



**US Army Corps
of Engineers®**
Engineer Research and
Development Center



Military Engineering

A General-Purpose Multiplatform GPU-Accelerated Ray Tracing API

R. Harrison Hunter, S. Ross Glandon, and Jing-Ru C. Cheng

July 2023

The US Army Engineer Research and Development Center (ERDC) solves the nation's toughest engineering and environmental challenges. ERDC develops innovative solutions in civil and military engineering, geospatial sciences, water resources, and environmental sciences for the Army, the Department of Defense, civilian agencies, and our nation's public good. Find out more at www.erdclibrary.on.worldcat.org/discovery.

To search for other technical reports published by ERDC, visit the ERDC online library at <http://www.erdclibrary.on.worldcat.org/discovery>.

A General-Purpose Multiplatform GPU-Accelerated Ray Tracing API

R. Harrison Hunter, S. Ross Glandon, and Jing-Ru C. Cheng

*US Army Engineer Research and Development Center (ERDC)
Information Technology Laboratory (ITL)
3909 Halls Ferry Road
Vicksburg, MS 39180-6199*

Final Technical Report (TR)

DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited.

Prepared for Headquarters, US Army Corps of Engineers
Washington, DC 20314-1000

Under PE 0602145A, Project BG2, Task 2, "Military Engineering"

Abstract

Real-time ray tracing is an important tool in computational research. Among other things, it is used to model sensors for autonomous vehicle simulation, efficiently simulate radiative energy propagation, and create effective data visualizations. However, raytracing libraries currently offered for GPU platforms have a high level of complexity to facilitate the detailed configuration needed by gaming engines and high-fidelity renderers. A researcher wishing to take advantage of the performance gains offered by the GPU for simple ray casting routines would need to learn how to use these ray tracing libraries. Additionally, they would have to adapt this code to each GPU platform they run on. Therefore, a C++ API has been developed that exposes simple ray casting endpoints that are implemented in GPU-specific code for several contemporary device platforms. This API currently supports the NVIDIA OptiX ray tracing library, Vulkan, AMD Radeon Rays, and even Intel Embree. Benchmarking tests using this API provide insight to help users determine the optimal backend library to select for their ray tracing needs. HPC research will be well-served by the ability to perform general purpose raytracing on the increasing amount of graphics and machine learning nodes offered by the DoD High Performance Computing Modernization Program.

DISCLAIMER: The contents of this report are not to be used for advertising, publication, or promotional purposes. Citation of trade names does not constitute an official endorsement or approval of the use of such commercial products. All product names and trademarks cited are the property of their respective owners. The findings of this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

DESTROY THIS REPORT WHEN NO LONGER NEEDED. DO NOT RETURN IT TO THE ORIGINATOR.

Contents

Abstract	ii
Figures	v
Preface	vi
1 Introduction.....	1
1.1 Background.....	1
1.1.1 Virtual Autonomous Navigation Environment	1
1.1.2 Intel Embree.....	1
1.2 Objectives.....	2
1.3 Approach and Scope	3
2 Wrapper Library API.....	4
2.1 Interface Specifics	4
2.2 Unit Tests	5
2.3 Ray Tracing Samples	5
2.4 Rendering Engine	7
3 Backend Implementations.....	11
3.1 NVIDIA OptiX 7	11
3.1.1 OptiX Wrapper Overview	11
3.1.2 Optix Wrapper Implementation Details.....	12
3.1.2.1 Initialization	12
3.1.2.2 API Call: CommitGeometry	12
3.1.2.3 API Call: CreateInstance.....	12
3.1.2.4 API Call: TransformInstance.....	13
3.1.2.5 API Call: BuildScene	13
3.1.2.6 API Call: GetIntersections.....	13
3.2 Vulkan.....	14
3.2.1 Vulkan Ray Tracing	14
3.2.1.1 Initialization	14
3.2.1.2 Scene Building.....	15
3.2.1.3 Pipeline and Shaders	16
3.2.1.4 Intersections.....	17
3.2.2 VulkanWrapper Implementation Details.....	17
3.2.2.1 Initialization	18
3.2.2.2 API Call: CommitGeometry	18
3.2.2.3 API Call: CreateInstance.....	19
3.2.2.4 API Call: TransformInstance.....	19
3.2.2.5 API Call: BuildScene	19
3.2.2.6 API Call: GetIntersections.....	19
3.3 AMD Radeon Rays	20
3.3.1 Radeon Rays SDK Overview	20

3.3.2	Radeon Ray Tracing Backend	21
3.3.2.1	Initialization	21
3.3.2.2	Preparation for the Build on Device	23
3.3.2.3	Create Command Stream	24
3.3.2.4	Register Build Command	24
3.3.2.5	Submit Command Stream	25
3.3.2.6	Release Command Stream	25
3.4	Intel Embree.....	25
3.4.1	EmbreeWrapper Overview.....	26
3.4.2	EmbreeWrapper Implementation Details.....	26
4	Benchmarking.....	27
4.1	Method	27
4.2	Results.....	27
4.2.1	Geometry Acceleration Structures	27
4.2.2	Ray Tracing.....	28
5	Conclusion.....	32
	Bibliography	33
	Appendix : User Guide	35
A.1	Wrapper Compilation.....	35
A.2	API Reference	35
A.3	Running Supplementary Code	37
A.3.1	Tests	37
A.3.2	Benchmarking.....	37
A.3.3	Samples.....	38
	Report Documentation Page (SF 298)	39

Figures

Figures

1. Utilizing the library API to perform portable API calls.	5
2. Chess still life. It uses glass, metal, and gradient shaders to demonstrate how to use the wrapper API to implement multi-bounce ray casts efficiently.	6
3. Floating cubes example. It uses a multiplicity of instances of a single cube geometry to demonstrate how instancing works.	6
4. Grass example. Each blade of grass is constructed out of segments that are instances of a grass segment model. The full example dynamically transforms each segment to emulate the blades of grass waving back and forth.	7
5. The construction of the path-tracing tree. While most ray tracers use a depth-first approach to tree construction, a breadth-first approach results in the most efficient form of construction utilizing the generic ray tracing wrapper since rays are generated on the host at every bounce.	9
6. The backpropagation of colors from leaf nodes back to the root pixel samples.....	10
7. Time spent constructing a geometry acceleration structure (GAS) containing over 475,000 triangles. For OptiX, this includes time spent copying the geometry from the host to the device, which is one reason Embree outperforms OptiX in this instance.....	28
8. The final size of the GAS constructed for a scene after memory optimization has been performed.	29
9. A single ray is intersected with a single triangle. The lack of overhead required by Embree, along with caching advantages, make it by far the faster choice for performing single ray casts.....	29
10. A single ray is intersected with over 475,000 triangles. Embree still outperforms OptiX here for the reasons specified in Figure 9.	30
11. One million rays cast into a one-triangle GAS. Scaling the number of rays gives OptiX the clear advantage. With one million rays, the intersection speed advantage offered by the GPU outweighs the overhead.	30
12. One million rays cast into a 475,000-triangle GAS.	31

Preface

This study was conducted for the US Army Corps of Engineers under PE 0602145A, Project BG2, Task 2, “Military Engineering.”

The work was performed by the Edge Computing Lab of the Supercomputing Research Center, US Army Engineer Research and Development Center, Information Technology Laboratory (ITL). At the time of publication, Mr. Robert M. Hunter was chief of the Supercomputing Research Center, and Mr. Nicholas Boone was the technical director for Military Engineering. The deputy director of ITL was Dr. Jackie S. Pettway, and the director was Dr. David A. Horner.

COL Christian Patterson was commander of ERDC, and Dr. David W. Pittman was the director.

1 Introduction

1.1 Background

The high performance computing (HPC) world has transformed to heavily equipped graphic processing units (GPUs) from massive many-core CPUs, resulted from the workload change by artificial intelligence / machine learning (AI/ML applications). Ray tracing is full of graphics-rendering technology that produces photorealistic graphics with real-world lighting and shadow formations. However, ray tracing is computationally intensive. NVIDIA has been pushing ray tracing technology for over a decade since it acquired a ray tracing company called RayScale in 2008 and is working toward real-time ray tracing capability with GPU hardware support of ray tracing. AMD also has GPUs with hardware ray tracing support. In addition, Intel also provides laptop- and desktop-based GPUs with native support of hardware ray tracing.

1.1.1 Virtual Autonomous Navigation Environment

The Virtual Autonomous Navigation Environment (VANE) is a collection of computational tools for high-fidelity modeling and simulation of the coupled systems of vehicle, terrain, environment, and sensors. The VANE Environment and Sensor Engine (VANE::ESE) encapsulates the simulations of detailed 3D environments and a variety of sensor models, such as RGB, infrared (IR), or hyperspectral cameras; lidar detectors; and even GPS positioning sensors. In the VANE::ESE, sensor simulations are performed using a physics-based ray tracing algorithm in order to provide the highest accuracy. Carrillo et al. (2020) give a full description of the various sensor models available in the VANE::ESE. Additional details about the camera and lidar models are given in Goodin et al. (2009), with recent lidar developments in Goodin et al. (2018), and details for the GPS model appear in Goodin et al. (2010). Examples of full simulations have been published in Carrillo et al. (2016).

1.1.2 Intel Embree

Embree has been the ray tracing framework of choice for programs like VANE thanks to its ease of use and native CPU execution. VANE uses it to perform efficient scene building and ray casting for its synthetic sensor platforms. A Scene class encapsulates all of the ray casting operations

employed by VANE, and a subclass called EmbreeScene is used to implement the same operations with the Embree application programming interface (API), allowing it to be used interchangeably with the original Scene class. One of the advantages to this approach is the scalability. Although Embree is not as fast at performing batch intersections as a GPU ray tracing platform, Embree can be leveraged across multiple compute nodes on an HPC system to achieve a high level of performance at scale. Tests still need to be run to determine the magnitude of compute nodes required for Embree to match the performance of a GPU platform. The advantage of the framework described in this report is the ability to leverage Embree or a GPU platform with the exact same ray tracing code. If sufficient compute nodes are available to give Embree an edge over a single-node GPU device, Embree can be selected (at runtime if needed) as the ray tracing platform to use. Otherwise, a GPU wrapper can be selected for smaller clusters. Technically, thanks to runtime backend selection, each compute node of a problem set can employ a different backend on each node without using unique code on each node, allowing GPU compute nodes to be combined with standard compute nodes on a single message passing interface cluster.

