AN USER PROGRAMMABLE SYSTEM AGNOSTIC REAL-TIME RAY TRACING FRAMEWORK

CHO CHARK JOE LEUNG

A thesis

submitted in partial fulfillment of the

requirements for the degree of

Master of Science

University of Washington

2021

Reading Committee:

Kelvin Sung, Chair

Michael Stiber

Yusuf Pisan

Program Authorized to Offer Degree:

Computing & Software Systems

Abstract

Ray tracing rendering is a well-understood rendering technique that can produce photorealistic images or complex visual effects. Until recently, customizing a ray tracing pipeline posed challenges because of the lack of infrastructure to program such a pipeline. Industries have been proposing their own ray tracing frameworks to address the customizability issue by supporting user programs integration. However, most of the existing solutions target proprietary platforms, which limits the access to ray tracing for the general public. In this project, we propose a platform-agnostic ray tracing framework that is based on the general programming capacity found in commodity graphical processing units (GPUs). The framework supports six types of customizable user programs, including custom geometries, shading, and lighting. The framework also provides the developers with utilities such as built-in acceleration structure, secondary ray generation, and material instancing. Our framework has been demonstrated that it is able to fulfill the common functional requirements found in existing commercial products without the platform limitations. We also identified tradeoffs in developing a flexible, cross-platform ray tracing framework running on GPUs. Our results provide insights into the future development of a similar ray tracing framework.

Table of contents

Table of figures

1 Introduction

Ray tracing is a graphical rendering technique that supports accurate and flexible lighting modeling. Ray tracing renderer computes the color of each pixel on the screen by first emitting virtual rays, known as primary rays or view rays, from a viewing camera. These rays may intersect with multiple primitives in a virtual scene. The location where the ray hits the object will be used to compute the illumination at that location [1]. In some cases, such as on a reflective surface or in a semi-transparent volume, one or many secondary rays will be spawned from the hit position to repeat the ray tracing process. This is an analog to reflection and refraction behavior of physical light rays. This analog allows graphical programmers to create physically accurate or interesting optical phenomena. Examples of complex rendering by ray tracing are shown in Figure 1 [2][3].

Figure 1. Examples of visual effects rendered by real-time ray tracing system. Top: Multiple reflections can be achieved by ray tracing. The reflection of the amber light on the soldier's helmet can be seen on the chest-plate of the middle character [2]; Middle: Demonstration of volumetric rendering and soft shadowing by ray tracing. The light shaft is visible inside the room. The shadow of the pillars on the wall has a soft edge [3]; Bottom: Ray tracing can be used to simulate the effect of translucent materials to the light color. The color patterns of the stained glass windows on the right are projected onto the left wall [3].

Ray tracing rendering has the potential to augment or replace the current mainstream real-time rendering methods. Ray tracing rendering has already been used in many offline rendering scenarios including computer generated movies such as "Cars" by Disney & Pixar [4], "Rogue One: A Star Wars Story" by Lucasfilm [5]. The comparably higher computational cost and the lack of customization support are two of many inhibiting factors for ray tracing rendering to replace the current mainstream real-time rendering systems.

The computational cost of ray tracing algorithms arises from the requirements for solving a large number of ray-primitive intersections, a number that is proportional to the number of geometries and the resolution of the image being computed. During the ray tracing process, in addition to every pixel, rays may be spawned at every ray-primitive intersection position, such as secondary rays for reflection determination and shadow rays for illumination calculation. Intersection tests

against all primitives must be computed for each of these rays. Thus, the overall computational complexity of ray tracing grows by the products of the number of rays and number of objects in the scene.

The second limiting factor of ray tracing rendering is the lack of customization support or development framework. In the current mainstream rendering systems, in order to achieve various visual effects, specialized programs known as shaders are created to instruct the renderer in generating the final images. For example, a shader that is capable of simulating the wrinkling of skins. Until recently [6], ray tracing rendering systems did not support programmers in defining similar customized shaders. For example, a programmer did not have the option to spawn a reflection ray in a customized direction, e.g., in the direction of the incoming ray. Supporting programmable modules increases the overall complexity of the pipeline, resulting in higher performance demands. For this reason, until recently, ray tracing rendering systems did not support programmable modules and thus limited the use of ray tracing rendering systems.

With the increasing computational capacity found in modern devices, one may expect the performance hurdle of ray tracing could finally be overcome. GPUs were introduced to the mass market as a hardware add-on in the late 90s to accelerate 3D graphics rendering. They were packed with fixed-function units such as triangles clipping, lighting, and rasterization units. Over the past thirty years, advancement in semiconductor fabrication has allowed more transistors to be packaged on a chip, resulting in increasing computational power of GPUs. For example, NVIDIA's 8800 GTX released in November 2006 has 681M transistors, 128 shader cores on 90nm fabrication, capable of 345 GFLOPS, Single Precision. Today, NVIDIA's RTX3090 has 28.3B transistors, 328 shader cores on an 8nm fabricated chip, delivering 29284 GFLOPS, Single Precision [7]. GPUs have become more flexible in the instructions they could execute, such as lifted limitations on branching and loops, untyped variables, and access to graphic memory [8]. The advantage of running highly parallel data processing applications on GPUs has encouraged manufacturers to invest in general programming support on their products. In 2006, NVIDIA introduced CUDA [9], a general programming framework for their GPUs, followed by Microsoft's DirectCompute [10] and Khronos Group's OpenCL [11].

A logical next step to promote the use of ray tracing rendering would be taking advantage of the parallel GPU processing potentials and creating a programmable ray tracing framework. Such a framework can reduce the time, effort, and development cost of a ray tracing application by allowing developers to reuse shared functionality in ray tracing rendering. Fundamentals support for ray tracing algorithms, such as acceleration structure building and traversal, and communication between different ray tracing modules, can be designed as the core of the framework. Developers using the framework can focus on writing custom visual effects. This is analogous to the success of promoting the rasterization pipeline in the early 90s by creating programmable rasterization pipelines such as OpenGL [12] and DirectX [13].

Recently, many manufacturers have proposed their solutions of programmable ray tracing frameworks. This is an indication that the industries are showing interest and potential market value in supporting ray tracing rendering. Industry leaders, such as NVIDIA [6], Intel [14], Microsoft [15], Apple [16], have developed their custom ray tracing frameworks based on their respective products. However, the existing solutions are typically designed for specific hardware or firmware platforms, which limit access and code reuse.

A hardware- and platform-agnostic ray tracing framework can be a low-cost, less-risk option to the existing ray tracing frameworks. One possible solution is using general programming capacity on GPUs. User programs and ray tracing framework can be implemented as compute shaders. Compute shaders abstract the hardware details from the application developers. This facilitates portability of the ray tracing framework and the custom user programs developed for the framework.

In this project, we will design and implement a ray tracing framework based on general programming models presented by modern commodity GPUs. The framework will integrate customizable user programs to the ray tracing pipeline, including ray generation, ray-object intersections, and illumination, while providing fundamental supporting operations such as acceleration structure building and input output assemblers.

A prototype of our framework has been implemented on Unity's scriptable render pipeline [17] and the Microsoft DirectCompute 11 compute shader system [10] in the 2019 Unity 3D engine [18]. Building on an existing 3D engine save us the development on user interface, shader

compiling infrastructure, and 3D scene editing. The types of supported user-defined ray tracing programs are comparable to existing commercial products. Our framework supports custom shaders including: ray generation, intersection, closest-hit, secondary ray generation, ray missed, and custom lighting. Our system supports real-time reflections and volumetric rendering effects with a maximum of 4 secondary rays per generation of the maximum of 4 generations. Our framework also improves the performance of the ray tracing rendering by providing a built-in scenebounding volume hierarchy acceleration structure (BVH) and object-level BVH support through the intersection program.

Our proposed ray tracing framework is portable across different platforms and GPU architectures. It does not require proprietary technology from the processors. Therefore, applications can target multiple platforms from a single codebase and the users does not need to purchase additional hardware.

2 Related Work

Industry have been developing various programmable ray tracing frameworks to address the lack of customizability limitation. Processors manufacturers design their ray tracing framework around their products to take advantage of special computation units or instruction sets found on their chips. For the operation system and platform providers, they integrate ray tracing framework to their existing graphic APIs found in their software products. In this chapter, we will discuss a selection of commercial ray tracing frameworks.

2.1 **NVIDIA OptiX Ray Tracing Engine**

NVIDIA OptiX Ray Tracing Engine [6] is an application framework based on NVIDIA's GPU general programming capacity. It utilizes NVIDIA's CUDA framework as the application runtime.

The customizability of OptiX engine comes from the user programmable modules, known as programs. Users supply these programs to the framework to specify the behavior of a part of the ray tracing pipeline. For example, a user can program a specific program called closest-hit program and assign it on a geometry in the scene. The closest-hit program assigned controls the color calculation (shading) on that geometry shall there be a view ray hitting the geometry. Another geometry can assign a closest-hit program of different shading model.

The OptiX architecture supports 8 different kinds of user programs. We have selected a subset of supported user program types from OptiX in our framework design. We will introduce each selected program in the next chapter.

The OptiX is designed around Mega kernel execution model. User programs are compiled into a single assembly. The assembly includes a software scheduler that controls which programs should be execute on the GPU cores. This model achieves better rendering performance by overriding the hardware default tasks scheduling to reduce the occurrence of long running operations holding up the GPU cores. However, implementation of a software scheduler requires access to the device's

low-level APIs. This coupled the ray tracing framework with the underlying runtime or GPUs. Thus, reducing the accessibility and portability of the application framework.

2.2 Microsoft DirectX Raytracing

Microsoft introduces DirectX Raytracing (DXR) [15], a ray tracing framework in DirectX 12. The design goal of this ray tracing framework is to have a single programming model across different graphical hardware. DXR is supported by some of the latest GPUs from NVIDIA and AMD. In DXR, users write their programs in High Level Shader Language (HLSL). The framework also exposes some fixed functions to assist and optimize common tasks in ray tracing.

DXR shares a similar set of user programmable modules and execution model with NVIDIA's OptiX. However, they are differing in the organization of user programs. In DXR, user programs are registered into 4 shader tables: ray-generation, miss, hit group, and callable. A hit group consists of an intersection program, any hit program, and closest hit program. This means a primitive assigned a hit group will bundle these three programs, as opposed to OptiX where these three programs are independently assigned.

The design of hit group in DXR is reasonable because the programs in a hit group are describing the appearance of a geometry. Having these programs coupled simplify the design of user program lookup and reduce the overhead. One may argue that having the intersection program also inside a hit group will causes multiple records registered on the shader tables for different geometries of the same shading. Yet, consider most geometries are represented by triangle meshes, the duplication of records may not be significant in most cases.

2.3 Intel Embree

Embree [14] is a CPU based ray tracing framework. Unlike other frameworks described in this report that runs on GPUs, Embree is designed to take advantage of the vectorization and instruction sets optimized for parallel computing on x86 CPUs. The framework has later expanded to include other CPU architectures such as Apple M1. Intel has reported the performance of Embree,

measured by BVH build time and number of ray traced per frame, is on par with or exceeding GPU based solution.

Embree consists of a library of ray tracing kernels written in C. Given that the running device is the CPUs, it has fewer programming restriction compared to GPUs. For example, recursive programs are allowed on CPU runtime. Recursion can structured ray tracing algorithm better than iterative approach. In ray tracing, the multiple rays spawned by a ray tracer function can use the same ray tracer function to resolve. Since most GPUs does not support recursion and dynamic allocation of stack, GPU based framework will have a more complicated solution to support multiple ray generations compared to CPU based framework like Embree.

Running rendering on CPUs does comes with a cost. Because rendering happens every frame, the CPUs are expected to be occupied. Other processes may compete for the processing units and causes frame drops. For example, in computer games, the CPU is also tasked to perform game logic updates and physics calculation. These computations are also happening in every frame. A powerful CPU capable to handle such workload maybe more expensive than a system that can have a less capable CPU with workload shared by the GPUs.

2.4 Apple Metal Performance Shaders

Apple proposed a ray tracing library as part of their Metal Performance Shaders (MPS) API [16] on their Metal API targeting Mac OS X machines. MPS supports fewer user programmable programs compared to OptiX and DXR. For example, the intersection program is not open to program by users because MPS assume all geometries are in the form of triangle meshes. This is a tradeoff of MPS to optimize the acceleration structure building and traversing by the framework.

MPS has 2 types of acceleration structures unlike other frameworks introduced in this report. The first type is designed for capturing rigid body movement, an Affine translation on a primitive but the geometry mesh of the primitive remains the same. MPS uses a two-level BVH acceleration structure like other frameworks for this scenario. The second type is designed for capturing vertex deformation. MPS performs a special operation called Refitting where the position and size of the

bounding box corresponding to the deformed vertex is updated but the two-level BVH hierarchy is preserved. This bypass rebuilding the entire BVH when a vertex is moved, hence speeding up the acceleration structure updates.

