Ray Tracing API Integration for OpenGL Applications

Wei-Hao Lai
Industrial Technology Research Institute
195, Sec. 4, Chung Hsing Rd., Chutung, Hsinchu, 31040, Taiwan
WeiHaoLai@itri.org.tw

Chang-Yu Tang
Industrial Technology Research Institute
195, Sec. 4, Chung Hsing Rd., Chutung, Hsinchu, 31040, Taiwan
CYTang@itri.org.tw

Chun-Fa Chang
National Taiwan Normal University
162, Sec. 1, Heping E. Rd., Taipei City 106, Taiwan
chunfa@ntnu.edu.com

ABSTRACT

Ray tracing is one of the most important rendering techniques in computer graphics. By means of simulating reflection and refraction of light transportation, ray tracing generates more photorealistic images than scanline rendering. However, the high computational cost is the main disadvantage of ray tracing algorithm. In recent years, the computing power of GPU has increased dramatically, and general-purpose computing on graphics processing units (GPGPU) has become popular. Many scholars have presented some physically based rendering methods with CUDA or OpenCL in order to improve image quality and increase rendering speed. Because rasterization is the mainstream in the gaming industry, there is still a long way to go to make ray tracing accepted by the industry in the near future. We introduce a ray tracing API integration for OpenGL applications that can replace the original OpenGL rasterization with ray tracing by simply adding a few lines of code, and the ray tracing algorithm in this API is parallelized by OpenCL.

Keywords

3D Rendering, Ray Tracing, Game Engine, Parallel Computing, OpenCL

1. INTRODUCTION

Ray tracing is widely used in the special effects and 3D animation industry. These commercial products emphasize photorealistic scenes and high quality lighting effects. Scenes and animations are rendered offline, so the speed is not the primary concern. However, due to the boom of 3D game industry, customers nowadays are not only asking for vivid scenes, but also instant interactions. Using general-purpose computing on graphics processing units (GPGPU) architecture, ray tracing now can be parallelized by GPU to make real-time rendering possible.

Rasterization is the main image synthesis method in the 3D game industry (e.g., OpenGL, Direct3D, and Vulkan). Beside the consideration of speed, the reason why it might be difficult to replace rasterization with ray tracing is that developers need to be familiar with the new structure of API to alter the original source code. In this paper, we propose an OpenGL-like API to substitute the OpenGL native rasterization to ray tracing, helping developers achieve the global illumination effects with minimal modifications.

OpenGL 3.0 introduced a deprecation mechanism to simplify future revisions of the API [Shr09a]. The direct-mode rendering using *glBegin* and *glEnd* function calls is one of the deprecated features. Vertex buffer object (VBO) is considered to be a more efficient way to make draw calls instead. VBO allows vertex array data to be stored in high-performance graphics memory on the server side and promotes efficient data transfer. We will show a few OpenGL applications with our ray tracing API integration to demonstrate its feasibility.

The ray tracing API integration is separated into two parts. The first part is collecting parameters from OpenGL API calls, redirecting each call to our API and capture parameters from original OpenGL source code at the same time. The other part is the ray tracing program written in OpenCL based on C99.

Due to the innate limitation, ray tracing algorithm starts when the whole scene data is ready. Once the ray tracing kernel is called, the API returns the rendered frame back to OpenGL and displays the frame on screen by the render-to-texture method. In other words, OpenGL would not rasterize the frame, but only display the frame rendered by our ray tracing API.

The paper is organized as follows. Section 2 provides an overview of the related work. In section 3, we describe the implementation and structure of our ray tracing API integration. More details about ray tracing

algorithm implementation are shown in section 4. Some experiment results and comparison between our method and the original OpenGL are in section 5. Finally, we conclude in section 6.

2. Related Work

The PowerVR OpenRL is a flexible low level API for accelerating ray tracing in both graphics and non-graphics applications [Img17a]. By writing OpenRL shading language (RLSL), developers can obtain full control of ray tracing logic. There are three types of shaders in OpenRL. Vertex shader handles vertex positions in the scene. Frame shader sets rays and sampling methods. A ray shader defines the ray color and the energy that a ray contains.