1.2 Objectives

This work has been funded by the Military Engineering (ME) Simulation Tools for Vehicle Engineering and Research (STCoVER) since October 2021. We seek to enhance the VANE::ESE software to better leverage current heterogeneous hardware accelerators and prepare for future accelerators in both HPC and edge computing environments. VANE::ESE aims to realistically simulate sensors in complex outdoor environments used for autonomous unmanned ground vehicle (UGV) operations. These sensors include lidar, GPS, visible and near-infrared wave range, and automotive radar. From Goodin et al. (2017), a typical VANE::ESE analysis might simulate a 3 to 10 min autonomous mission featuring specific, sensor-dependent autonomous functionalities such as obstacle detection and avoidance (ODOA), terrain classification, or map generation. Thousands of images and millions of lidar points may be generated in such a simulation that could be run through within a day. Therefore, performance improvement toward feeding the simulated data/images to real-time kinematic positioning (or equivalent real sensor's response time such as shutter lag for a camera) is highly anticipated. We

propose a testable hypothesis: a VANE::ESE simulation can be improved to effectively leverage heterogeneous hardware accelerator architectures, such as GPUs, resulting in significant performance improvements and better time-to-solution on HPC systems and edge computing platforms over the current state of VANE::ESE. This work will seek to upgrade VANE::ESE to make effective use of current GPU accelerators—that is, no lock-in accelerator architecture, and future accelerated system, in addition to the CPU-only option.

1.3 Approach and Scope

As part of the upgrade process, we aim to develop a software test suite for VANE::ESE. Software testing methodologies encompass everything from front- to backend testing, which often includes unit and system testing to meet the quality assurance of the software product. The goal of the software testing is to ensure the software can successfully operate in multiple environmental scenarios across different platforms and that performance and correctness are retained during adaptation to new hardware platforms and addition of new features. We will set up the SOP and the test suite for code release criteria to ensure each part of software behaves as expected. The testing methods may include unit testing, integration testing, system testing, and acceptance testing, which all belong to the category of functional testing. Although there are four other test types in the category of nonfunctional testing methods, we will be mainly focused on functional testing.

2 Wrapper Library API

To facilitate general purpose ray tracing, a C++ API was developed that provided an interface for basic ray intersection methods. A client of the wrapper can use these methods without assuming any knowledge of the backend that they are implemented with. The ultimate goal of developing such a general purpose base wrapper library does not target specific applications such as VANE but all applications that require GPU ray tracing.

2.1 Interface Specifics

Simplicity and readability are major goals in the development of the raytracing wrapper. API methods are needed for

- copying geometry data to the device and building the acceleration structure;
- creating and transforming instances of geometry data; and
- copying rays to the device and performing ray intersection tests.

Converting these requirements into a C++ API leads to the following methods:

```
CommitGeometry(uint facet_indices , uint num_facets ,
               float *vertices , uint num_vertices ,
               uint geom_id)
CreateInstance(uint geom_model_id , uint instance_id)
TransformInstance(float x, float y, float z,
                  float x_rot , float y_rot ,
                  float z_rot , float x_scale ,
                  float y_scale , float z_scale ,
                  uint instance_id)

BuildScene ()
GetIntersections(float *ray_origins ,
                 float *ray_directions ,
                 uint num_rays ,
                 Intersection intersection)
```

To facilitate the `GetIntersection` method, a common `Intersection` structure is also required:

```
struct Intersection {
```

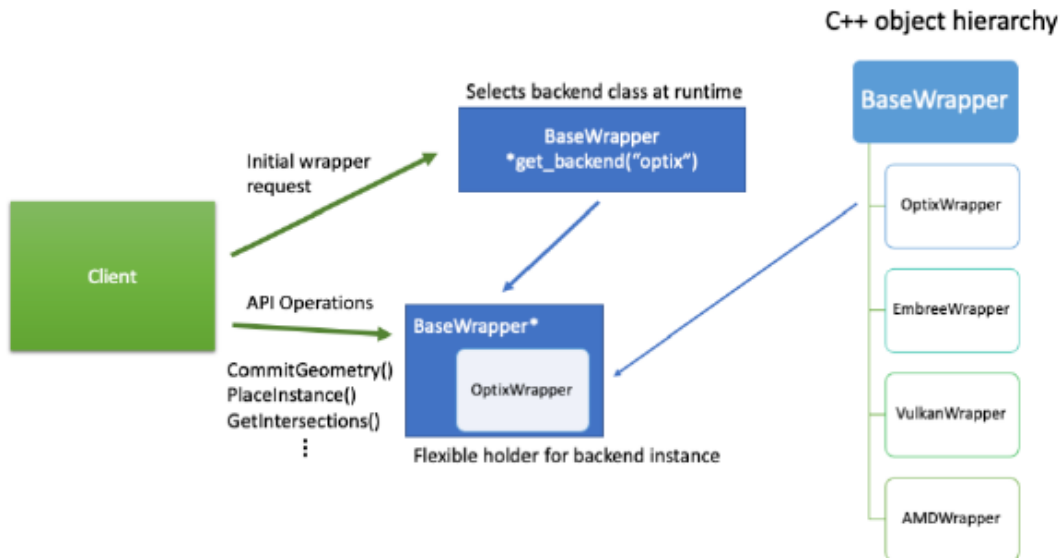
```

float distance;
uint geometry_id;
float normal_x , normal_y , normal_z;
}

```

The wrapper is written to be as flexible as possible. All backends implement the same API methods. The code even supplies a function that can be used to supply any backend at runtime, as shown in Figure 1.

Figure 1. Utilizing the library API to perform portable API calls.



2.2 Unit Tests

A suite of C++ unit tests has been implemented using the Google Test framework (Google 2022). These tests are used to ensure correct compilation and continuously validate the execution of available backend implementations. Specific tests include simple ray casts, dynamic transformation, and instancing calls.

2.3 Ray Tracing Samples

To provide a visual validation and demonstration of the effectiveness of the backend implementations of the raytracing wrapper, a simple rendering engine (with corresponding material shaders) was written using the wrapper API to construct scenes out of glass, metal, or solid objects. These scenes can be rendered into image files using any of the backends currently compiled into the working directory. Specific examples have been preconstructed using this rendering engine to demonstrate the

wrapper's ability to process multiple ray bounces, large amounts of instances, and dynamic transformations. Current examples representing these three capabilities include (respectively) a chess piece still life (Figure 2), floating cubes (Figure 3), and waving grass (Figure 4).

Figure 2. Chess still life. It uses glass, metal, and gradient shaders to demonstrate how to use the wrapper API to implement multi-bounce ray casts efficiently.

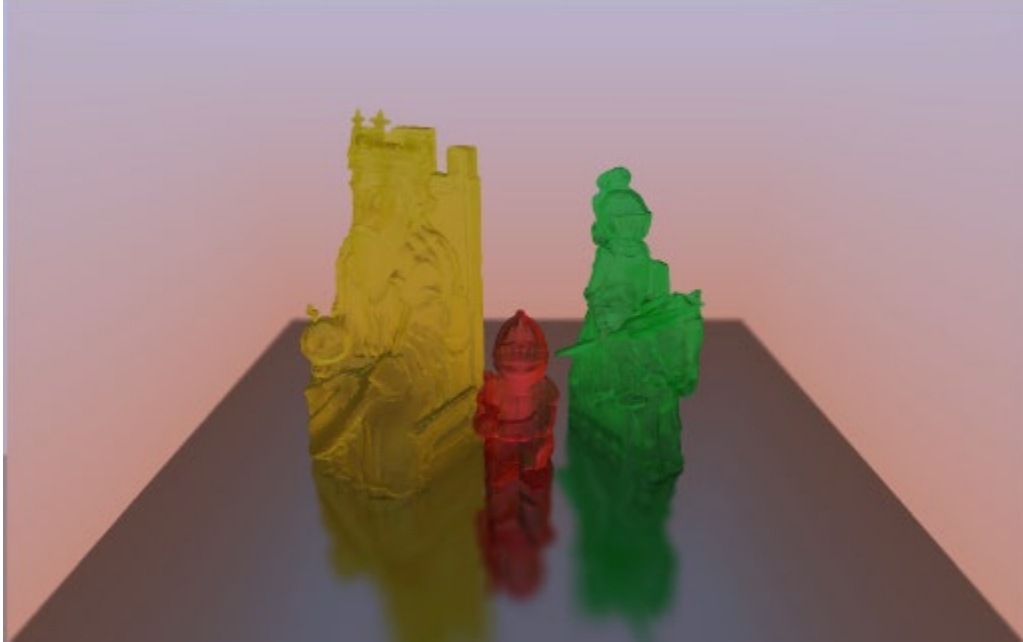
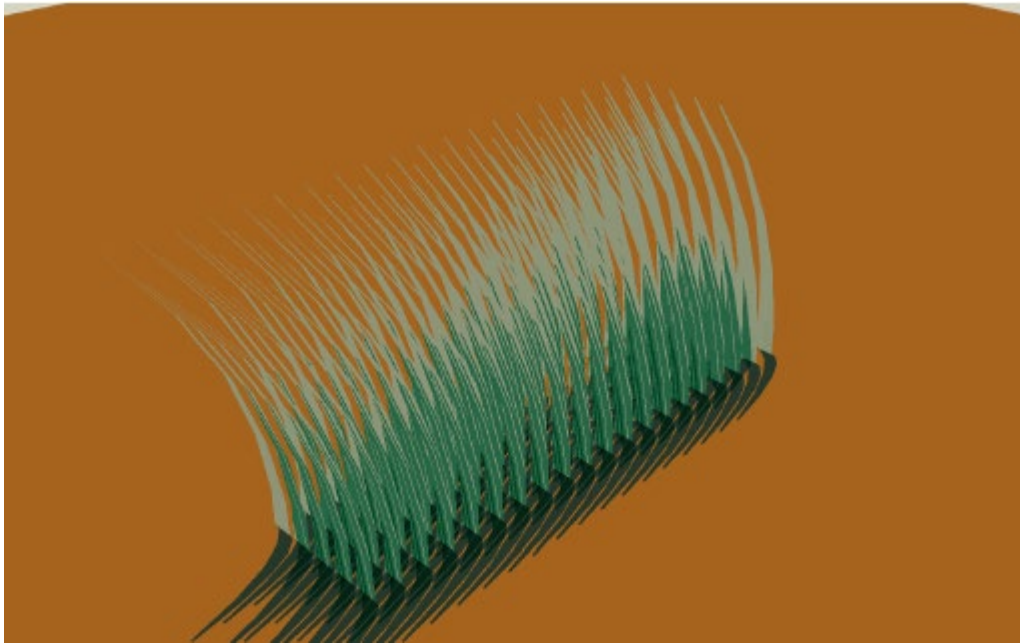


Figure 3. Floating cubes example. It uses a multiplicity of instances of a single cube geometry to demonstrate how instancing works.