2.5 Unity Ray Tracing Framework

Unity 3D game engine has an editor integrated toolset to implement ray tracing rendering on Unity projects [19]. The framework is based on DirectX 12 and NVIDIA RTX technologies. There are 4 different types of user program supported by Unity: ray generation program, miss program, closest-hit program, and any hit program. Users integrate their custom program on a special file called Ray Tracing Shader (.raytrace). The type of the program is annotated by tags. Metadata such as shader variable binding are done through ShaderLab language as in traditional surface shaders in Unity.

Unity presented a high usability solution for ray tracing application development. Developers has a graphical user interface to create scenes and visualize the results in real-time. Assets management, shader compilation, and application deployment are also handled by the engine. However, Unity solution is less flexible in terms of supported user programs. For example, only triangle meshes is supported and users has no control on the choice of acceleration structure. At the moment of this writing, Unity framework is tie with NVIDIA RTX capable GPUs. Considering RTX is a proprietary technology, this limits the accessibility of ray tracing application developed on Unity.

2.6 Summary

We have covered four programmable ray tracing frameworks available in the market. They support a similar set of user programs and use of acceleration structure, primarily bounding box volume (BVH), to improve the rendering performance. These vendors proposed frameworks are designed to run on the vendor's specific, usually high-end, commercial products. Such requirements segmentate the market where consumers are forced to select a particular system. This would result in limiting access to ray tracing rendering for the public. A platform agnostic ray tracing framework that based on general programming capacity found in consumer-grade GPUs could be a more economic approach to promote the adoption of ray tracing rendering.

3 Design, and Architecture

In the previous chapter, we have identified a set of functional requirements from four existing ray tracing frameworks in the market. In this chapter, we will design a framework that fulfill the functional requirements of customizable real-time ray tracing framework and capable to run on commodity GPUs that supports general programming.

3.1 Functional Requirements

A customizable, real-time ray tracing framework should fulfill the following requirements:

- Facilitate data feeding to the ray tracing renderer (ray tracer) running on the GPUs.
- Support user programs, including ray-generation, ray intersection, secondary rays, closest-hit, miss, and lighting.
- Build scene-level acceleration structure.

In the following sections, we will discuss the features of our framework design that meet each of the requirements.

3.2 Data preparation for the ray tracer

3.2.1 Overview

The core component in our proposed framework is the ray tracer. It is responsible for executing ray tracing algorithms on the GPU's general programming unit to render images. The ray tracer requires two kinds of data for the rendering tasks: data describing a scene and a collection of user programs to be invoked. Scene data preparation is handled by the CPU rendering pipeline while the user programs is managed by the shader manager component.

Figure 2 shows the relationships between these three components of our system. We are going to describe each component in the following paragraphs.

(A) CPU Render Pipeline: This component is responsible for parsing the current scene assets, including geometries, materials, and lighting. Scene-level acceleration structure is constructed from the parsing result. These data will then be serialized and loaded into the graphic memory for the ray tracer. This component is invoked every frame to capture the changes in the scene.

(B) Compute buffers: Compute buffers holds serialized data to be passed to graphic memory. For example, an array of floating-point numbers or an array of 2D textures.

(C) Shader manager: This component is responsible for tracking all user program files and compile the programs and framework's ray tracer into assembly to be executed on the GPU. When a user submits, updates, or removes their programs in our ray tracing framework, this component will update the internal program collections by the types of the program (e.g. ray intersection program or closest hit program). Then the component will invoke the shader compiler to compile both the updated shader collections and the framework's ray tracer program into an executable.

(D) Shader executable: The output from the shader compiler that combined the user programs and the ray tracer program into a single assembly to be executed on the GPU.

(E) Ray Tracer: This component is responsible for carrying out the ray tracing algorithm on the target GPUs. It binds the shader executable with the compute buffers, execute the ray tracing algorithm, and returns the color of each screen pixel as its output.

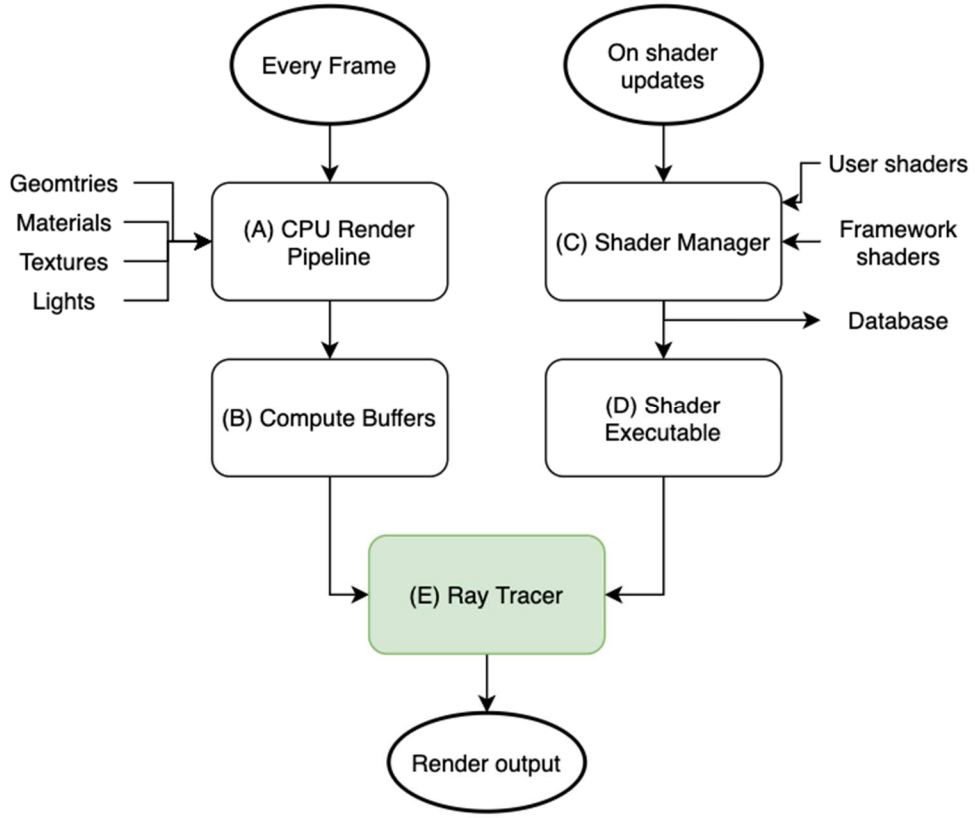


Figure 2. An overview of the ray tracing framework system architecture. Green box indicates the component is running on GPUs.

We have introduced the CPU render pipeline, shader manager, and ray tracer that form our ray tracing framework. In the remaining section, we will describe the subcomponents of the first two components. The detail of our ray tracer design will be covered in the next section.

3.2.2 Shader Manager

Shader Manager is responsible for tracking the files of all users developed programs and combining the user programs with the framework shaders into an executable targeting the host GPUs. There are 5 subcomponents in Shader Manager. Their call graph is illustrated in Figure 3.

(A) **Preprocessor**: This component reads the annotation of each user program to determine the program category. For example, a custom intersection program will have a tag [intersection("CustomSphere")].

(B) **Assets Registration**: This component performs create, read, update, and delete operations of program in each category. Each program is assigned a unique identifier to allow referencing a program on a primitive, material, or light. The information is written to a User Program Assets Database.

(C) **Assets Database**: This component stores the shader registry from Shader Assets Registration component in a persistence storage.

(D) **User Program Collection Updater**: This component integrates or removes a user program to/from the codebase that is going to be compiled. Based on the GPU execution model selection, the integration would be implemented differently. For example, a run-to-complete execution model will put all user programs inside a switch statement to perform operation switching.

(E) **Shader Compiler**: This component compiles the user programs and framework into assembly targeting the host GPU.

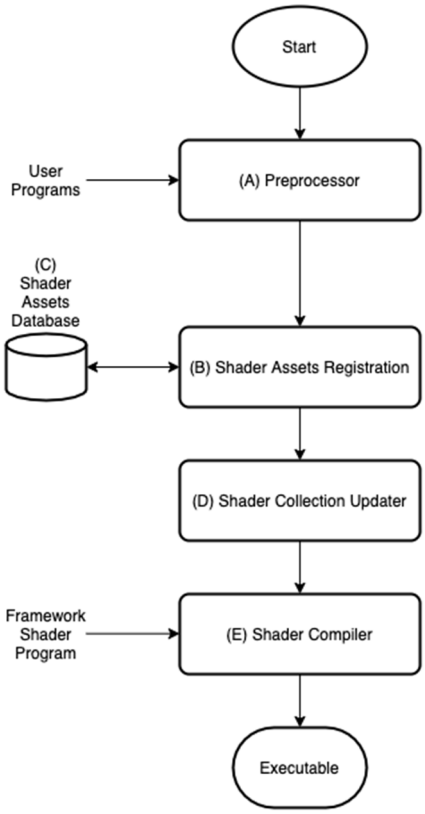


Figure 3. Shader manager call graph.

The shader manager supplied our ray tracer with complied user programs. This is the essential step to bring customizability to our ray tracing framework. In the following paragraph, we will introduce the CPU render pipeline which is responsible for parsing the scene information to feed our ray tracer.

3.2.3 CPU Render Pipeline

CPU Render Pipeline is responsible for scene information collection and preparation before the scene can be passed to the ray tracers running on the GPUs. It also builds the scene-level acceleration structure. The flowchart of the pipeline can be seen in Figure 4.

(A) **Extract geometry information**: The scene parser will query the geometry and material information from all primitives presented in the scene. This component provides such information by parsing external components, such as meshes, materials, or other custom scripts. Users can also extend this component to configure how geometry and material information is parsed.

(B) **Scene Parser**: The scene parser queries all information from each primitive in the scene and construct a representation of the current scene state. This allows the scene parser to determine if any acceleration structures need to be rebuilt.

(C) **Acceleration Structure Builder**: This component constructs the scene space (top-level) acceleration structure for the entire scene. The type of acceleration structure depends on the design choice. A matching traversal program on the GPU pipeline is required to instruct the ray tracer on how to traverse a given acceleration structure.

(D) **Acceleration Structure Cache**: If any primitives in the scene did not perform an affine transformation in the current frame, we can reuse the generated acceleration structure from the previous frame. Note that camera movement does not require rebuilding acceleration structure.

(E) **Data Serialization**: Before the geometries, textures, and materials can be passed to the graphic memory buffers, they are required to be serialized and flatten into a binary array. This component performs such operations.

(F) **Resources Binding**: The binary arrays are loaded to the GPU memory by this component.

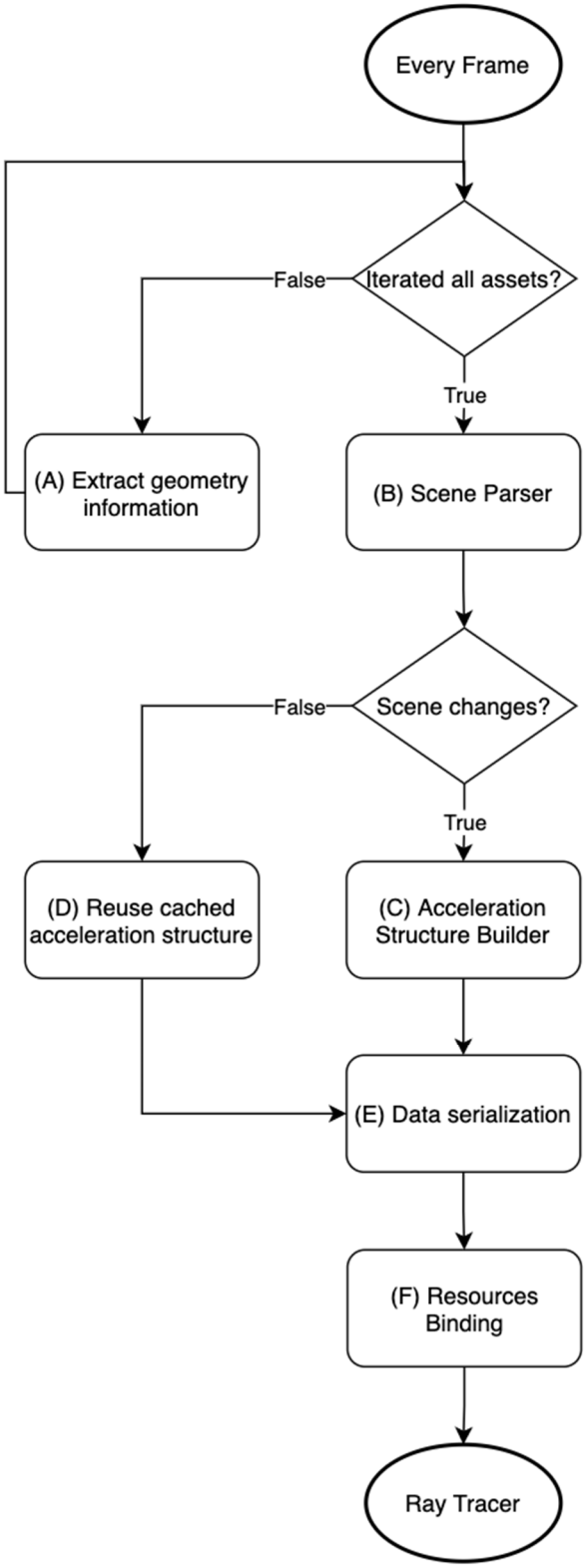


Figure 4 The call graph of the CPU render pipeline.