The OptiX is an application framework for achieving optimal ray tracing performance on the GPU [Nvi17a]. OptiX provides a simple, recursive, and flexible pipeline for accelerating ray tracing algorithms. Many professional 3D computer graphics software are using it as a plug-in (e.g., AutoCAD, Maya, and Lightworks). Though Optix is a powerful and mature framework, it only works on Nvidia graphic cards. Considering OpenGL cross-platform feature, our ray tracing API integration has to minimize dependency as far as possible.

Brian Paul started the Mesa project in 1993 [Pau17a], which is an open-source implementation of the OpenGL specification. Mesa implements a translation layer between a graphics API such as OpenGL and the graphics hardware drivers in the operating system kernel. Our method is quite similar to Mesa. The difference is instead of simulating the OpenGL pipeline, we use ray tracing instead.

OpenCL is the open standard for cross-platform and parallel programming of diverse processors found in personal computers, servers, mobile devices, and embedded platforms [Khr17a]. An OpenCL program is divided to run on host and device. The host is the main CPU used to configure kernel execution. The device is the component that contains the processing units that will execute the kernel. A host can trigger multiple kernels to do diverse missions, and assign to different devices.

3. Ray Tracing API Integration

The ray tracing API integration has two parts. The first part is called the function calls redirection, which is responsible for parameters collection. When OpenGL API works, the parameters are packed and redirected to our API for later use. The second part is a ray tracer written in OpenCL, which undertakes the parameters from the previous step and passes them to a kernel program. After receiving the parameters from the host, the kernel program implements ray tracing in parallel.

In this paper, the ray tracing API integration is easy to use. Programmers only need to include a header file, and then an OpenGL application would automatically render with ray tracing, but without the original rasterization.

3.1 Supported OpenGL Versions

According to OpenGL 3.0 specification, the fixed function pipeline as well as most of the related OpenGL functions and constants were declared deprecated. These deprecated elements and concepts were commonly referred to legacy OpenGL [Seg17a]. How to distinguish a legacy OpenGL source code is not a difficult task. One typical sign that a program is using Legacy OpenGL is immediate mode. Immediate mode is using *glBegin* and *glEnd* with *glVertex* and *glColor* in between them

The advantages of legacy OpenGL are built-in lighting model, procedural-base method, etc. GPU hardware architecture improves in leaps and bounds. The fixed function pipeline is unable to match the flexibility. The OpenGL Shading Language (GLSL) has been added to allow for increasing flexibility of the rendering pipeline at the vertex and fragment level. Nevertheless, there are still some OpenGL applications use the immediate mode, such as education courses and tutorial websites because it is easier to learn.

We concentrate on applications that use client-side VBO, and give those applications the ray tracing integration support, but we do not support immediate mode starts with *glBegin* and ends with *glEnd*. Due to unified shading architecture, we do not know how the vertex attribute pointers are used in GLSL. The applications containing GLSL program would be skipped by our ray tracing integration API. However, modern OpenGL programs must contain at least one GLSL program. The problem is solved by creating a hint module added at the front of every vertex attribute pointer to tell our API what those VBOs actually do, and then read GLSL programs, trying to interpret by our API. The idea is inspired by the OpenRL programming model [Img17a].

3.2 API structure

Figure 1 shows the ray tracing API integration flow chart. As we can see on the leftmost side is a common OpenGL application flow containing model data loading, light adjusting, data pointer definition, draw calls, and synchronization. At the middle is what our API does to the OpenGL API. These OpenGL API calls would not present their original behaviors, but execute data collection and transfer data to our specification. The rightmost is the ray tracing flow chart. We start up a ray tracing kernel when the program meets the synchronization point. A space-partitioning data structure called KD-tree is used to boost the ray-triangle intersection test in the algorithm. The test scenes in our experiments are static because every time an object moves, the tree must be rebuilt

