SCIPEDIA

A Hybrid Approach for Vulkan-Based Ray Tracing Implementations

Mingyu Kim[1] and Nakhoon Baek[1,2,*]

[1] School of Computer Science and Engineering, Kyungpook National University, Daegu, 41566, Republic of Korea

[2] Data-Driven Intelligent Mobility ICT Research Center, Kyungpook National University, Daegu, 41566, Republic of Korea

INFORMATION

Keywords:
Ray tracing
hybrid rendering
Vulkan
acceleration
experimental results

DOI: 10.23967/j.rimni.2026.10.72287

SCIPEDIA Revista Internacional de Métodos Numéricos para Cálculo y Diseño en Ingeniería

Revista Internacional Métodos numéricos para cálculo y diseño en ingeniería RIMNI in cooperation with CIMNE

A Hybrid Approach for Vulkan-Based Ray Tracing Implementations

Mingyu Kim[1] **and Nakhoon Baek**[1,2,*]

[1]School of Computer Science and Engineering, Kyungpook National University, Daegu, 41566, Republic of Korea

[2]Data-Driven Intelligent Mobility ICT Research Center, Kyungpook National University, Daegu, 41566, Republic of Korea

ABSTRACT

Three-dimensional (3D) graphics output started with traditional local shading models processed in real time. Ray-tracing techniques are actively introduced for high-quality output. These traditional local shading pipelines and ray tracing pipelines are typically provided independently. Recently, hybrid rendering methods were introduced to integrate the results of these pipelines for better real-time graphics results. However, technical problems arise with heterogeneous application programming interfaces (APIs). Vulkan was recently introduced as a new low-level graphics API in practical 3D graphics implementations. With its new ray tracing extensions, Vulkan has become one of the 3D graphics environments that simultaneously supports traditional local shading and modern ray tracing pipelines. Stand-alone ray tracing and hybrid rendering techniques are implemented to determine a practical implementation of the ray tracing technique in this latest Vulkan environment. This work reveals the completeness of both implementations and compares the graphics performance with experimental computer animation sequences and different camera configurations. The hybrid rendering techniques perform better with previous graphics processing unit models, and the core ray tracing implementation achieves faster processing speeds of 39% to 64% compared to the stand-alone ray tracing method, at least in the Vulkan environment with the latest NVIDIA graphics card. In contrast, the preprocessing of the geometry pass takes noticeable time, which reduces our overall performance improvements. The experimental results can be applied as guidelines for implementing practical real-time 3D graphics applications and as a new benchmark model for ray tracing implementations.

OPEN ACCESS

Received: 23/08/2025

Accepted: 28/11/2025

Published: 03/02/2026

DOI
10.23967/j.rimni.2026.10.72287

Keywords:
Ray tracing
hybrid rendering
Vulkan
acceleration
experimental results

1 Introduction

Computer graphics has steadily and rapidly grown since its start in the 1950s [1,2]. Fundamentally, the goal of computer graphics is to generate high-quality output in real time to satisfy human vision systems. However, it still demonstrates performance limitations. Therefore, there have been practical attempts to improve the graphics processing time while maintaining the final output quality [3–5]. In early graphics systems, *local shading models* [6], such as the *Phong illumination model* [7] (Fig. 1), were

*Correspondence: Nakhoon Baek (nbaek@knu.ac.kr).

developed to achieve real-time processing. Optimized three-dimensional (3D) shading pipelines exist, even at the semiconductor chip level, where additional techniques, such as *deferred rendering* [8], are applied.

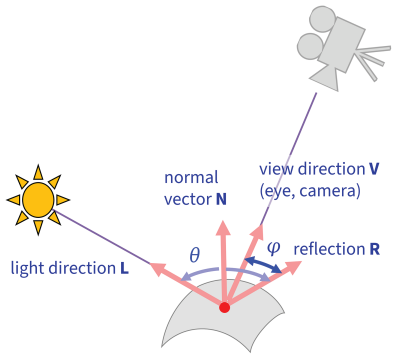


Figure 1: Local illumination model

Global illumination models, such as *ray tracing* [9,10] and *radiosity* [11], have been developed to improve graphics output quality. Along with the advancements in graphics hardware, ray tracing techniques are processed at the semiconductor chip level, and dedicated hardware systems are commercially available [12,13]. *Hybrid approaches* simultaneously employing existing traditional local shading and high-quality ray tracing pipelines have been developed to achieve appropriate graphics quality in a relatively fast processing time [14,15]. This paper employs a hybrid approach between the *traditional local shading pipeline* and the *ray tracing pipeline* in the personal computer environment, with *ray tracing extensions* [16] in *Vulkan* [17–19]. Vulkan is currently one of the most widely used 3D graphics application programming interfaces (APIs). Practical 3D graphics implementations can use Vulkan and its ray tracing extensions for high-quality graphics output, whereas graphics implementors require sufficiently tested implementation models and guidelines.