3.2.4 Summary

In this section, we have covered the CPU Render Pipeline and Shader Manager components. These two components are executed on the CPU to supply the scene information and the user programs to the ray tracer running on the GPU. In the next section, we will describe our ray tracer design and how does it fulfill the functional requirement of customizability in a ray tracing framework.

3.3 Ray Tracer Design

Ray tracer is responsible for executing the ray tracing algorithm and invoking each custom programs at designated execution points. Multiple ray tracers run for every screen pixel per frame on the GPU. It outputs the color of the corresponding screen pixel.

There are six types of user programs open to application developers for customization in our ray tracer design. In Figure 5, user programs are represented in blue boxes. Summaries of each type of user programs are as follows:

(A) Ray Generation Program: Ray generation program is responsible for generation of primary (view) rays. Developers can control the number of rays spawned per pixel. This allows developers to control the image quality. At the same time, developers can also modify the ray's direction to implement various camera projection models, such as perspective and orthographic. The Ray generation program is invoked for every screen pixel.

(C) Ray Intersection Program: This program is responsible for calculating the intersection between a ray and a geometry. If there is an intersection, the intersection point's position, its normal vector, and its material will be returned. If there is no intersection, no intersection position will be returned to the ray tracer to signal a miss scenario, which is handled by the miss program. Custom object-level (per object) acceleration structure is also implemented in this program. A ray can traverse an object-level acceleration structure inside this program to deduce if there is a hit. This program is invoked for every intersection along the ray traversal.

(D) Secondary Ray Generation Program: This program is responsible for spawning secondary rays at the point of intersection. For example, when a ray hits a translucent reflective surface, a reflection ray and a translucent ray are spawned from the intersection point towards the reflective

and refractive direction, respectively. Based on the hardware limitations, users can implement a hard limit of maximum number of generations a primary ray can spawn, and a hard limit of maximum number of ray an intersection can spawn. This program is invoked for every intersection.

(F) Closest Hit Program: This program is responsible for computing the color of the intersection point based on the lighting model and material supplied by the users. The intersection point color will then be assigned to the inclining rays for back-tracking. To implement different lighting models, application developers can invoke different lighting programs inside this program to collect the colors coming from each light source in the scene. This program is invoked for every intersection.

(G) Light Program: This program is responsible for generating light ray(s) from each light source. Different lighting models can be implemented in this program, such as directional light, point light, and spot light. This program is invoked by the closest hit programs.

(H) Miss Program: This program is responsible for assigning a ray color shall a ray not hitting any objects along its traversal. For example, a ray shooting towards the sky or background. This program is invoked for every ray that does not hit any object.

In addition to user programs, there are 2 framework owned programs to assist the ray tracer on scene level acceleration structure traversal and back-tracking the ray traced result from secondary rays. In Figure 5, they are represented by box B and Box E.

(B) Scene Level Traversal Program: This framework owned program is responsible for traversing the scene level BVH acceleration structure for each ray. When a ray enters the scene, this program will return a list of potential bounding boxes that would intercept with the ray. Together with the ray-intersection program discussed later, the closest intersection, if any, of a ray to an object on the scene can be identified. This program is implemented by the framework and close to modification. This program is invoked for every primary and secondary ray.

(E) Update Back-Tracking Data Program: Upon a list of secondary rays are generated at the intersection point, the intersecting ray data is recorded for later computation. Once all the

secondary rays return with their ray color to the closest hit program of that intersection point, the recorded ray will be retrieved and carries the resulting color from the closest hit program back to its original, either another intersection point or back to the screen pixel. This program is invoked when an intersection point spawns secondary rays.

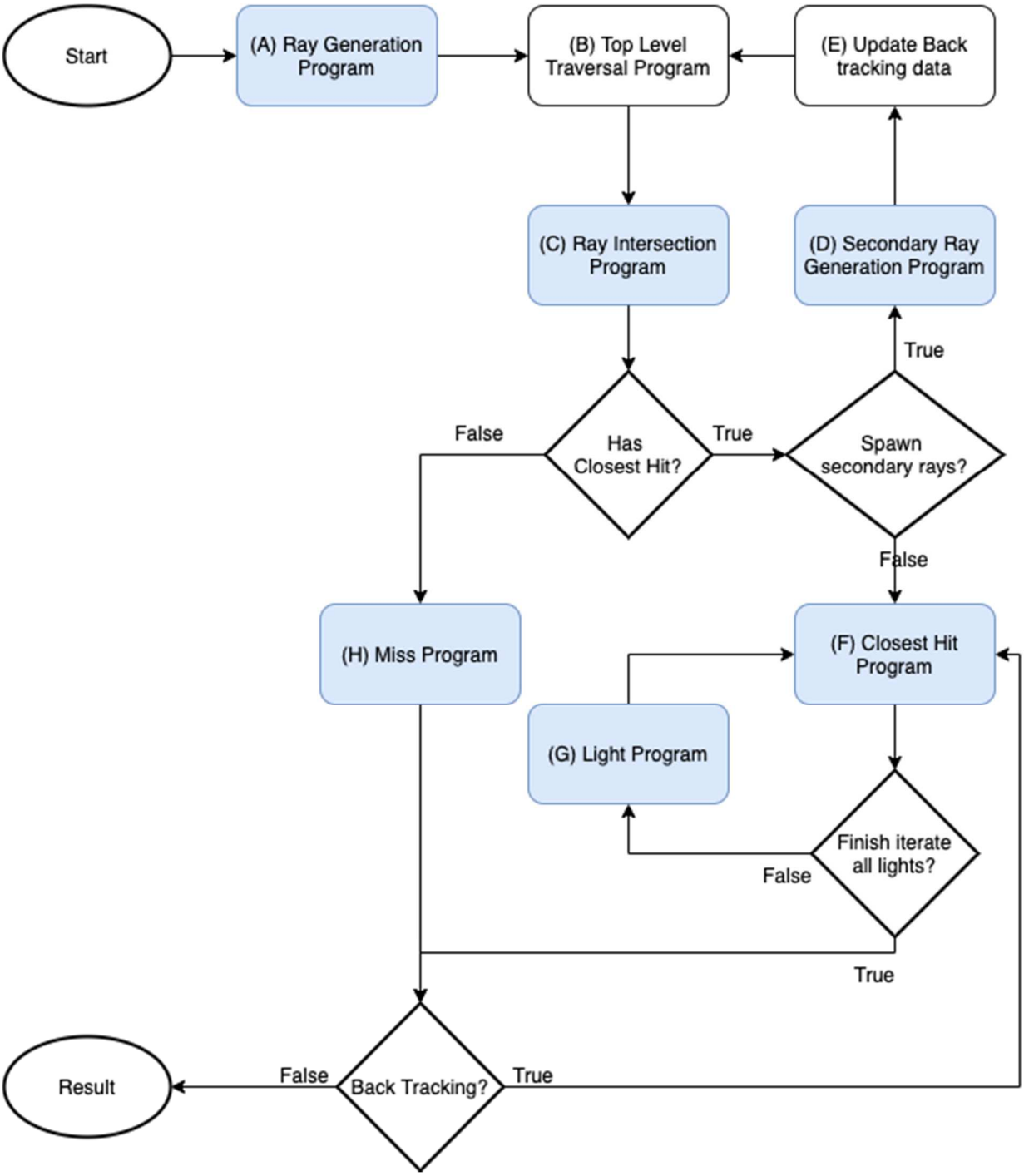


Figure 5. The architecture of the ray tracer. Blue rectangle boxes represent user programs (Box A, C, D, F, G, H). White rectangle boxes (Box B and E) represent framework owned programs to assist ray tracer.

We have covered our ray tracer design and how each custom program linked by the ray tracer to form a customizable ray tracing framework. Our ray tracer and the user programs must be compiled into compute shader assembly by a shader compiler targeting the GPU that the application will run

on. The GPU can treat the assembly as a general compute shader and does not require a specific hardware feature to operate. In the next chapter, we describe our implementation of our design on Unity 3D engine to demonstrate the customizability, portability, and performance of our proposed framework.

4 Implementation

In this chapter, we will discuss our implementation of a prototype of our design described in previous chapter based on Unity 3D engine's Scriptable Render Pipeline (SRP). We will share the rationale of selecting Unity as the demonstration platform, our effort in integrating our design into Unity 3D editor, and details of the communication between the components of our framework.

4.1 Selection of Development Environment

Develop our prototype on an existing 3D engine could save us the effort to develop supporting features for enhancing the usability of our framework. A good candidate of an 3D engine for our use case should include the following features:

- Cross platform compute shader compiler.
- Assets management infrastructure such as 3D model and texture importing.
- Graphical user interface for scene editing.

We have selected Unity 3D engine 2019's Scriptable Render Pipeline [17] as our development environment. Unity supports general programming on GPU. Its scriptable render pipeline feature allows us to replace the default Unity rasterization render pipeline with our ray tracing framework while maintaining the access to utility functions such as scene management, compute buffer loading, and compute shader dispatch. Another significant benefit provided by the Unity 3D engine is its compute shader compiler. The compiler can generate assemblies for various graphical API system such as DirectX for Windows machine and Metal for Mac OS X machine. It is worth noting that we did not utilize any Unity's own ray tracing framework solution as their solution requires special GPUs equipped with NVIDIA RTX technology. Our shader programs are straightforward compute shader features in Unity.

4.2 Integrating our framework with Unity Editor

4.2.1 Scriptable Render Pipeline Setup

Scriptable Render Pipeline (SRP) is a feature of Unity Editor that allows developers replace the editor's default rendering pipeline with their own. All cameras, including the editor windows, will be using the replaced render pipeline to generate images to be displayed. Figure 6 shows a typical setup of the Unity Editor with our ray tracing framework implemented on an SRP. The images on both windows are rendered by our ray tracing framework in real-time. The viewing windows on the left is a live editing window that allow developers to inspect and modify the scene. The window on the right is the render image from the main camera where the user of the application will see. Running our framework on SRP allows us to achieve "what you see is what you get" user interface. Developers no longer need to recompile the application to see the result. This greatly improves the productivity of ray tracing application development.

Figure 6. A typical Unity Editor user interface with our ray tracing framework setup on scriptable render pipeline.

There are three steps to setup the implementation of our ray tracing framework on the SRP. Firstly, the CPU render pipeline is programed as an integral part of an SRP. When the SRP is called by Unity engine when there is a need to render, our CPU render pipeline is invoked to perform operations outlined in the Design chapter. The outputs of the CPU render pipeline are stored in multiple compute buffers to be passed to the ray tracer in the GPU. In the second step, we instruct the SRP to bind the compute buffers containing the CPU render pipeline outputs to the

corresponding global variables on our ray tracer. Then we dispatch our ray tracer compute shader executable for every screen pixel through SRP. The third step is to retrieve the rendered image from the ray tracer's output texture field and submit the image to Unity engine for display.

4.2.2 Shader Manager

The shader manager of our prototype is implemented as Unity's assets database callbacks. Our shader manager is triggered whenever the developers add, modify, or remove their custom programs in the Unity Editor. This behavior is achieved by registering the shader manager to monitor the file change events provided by Unity Editor's assets database. When such event is triggered, the shader manager will check if the file change is relevant. The shader database and the shader collection will then update accordingly. The shader database is a set of JSON files that store the user program information for tracking and shader collection generation. Details of the shader collection update is covered in Section 4.3.

