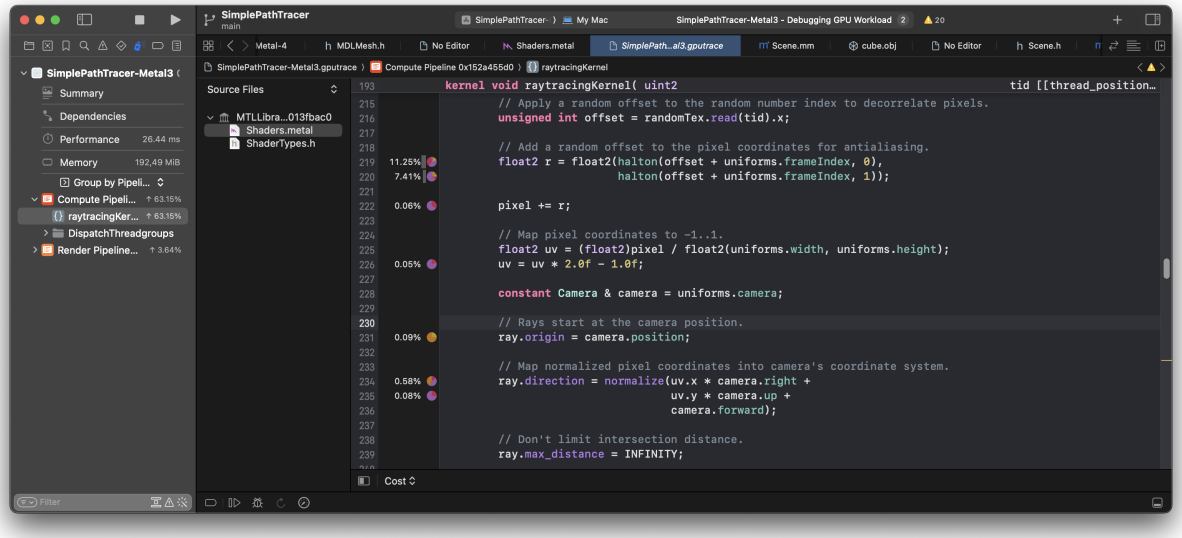

Figure 6: Profiler in Xcode

4 Results

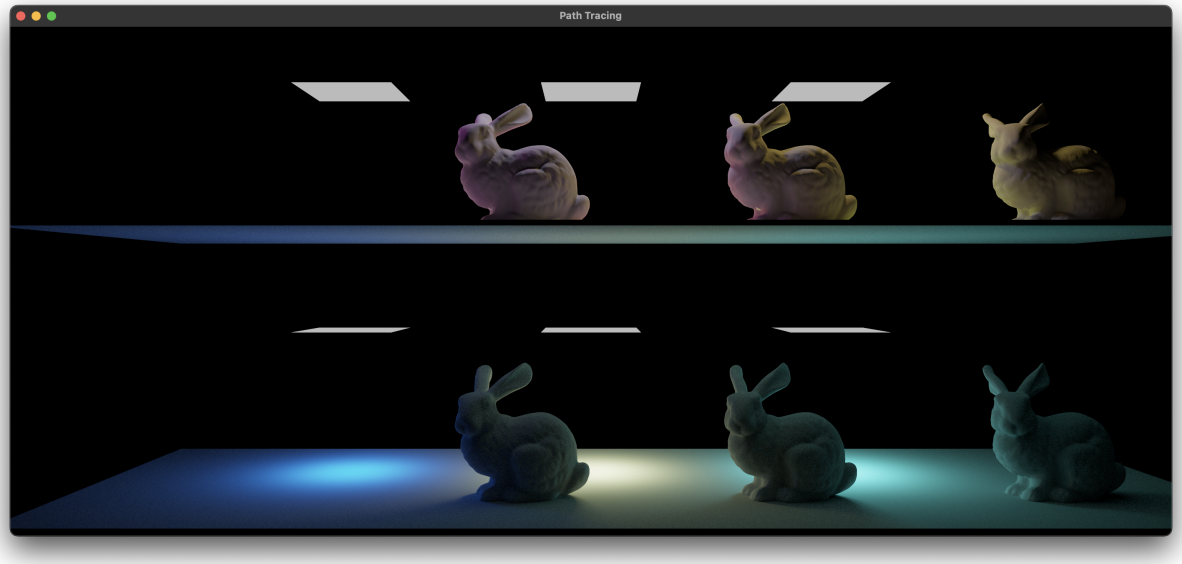

Figure 7: Rendered scene with bunny.obj

This section will describe the results of the project. It will describe the results of the implementation and the goals of the project.

Overall, the goals of the project to apply ray tracing knowledge to a different API (in this case Metal) was achieved! Conceptually the ray tracing pipeline is very similar to what we've done in WebGPU, except for setting up/using acceleration structures instead of BSP trees and bounding box/triangle intersection functions being pre-defined in this case. Furthermore, there is a difference that the calculations for pixel colors are done in a kernel function instead of a fragment shader. Moreover, I was able to import the same object as in the WebGPU worksheet and store it in the acceleration structure to optimize the rendering of the scene. Lastly, the results of making the acceleration structure from the imported OBJ file were verified by using the acceleration structure viewer and shader debugging tools were used to see the results of the calculations and the values of the variables for the executed code.