Figure 4. Grass example. Each blade of grass is constructed out of segments that are instances of a grass segment model. The full example dynamically transforms each segment to emulate the blades of grass waving back and forth.



2.4 Rendering Engine

While it is possible to perform a single ray intersection with the ray tracing wrapper's `GetIntersection` function, the performance advantage of using the wrapper increases with the number of rays passed into the function due to the input/output (I/O) overhead associated with copying the rays to the backend device. Thus, while most renderers programmed in device code utilize a tree of ray generation calls where each shader can generate and launch single reflective rays (a kind of depth-first path tracing), the rendering engine used for these examples takes a different approach. A form of breadth-first path tracing is used where the new rays generated from each bounce are bundled into a buffer as they are created and then all launched in one call of `GetIntersections`. This requires some slightly more complicated bookkeeping to construct the path tracing tree, but it minimizes the overhead spent copying rays and results to and from the device. The tracing loop looks something like this:

```
Call GetIntersections on initial N camera rays
```

```
Save hit results
```

```
Create empty path tree with N parent  
nodes while there are more than 0 ray  
hit results do
```

```
    for each ray hit result
```

```
        call shader code to generate  
        bounce/shadow ray save bounce ray to  
        buffer for next ray cast link ray to its  
        parent in path tree  
  
    end  
    Call GetIntersections on new bounce rays Gather hit results  
  
end
```

The result of this code is a creation of a path tracing tree structure as shown in Figure 5.

Rendering the image is then accomplished by starting at the leaves of the path tree and backpropagating colors through to the root samples. At each node, the color method of the appropriate shader is called with the current colors of that node's children as its parameters. This allows the shader to decide what to do with the color results of its bounce ray(s) (i.e., attenuation, occlusion, mixing, etc.). Figure 6 shows an example of this process for the tree constructed in Figure 5.

Figure 5. The construction of the path-tracing tree. While most ray tracers use a depth-first approach to tree construction, a breadth-first approach results in the most efficient form of construction utilizing the generic ray tracing wrapper since rays are generated on the host at every bounce.

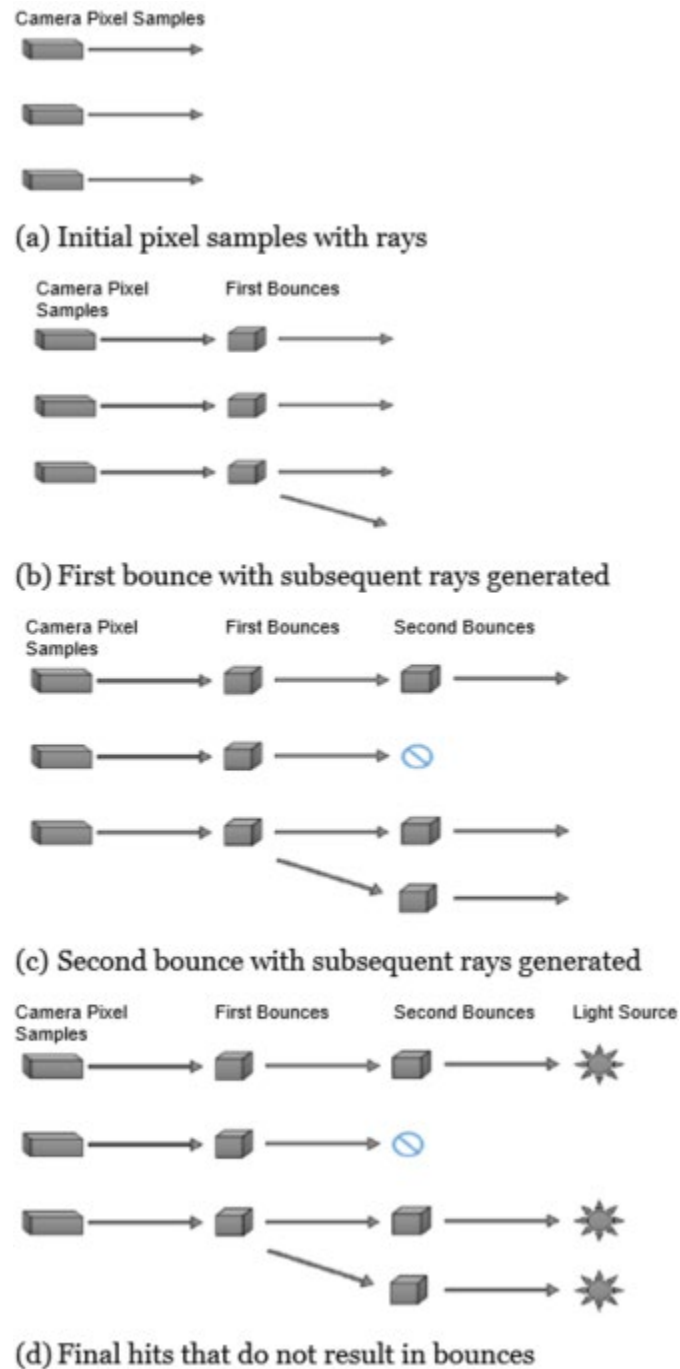
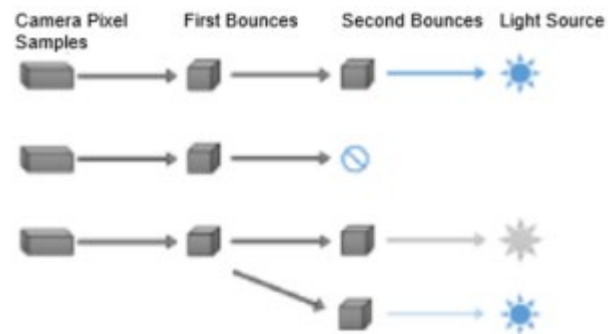
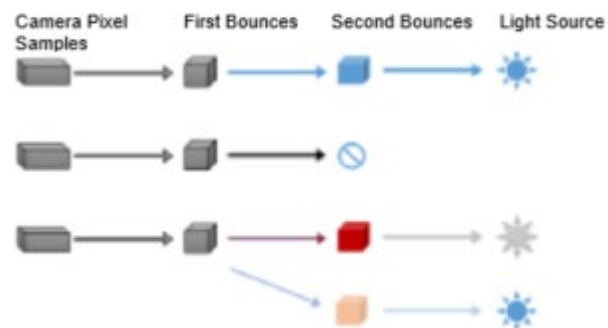


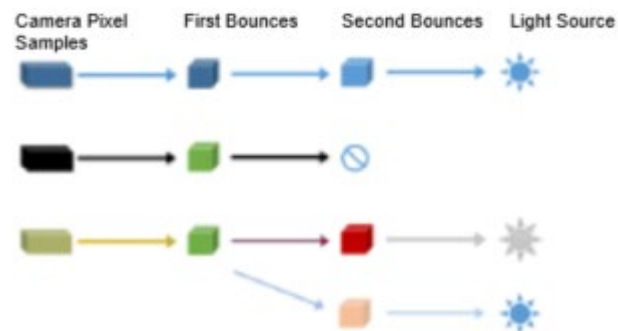
Figure 6. The backpropagation of colors from leaf nodes back to the root pixel samples.



(a) Resolving colors at leaf nodes



(b) Shaders handling the backpropagation of colors



(c) Final colors result at pixel samples

3 Backend Implementations

This chapter describes implementation details for the various ray tracing backend libraries used to fulfill the top level API described in Section 2.1. Currently, four backends are being developed: three based on GPU-accelerated libraries and a fourth for multithreaded CPUs. The NVIDIA OptiX backend (Section 3.1) supports a range of NVIDIA GPUs for acceleration based on the OptiX 7 ray tracing library. The Vulkan backend (Section 3.2) is built around the Vulkan general purpose accelerated 3D graphics library, using its ray tracing extension, and supports GPUs from NVIDIA, AMD, and Intel. The AMD Radeon Rays backend (Section 3.3) is also cross-platform and is built on Radeon Rays, a ray tracing library that provides an interface to both Vulkan and DirectX 12 ray tracing capabilities. Finally, the last backend is based on the Intel Embree ray tracing library, which efficiently leverages single instruction, multiple data parallelism on CPUs (Section 3.4).

3.1 NVIDIA OptiX 7

NVIDIA is one of the leading producers of computer graphics hardware and software. NVIDIA graphics cards run kernels programmed in CUDA, a low-level C language with built-in device macros. As ray tracing grew in prominence for computer graphics applications, NVIDIA released OptiX as a library that implements common ray tracing operations in CUDA for their graphics cards (Parker et al. 2010), particularly the new RTX series built for real-time ray tracing. At the time of writing, OptiX 7 is the latest major version that NVIDIA has released (Morley 2019).

3.1.1 OptiX Wrapper Overview

OptiX, like other GPU libraries, divides code up into two categories: *host* code and *device* code. Host code operates on the CPU and drives the data pipeline to the GPU device in addition to dictating when and how the device code runs. Device code is written by the user solely to handle how the GPU handles different ray events. The host code sends commands to the device through a CUDA *stream*. Operations can be performed on this stream synchronously or asynchronously from the host perspective. The OptiX backend implemented for the general-purpose ray tracing wrapper described in this report uses asynchronous stream methods whenever possible to increase the concurrency between host API calls and device

processing. Geometry data is stored on the host and pushed to the device through acceleration structure build calls when `BuildScene` is called.