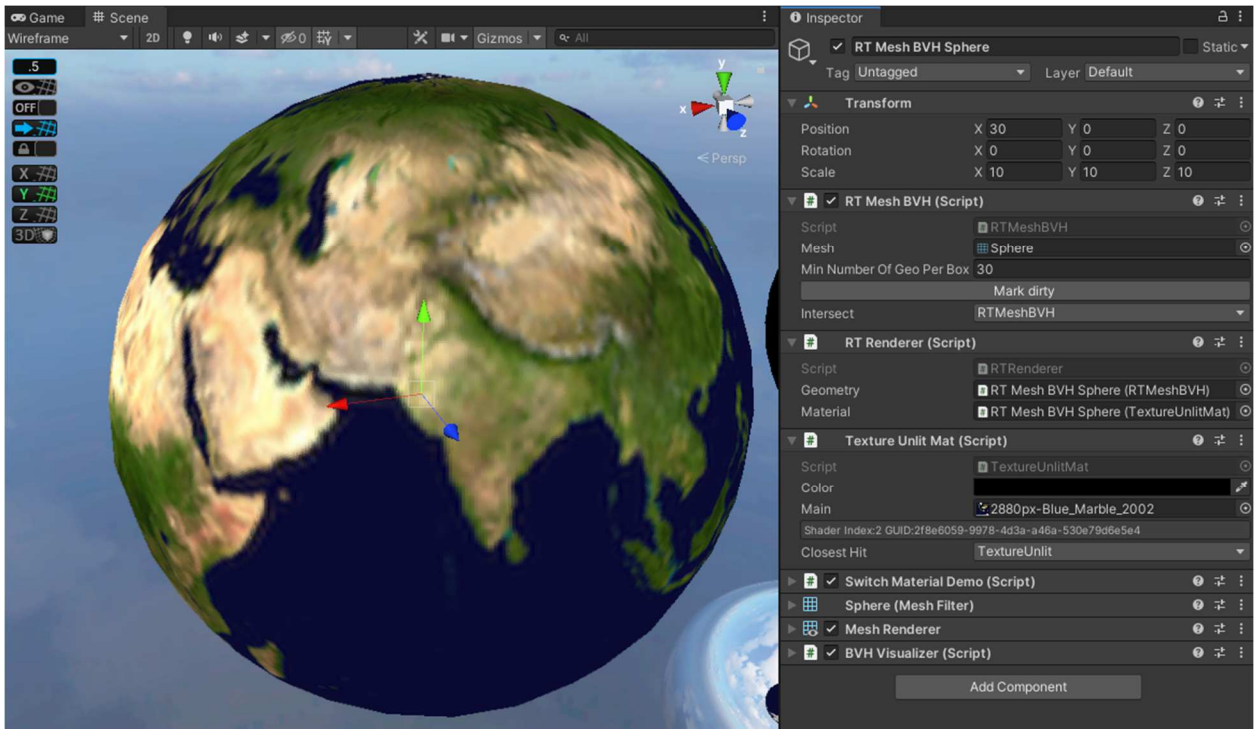
4.2.3 Geometry, Material, and Light Adaptors

When the scene parser of the CPU render pipeline collecting information of the current scene, it needs a way to identify the object type and extract the object's properties accordingly. This is achieved by the adaptors attached to each object.

There are three classes of adapters in our prototype where each provide a specific type of property of an object. Geometry adapter provides the geometry data and the intersection user program to be used. Material adapter provides the material properties and the closest-hit user program to be used. Light adapter is attached on objects representing lights and provides the light parameters and the light user program to be used.

The framework provides base classes for each type of adapters to provide a common interface for the scene parser to query the properties of an object. Developers implements these adaptors based on the data requirement of their ray tracing user programs. With the common interface on each type of adaptor, we can decouple any customized adaptors from the framework code. This allows arbitrary number of adaptors to be created without the user modifies the framework.

Figure 7 shows the adapter assignment on a sphere and a point light. The BVH mesh geometry adapter (denoted as RT Mesh BVH) is responsible for providing the BVH data of the sphere model and the ID of the custom intersection program to be used, which is RTMeshBVH intersection program. The material adapter (denoted as Texture Unlit Mat) provides the texture and the ID of the closest-hit program assigned to the sphere model. The point light adapter (denoted as Custom Point Light) provides the light parameters such as attenuation and the light color and the light program to be used.



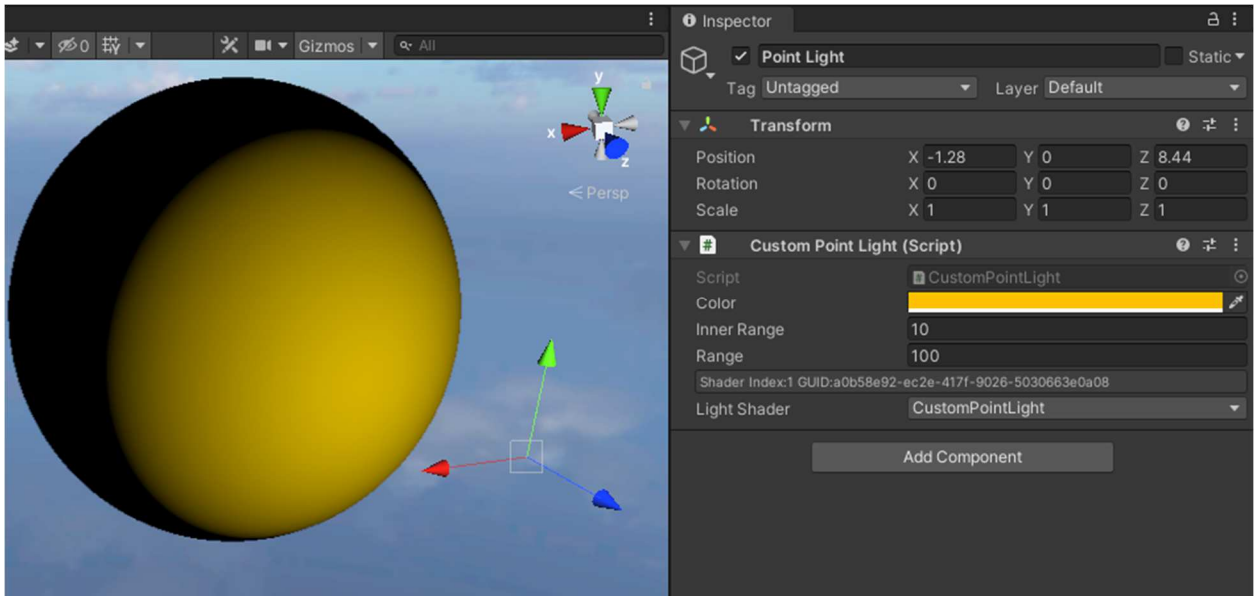


Figure 7. The component assignment of a sphere geometry (upper image) and a point light (lower image).

4.2.4 BVH visualizer

To facilitate fine tuning and debugging of the generated scene- and object-level BVH acceleration structures, we have defined a helper class, the BVH visualizer, that renders the bounding boxes of a BVH for visual inspection by the developers. The BVH visualizer renders the bounding boxes at different levels with a unique color. It is also capable of showing the boxes of a particular level in the BVH hierarchy. Examples of the BVH visualizer can be found in Figure 17 and Figure 18.

4.3 User programs collection

We decided to aggregate user programs by types in switch statements to allow ray tracer selects and invokes a particular user program. The shader manager creates and update the switch statements within the ray tracer where each user program is a unique case in the switch statement and assigned a unique user program ID. During runtime, the ray tracer selects a particular user program by passing the user program ID to the switch statement. A snippet of the closest-hit program collection switch statement can be found in Appendix B.

4.4 Communication between CPU Render Pipeline and the GPU Ray Tracer

Since GPU does not have memory access to the CPU main memory, scene data prepared by the CPU render pipeline must be copied to the GPU memory before the GPU ray tracer can access

them. In the following sections, we will discuss the implementation considerations of passing data from CPU render pipeline to the GPU ray tracer in our prototype.

4.4.1 Serialization

The copy operation from main CPU memory to GPU memory requires data to be serialized. All scene data, except Texture, are first serialized into arrays of floating-point numbers. Texture by design is a matrix of floating-point numbers thus does not require additional serialization. The data serialization is done by the corresponding adaptor. The encoding format must be agreed between the adaptor and the custom program who is going to use the data. Figure 8 shows an example layout of a triangle geometry. The same data type are aggerated into one single array. On the ray tracer side, the triangle is retrieved by computing the index offset of the target triangle in the entire triangle list. Noted that it is up to the developer of the adaptor and custom program to design the data layout.

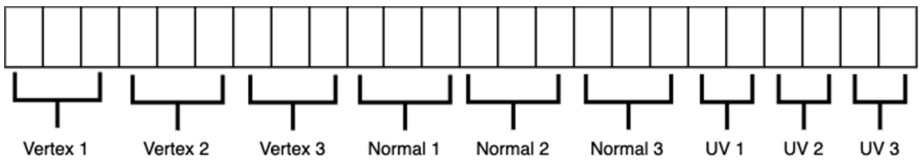


Figure 8. Data layout of a serialized triangle in the memory to be passed to graphical memory

4.4.2 Support multiple primitives in a single node in BVH

A special case of data serialization is BVH with multiple geometries in leaf node. Since the byte length of a bounding box is different from a triangle, putting them on the same array will complicate the access index calculation. Additionally, a bounding box may contain triangles that are not next to each other on the serialized triangle list. We will need a list of triangle index for each bounding box entry. To address the above two requirements, we have designed a data structure consists of three arrays as shown in Figure 9. The first array contains the serialized bounding boxes of the BVH. The leaf node of the BVH (box A) holds the index to an entry on the second array (box B) where each item holds a list of geometry encapsulated by the leaf node's bounding box (box D & E). The length of the list is stored in box C for traversal the second array by index calculation. The value of box D and box E holds the index to the actual triangles stored in the third array (box F & G).

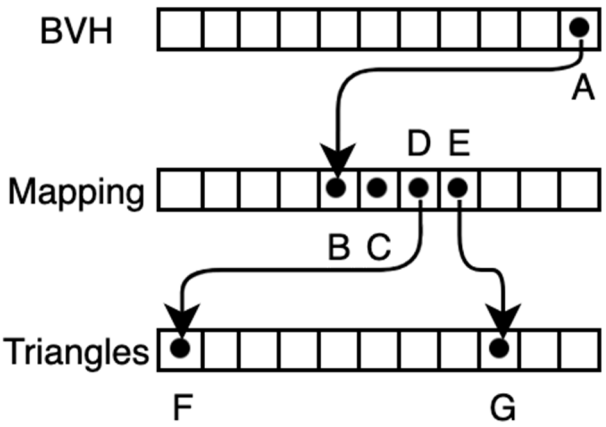


Figure 9. The serialized BVH layout to support leaf nodes with multiple triangles.

4.5 Secondary ray

One of the features of ray tracing is the ability to spawn secondary rays from the intersection location to simulate reflection or translucent. In theory, solution to secondary ray rendering can be structured as recursion. However, recursive functions are not supported in Unity compute shader. We will need to unwind the recursive ray tracing algorithm into iterative solution equivalent. In the following sections, we will discuss how we support secondary rays rendering under this GPU programming limitation.

4.5.1 Iterative approach

Solving recursive problem by iteration usually involves an iteration flow control and an explicit stack to keep track of the invocation order. Implementing iterations is straightforward as Unity's compute shader supports common iteration flow control statements such as For Loop and While Loop. However, creating a resizable stack poses a challenge. Array's length in Unity's compute shader must be defined as compile time constants. This result in a fixed recursive depth. In our implementation, the hard limit of secondary ray generation is set to four.

4.5.2 Limited number of secondary rays spawned at one location

The fixed length array limitation also affects the number of secondary rays spawned at a given location. For certain rendering technique such as ambient occlusion and multiple secondary rays sampling for rough surfaces, multiple secondary rays can be dispatched from the intersection location with different ray parameters. Since we cannot use recursion to follow each secondary

ray, the secondary rays that haven't processed will need to be cached in an array. The array that holds the secondary rays must also be fixed length. Therefore, there is a predefined hard limit on how many secondary rays that can be spawned at a given location. In our implementation, that limit is also set to four.

4.5.3 Performance Consideration

Setting the secondary rays spawning limit should be verified by the available GPU memory. The arrays for tracking secondary rays are created for each ray tracer instance. Given the number of ray tracer instances are equals to the screen resolution, the memory demand of a slight increase in the array size will be multiply by the screen resolution. Therefore, we opt to a smaller array size (four) to relax the system requirement. With more powerful GPU and future improvement of GPU programming framework, these limitations would be lifted to allow a more dynamic scene being rendered.

4.6 Summary

We have covered our efforts in implementing our framework design based on Unity 3D engine's Scriptable Render Pipeline. Our implementation can take advantage of out-of-the-box 3D engine features to enhance the usability of our framework. The programming restrictions of Unity's compute shader runtime and the migration measures were discussed. There are challenges in the integration work that worth further effort to address, such as the lack of support of shader pre-compilation on Unity platform. Nevertheless, the described implementation is a functional prototype of our framework that can support real-time ray tracing rendering in the Unity editor environment. Unity application developers can now use our framework to create real-time reflections and volumetric fog effects. In the next chapter, we will showcase the flexibility of our framework in achieving various rendering techniques.

5 Results

This chapter presents the results of our system design and implementation from two perspectives: demonstrate effective support of customizable visual effects that a programmable ray tracing framework can achieve and analyze the framework real-time performance. The results show that a ray tracing framework based on general programming on GPUs is feasible and flexible on the choice of hardware.

5.1 Programmable Ray Trace System

Our framework supports six different types of user programs. In the following sections, we will discuss the potential uses cases of each type of user program and show our examples.

5.1.1 Custom ray generation

Users can control the render quality and implement various camera projection models in ray generation program. One of the following examples implements anti-aliasing by super-sampling technique [20]. To determine the portions of a pixel that is covered by an object, multiple rays can be emitted at different locations within each screen pixel. Developers can control both the number of rays and sampling strategies and locations within each screen pixel in the ray generation program.