5 Discussion

This section will discuss the challenges faced and the lessons learned.

While it is certainly beneficial to have tools to debug the shaders, it is a challenge to find these tools and understand how to use them. Comparatively speaking, the available sources online for Metal API are slim to those available for OpenGL. This makes it significantly more time consuming to understand the code. As mentioned before, plethora of examples

found online or in the official documentation are in Swift, while others are in Objective-C, and while both can be used to achieve the same results, the unfamiliarty of syntax of both languages, it means that it is significantly harder to follow guides and examples.

Disregarding the entire setup required to render a scene, i think Metal and Xcode can be a a great tool for shader development, optimization and debugging. having an option to quickly investigate calculations of individual pixel values and seeing the results of intermediate calculations is a great feature allowing for debugging without having to set up a specific sequence of events to see results of intermediate calculations.

Unfortunately as an IDE Xcode is limited in its platform availability but is also not so great to work with questionable project settings and general stability issues as an application.

References

[1] "Accelerating Ray Tracing Using Metal." https://developer.apple.com/documentation/metal/metal_sample_code_library/accelerating_ray_tracing_using_metal?language=objc.

[2] "Rendering Reflections in Real Time Using Ray Tracing." https://developer.apple.com/documentation/metal/metal_sample_code_library/rendering_reflections_in_real_time_using_ray_tracing?language=objc.

[3] "Rendering a Curve Primitive in a Ray Tracing Scene." https://developer.apple.com/documentation/metal/metal_sample_code_library/rendering_a_curve_primitive_in_a_ray_tracing_scene?language=objc.

[4] "Accelerating Ray Tracing Using Metal at WWDC 2023." https://developer.apple.com/videos/play/wwdc2023/10128/.

6 Appendix A

6.1 Importing .obj files

```
-(void)addObjFile:(NSString *)filename transform:(matrix_float4x4)transform {
NSError *error = nil;
NSString *objFile = [[NSBundle mainBundle] pathForResource:filename ofType:@"obj"];
if (!objFile) {
    NSLog(@"Error: File not found: %@", filename);
    return;
}
NSString *contents = [NSString stringWithContentsOfFile:objFile encoding:NSUTF8StringEncoding error:&
    error];
if (error) {
    NSLog(@"Error reading file: %@", error);
    return;
}
NSArray *lines = [contents componentsSeparatedByString:@"\n"];
const size_t firstIndex = _vertices.size();
std::vector<uint16_t> temp_indices;
for (NSString *line in lines) {
    if ([line hasPrefix:@"v "]) { // Vertex positions
        NSArray *components = [line componentsSeparatedByString:@" "];
        float x = [components[1] floatValue];
        float y = [components[2] floatValue];
        float z = [components[3] floatValue];
        float4 vertex = transform * vector4(x, y, z, 1.0f);
        _vertices.push_back(vertex.xyz);
        _colors.push_back(vector3(1.0f, 1.0f, 1.0f)); // Default white color
    }
    else if ([line hasPrefix:@"vn "]) { // Normals
        NSArray *components = [line componentsSeparatedByString:@" "];
        float x = [components[1] floatValue];
        float y = [components[2] floatValue];
        float z = [components[3] floatValue];
        _normals.push_back(normalize(vector3(x, y, z)));
    }
    else if ([line hasPrefix:@"f "]) { // Faces
        NSArray *components = [line componentsSeparatedByString:@" "];
        size_t vertexCount = components.count - 1; // Exclude the "f"
        std::vector<uint16_t> faceIndices;
        std::vector<uint16_t> normalIndices;
        // Parse all vertex data for the face
        for (int i = 1; i <= vertexCount; i++) {
            NSArray *indices = [components[i] componentsSeparatedByString:@"/"];

            // Parse vertex position index
            uint16_t vIndex = [indices[0] intValue] - 1;
            faceIndices.push_back(vIndex);

            // Parse normal index (if it exists)
            uint16_t nIndex = (indices.count > 2 && ![indices[2] isEqualToString:@""]) ? [indices[2]
                intValue] - 1 : 0;
            normalIndices.push_back(nIndex);
        }
        // Triangulate the face if it has more than 3 vertices
        for (int i = 1; i < vertexCount - 1; i++) {
            _indices.push_back(faceIndices[0]);
            _indices.push_back(faceIndices[i]);
            _indices.push_back(faceIndices[i + 1]);
            // Create triangle
            Triangle triangle;
            triangle.normals[0] = _normals[normalIndices[0]];
            triangle.normals[1] = _normals[normalIndices[i]];
            triangle.normals[2] = _normals[normalIndices[i + 1]];
            triangle.colors[0] = _colors[faceIndices[0]];
            triangle.colors[1] = _colors[faceIndices[i]];
            triangle.colors[2] = _colors[faceIndices[i + 1]];
            _triangles.push_back(triangle);
        }
    }
}
}
```

Code for main simplified rendering pipeline - *Renderer.mm*
Ray tracing kernel and other shaders - *Shaders.metal*