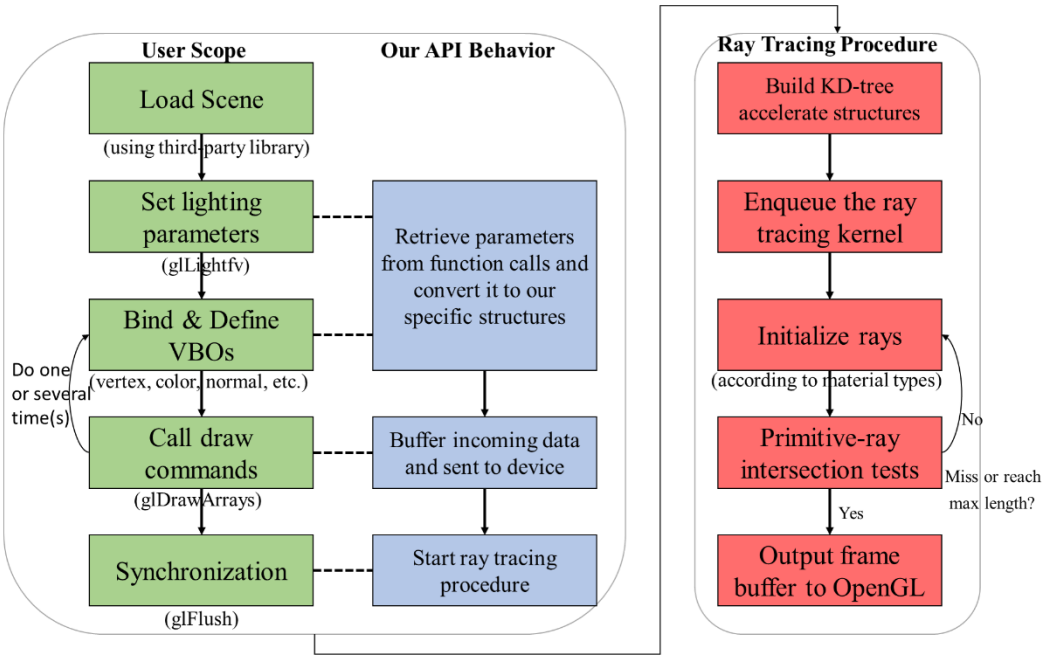

Figure 1: Ray tracing API integration flow chart. The leftmost green flow is user scope, and it is about the execution sequence of an OpenGL application. The middle is about capturing parameters from original OpenGL functions. The rightmost is the ray tracing procedure written in OpenCL.

and our implementation would not be able to reach real-time performance. Danilewski, et al. [Dan10a] constructed a binned SAH KD-tree with CUDA to solve the dynamic scene problem and declared their method could reach real-time, but now we just focus on our method that works on static scenes.

3.3 Supported Function Calls

Legacy OpenGL applications use built-in matrix manipulation functions to change the position of 3D objects, fixed-function lighting model to add effects, and client-side VBO to upload data. These actions are done during the data preparation stage in our ray tracing API integration. The ray tracing kernel in our API would wait until synchronization call signal. Table 1 shows all supported function calls, which are enough for a simple OpenGL application execute smoothly. Chapter 5 shows some results rendered by our ray tracing API and detail analysis would be discussed later.

Buffer	Rendering	Capability
glGenBuffers	glVertexPointer	glEnableClientState
glBindBuffer	glColorPointer	glDisableClientState
glBufferData	glNormalPointer	glEnable
	glDrawArrays	glDisable
	glFlush	
Lighting	**GLU(OpenGL Utility Library)**	**Extension**
glLightfv	gluPerspective	rtMaterialEXT
	gluLookAt	rtBuildAcclStructEXT

Table 1: Supported function calls. The extension column represents some features that legacy OpenGL does not support.

4. Ray Tracing

In this section, we will talk about the rendering algorithm in our ray tracing API. The rendering method is a ray tracing program developed with OpenCL.

4.1 Overview

Ray tracing describes a method for producing visual images constructed in 3D computer graphics environments [Whi05a]. To implement ray tracing algorithm, the steps are as follows. First, construct vectors from observer to each sample on the image plane. These vectors are deemed as lights in the space. Second, calculate intersections with all 3D geometrics in the scene and compute the color using lighting models. Then, the third step is to generate new rays according to reflection models and refraction models.

4.2 Implementation Details

Back to Figure 1, rightmost: we build an acceleration data structure called KD-tree. This step helps us conserve some unnecessary intersection calculations and improve rendering speed. Our ray tracing kernel is inspired by Whitted ray tracer model [Whi05a], but there are some differences in detail. In OpenCL, we create a kernel that execute in parallel, though the model is not. The second point is that the original model expresses in recursive style. However, OpenCL does not support recursion. Consequently, it is necessary to convert recursion to iteration-based implementation.