Another use case would be implementing different camera projection models. In perspective camera projection, ray travels from the camera position, through the pixel center position, and onwards. On the other hand, rays travel parallelly to the camera viewing direction in an orthographic camera. Developers can control the initial direction of each ray in each pixel in the ray generation program as well.

A comparison of images between single sampling point ray generation program and super-sampling ray generation program is shown in Figure 10. The edges between the red wall and the white wall, as well as the top edges of the box, is less rough in the image from super-sampling ray generation program. The code snippet of the super-sampling module can be found in Appendix A.

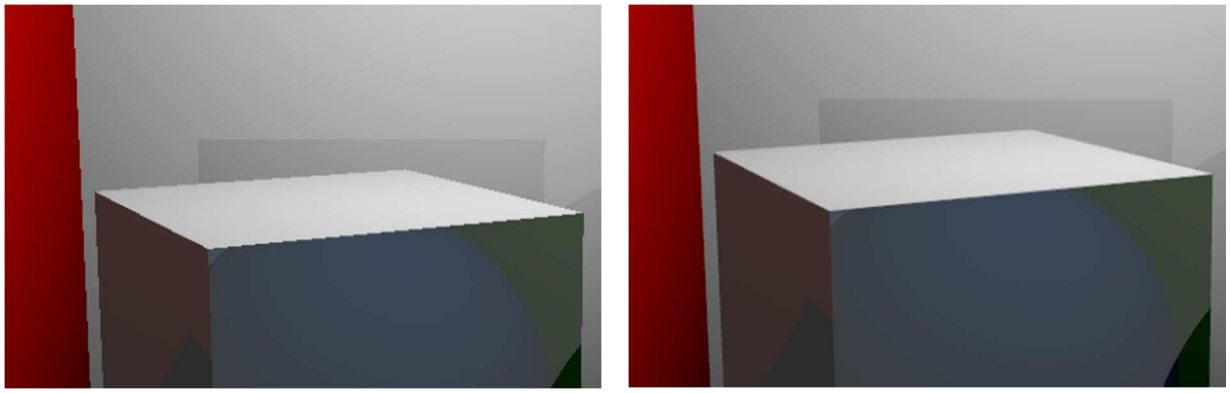


Figure 10. Comparison of images using single sampling ray generation module (left) and that from super-sampling ray generation module (right).

Figure 11 shows the results of ray trace results with perspective (left) and orthographic (right) camera rays. Because of perspective, the image on left clearly displays the foreshortening of objects where the rear of the automobile appears to be smaller than the front. In contrast, the image on the right shows that objects of equal size at different distances from the camera will still be rendered as equal size.

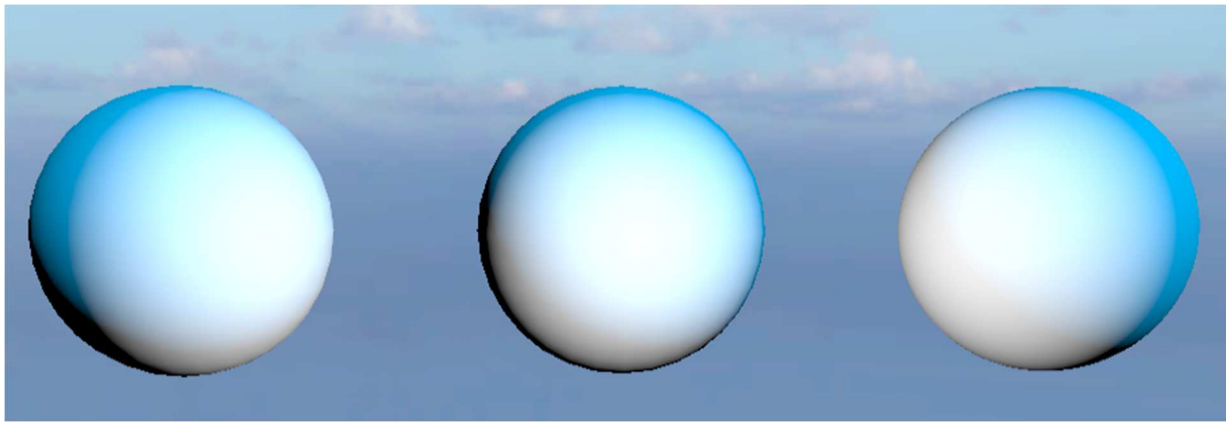
Figure 11. Comparison of images using perspective camera ray generation module (left) and orthographic camera ray generation module (right).

5.1.2 Custom geometries

3D models are usually represented as triangle meshes. In many cases, there exists different representations of a primitive. For example, a sphere can be represented by a center point with a radius, or it can be represented by a collection of triangles based on an explicit tessellation of the quadratic equation. There may also be a need to include an object-level acceleration structure to improve rendering performance on large vertex count 3D meshes. In our framework, we allow

users to create custom ray-object intersection algorithms and object-level acceleration structure traversal program inside an intersection program to support these use cases.

To demonstrate the flexibility of our intersection program, we have implemented three intersection programs that displays a sphere, as shown in Figure 12. These intersection programs are based on three different geometry representations: a spherical equation (right sphere); a triangular mesh (center sphere); and an object-level BVH accelerated triangular mesh of a sphere (left). In Figure 13, we visualized the lowest level of the object-level BVH structure of the left sphere in Figure 12. We also show the wireframe of the triangular mesh sphere in the center of Figure 13. Noted that the right sphere in Figure 12 is based on the quadratic spherical equation and therefore no triangle mesh is shown in the corresponding position in Figure 13.



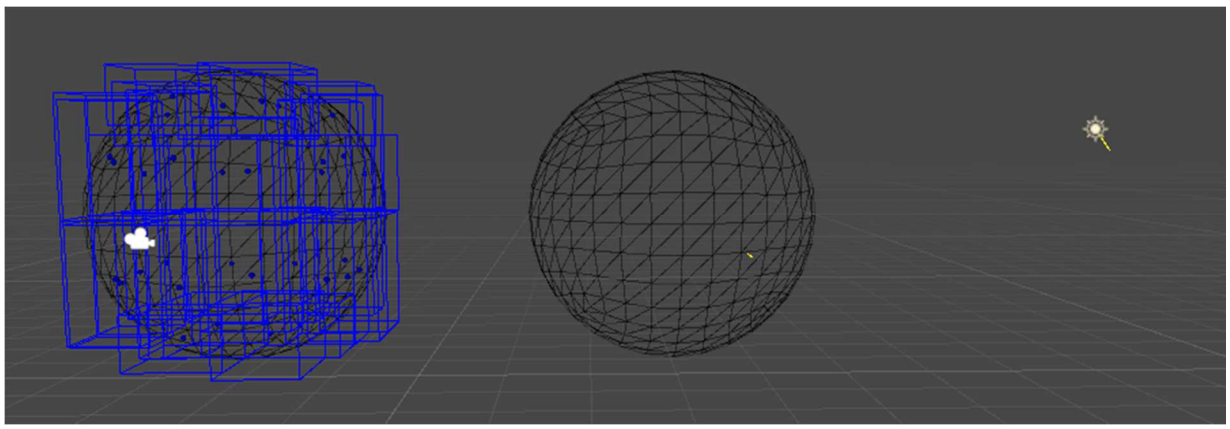


Figure 12 (upper image). Three spheres are rendered using different intersection modules. From left to right, object-level BVH accelerated triangular mesh intersection module; triangular mesh intersection module; spherical equation custom intersection module.

Figure 13 (lower image). From left to right: The lowest level of the object-level BVH structure of the left sphere in Figure 12; The triangular mesh used to generate the left and center spheres in Figure 12.

5.1.3 Custom surface shading by programmable shaders

An essential part of rendering is to determine the color on an object's surface, known as shading. For an opaque object, its shading is affected by the surface properties and the light(s) that illuminate on it. For a translucent object, additional reflective or refractive rays are to be included in shading. These scenarios can be accomplished by a combination of secondary ray generation program, closest-hit program, and light program.

In Figure 14, we have demonstrated 3 scenarios on how a user can control the color of the intersection points on a 3D model surface.

On the left image, we have implemented an unlit texture mapping closest hit shader. This shader maps the intersection point's position to a 2D texture and returns the color value from that texture. The shader is passed with the unique ID of the primitive being hit to retrieve the texture assigned to that primitive.

On the center image, we have implemented a reflective closest hit shader. Our reflective closest hit shader works together with the reflective secondary ray generation shader. At the time a ray intersects a primitive with our reflective secondary ray generation shader, a secondary ray is spawned in the reflection direction. When the secondary ray returns with a color, our reflective closest hit shader combines the reflective color and the Albedo color assigned on the surface as the resulting color. In our example, we set the resulting color directly from the reflective ray's color. Most of the reflective rays hit the skybox as shown in Figure 14. Thus, the reflection image of the skybox is rendered on the 3D model surface.

On the right image, we have implemented a closest-hit shader that follows the Blinn-Phong shading model. Our closest hit shader will dispatch light rays to each light in the scene. Each light can return a color to our closest hit shader. In our example, there is a green point light in front of the 3D model. The intensity of the green color depends on the normal direction of the surface as specified in the Blinn-Phong shading model.

Figure 14. Demonstration of applying closest-hit user program on a 3D model rendering. The closest-hit shader assigned: (left) Unlit texture mapping shader; (center) reflective shader; (right) Blinn-Phong shader with a green point light in front of the model.

Together these three example shader programs demonstrated the framework's support for straightforward 2D file texture mapping, secondary ray spawning and shading, light source sampling, and the general support for illumination computation. These provide flexibility for programmers of our framework to define shader programs with sophisticated, or even strange, capabilities. For example, determining reflection direction based on texture map values.

5.1.4 Custom volumetric fog

Volumetric fog gives visual cue to the viewer on the environment of a scene. For example, the moisture level of a scene can be display as a heavy ground fog, or a light shaft shooting from a window shows the viewer the location of the light sources. In our framework, a closest-hit program in tandem with a secondary ray generation program can be used to support some volumetric rendering techniques such as volumetric fog.

In the two examples shown in Figure 15, the entire scene is encapsulated by a cube volume assigned a volumetric closest-hit program and secondary ray generation program. When a ray hits the surface of the volume, the secondary ray generation program spawns a ray in the direction of incoming ray, traveling inside the body of the geometry. When the secondary ray returns, our closest-hit program will calculate the attenuation by the secondary ray, factoring the scattering effect along the travelled direction.

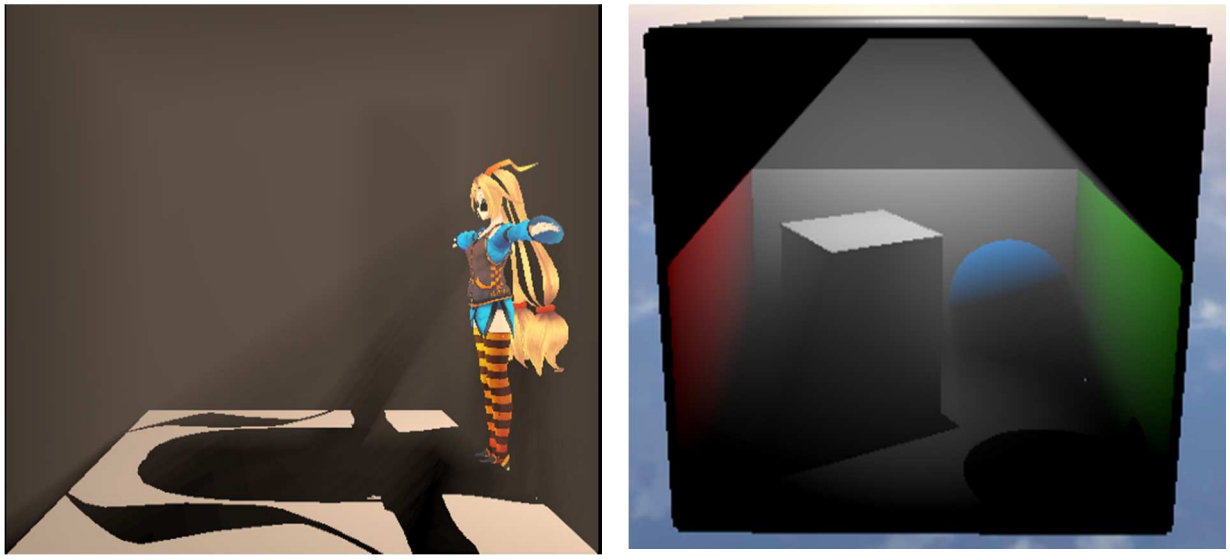


Figure 15. Volumetric fog and shadow effect demonstrations. (Left) Demonstrating light shaft effect by a volumetric fog closest hit shader. A point light is positioned on the righthand side of the image, behind the 3D model; (Right) Demonstrating the light volume of a point light through the square hole on the ceiling of a Cornell Box.

5.1.5 Custom lights

Developers use different types of light sources in a virtual scene to mimic the lighting in physical world. For example, a directional light is used for sunlight and a point light is used in place of light bulbs. Our framework allows developers to implement different types of light source by programming the light ray direction, colors, and attenuation in the light program.