3.1.2 Optix Wrapper Implementation Details

3.1.2.1 Initialization

The OptiX wrapper uses the initialization phase to set up the boilerplate required by OptiX. A CUDA stream is created for all device commands, OptiX contexts are created for each connected NVIDIA device, and all compiled device code is loaded into a pipeline and copied to the device(s) in the form of a shader binding table. All the terms emphasized here have specific meanings for OptiX that are beyond the scope of this report to discuss, but they should provide an avenue for further research if desired.

3.1.2.2 API Call: *CommitGeometry*

The OptiX wrapper lazily creates geometry models on the device. In other words, the `CommitGeometry` implementation sets up host buffers for the geometry data but doesn't do any actual copying to the device or geometry acceleration structure (GAS) building. Those operations are handled later when the user calls `BuildScene`.

3.1.2.3 API Call: *CreateInstance*

When the user creates a new geometry model with `CommitGeometry`, the wrapper doesn't actually add an object to the scene yet. To actually push geometry into the scene where rays will intersect with it, the `CreateInstance` method must be called. This method creates a single instance of the geometry model to place at a specific location and orientation in the scene. OptiX, like most other ray tracing libraries, provides an instance acceleration structure (AS) to facilitate this operation. The OptiX wrapper creates an instance acceleration structure that points to the GAS created for the geometry model being instanced. However, just as in `CommitGeometry`, these acceleration structure operations are performed lazily; `CreateInstance` merely records the instance data on the host and waits for `BuildScene` to be called before copying the data to the device and building the instance AS.

3.1.2.4 API Call: *TransformInstance*

`CreateInstance` by itself does not actually place an instance at a specific location in a scene. The `TransformInstance` call translates, rotates, and scales a specific instance by parameters specified by a user. Internally, it generates a 3×4 matrix and attaches it to the instance AS built by `CreateInstance`. This tells OptiX exactly what transformations apply to rays before performing intersection tests for a specific geometry object.

3.1.2.5 API Call: *BuildScene*

As indicated in the above sections, the `BuildScene` operation is where most of the geometry processing takes place in the OptiX wrapper. It starts by building a GAS for each geometry model that has been committed. It follows this by building the instance acceleration structure for every instance of those models placed in the scene. If this is not the first time `BuildScene` has been called, then it will only perform these actions for any models or instances that have been modified since the last call to `BuildScene`.

3.1.2.6 API Call: *GetIntersections*

`GetIntersections()` fulfills the purpose for which the wrapper exists: performing ray queries. The method takes a list of ray origins and ray directions, copies them to the devices, and initializes the OptiX ray tracing pipeline with a call to `optixLaunch`. This transfers processing over to the device, where the shader programs provided by the wrapper kick in. There are three shader programs used by OptiX to process ray casts: ray generation, miss, and closest hit. Ray generation for the OptiX wrapper simply involves tracing a ray pulled from the device buffer holding all of the ray origins and directions. The specific ray that is traced depends on the block and grid index of the kernel in the OptiX launch matrix. The miss shader simply reports a negative distance value to the intersection structure so the host code can record a ray miss. The closest hit shader uses OptiX device functions to pull geometry information from the primitive that was intersected so that a normal vector can be computed, in addition to querying the ID of the geometry instance and the distance to the hit. All of this information is passed back to the host through the intersection data structure. OptiX collates these intersection data structures into a device buffer (the address of which is recorded on the host), allowing the host code to take over and copy these results out of the

device before handing them over to the user. If multiple GPUs are present and have been selected (an option that the user can specify in the constructor), then `GetIntersections()` will evenly split the ray buffer between the available GPU devices and attempt to find intersections in parallel.

3.2 Vulkan

Vulkan is a cross-platform 3D graphics API published by the Khronos Group (2022) that also maintains a number of open, cross-platform GPU APIs, such as OpenGL (a high-level graphics API), OpenCL (a parallel computing API), and SYCL (a programming platform for heterogeneous systems). Vulkan is a very large API targeted at high-performance real-time rendering of graphics for video gaming or visualization, first announced in 2015. Initially, Vulkan supported only rasterization, which has been the dominant method for producing interactive graphics for many years, but as NVIDIA, AMD, and now Intel have released GPUs with dedicated ray tracing hardware, Vulkan finalized a set of ray tracing extensions in November 2020 (Koch 2020).

3.2.1 Vulkan Ray Tracing

Before describing the details of our Vulkan backend implementation, we will use this section to provide some background details of the Vulkan ray tracing extension and how accelerated ray tracing can be achieved using Vulkan. First, we will describe how to initialize the Vulkan library (which is a general process for all Vulkan-accelerated programs, not just to ray tracing), then we will cover scene construction, creating the Vulkan ray tracing pipeline, and finally, how to execute the pipeline to compute ray intersections.

3.2.1.1 Initialization

Initializing Vulkan for use in an application requires creating a Vulkan context, initializing one or more devices (GPUs) for use, and setting up execution queues linked to a device.

- **Vulkan Context:** The first step in initializing Vulkan in an application is to create a `VkInstance` object, which we will call a Vulkan context due to the use of instances in scene building. Building the Vulkan context makes the connection to the Vulkan shared library and loads the

- system GPU drivers that will ultimately satisfy most Vulkan API calls. It also checks for and loads the Vulkan extensions required by the application (such as the ray tracing extension) and sets up validation layers for debugging purposes.
- **Vulkan Device:** The second step involves selecting a physical device (a GPU) to use for Vulkan API calls and generating a logical device handle for it. We query the Vulkan context for available GPUs and features available on those devices, comparing against application requirements, such as supporting the ray tracing extension or parallel execution queue counts, in order to select the appropriate device for the application. Then, upon selecting a physical device, we create a logical device handle (a `VkDevice`) with which to interact with that GPU. When creating the logical device, we also enable ray tracing extension features for the device, as well as specify the type and quantity of parallel execution queues needed by the application.
 - **Execution Queues:** After building the `VkDevice` handle, we need to initialize the execution queues for that device. Each `VkDevice` handle is used to build one or more `VkQueue` objects, to which tasks can be submitted. In Vulkan, any task that needs to be performed on a GPU must be submitted as a command buffer to an execution queue, and the more queues that are available, the more GPU tasks can be added in parallel. Essentially all GPU operations in Vulkan follow a two-stage process: prepare a command buffer, fill it with tasks called commands (i.e., memory transfer, build acceleration structure, trace rays, etc.), then submit the command buffer to an execution queue for asynchronous processing on the GPU and await the result.

With the Vulkan context, logical device(s), and associated execution queues all initialized, we can construct and prepare the 3D scene for ray tracing.

3.2.1.2 Scene Building

Building a scene for ray tracing involves creating and uploading geometries to the device, creating and transforming instances of those geometries, and building acceleration structures encapsulating those scene components.

- **Geometries:** A geometry is represented by a vertex buffer and an index buffer, containing the 3D coordinates of the triangles making up the object. These buffers start as simple arrays on the host and are then

- copied to device buffers via a simple transfer command submitted to an execution queue.
- **Instances:** An instance is a kind of reference or handle for a geometry (and/or a bottom-level acceleration structure made from a geometry) associated with some spatial transformation (i.e., a change in size, rotation, or position). This allows for lightweight reuse of a geometry in the scene, potentially providing significant memory savings.
 - **Acceleration Structures:** A ray tracing AS is a data structure describing a geometry (or an entire scene with multiple geometries) in a format optimized for ray traversal. In Vulkan, a scene is described by two tiers of acceleration structures: bottom-level acceleration structures (BLAS) containing geometries, and a top-level acceleration structure (TLAS) containing instances that refer to those BLAS. Building an acceleration structure involves gathering references to the input data (vertex and index buffers for BLAS, transforms and BLAS for TLAS), allocating a buffer for the AS on the GPU and submitting a compute command to an execution queue for building the AS on the device.

Once the TLAS is built, encapsulating all the components of the 3D scene, the scene is ready to accept ray traversal requests.

3.2.1.3 Pipeline and Shaders

Before we can start tracing rays through the scene, we need to prepare the commands needed to do so. This requires loading shader programs and building the ray tracing pipeline, creating the shader binding table, and lastly recording the pipeline in a command buffer for execution.

- **Shaders:** A shader (or shader program) is a customizable compute kernel that is executed in parallel on the GPU. For Vulkan, shaders are imported as SPIR-V (Khronos Group 2021) bytecode at runtime and compiled into a shader module. For ray tracing, three types of shaders are required:
 - **Ray Generation:** The ray generation (raygen) shader does any setup needed to generate an initial ray and uses a handle to the TLAS to call a `TraceRays()` primitive, which begins ray tracing for that ray.
 - **Miss:** The miss shader is automatically called for rays that miss all objects in the scene and performs any finalizing operations for that ray.

- Closest-Hit: The closest-hit shader is automatically called for rays that hit objects in the scene, with a built-in variable containing the index of the object (and associated triangle) that was hit, so that handling of the ray intersection can be done differently on an object-by-object basis.
- Descriptors: Descriptors act as parameterized references to external resources required by the shader programs. For example, the raygen shader needs a reference to a TLAS to begin tracing rays, and the miss and closest-hit shaders need output buffers in which to record tracing results. So descriptors are created for these resources and are dynamically attached to the shaders before execution, allowing precompiled shader programs to be reused with different scenes and output buffers.
- Pipeline: The ray tracing pipeline holds everything together: it binds together a descriptor layout with a set of shaders to define the host side interface to the ray tracing process.
- Shader Binding Table: The shader binding table is a device memory buffer that contains handles to each of the shader programs associated with each geometry in the scene (as for complex scenes, different shaders may be needed for different objects), along with any extra data needed for each shader (such as pointers to the vertex and index buffer for each geometry, as needed for many shading calculations).

Once the Vulkan ray tracing pipeline and shader binding table have been built, we can begin to compute intersections.

3.2.1.4 *Intersections*

With a completed scene built into a TLAS and a pipeline containing all the required shaders, we can finally begin the ray tracing process on the device. As before when submitting tasks to the device, we build a command buffer, first binding the pipeline, then the descriptors for the TLAS and other input or output buffers, finally adding the Vulkan command to trace rays. After filling the command buffer, we submit it to an execution queue and wait for the results computed by the shaders.

3.2.2 VulkanWrapper Implementation Details

This sections describes some of the details of our Vulkan backend for the wrapper API described in Section 2.1.