The intersection test is one of the most time-consuming computation in ray tracing algorithm. A reputable intersection computation method gains more computational efficiency. The Möller-Trumbore algorithm is a fast ray-triangle intersection algorithm [Mol05a]. It calculates the intersection of a ray and a triangle in three dimensional space without needing precomputation of the plane equation of the plane that contains the triangle. In our implementation, triangle and sphere object types are supported.

We apply the diffuse part of Phong shading [Pho75a] of illumination when intersection points are calculated. Some other physical effects such as reflection and refraction are also implemented in our API. It is worth notice that while using our ray tracing integration API, we can add these effects in an OpenGL application program though the original source code does not have these effects. The most common used accelerating structure of ray tracing algorithm are BVH-tree and KD-tree. We wrote a KD-tree for our API, and we refer to Pharr et al [Pha04a]. The KD-tree is a binary tree in which every node is a k-dimensional point. Every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts. While doing a ray intersection test, traverse through a KD-tree can prune half of the triangles that are not hit, preventing the unnecessary intersection being computed.

5. RESULTS

This section presents the results obtained by our ray tracing API integration. The experiment environment is according to Table 2. The image resolution is set to 800*600.

Specifications	
OS	Windows 10 pro x64
CPU	Intel i7-6700K
RAM	DDR4 2666 32GB
NVIDIA® GeForce GTX TITAN X	
CUDA Cores	3072
Memory	12 GB G5
TFLOPS	7

Table 2: Experiment environment

Figure 2 are images rendered by the original OpenGL rasterization and our ray tracing API integration. The left pictures are rendered by OpenGL and the right ones are rendered by our API. The picture at the bottom-right has a hard shadow effect that is undoubtedly easy to ray tracing, but OpenGL would need a bunch of GLSL code to achieve the same result. Figure 3 shows more results rendered by our API. The top-left is a Cornell box with a reflective red wall and a dragon in it. The top-right is a glass cube and a dragon in a Cornell box. The paired images at the bottom is a dragon and Sponza separately. Table 3 shows the frame per second (FPS) of each scene. B. & D. represents Cornell box and dragon. The "w/ Reflec." and "w/ Refrac." mean "with reflection" and "with refraction". These features only affect our API integration, so OpenGL column has no test results. Although our ray tracing API is still slower than rasterization. However, it is worth to be mentioned that in most test cases, the FPS data of our API are above 30 FPS, which is the minimum threshold of real-time rendering. This means that the hardware is now able to bear such large amount of computing throughput.

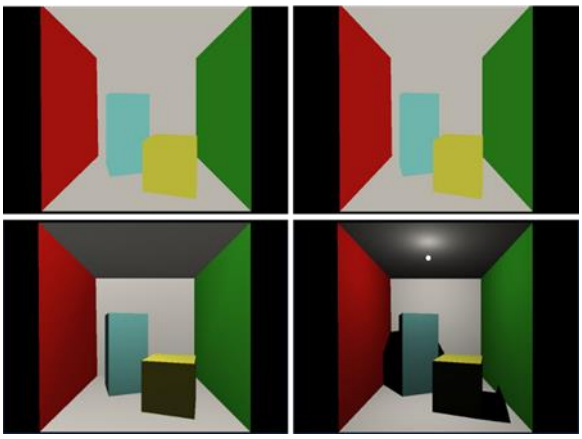

Figure 2: The comparison of rendering results between original OpenGL (left) and our ray tracing API integration (right).

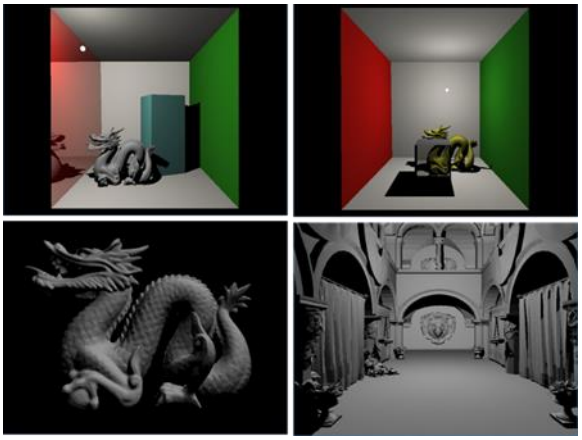

Figure 3: More experiments by proposed method.