In Figure 16, we demonstrate our implementation of directional and point light sources through two separate light programs. The source code of both light programs can be found in Appendix A.

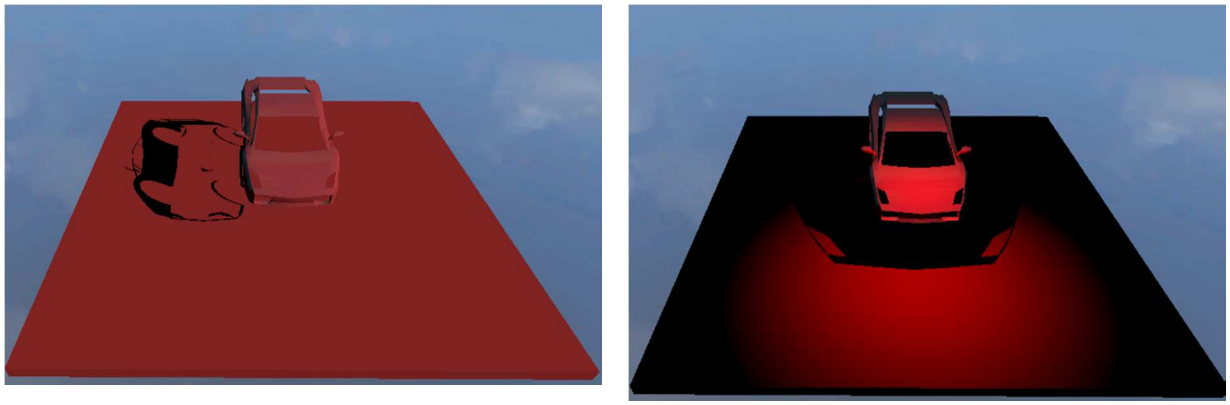


Figure 16. Demonstration of different light source types: (Left) a directional light, pointing down leftward; (Right) a point light, above the front edge of the car model.

5.1.6 Scene-Level Acceleration Structures

The use of acceleration structures improves the rendering performance by selecting a subset of objects participating in ray-object intersection of a given ray. This optimization allows a more complex scene, or the number of objects in a scene, to be rendered. Our framework automatically builds a single bounding box hierarchy (BVH) acceleration structure to include all objects in a scene. This saves the developers from implementing the acceleration structure building and traversing programs.

We visualize the scene-level acceleration structure hierarchy of a demo scene in Figure 17. The root node of the BVH is always encapsulated in the entire scene, shown as a red box. The second level node of the BVH is shown as orange boxes. Depending on the number of geometries and their location distribution, a second level node may contain any number of geometries. If there are multiple geometries within a node, the BVH algorithm will continue bisecting the node to smaller ones.

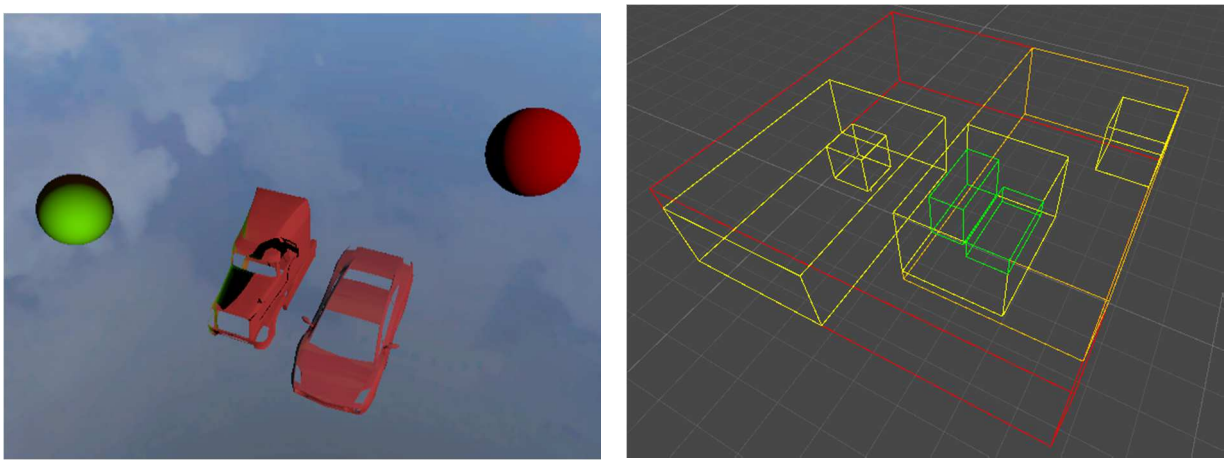


Figure 17. Visualization of the acceleration structure. (Left) The rendered scene; (Right) The bounding boxes of the BVH acceleration structure. Root node shown as a red box. Subsequent levels are shown in the order of orange, yellow, and green boxes.

5.1.7 Object-Level Acceleration Structure

In addition to the scene-level acceleration structure, users can also include a separate, object-level acceleration structure for any object. An object-level acceleration structure can speed up the rendering of an object compose of a large number of geometries. In our framework, an object-level acceleration structure can be implemented in two parts. Firstly, a custom geometry parser is created on the CPU rendering pipeline to construct the acceleration structure and convert it into a data collection that can be used by the ray tracer on the GPU. Secondly, a custom intersection program is created to traverse the acceleration structure during ray tracing.

In our example shown in Figure 18, we have visualized the object-level acceleration structure of a truck frame consists of more than 4000 triangles. The acceleration structure building algorithm is BVH.

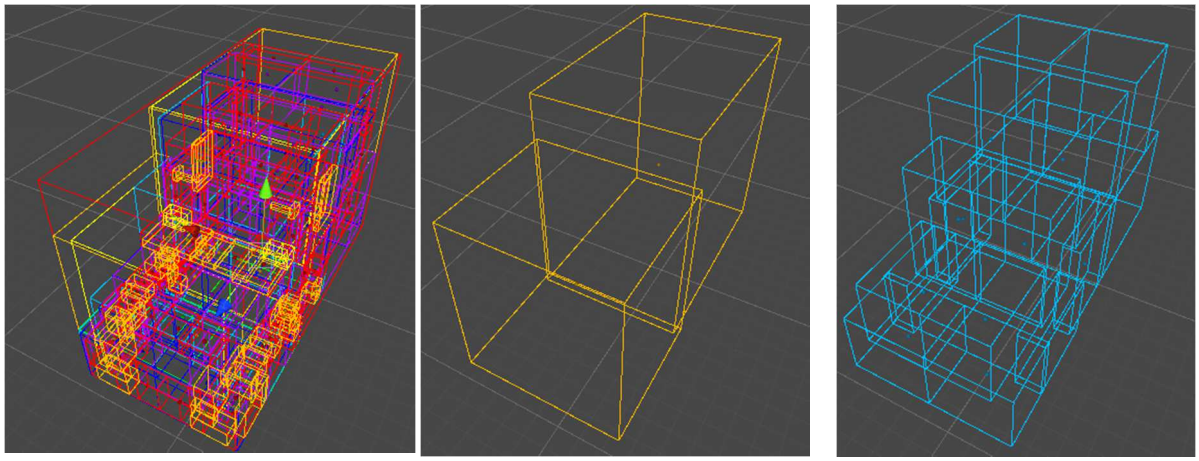


Figure 18. Visualization of all bounding boxes of a local level BVH acceleration structure of a truck model. (Left) Showing all levels; (Center) Showing second level; (Right) Showing forth level;

5.1.8 Summary

In this section, we have demonstrated the programmability of our ray tracing framework based on general programming on GPUs. The types of programmable modules supported are comparable to existing commercial products. Our framework does not require special hardware or platform, which will increase the access of ray tracing rendering to the public. In the next section, we are going to discuss the rendering performance of our framework.

5.2 Analysis of Real-Time Performance

Most ray tracing rendering systems report and compare their performance in terms of number of triangles presented in a testing scene [6], [14]. This assumes an optimized ray tracing framework should be able to render a scene contains a large number of objects in real-time and achieve higher screen resolution by dispatching more rays. One way to measure the number of triangles supported by a ray tracing framework in a real-time rendering context is to graduate increases the number of triangles presented in a scene until the frame per second (FPS) drops to certain level. We selected 30 FPS as used in other reports [6]. Note that many testing scenes are static, meaning the objects inside the scenes are not undergoing Affine transformations during the measurement. This minimizes the fluctuations in the measurement causes by object transformation calculation and acceleration structure rebuilding.

Our experiment consists of a scene with two shadow casting light sources and a number of 3D robot models, each consists of 6132 triangles. The entire mesh is texture mapped with a 1024 * 1024 px RGB texture. Object-level acceleration structures have been built for each robot model. The closest-hit shader assigned is the traditional Blinn-Phong shading model that supports texture mapping as the material color. Each intersection point performs two shadow ray sampling to determine the illumination by the two light sources in the scene. There is no reflective or translucent effect in the scene. We continue to increase the number of the robot models until the FPS of the rendered scene drops below 30. The FPS figure reported by Unity Editor is used (Figure 20). Noted that the triangle count shown in Figure 20 only reports triangles passing to the default Unity's rasterization render pipeline. Therefore, the triangles submitted to our ray tracer are not counted. The screen resolution is kept at 1920 * 1080 pixel (HD 1080p). The system configuration is as follows: CPU: Intel i7-9750H@2.6GHz; RAM: 16 GB; GPUs: NVIDIA Quadro T2000 with 4GB dedicated memory.

We have recorded 104k triangles with 2 light sources presented in our testing scene, shown in Figure 19, before the FPS drops below 30. Table 1 summarize the performance statistic of the test scene when the FPS just drop below 30. Our framework on the given system can supports around 106k triangles @ 30 FPS. CPU thread time (16ms) and GPU thread time (10ms) were similar when the scene is static. In this scenario, the load distribution between the CPU and GPU are rather balanced without obvious bottleneck. However, when objects are moving, we observe a noticeable increase in CPU thread time (1600ms): a result of acceleration structure rebuilding on the CPU render pipeline.

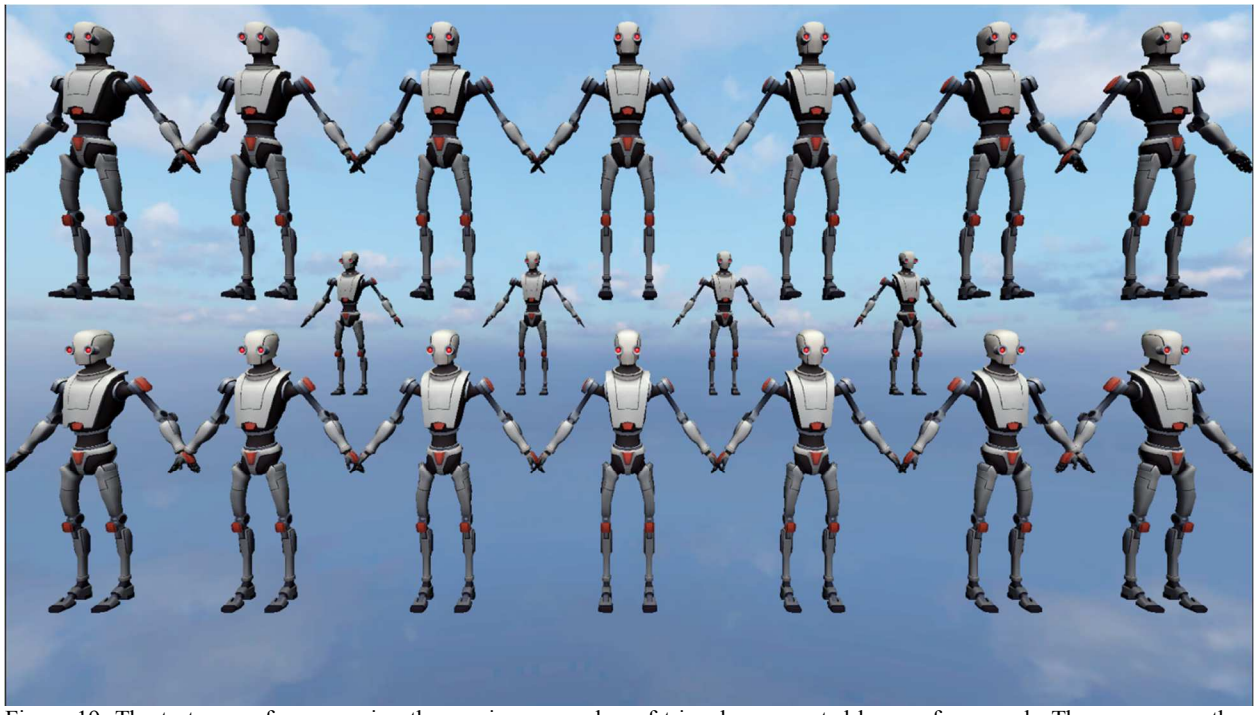


Figure 19. The test scene for measuring the maximum number of triangles supported by our framework. There are more than 104,000 triangles presented in the scene.