3.2.2.1 Initialization

During initialization of the Vulkan backend, we perform the steps described in Section 3.2.1, building the Vulkan context, selecting one or more GPU devices, and building a set of execution queues. Building the context, we check whether the system has a version of the Vulkan library (and GPU drivers) that support the `VK_KHR_ray_tracing_pipeline` extension, along with other required extensions, and choose whether to enable validation layers for debugging. For device selection, we again check for ray tracing support on each device as well as the number and type of available execution queues. Our support of parallel threading is limited by the number of queues available on the device, so if we know the number of threads that will be calling the API, we can select a device with an appropriate number of queues. Once the device is selected, we initialize its execution queues. Alongside the execution queues, we also build some infrastructure to simplify threading, ensuring only a single thread can submit to a queue at any point in time, along with a simple round-robin scheduling setup so that queues can be distributed to threads without any API user intervention.

As part of the initialization, we also set up some host-side structures for containing scene elements to allow for fast and thread-safe adding and modifying of geometries. We use unordered hash maps to map geometry and instance IDs to geometries, their BLASs, and instances, along with locks to enable multiple threads to build these structures independently.

The initialization process also involves compiling the shader programs and, if possible, prebuilding the ray tracing pipeline (which will significantly improve the performance compared to building it in `GetIntersections`) and preallocating ray input buffers and intersection output buffers. If the number of concurrent calling threads is already known, we can build the pipelines and allocate command buffers for that many threads. If we know the thread count and the maximum ray count for intersections, we can also pre-allocate a set of input and output buffers along with their descriptors for later calls to `GetIntersections`.

3.2.2.2 API Call: `CommitGeometry`

The `CommitGeometry` call adds a new geometry to the scene, or updates a geometry if the given geometry ID already exists. For the Vulkan backend, this operation creates and fills device buffers for the vertex and index

buffers and builds the BLAS for the geometry. If a geometry is updated, it copies the new vertices into the existing vertex buffer (the number and order of the new vertices must match) and updates the associated BLAS. The entire process is thread-safe, with insertion of the new geometry and BLAS structures into their host-side maps protected by locks.

3.2.2.3 API Call: *CreateInstance*

The `CreateInstance` call adds an instance of an existing geometry to the scene, initialized with an identity transform. This call is thread-safe, as insertion of the new instance into the scene map is protected by a lock.

3.2.2.4 API Call: *TransformInstance*

The `TransformInstance` call updates an existing instance with a new transformation matrix. This marks that instance as updated to ensure the instance is buffered to the device before the TLAS is built. This call is thread-safe without locks so long as no other threads access the same instance concurrently.

3.2.2.5 API Call: *BuildScene*

The `BuildScene` call buffers all new or updated instances to the GPU device and builds the TLAS from the instances and BLASs. It also creates a descriptor for the TLAS, completing the scene building process.

Additionally, `BuildScene` constructs the shader binding table for the scene, inserting pointers to each geometry's vertex and index buffers alongside the handle to the closest-hit shader.

This call is not thread-safe and should be called by only a single thread for each scene.

3.2.2.6 API Call: *GetIntersections*

The call to `GetIntersections` actually sends rays into the scene and computes intersections. If the ray tracing pipelines, input buffers, and output buffers have been prebuilt, we select a free pipeline, binding the scene and free I/O buffer descriptors, and submit the pipeline to a free execution queue to wait for the intersections. Then, we copy the intersections from the output buffer to return to the caller.

If the pipeline and buffers could not be prebuilt in the initialization step, we instead build the pipeline here, allocate ray input buffers and an intersection output buffer, build descriptors for the I/O buffers, and allocate and record the command buffer. Then, we can submit to an execution queue and wait for intersections to be copied out.

This call is thread-safe after `BuildScene` has been called.

3.3 AMD Radeon Rays

AMD Radeon Graphics Rays, currently 4.1, was released as an open source software on the GPUOpen site (AMD 2021b). From their release announcement, we know that AMD developed Radeon Rays, a high-efficiency, high-performance ray intersection acceleration library for DirectX 12 and Vulkan, to help developers get the most out of AMD GPUs, though it is not limited to AMD hardware or a specific operating system. The lightweight library has a set of well-defined C API for scene construction and asynchronous intersection queries and supports the AMD C++ parallel computing platform. That means they also support Heterogeneous-Computing Inference for Portability. Features of the Radeon Rays library include custom axis-aligned bounding box, bounding volume hierarchy optimization, and a logging mechanism for easier debugging.

3.3.1 Radeon Rays SDK Overview

Radeon Rays, like other GPU libraries, embeds similar concepts implemented in Optix and Vulkan, such as dividing code into two categories—host code and device code—where the host code operates on the CPU and drives the data pipeline to the GPU device whenever required and the device code is to handle how the GPU handles different ray events. The host code sends commands to the device through an `rrCommandStream` via four steps: (1) create a command stream to build an acceleration structure, (2) register a build command, (3) submit the command stream and wait until completion, and (4) release the command stream. Geometry data are stored on the host and pushed to the device through acceleration structure build calls when `BuildScene` is called.

3.3.2 Radeon Ray Tracing Backend

3.3.2.1 Initialization

The Radeon wrapper uses the initialization phase to set up the boilerplate required by Radeon Rays. Radeon Rays' *context* is created for each connected GPU device, along with the `build_flags` set to something like `RR_BUILD_FLAG_BITS_PREFER_FAST_BUILD`. In addition to the two vectors `std::vector<RRDevicePtr> geometry_ptrs` and `std::vector<RRInstance> instances` created in the `RadeonWrapper` class to hold all loaded geometry objects and instances, respectively, the `RRDevicePtr scene_ptr` holding the entire scene is initialized here. Detailed steps are listed in the following.

```
// Create context
RRError ret_code;
for (int i=0; i<use_devices; i++) {
    RadeonWrapper::Device device;
    device.idx = i;
    device.context = nullptr;
    ret_code = rrCreateContext( RR_API_VERSION , RR_API_VK , &
context); std::cout << " rrCreateContext: " << ret_code <<
std::endl; devices.push_back(device);
}
options.build_flags = RR_BUILD_FLAG_BITS_PREFER_FAST_BUILD;
// initialize the scene pointer
scene_ptr = nullptr;
```

Note that the related data structures are already defined as follows.

```
std::vector <Device > devices;

struct Device
{ int idx;

    RRContext context;
    std::vector<DeviceGeometry >
models;

};

RRBuildOptions options;
```

Additionally, RadeonRays supports the basic data types including `RRBuildOptions`, `RRCommandStream`, `RRContext`, `RRDevicePtr`, `RREvent`, and

RRRayMask, as well as a set of API with the convention prefixed by `rr`, e.g., `rrCreateContext` (AMD 2021a).

All the remaining five APIs—`CommitGeometry`, `CreateInstance`, `TransformInstance`, `BuildScene`, and `GetIntersections`—follow the backend design described in Section 3.1.2. Therefore, the backend construct for Vulkan is described based on the four steps—create, register, commit, and release—of the command stream.

Prior to sending commands from the host to the device to build a geometry object or to build the scene that could be composed of many instances by transforming geometry objects, we first need to allocate device buffers to store data (e.g., the geometry object data), followed by creating the `RRGeometryBuildInput` object and allocating device buffers for builders and resulting geometry on GPU devices. The workflow is demonstrated as follows.

```
do // geometry build on device
  for each geometry object load mesh data
    geom_ptr = Geometry build on devices
    geometry_ptrs. push_back( geom_ptr);
  end for
end do
do // create instances for later scene build on
  devices for each geom_ptr popped from
    geometry_ptrs
      Create <RRInstance > instance
      set instance. geometry = geom_ptr
      copy transformation matrix to instance.
      transform instances. push_back( instance)
    end for
  end do
do

  Scene Build on
  devices end do

  Create rays

  Trace rays on devices
```

3.3.2.2 Preparation for the Build on Device

The host reads in mesh data for a geometry object, say `geom`. The following lists the preparation process for building geometry acceleration structure, instance acceleration structure, and efficient ray queries.

```
RRDevicePtr vertex_ptr = nullptr , index_ptr = nullptr;
// allocate device buffer for vertices and triangles
rrAllocateDeviceBuffer(context , num_vertices*sizeof(float3 ) ,
                        & vertex_ptr);
rrAllocateDeviceBuffer(context , num_facets*sizeof(
uint32_t)*3 ,
                        & index_ptr);

// copy vertex data to device
void* ptr = nullptr; // ptr holder for mapping and unmapping
rrMapDevicePtr(context , vertex_ptr , & ptr); // map device
vertex_ptr memcpy(vtx_ptr , geom ->vertices , sizeof(float3 )*
num_vertices); rrUnmapDevicePtr(context , vertex_ptr , & ptr); //
unmap device

// copy facets (i.e., indices) data to device
rrMapDevicePtr(context , index_ptr , & ptr); // map device
index_ptr memcpy(index_ptr , geom ->facets ,
sizeof(uint32_t)*3* num_facets); rrUnmapDevicePtr(context ,
index_ptr , & ptr); // unmap device

// Create RRGGeometryBuildInput object
RRGeometryBuildInput geom_build_input = {}
RRTriangleMeshPrimitive mesh = {}
geom_build_input. triangle_mesh_primitives = & mesh;
geom_build_input. primitive_type =
RR_PRIMITIVE_TYPE_TRIANGLE_MESH; mesh. vertices = vertex_ptr;
mesh. vertex_count = num_vertices;
mesh. vertex_stride = 3*
sizeof(float); mesh.
triangle_indices = index_ptr; mesh.
triangle_count = num_facets; mesh.
index_type = RR_INDEX_TYPE_UINT32 ;
geom_build_input. primitive_count =
1u;

// Calculate memory requirement for the geometry build
RRMemoryRequirements geom_reqs;
rrGetGeometryBuildMemoryRequirements(context , &
geom_build_input ,
                                & options , & geom_reqs);

// allocate buffers for builders and resulting geometry
RRDevicePtr scratch_ptr , geom_ptr;
rrAllocateDeviceBuffer(context , geom_reqs. temporary_buffer_size
,
                                & scratch_ptr);
rrAllocateDeviceBuffer(context , geom_reqs.
result_buffer_size ,
                                & geom_ptr);
```