	# Tri.	**OpenGL**	**Our API**
Cornell Box	34	5730.58 FPS	261.72 FPS
B. & D.	100,034	5149.60 FPS	53.62 FPS
B. & D. w/ Reflec.	100,034	---------	50.40 FPS
B. & D. w/ Refrac.	100,034	---------	49.30 FPS
Sponza	262,267	2612.49 FPS	14.39 FPS

Table 3: FPS comparison between OpenGL and our API

	KD-tree build time (second)	Triangle intersections per frame	Sphere intersections per frame	Avg. Intersections per second
Cornell Box	0.000044	4113244	480000	17550.2216
B. & D.	0.503594	3805833	480000	79929.7464
B. & D. w/ Reflec.	0.503594	4868135	539321	107290.7937
B. & D. w/ Refrac.	0.503594	5125656	501409	114139.2495
Sponza	1.512600	15000425	480000	1075776.5810

Table 4: Detail information about ray intersections

Typically the intersection test is a bottleneck of the ray tracing algorithm. Except the construction of KD-tree, we make the rest actions all done by GPU to benefit from its parallelization. According to Table 3 and Table 4, if the average intersections per second is around a hundred thousand, the FPS can be 50 or even more. While rendering the Sponza scene, the average intersections come up to a million, but the FPS still remains around 14.

6. Conclusion

The contribution of this paper is the introduction of a ray tracing API integration for OpenGL applications. Our API makes it much easier to develop a ray tracing based application by using OpenGL code at hand. Moreover, a ray tracing based application can reach interactive speed is in practice. Our API still have a lot of room for improvement. For example, we can use Monte Carlo path tracing techniques mentioned by Keller et al. [Kel12a]. Another improving direction is to break the limit of static scenes. Danilewski et al. [Dan10a] constructed SAH KD-tree on GPU with CUDA, which could achieve real-time rendering of dynamic scenes.

7. Acknowledgement

This work is supported in part by the Ministry of Science and Technology (Taiwan) grant MOST 103-2221-E-003-010-MY3, the Industrial Technology Research Institute grant D0-10404-28-3, and by the Institute for Information Industry.

8. REFERENCES

[Dan10a] Danilewski, P., Popov, S., & Slusallek, P. Binned sah kd-tree construction on a gpu. Saarland University, 1-15.

[Img17a] Imagination T. OpenRL SDK - Imagination Community. Retrieved January 4, 2017, from https://community.imgtec.com/developers/powervr/openrl-sdk/

[Kel12a] Keller, A., Premoze, S., & Raab, M. Advanced (Quasi) Monte Carlo Methods for Image Synthesis. In ACM SIGGRAPH 2012 Courses.

[Khr17a] Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems. Retrieved January 4, 2017, from https://www.khronos.org/opencl/

[Pau17a] Paul, B. Mesa Introduction. Retrieved January 4, 2017, from http://www.mesa3d.org/

[Mol05a] Möller, T., & Trumbore, B. Fast, minimum storage ray/triangle intersection. In ACM SIGGRAPH 2005 Courses (p. 7). ACM.

[Nvi17a] NVIDIA C. NVIDIA OptiX Ray Tracing Engine | NVIDIA Developer. Retrieved January 4, 2017, from https://developer.nvidia.com/optix

[Pha04a] Pharr, M., & Humphreys, G. Physically based rendering: *From theory to implementation. Morgan Kaufmann*.

[Pho75a] Phong, B.T. Illumination for computer generated pictures. Communications of the ACM, 18(6), 311-317.

[Seg17a] Segal, M., & Akeley, K. The OpenGL Graphics System: A Specification (Version 3.0 - September 23, 2008). Retrieved January 4, 2017, from https://www.opengl.org/registry/doc/glspec30.20080923.pdf

[Shr09a] Shreiner, D., & Bill The Khronos OpenGL ARB Working Group. OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1. Pearson Education, 2009.

[Whi05a] Whitted, T. An improved illumination model for shaded display. In ACM Siggraph 2005 Courses (p. 4). ACM.