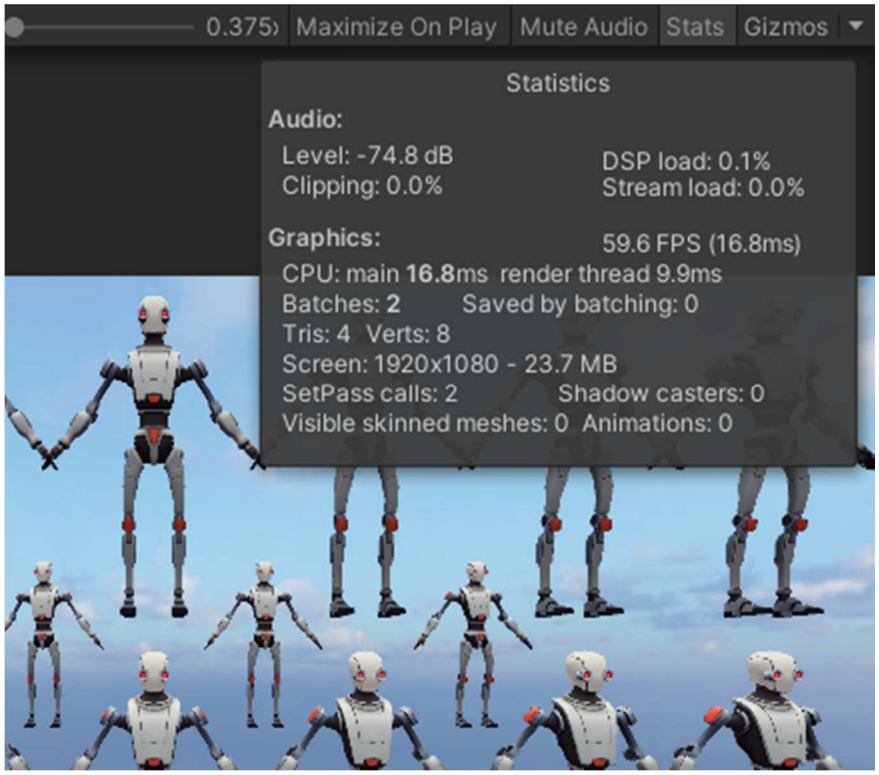


Figure 20. The FPS counter in Unity editor.

Table 1. Performance statistics of the test scene rendering when FPS is just below 30 FPS.	
Number of Triangle	106k
Frame per second (FPS)	30
CPU thread time (ms)	16
GPU thread time (ms)	10
BVH rebuild time (ms)	1600

Building acceleration structures for a large triangle meshes is an expensive operation compared to other operations in CPU render pipeline when acceleration structure building is not involved. The CPU render time when acceleration structure is rebuilt is around 1600ms compared to 16ms when the acceleration structure is taken from the cache. A significant FPS drops is observed when we move the 3D models around the scene as a result of prolonged CPU time to complete a frame.

Two optimizations could be applied to our acceleration structure builder. Firstly, all the vertex position should be set to object space. Since object space coordinates are unaffected by the Affine transformation of the entire object, the object-level acceleration structure based on local space coordinates is still valid after Affine transformation. This saves the rebuilding of object-level acceleration structure. Secondly, a better data structure to hold the object-level acceleration structure collection can reduce the number of rebuilding. As describe in 4.4.1, data of the same type are serialized and insert into the same array. Any modification on one object-level acceleration structure, such as mesh deformation or adding new meshes, will requires re-serializing all the object-level acceleration structures of the same type. This can be avoided by having each object-level acceleration structure as a separate entity on the serialized array. Although current GPU general programming runtime does not support an array of arrays, we can achieve similar data structure by using a list of texture where the texture behaves as a 1D or 2D array.

Note that the number of triangles supported is only one of the indicators of the quality of a ray tracing framework. There could be tradeoff in performance in return for a more flexible rendering. For example, a ray tracing framework that support multiple geometry types is requires to switch between geometry intersection programs, which would receive some performance penalty. Given that the actual performance measurements are affected by multiple factors, including hardware configuration, the system's current load, the scene composition, user actions, the raw number and

any direct comparisons may not yield decisive conclusion. Yet, these performance numbers do provide insights on the degree of complexity of a scene a ray tracing framework could handle.

6 Discussion

In this chapter, we will discuss the tradeoffs and limitations of our design and implementation. There are three kinds of tradeoffs. The first kind are tradeoffs we have made within our framework codebase for simplicity or performance gain. The second kind of tradeoffs or limitations are inherited from the underlying technology stack. The third kind are limitations of the current GPU execution model.

6.1 Functionality Limitations

We have identified seven features that are not included in our framework design:

- GPU instancing
- Post-processing support
- Multi-pass rendering
- Object space ray-geometry intersection
- Scene data caching
- Task scheduling
- Camera inside optical dense medium

6.1.1 GPU Instancing

GPU instancing allows creation or duplication of geometries by the renderer running on GPU. This supports rendering a scene consists of large number of objects of the same geometry, without explicitly duplicating the geometry of the objects in the GPU. Trees in a forest, or buildings in a city from afar are some examples that can significantly benefit from GPU instancing. Implementing GPU instancing on our design requires updating the scene level acceleration structure to include bounding boxes for those generated instancing. However, the scene level acceleration structure is built by the CPU render pipeline before submitted to GPU ray tracer. A solution would be creating a placeholder bounding box around the location where the instances may locate. The placeholder would have assigned a custom intersect program that perform GPU instancing on the ray tracer.

6.1.2 Post-Processing Support

Customizable post-processing program is another feature we did not included in the current framework. Post-processing allows developers apply additional filters to the rendered images before sending the images to a display device. Some typical use cases of post-processing include noise reduction, anti-aliasing, and motion blurring. A post-processing module can be supported by passing the raw rendered image to an additional processing stage in our rendering pipeline after the GPU ray tracer and retrieving the processed image from the stage output.

6.1.3 Multi-pass rendering

The third feature we decided not to include is support for multi-pass rendering. An example is the support for shadow depth map. In shadow map approach, each light pre-compute the visibility of the objects in the scene from the light perspective before the ray tracer. There are other multi-pass techniques, such as caustic [21] and motion blur [22], that will improve the expression of a rendered scene. The challenging part is providing an infrastructure for developers to perform pre-ray tracing rendering passes. Developers may design the multi-pass to run on the CPU render pipeline or the GPU ray tracer. An update on both components is needed to allow developers defining tasks on each multi-pass and how the result from each pass is combined. A closer study in the design of the existing multi-pass rasterizing or ray tracing frameworks may inspire future work on bringing multi-pass rendering support to our ray tracing framework.

6.1.4 Object space ray-geometry intersection

Geometry data and ray intersection positions in our framework are on the camera coordinate space, which simplifies the computations. However, geometries in the camera space require separated copies for each object instance. Performing affine transformation on an object also require resubmitting the updated object to the ray tracer. This increases the amount of data required to be transferred to the GPU per frame and may causes longer render time at frame when coordinates update is performed. For complex geometries represented by an object-level acceleration structure, reconstructing a large acceleration structure per transformation may stall the rendering. An

alternative would be to perform all computations in the object space. In such a case, objects of the same geometry with different transform can still share a single copy of the geometry or object-level acceleration structure. This can reduce the required data transfer amount and may improve the renderer performance.

6.1.5 Scene data caching

In our current CPU render pipeline design, the scene level acceleration structure and object data are cached to avoid unnecessary rebuilding both scene-level and object-level acceleration structures. Yet, in our Unity implementation, the cached result is sent to the ray tracer every frame because the data loaded to the GPU memory are lost after the ray tracer execution. We suggest future work to investigate the use of GPU memory persistency to cache scene data.

6.1.6 Task scheduling

Scheduling which tasks to execute on the GPU can improve the SIMD processor utilization. Multiple GPU execution models were proposed to optimize certain types of workloads [23]. Our ray tracer implementation follows the “Run-To-Complete” model where we dispatch each thread with the entire ray tracer executable. “Run-To-Complete” model is simple to implement and does not requires overriding the default GPU thread scheduler. However, a long running task in one thread may hold up all the processing units. For example, a view ray that hits multiple reflective surfaces will take more steps to complete while another view ray that does not hit any object in the scene will finish ray tracing earlier. Under “Run-To-Complete”, the whole batch of threads will have to wait for the multiple reflection ray finish execution before releasing the processors. Alternate tasks scheduling approaches, including the Mega Kernel model where tasks are dispatched by a dedicated software scheduler owning the GPU threads, or the Kernel-by-Kernel model where a CPU scheduler identifies and dispatches similar threads in batches, should be examined and can significantly improve the system performance.

6.1.7 Camera inside optical dense medium

The current framework does not support volumetric rendering when a camera is placed within an optical dense medium. Camera inside a medium will be treated as it is in a clear medium, meaning no scattering or other volumetric rendering effects are applied on the view rays. Addressing this shortcoming requires each primary ray to detect the medium it is currently in every frame. A future work on this issue could be searching for the volume that encapsulate the position of the ray origin with the help of the scene-level acceleration structure.

6.2 Platform Limitation

In this section, we will discuss the limitations we have encountered when implementing our design on Unity 3D engine. Note that integrating on a different platform may have a different set of engineering challenges.

The usability of our framework is impacted by the lack of support of precompiled shaders. Since we are packing all user programs and framework code into a single assembly, any changes in one of the shader programs will require a recompilation of the entire codebase. When there is a large collection of user programs, the compilation may take longer time to complete.

A possible solution would be shader binding. In shader binding, user programs and frameworks are bonded by interfaces. As long as the interfaces are unchanged, only the user program that has been modified will need to recompile. This will speed up the compilation of very large projects in order to maintain usability. Similar technology can be found in DirectX 11. However, on Unity 3D engine, shader compilation is managed by the engine and is not exposed to the application developers. At the moment of this writing, it is not possible to have shader binding on Unity 3D engine.

6.3 GPU Limitation

Our last discussion is on the limitation of general programming on GPUs that affects our design. Certain basic operations, such as recursion and dynamic allocations, are not supported on GPUs. As discussed in 4.5, the lack of support for recursion is particularly impacting our design decisions. Ray tracing algorithm solves the rendering problem of a ray and may spawn multiple secondary

rays. Those newly spawned secondary rays then use the same ray tracing algorithm to render and spawn subsequent rays. Thus, the entire ray tracing rendering can be structured as recursion. Since recursion is not supported, we are required to rewrite the algorithm in an iterative approach and the use of a stack. Because dynamic allocation is not supported as well, a fixed length array is used to implement a stack. These result in a hard limit for maximum number of secondary rays that can be spawned at a given location and number of secondary ray generations. Although declaring a large enough array may relax this limitation, given the arrays are locally owned by each ray tracer instance and the number of ray tracer instances is equals to the screen dimension, a slight increase in the array size will be multiplied by the screen dimension, result in a much larger graphical memory demand.

7 Conclusion

This project demonstrated the feasibility of a user programmable real-time ray tracing framework based on general programming capacity on commodity GPUs. The current iteration of our framework supports user defined ray generation, ray-geometry intersection, custom acceleration structure, surface shading, secondary ray effects, lighting model, and miss program. The number of programmable modules supported is comparable to existing commercial solutions.

We have integrated our ray tracing framework on Unity 3D engine to demonstrate potential use cases of the framework. The demonstrations include custom geometries, real-time reflection, volumetric shadows, and custom lighting. In our integration, the only required component provided by Unity is the compute shader compiler. Since most integrated 3D application environments has their own compute shader compiler, we believe our framework design is portable to other development platforms with proper compute shader compilation features.

With more powerful GPUs being constantly introduced and the associated programming models continuously improving, it is expected that the limitations discussed in the previous chapter would be addressed in the next few years. One future direction of this project is to design an execution model that can increase the utilization rate of GPU cores by dividing and scheduling ray tracing operations to minimize long running threads holding up all the execution units. A recent trend is that GPU manufacturers have begun to support hardware accelerated ray tracing specific operations on their chips, such as acceleration structure building and traversal, and ray-triangle intersection. Another direction is to detect and take advantage of such hardware features on the target devices at ray tracing system configuration time.

We believe that real-time ray tracing framework based on basic consumer grade programmable graphics processors is feasible and should be able to compete with the current platform specific but unportable solutions. Such solutions would allow the coexistence of ray tracing and traditional feed forward rendering pipeline on a single chip. Existing software will not need to be rewritten and newer GPUs can maintain backward compatibility.

References

[1] Kevin G. Suffern, “What Is Ray Tracing?,” in *Ray tracing from the ground up*, A K Peters, Ltd, 2007, p. xiii.