The steps prior to building the scene are:

```
RRSceneBuildInput scene_build_input = {};
// each scene was already pushed to instances
scene_build_input. instances = instances.data();
scene_build_input. instance_count = ( uint32_t)
instances.size();

// Calculate memory requirement for the scene build
RRMemoryRequirements scene_reqs;
rrGetSceneBuildMemoryRequirements(context , &
scene_build_input ,

                                & options , & scene_reqs);
// allocate buffers for builder and resulting geometry
RRDevicePtr scratch_scene_ptr = nullptr;
rrAllocateDeviceBuffer(context ,
    scene_reqs. temporary_build_buffer_size , &
scratch_scene_ptr); rrAllocateDeviceBuffer(context , scene_reqs.
result_buffer_size ,

                                & scene_ptr);
```

and those prior to tracing rays are

```
RRDevicePtr rays_ptr , hits_ptr , scratch_trace_ptr;
rrAllocateDeviceBuffer(context , rays.size()* sizeof( Ray), &
rays_ptr);
\\ copy rays to device
rrMapDevicePtr(context , rays_ptr ,
& ptr);
memcpy(ptr, rays.data(), sizeof(Ray)* rays.size());
rrUnmapDevicePtr(context , rays_ptr , & ptr);
```

3.3.2.3 Create Command Stream

After preparing for building the acceleration structure (e.g., for the geometry build), we then create a command stream for the execution on the device. It is simply an API function call.

```
RRCommandStream command_stream = nullptr;
rrAllocateCommandStream(context , & command stream);
```

3.3.2.4 Register Build Command

Next, we get to register the build command—for example, the `rrCmdBuildGeometry` API is to build a geometry object, `rrCmdBuildScene` to build the scene, and `rrCmdIntersect` to compute ray tracing. For building the geometry, the code is shown in the following.

```
rrCmdBuildGeometry(context , RR_BUILD_OPERATION_BUILD ,
                  & geom_build_input , & options ,
                  scratch_ptr , geom_ptr , command_stream);
geometry_ptrs. push_back( geom_ptr); // collect the device ptr
```

The other two APIs for registering the build command are specified as follows.

```
rrCmdBuildScene(context , & scene_build_input , & options ,
                scratch_scene_ptr , scene_ptr ,
                command_stream_scene); rrCmdIntersect(context , scene_ptr ,
                RR_INTERSECT_QUERY_CLOSEST ,
                rays_ptr , num_rays , nullptr
                ,
                RR_INTERSECT_QUERY_OUTPUT_FULL
                _HIT ,
                hits_ptr , scratch_trace_ptr , command_stream_trace)
```

3.3.2.5 Submit Command Stream

Once the command stream is created followed by the registration of the build command, we can now commit it and wait for its completion, shown as follows.

```
// rr commit the command stream and wait for the event to
complete RREvent wait_event = nullptr;
rrSubmitCommandStream(context , command_stream , nullptr , &
wait_event);
rrWaitEvent(context , wait_event);
```

3.3.2.6 Release Command Stream

Finally, we need to release the command stream and the event.

```
// rr release
rrReleaseEvent(context , wait_event );
rrReleaseCommandStream(context , command_stream );
```

3.4 Intel Embree

Thus far, each of the backend wrappers that have been discussed were designed to utilize dedicated graphics devices connected to the CPU. However, what if we are not connected to a graphics card? Is it impossible to perform efficient ray tracing on a general purpose CPU? Thankfully, Intel introduced Embree to provide this exact service. Intel Embree is an open-source kernel framework designed to perform rapid ray tracing on

regular x86 CPUs (Wald et al. 2014). It provides a cross-platform API written in C++ offering a set of methods for constructing acceleration structures and solving ray intersections or occlusions. The EmbreeWrapper class implements the basic ray casting methods described in this report using the Embree API, providing users the ability to run their ray tracing codes on CPUs when a supported graphics card is not available.

3.4.1 EmbreeWrapper Overview

Compared to the other GPU backends discussed in this report, Intel Embree is remarkably simple to use. Since there is no host/device distinction in Embree, there is no dealing with device buffers or command streams. Embree does utilize a device concept but mainly as a container for all the scene data and operations.

Using the EmbreeWrapper also doesn't require any special device drivers or software installation. The Embree library can be "installed" merely by downloading the source code and running an included shell script that sets up a few necessary environment variables.

3.4.2 EmbreeWrapper Implementation Details

The implementation of the EmbreeWrapper is fairly straightforward. When a geometry model is committed, the primitive data are bundled into an Embree geometry object, which is then attached to a scene object. As instances of geometry models are placed in the scene, new Embree instance geometry objects are created, their respective model "scenes" (each representing the model's GAS) are attached to the instance, and the instance geometry is subsequently attached to a global "scene" object representing the root acceleration structure.

Retrieving intersections merely involves converting ray origin and direction values into Embree's RayHit data structure and calling Embree's own `GetIntersection` method. The results are then reinterpreted back into the global `Intersection` structure recognized by all the backend wrappers.

4 Benchmarking

One of the motivations for this work is to obtain comparisons of different ray casting backends on identical problems. A common API with various backend implementations makes accomplishing this at a high level very straightforward. Benchmarking tests that generate timing results for a user-specified list of backends were incorporated into the wrapper code base. Thus, the results shown below can be generated by a user to get device specific timing comparisons.

4.1 Method

Results described below were performed on a Supermicro system named rCuda. The Supermicro SYS-4029GP-TRT3 SuperServer has two Intel Xeon Gold 5220 2.2 GHz 18-core processors on the dual socket LGA-3647. Each processor can boot up to 36 threads, 3.90 GHz maximum turbo frequency, and has 24.75 MB cache. In addition, the system has four PNY RTX5KNVLINKX8S2RKIT RTX 5000, an NVLINK Quadro 2W2S Kit used to provide an NVLink connector for seven NVIDIA RTX 5000 graphic cards—each with 16 GB memory—by fusing two RTX 5000 boards via each kit. rCuda is equipped with twelve Samsung M393A4K40CB2-CTD 32 GB 2666 MHz PC4-21300 DDR4—in total 384 GB memory. The 2 TB solid state drive has ultra-fast PCIe NVMe Gen3x4 interface supported by the motherboard's M.2 slots.

4.2 Results

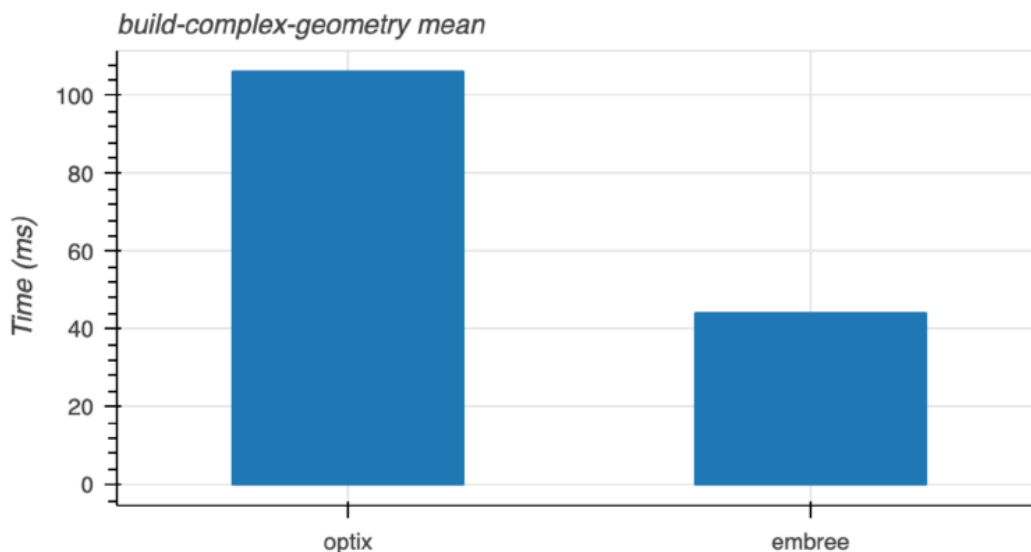
Preliminary benchmarking results have been generated using the Embree and OptiX backends. This provides a good point of comparison between a CPU-based backend and a GPU-based backend. The advantages and disadvantages of these two approaches can be seen clearly even in these high-level results.

4.2.1 Geometry Acceleration Structures

Figure 7 shows the average time required to copy and build an acceleration structure for a geometry of over 475,000 triangles. Because Embree does not have to copy geometry buffers to a separate device, it outperforms OptiX by a significant amount. However, OptiX is by no means slow at producing the acceleration structure. It will be insightful to see how OptiX

compares to other GPU backends like Vulkan and AMD Radeon once they are ready for benchmarking.

Figure 7. Time spent constructing a geometry acceleration structure (GAS) containing over 475,000 triangles. For OptiX, this includes time spent copying the geometry from the host to the device, which is one reason Embree outperforms OptiX in this instance.



Another point of comparison for geometry handling is the size of the finished geometry acceleration structure. This could be a useful metric if a specific problem requires very large geometries with limited memory resources. Both OptiX and Embree offer the capability to optimize the memory requirements of a specific acceleration structure with additional build time, essentially trading speed for memory. Figure 8 displays the results reported by each backend on the size of the final acceleration structure.

4.2.2 Ray Tracing

Four test cases were constructed to measure the speed of the `GetIntersections` call for each backend, single ray intersections for both small and large geometries and high-quantity (10,000) ray intersections for both small and large geometries. The results are shown in Figure 9, Figure 10, Figure 11, and Figure 12, respectively.

Figure 8. The final size of the GAS constructed for a scene after memory optimization has been performed.

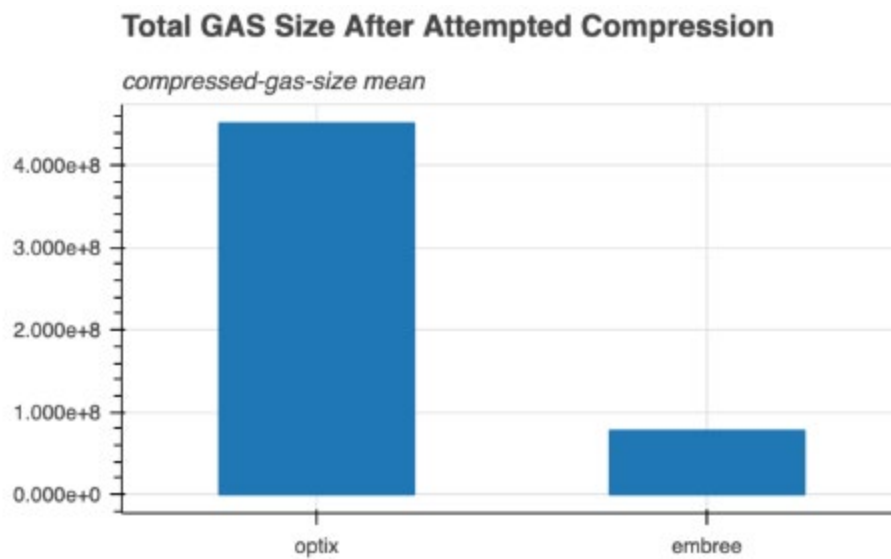


Figure 9. A single ray is intersected with a single triangle. The lack of overhead required by Embree, along with caching advantages, make it by far the faster choice for performing single ray casts.

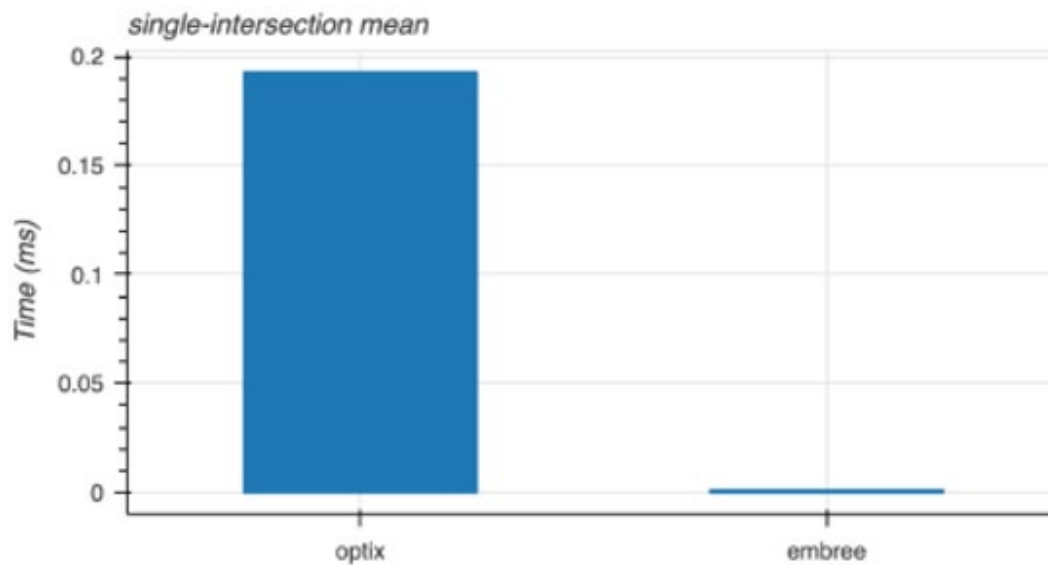


Figure 10. A single ray is intersected with over 475,000 triangles. Embree still outperforms OptiX here for the reasons specified in Figure 9.

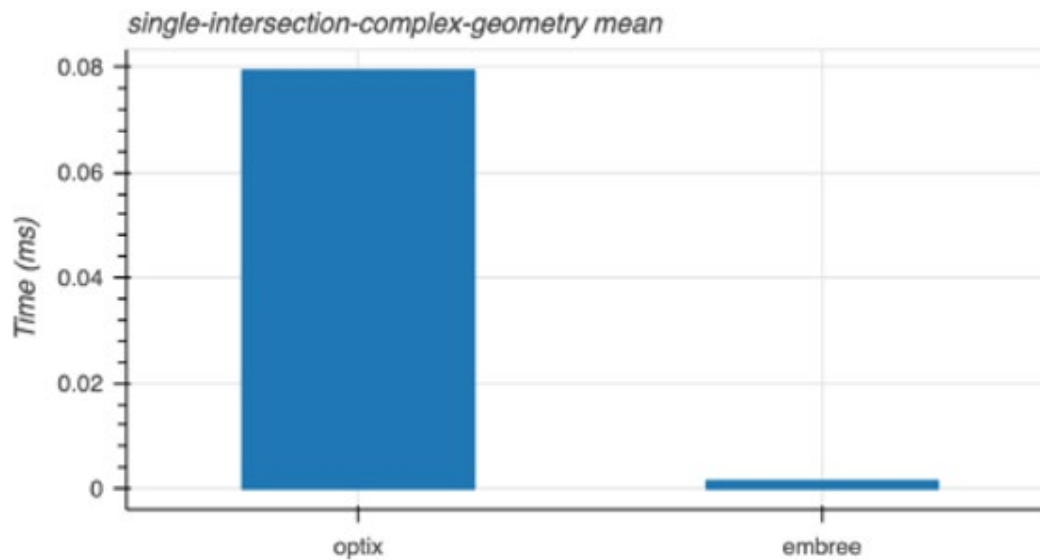


Figure 11. One million rays cast into a one-triangle GAS. Scaling the number of rays gives OptiX the clear advantage. With one million rays, the intersection speed advantage offered by the GPU outweighs the overhead.

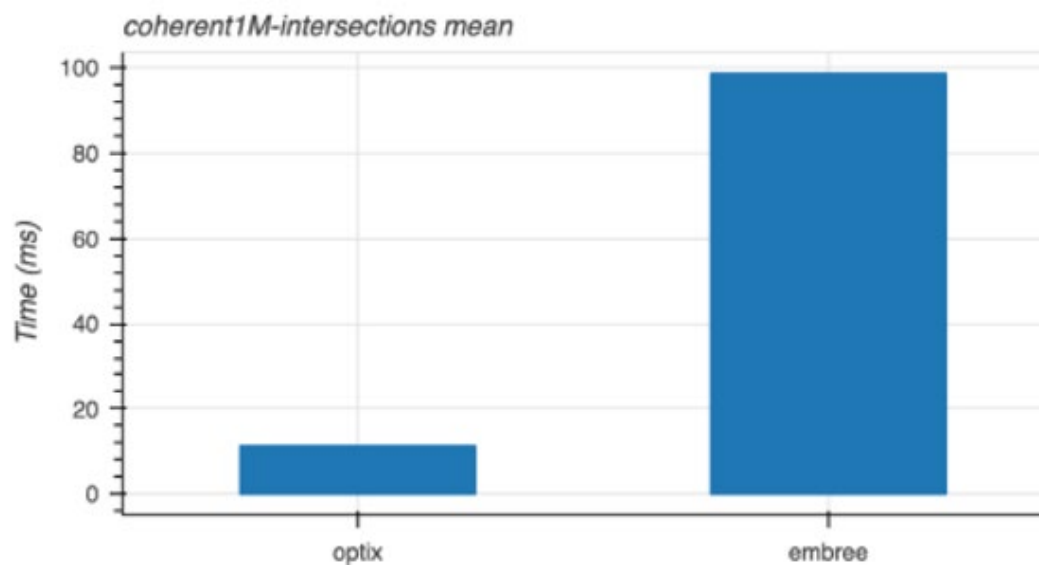
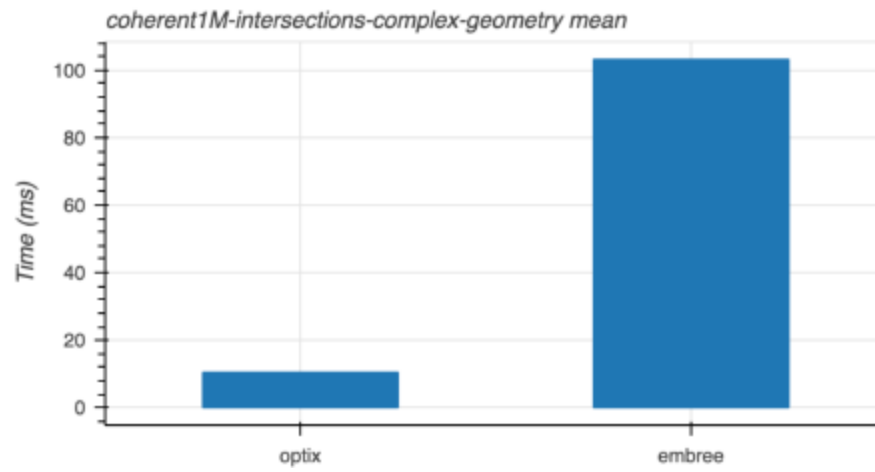


Figure 12. One million rays cast into a 475,000-triangle GAS.



5 Conclusion

This document describes the development and first implementation of a cross platform GPU accelerated ray tracing library in C++ for use in simulation and visualization of scientific models. The top level API is intentionally of low complexity, with only a few functions for building a 3D scene and casting rays into that scene, to simplify use in new and existing simulation codebases, such as VANE::ESE. To support multiple platforms and GPUs, several ray tracing and graphics programming libraries are used to implement the top-level API, including Nvidia Optix, Intel Embree, AMD Radeon Rays, and Vulkan, with some implementation details presented for each backend.

A variety of future work is planned for later versions of this library, such as full integration into VANE::ESE. Additional features are also being considered, such as supporting full path tracing on the GPU, with the addition of scene material support and ray bounce calculations all performed on the GPU. Finally, more backend libraries could be added to broaden available hardware support, such as Apple's Metal for macOS support or DirectX 12 for Microsoft Windows.