[2] Unreal Engine, “Reflections Real-Time Ray Tracing Demo | Project Spotlight | Unreal Engine.” https://www.youtube.com/watch?v=J3ue35ago3Y

[3] NVIDIA GeForce, “Minecraft with RTX Beta | Official Worlds Tour,” *Youtube*, Apr. 16, 2020. https://www.youtube.com/watch?v=wUCRr04DIHU

[4] P. H. Christensen, J. Fong, D. M. Laur, and D. Batali, “Ray Tracing for the Movie `Cars’,” in *2006 IEEE Symposium on Interactive Ray Tracing*, Salt Lake City, UT, Sep. 2006, pp. 1–6. doi: 10.1109/RT.2006.280208.

[5] P. Christensen *et al.*, “RenderMan: An Advanced Path-Tracing Architecture for Movie Rendering,” *ACM Trans. Graph.*, vol. 37, no. 3, pp. 1–21, Aug. 2018, doi: 10.1145/3182162.

[6] S. G. Parker *et al.*, “OptiX: a general purpose ray tracing engine,” *ACM Trans. Graph.*, vol. 29, no. 4, p. 1, Jul. 2010, doi: 10.1145/1778765.1778803.

[7] “GeForce 30 series,” *Wikipedia*. https://en.wikipedia.org/wiki/GeForce_30_series

[8] I. Buck, “The Evolution of GPUs for General Purpose Computing,” p. 38.

[9] NVIDIA, “CUDA Toolkit.” https://developer.nvidia.com/cuda-toolkit

[10] Microsoft, “Compute Shader Overview.” https://docs.microsoft.com/en-us/windows/win32/direct3d11/direct3d-11-advanced-stages-compute-shader

[11] The Khronos® Group Inc, “OpenCL Overview.” https://www.khronos.org/opencl/

[12] The Khronos® Group Inc., “OpenGL Overview.” https://www.khronos.org/opengl/

[13] Microsoft, “DirectX Developer Blog.” https://devblogs.microsoft.com/directx/landing-page/

[14] I. Wald, S. Woop, C. Benthin, G. Johnson, and M. Ernst, “Embree: a kernel framework for efficient CPU ray tracing,” *ACM Transactions on Graphics (TOG)*, vol. 33, no. 4, pp. 1–8, 2014, doi: 10.1145/2601097.2601199.

[15] Microsoft, “DirectX Raytracing (DXR) Functional Spec.” https://github.com/microsoft/DirectX-Specs/blob/master/d3d/Raytracing.md (accessed Apr. 28, 2020).

[16] Apple Inc, “Developing and Debugging Metal Shaders.” https://developer.apple.com/documentation/metal/shader_authoring/developing_and_debugging_metal_shaders (accessed Apr. 28, 2020).

[17] Unity Technologies, “Scriptable Render Pipeline.” https://docs.unity3d.com/Manual/ScriptableRenderPipeline.html

[18] Unity Technologies, “Unity 2019.3.” https://unity.com/releases/2019-3

[19] Unity Technologies, “Getting started with ray tracing.” https://docs.unity3d.com/Packages/com.unity.render-pipelines.high-definition@7.1/manual/Ray-Tracing-Getting-Started.html

[20] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars, Eds., “Arrangements and Duality,” in *Computational Geometry: Algorithms and Applications*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 173–190. doi: 10.1007/978-3-540-77974-2_8.

[21] X. Wang and R. Zhang, “Rendering transparent objects with caustics using real-time ray tracing,” *Computers & Graphics*, vol. 96, pp. 36–47, May 2021, doi: 10.1016/j.cag.2021.03.003.

[22] Q. Hou, H. Qin, W. Li, B. Guo, and K. Zhou, “Micropolygon ray tracing with defocus and motion blur,” *ACM Trans. Graph.*, vol. 29, no. 4, pp. 1–10, Jul. 2010, doi: 10.1145/1778765.1778801.

Appendix A – User program examples

The following are examples of each user program type. Description of each user program type can be found in 3.3.

Ray Generation Program

The example ray generation program shown spawns view rays based on perspective camera projection. The number of view rays from each pixel is set to four to achieve super-sampling.

```
#ifndef CustomPinCameraRandomSS_RayGen_Compute
#define CustomPinCameraRandomSS_RayGen_Compute

// [RayGeneration(CustomPinCameraRandomSS)]
Ray CustomPinCameraRandomSS(uint width, uint height, uint3 id, uint iter)
{
    // Transform pixel to [-1,1] range
    // https://stackoverflow.com/questions/4200224/random-noise-functions-for-glsl
    float rd = frac(sin(dot((id.xy + float2(0.25f + iter * 0.25f, 0.25f + iter * 0.25f)),
float2(12.9898,78.233))) * 43758.5453);
    float2 uv = float2((id.xy + float2(rd, rd)) / float2(width, height) * 2.0f - 1.0f);

    // Transform the camera origin to world space
    float3 origin = mul(_CameraToWorld, float4(0.0f, 0.0f, 0.0f, 1.0f)).xyz;

    // Invert the perspective projection of the view-space position
    float3 direction = mul(_CameraInverseProjection, float4(uv, 0.0f, 1.0f)).xyz;
    // Transform the direction from camera to world space and normalize
    direction = mul(_CameraToWorld, float4(direction, 0.0f)).xyz;
    direction = normalize(direction);
    return CreateRay(origin, direction, 1);
}

uint CustomPinCameraRandomSSNumberOfRay(uint width, uint height, uint3 id)
{
    return 4;
}

#endif
```

Intersection Program

The example intersection program shown represents a sphere. It performs ray-sphere intersection calculation to determine the location of the intersection point for shading or as the ray origin of secondary rays. *SphereIntersectUtils* is a helper function provided by our framework to perform ray-sphere intersection calculation on the GPU.

```
#ifndef RTSphere_Intersect_Compute
#define RTSphere_Intersect_Compute

// [intersect(RTSphere)]

struct RTSphere
{
    float3 center;
    float radius;
};

StructuredBuffer<RTSphere> _RTSphere;

void RTSphereIntersect(Ray ray, inout RayHit bestHit, Primitive primitive, int primitiveId)
{
    // For every sphere in the primitive, we do:

    [fastopt] for(int s = 0; s < primitive.geometryInstanceCount; s++)
    {
        RTSphere sphere = _RTSphere[primitive.geometryInstanceBegin + s];

        float2 result = SphereIntersectUtils(ray, bestHit, sphere.center, sphere.radius, primitiveId);

        if (result.x != -1) // Has intersect?
        {
            bestHit.primitiveId = primitiveId;

            if (result.y <= 0)  // Ray is inside->outside?
            {
                // Ray is inside->outside
                bestHit.mediumToEnter = 0; // FIXME: Even if an inside ray is leaving, it may not necessar
y entering the air
                bestHit.mediumToLeave = primitiveId;
            }
            else
            {
                bestHit.mediumToEnter = primitiveId;  // Ray is outside->inside
                bestHit.mediumToLeave = ray.medium;
```

```
            }
        }
    }
}

#endif
```

Secondary Ray Generation Program

The example secondary ray generation program represents a reflective and/or a translucent surface. This program outputs an array of secondary rays to the caller. The actual shading of the object is performed by the closest-hit program of the same object. The closest-hit program will blend the ray color returned from the secondary rays with other shading result. The degree of reflectivity and transparency is controlled by the *TranslucentMat_reflectivity* and *TranslucentMat_transparency*.

```
StructuredBuffer<float4> TranslucentMat_color;
StructuredBuffer<float> TranslucentMat_reflectivity;
StructuredBuffer<float> TranslucentMat_secondaryRayEffect;
StructuredBuffer<float> TranslucentMat_transparency;

// [shader(Translucent)]
void Translucent_SecRays(
    inout Ray ray,
    RayHit hit,
    inout SecRaysAtHit secRaysAtHit
)
{
    if (ray.gen <= 1) {
        return;
    }

    // Translucent
    secRaysAtHit.srays[0].origin = hit.position;
    secRaysAtHit.srays[0].direction = ray.direction;
    secRaysAtHit.srays[0].color = float3(0, 0, 0);
    secRaysAtHit.srays[0].gen = ray.gen - 1;
    secRaysAtHit.srays[0].medium = hit.mediumToEnter;   // Translucent ray enter the object
    secRaysAtHit.srays[0].tmin = 0.01f;
    secRaysAtHit.srays[0].weight = TranslucentMat_transparency[_Primitives[hit.primitiveId].materialInstan
ceIndex];

    // Reflective
    secRaysAtHit.srays[1].origin = hit.position;
```

```
    secRaysAtHit.srays[1].direction = normalize(reflect(ray.direction, hit.normal));
    secRaysAtHit.srays[1].color = float3(0, 0, 0);
    secRaysAtHit.srays[1].gen = ray.gen - 1;
    secRaysAtHit.srays[1].medium = hit.mediumToLeave;   // Reflective ray did not enter the object, medium
 is the surrounding
    secRaysAtHit.srays[1].tmin = 0.01f;
    secRaysAtHit.srays[1].weight = TranslucentMat_reflectivity[_Primitives[hit.primitiveId].materialInstan
ceIndex];

}
```

Closest-Hit Program

This example of closest-hit program complements the secondary ray generation program shown above to render a reflective or translucent object. It takes the ray color from the secondary ray generation and blends with the solid color of the object from *TranslucentMat_color*. *GetIlluminate* is a helper function provided by the framework to perform lighting and shadowing.

```
float3 Translucent(
   inout Ray ray,
   RayHit hit,
   float3 ambientLightUpper,
   float3 secondaryRayColor
)
{
    float3 color = float3(0, 0, 0);
    float4 matColor = TranslucentMat_color[_Primitives[hit.primitiveId].materialInstanceIndex];

    [fastopt] for(int l = 0; l < _NumOfLights; l++)
    {
        LightHit light0 = GetIlluminate(hit.position, 0, hit.primitiveId, l);   // We are calculating the
surface color, so medium = 0

        float nDotL = dot(hit.normal, -1 * light0.direction);
        if (nDotL >= 0)
        {
            color += matColor.xyz * light0.color * nDotL;
        }
    }

    float sec = TranslucentMat_secondaryRayEffect[_Primitives[hit.primitiveId].materialInstanceIndex];

    return (1 - sec) * color + sec * secondaryRayColor;
}
```

Light Program

This example of light program represents a point light with attenuation. *ShadowTrace()* is a helper function provided by the framework to determine if an intersection point has direct line-of-sight to this light source. If an intersection point has no line-of-sight, it is considered to be in the shadow from the perspective of this light source. A black color is returned as the light color.

```
StructuredBuffer<float4> CustomPointLight_color;
StructuredBuffer<float> CustomPointLight_innerRange;
StructuredBuffer<float> CustomPointLight_range;

LightHit GetIlluminate_CustomPointLight(LightInfo lightInfo, float3 hitPos, int medium, int primitiveId)
{
    float3 fullColor = CustomPointLight_color[lightInfo.instanceIndex];
    float range = CustomPointLight_range[lightInfo.instanceIndex];
    float innerRange = CustomPointLight_innerRange[lightInfo.instanceIndex];
    float3 direction = normalize(hitPos - lightInfo.position);
    float dist = distance(hitPos, lightInfo.position);

    if (dist > range || range <= 0) {
        return CreateLightHit(float3(0, 0, 0), float3(0, 0, 0));    // Beyond the point light range
    }
    if (dist < innerRange) {
        return CreateLightHit(fullColor, direction);    // Full intensity inside inner range
    }

    Ray ray;
    ray.color = fullColor * (1 - smoothstep(innerRange, range, dist));
    ray.origin = lightInfo.position;
    ray.direction = direction;
    ray.gen = 1;
    ray.tmin = 0.0001f;
    ray.tmax = dist;    // Any hit behind me does not considered as blocking
    ray.medium = medium;
    ray.weight = 1;

    return CreateLightHit(ShadowTrace(ray, primitiveId), direction);
}
```

Miss Program

A simple miss program that uses a skybox texture as the background of the ray traced scene.

```
float3 missShader(float3 direction) {
    // Sample the skybox and write it
    float theta = acos(direction.y) / -3.14159265359f;
    float phi = atan2(direction.x, -direction.z) / -3.14159265359f * 0.5f;
    return _SkyboxTexture.SampleLevel(sampler_SkyboxTexture, float2(phi, theta), 0).xyz;
}
```

Appendix B – Example of a user program collection

The following is the code snippet of the closest-hit user programs collection generated by the ray tracing framework automatically. This generated program is in HLSL.

```
float3 ClosestHit(
        inout Ray ray,
        RayHit hit,
        float3 ambientLightUpper,
        float3 secondaryRayColor)
{
        switch(_Primitives[hit.primitiveId].materialIndex)
        {
                case 0:
                        return Phong(ray, hit, ambientLightUpper, secondaryRayColor);
                case 1:
                        return Refractive(ray, hit, ambientLightUpper, secondaryRayColor);
                case 2:
                        return TextureUnlit(ray, hit, ambientLightUpper, secondaryRayColor);
                case 3:
                        return Translucent(ray, hit, ambientLightUpper, secondaryRayColor);
                case 4:
                        return Volumetric(ray, hit, ambientLightUpper, secondaryRayColor);
                default:
                        return float3(0, 1, 1);
        }
}
```