Bibliography

- AMD. 2021a. "RadeonRays 4.0." AMD Radeon ProRender. <https://radeon-pro.github.io/RadeonProRenderDocs/en/rr/about.html>.
- AMD. 2021b. "AMD Radeon Rays." AMD GPUOpen | Let's Build Everything. <https://gpuopen.com/radeon-rays/>.
- Carrillo, J. T., C. T. Goodin, and A. E. Baylot. 2016. "NIR Sensitivity Analysis with the VANE." In G. C. Holst and K. A. Krapels, editors, *Infrared Imaging Systems: Design, Analysis, Modeling, and Testing XXVII* 9820: 98200I. International Society for Optics and Photonics (SPIE). <https://doi.org/10.1117/12.2222077>.
- Carrillo, J. T., C. T. Goodin, and J. D. Fernandez. 2020. Sensor and Environment Physics in the Virtual Autonomous Navigation Environment (VANE). Technical Report. ERDC/GSL TR-20-32. Vicksburg, MS: Engineer Research and Development Center. <https://hdl.handle.net/11681/37968>.
- Chambers, D. R., J. Gassaway, C. Goodin, and P. J. Durst. 2015. "Simulation of a Multispectral, Multicamera, Off-Road Autonomous Vehicle Perception System with Virtual Autonomous Navigation Environment (VANE)." In D. A. Huckridge, R. Ebert, M. T. Gruneisen, M. Dusek, and J. G. Rarity, editors, *Electro-Optical and Infrared Systems: Technology and Applications XII; and Quantum Information Science and Technology* 9648: 964802. International Society for Optics and Photonics (SPIE). <https://doi.org/10.1117/12.2194372>.
- Goodin, C., M. Doude, C. R. Hudson, and D. W. Carruth. 2018. "Enabling Off-Road Autonomous Navigation—Simulation of Lidar in Dense Vegetation." *Electronics* 7 (9): 2079–9292. doi: 10.3390/electronics7090154. <https://www.mdpi.com/2079-9292/7/9/154>.
- Goodin, C., P. J. Durst, B. Gates, C. Cummins, and J. Priddy. 2010. "High Fidelity Sensor Simulations for the Virtual Autonomous Navigation Environment." In N. Ando, S. Balakirsky, T. Hemker, M. Reggiani, and O. von Stryk, editors, *Simulation, Modeling, and Programming for Autonomous Robots*, 75–86. Berlin: Springer. doi: 10.1007/978-3-642-17319-6_10.
- Goodin, C., R. Kala, A. Carrillo, and L. Y. Liu. 2009. "Sensor Modeling for the Virtual Autonomous Navigation Environment." *IEEE Sensors Journal*: 1588–92. doi: 10.1109/ICSENS.2009.5398491.
- Google. 2022. "GoogleTest User's Guide." GoogleTest Framework. <https://google.github.io/googletest/>.
- Khronos Group. 2021. "Khronos SPIR-V Registry." Khronos Registry. <https://registry.khronos.org/SPIR-V/>.

- Khronos Group. 2022. "The Khronos Group Inc." <https://www.khronos.org/>.
- Koch, D. 2020. "Vulkan Ray Tracing Pipeline Extension." https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VK_KHR_ray_tracing_pipeline.html.
- Morley, K. 2019. "How to Get Started with OptiX 7." <https://developer.nvidia.com/blog/how-to-get-started-with-optix-7/>.
- Parker, S. G., J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich. 2010. "Optix: A General Purpose Ray Tracing Engine." *ACM Transactions of Graphics* 29 (4). <https://doi.org/10.1145/1778765.1778803>.
- Wald, I., S. Woop, C. Benthin, G. S. Johnson, and M. Ernst. 2014. "Embree: A Kernel Framework for Efficient CPU Ray Tracing." *ACM Transactions of Graphics* 33 (4). <https://doi.org/10.1145/2601097.2601199>.

Appendix A: User Guide

A.1 Wrapper Compilation

There are two steps to compiling a working wrapper executable: compiling the selected backend(s) and compiling the generalized wrapper. Each of these code bases should have detailed instructions for compilation included with the code itself. Start by downloading or cloning the base wrapper and whichever backends you would like to use. Then, go into each backend and compile them using their individual build systems. Finally, using `CMake`, configure the base wrapper, filling out the appropriate cache variables with the paths to each backend you've compiled. This will likely require pointing to both the wrapper code and the underlying libraries those wrappers are built on (e.g., the OptiX wrapper code and the NVIDIA OptiX SDK directory). Once the wrapper is compiled, you can include the resulting header files and library files in your own code or run the benchmarking code and rendering samples included in the base wrapper's code.

A.2 API Reference

```
unsigned int CommitGeometry(unsigned int *facet_indices,
                           unsigned int num_facets,
                           float *vertices,
                           unsigned int num_vertices,
                           unsigned int geom_id)
```

Creates a geometry model that can be instantiated using the `CreateInstance` method. Note: This does not insert the geometry into the scene. This is accomplished by instantiating this geometry model using the assigned ID.

Arguments:

`facet_indices`: An array of indexes into the vertex list
`num_facets`: The number of index triplets recorded in `facet_indices`
`vertices`: An array of vertex points of the form `x0,y0,z0,x1,y1,z1,...`
`num_vertices`: The number of coordinate vectors in the vertex array
`geom_id`: The unique ID to assign to this geometry instance

Returns: The geometry ID passed in as `geom_id` if successful.

```
unsigned int CreateInstance(unsigned int geom_model_id,
                           unsigned int inst_id)
unsigned int CreateInstance(unsigned int geom_model_id,
                           unsigned int inst_id,
```

```
float x, float y, float z,
float x_rot=0,
float y_rot=0,
float z_rot=0,
float x_scale=1,
float y_scale=1,
float z_scale=1)
```

Used to create a geometry instance and optionally place it in the scene. See `TransformInstance` below for a description of the parameters used to place the instance in the scene.

Arguments:

`geom_model_id`: The committed geometry model to instantiate
`inst_id`: The global identifier to assign to this instance

Returns: The instance ID passed in.

```
bool TransformInstance(float x, float y, float z,
                      float x_rot, float y_rot, float z_rot,
                      float x_scale, float y_scale, float
                      z_scale,
                      unsigned int instance_id)
```

Used to transform a geometry instance in world coordinate space.

Arguments:

`x`: The X translation offset
`y`: The Y translation offset
`z`: The Z translation offset
`x_rot`: The X rotation value (in radians)
`y_rot`: The Y rotation value (in radians)
`z_rot`: The Z rotation value (in radians)
`x_scale`: The X scale value
`y_scale`: The Y scale value
`z_scale`: The Z scale value
`instance_id`: The unique ID of the instance to apply the transform to

Returns: True of successful otherwise false

```
void BuildScene()
```

Used to prepare the scene for ray tracing. Can be used by wrappers to copy geometry to device, build acceleration structures, compact data, or any other necessary preparations for raytracing.

```
void GetIntersections(float *ray_origins,
                     float *ray_directions,
                     unsigned int num_rays,
                     Intersection *intersections)
```

Used to perform ray intersection tests.

Arguments:

`ray_origins`: The 3D coordinate vectors for the origin of each ray
`ray_directions`: The 3D coordinate vectors for the direction of each ray
`num_rays`: The number of rays to be cast
`intersections`: A preallocated space to store the intersection results in

A.3 Running Supplementary Code

A.3.1 Tests

Unit tests are included in the base wrapper code for testing backends. These tests are written in the Google Test framework and can be easily executed using the `RTWrapper_test` executable in the build directory. The `BUILD_TESTS` CMake option must be turned on in compilation.

A.3.2 Benchmarking

There is some high-level benchmarking code included in the base wrapper to allow users to gather metrics from backends on whichever device they are running on. To generate this basic report, run the `run_timings` executable found in the `timings/` directory. Just list the backends that you wish to compare as command-line arguments (e.g., `run_timings optix embree`). This will run the benchmarking code and generate a yaml report file called `rt_timings.yml`. This report is relatively human-readable and can be parsed to generate charts of backend comparisons. Additionally, there is a python tool included in the `timings/viz` directory that can be used to generate a html or png report containing charts for every test included in the benchmarking report. Some examples of the charts this tool generates are found in Figures 9, 10, 11, and 12.

A.3.3 Samples

To run the rendering samples in the base wrapper code, the `CMake` option for `COMPILE_SAMPLES` must be turned on when the code is compiled. After compilation, the `samples` directory in the build directory will have all of the examples included in the individual directories. Most of the sample executables require two command-line arguments: the backend to run and the file name of the output image. However, check the samples included documentation for specific documentation on running that example.

REPORT DOCUMENTATION PAGE

1. REPORT DATE July 2023		2. REPORT TYPE Final Technical Report		3. DATES COVERED	
				START DATE FY2022	END DATE FY2022
4. TITLE AND SUBTITLE A General-Purpose Multiplatform GPU-Accelerated Ray Tracing API					
5a. CONTRACT NUMBER		5b. GRANT NUMBER		5c. PROGRAM ELEMENT 0602145A	
5d. PROJECT NUMBER BG2		5e. TASK NUMBER 2		5f. WORK UNIT NUMBER	
6. AUTHOR(S) R. Harrison Hunter, S. Ross Glandon, and Jing-Ru C. Cheng					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) US Army Engineer Research and Development Center (ERDC) Information Technology Laboratory (ITL) 3909 Halls Ferry Road Vicksburg, MS 39180-6199				8. PERFORMING ORGANIZATION REPORT NUMBER ERDC/ITL TR-23-2	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Headquarters, US Army Corps of Engineers Washington, DC 20314-1000			10. SPONSOR/MONITOR'S ACRONYM(S) USACE		
12. DISTRIBUTION/AVAILABILITY STATEMENT DISTRIBUTION STATEMENT A. Approved for public release; distribution is unlimited.					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT Real-time ray tracing is an important tool in computational research. Among other things, it is used to model sensors for autonomous vehicle simulation, efficiently simulate radiative energy propagation, and create effective data visualizations. However, raytracing libraries currently of-fered for GPU platforms have a high level of complexity to facilitate the detailed configuration needed by gaming engines and high-fidelity ren-derers. A researcher wishing to take advantage of the performance gains offered by the GPU for simple ray casting routines would need to learn how to use these ray tracing libraries. Additionally, they would have to adapt this code to each GPU platform they run on. Therefore, a C++ API has been developed that exposes simple ray casting endpoints that are im-plemented in GPU-specific code for several contemporary device plat-forms. This API currently supports the NVIDIA OptiX ray tracing library, Vulkan, AMD Radeon Rays, and even Intel Embree. Benchmarking tests using this API provide insight to help users determine the optimal backend library to select for their ray tracing needs. HPC research will be well-served by the ability to perform general purpose raytracing on the increasing amount of graphics and machine learning nodes offered by the DoD High Performance Computing Modernization Program.					
15. SUBJECT TERMS Autonomous robots; Detectors--Simulation methods; High performance computing; Military robots; Optical radar; Ray tracing algorithms					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT		18. NUMBER OF PAGES
a. REPORT Unclassified	b. ABSTRACT Unclassified	c. THIS PAGE Unclassified	None		47
19a. NAME OF RESPONSIBLE PERSON				19b. TELEPHONE NUMBER (include area code)	