This paper implements both of a typical *stand-alone ray tracing method* and a new *hybrid rendering method*, using the version 1.2 or later of shader programs in Vulkan. More precisely, we plan to optimize the rendering process with a hybrid method, using the G-buffer in two ways. First, we effectively reduce the depth of recursive ray tracing, using the preprocessed values in the G-buffer. Second, we minimize unnecessary computations through selectively executing the ray tracing function only when the G-buffer indicates that the object has reflective or refractive characteristics.

We measure the processing speeds of both methods using various camera configurations for computer animation sequences that are widely used for graphics experiments [20]. The experimental results can be employed as guidelines for future practical 3D graphics implementations and as a new benchmark model for ray tracing implementations.

Section 2 presents background work and related previous findings. Section 3 presents the overall design and shader codes from the implementations. The experimental environment and measurement results follow in Section 4. Finally, the conclusions and future work are presented in Section 5.

2 Related Work

2.1 Traditional Local Shading Methods

In 3D computer graphics, a 3D virtual object undergoes various processing steps to present it realistically on a 2D computer screen. Typical 3D graphics processing steps can be expressed as a

rendering pipeline, as depicted in Fig. 2. The complex processing of real-world physical phenomena and their optical interpretation should be considerably simplified for real-time processing.

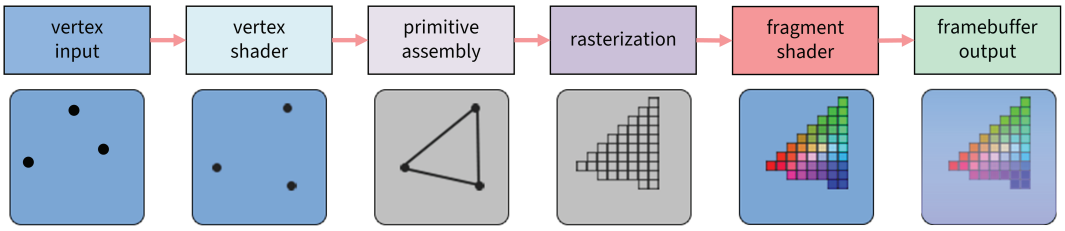


Figure 2: Traditional local shading pipeline

In early computer graphics, traditional local illumination models, including the famous Phong shading model [7], were developed, and based on these models, semiconductor chip-based hardware systems are now standard [6,21]. However, these models have limitations in accurately simulating realistic optics effects, such as light reflection and refraction.

Deferred rendering (or *deferred shading*) [8,22] is a technique that delays 3D shading calculations later in the pipeline for more accurate optical effects. This method sets up a separate memory storage area called the *geometry buffer* (G-buffer) in the traditional rendering pipeline and implements the entire process in two passes. In the first *geometry pass*, geometric information, such as the vertex position, normal vector, and vertex color value for each pixel in the scene, is calculated and stored in the G-buffer. Shading calculations are performed in the second *lighting pass* using geometric information stored in the G-buffer.

The deferred rendering method enables efficient rendering even for numerous light sources, enabling real-time rendering of complex scenes. However, this method also has the drawback of requiring an additional G-buffer, which may require considerable memory space, especially for high resolution images. Additionally, there are still limitations in processing translucent objects and optics effects, such as reflection and refraction.

2.2 Ray-Tracing Methods

Ray tracing [9,10] is a widely used global illumination technique that simulates phenomena, such as light propagation, reflection, and refraction, for each pixel on the screen (Fig. 3). This method simulates real-world optics phenomena and enables high-quality rendering of translucent objects, shadow generation, and light diffraction. The light ray traversal process should be recursively repeated to simulate the optical phenomena accurately until there are no more colliding objects (Fig. 4), requiring a high computational cost.

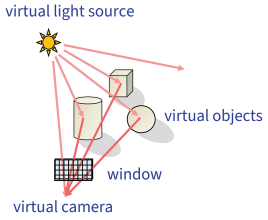


Figure 3: Ray-tracing model

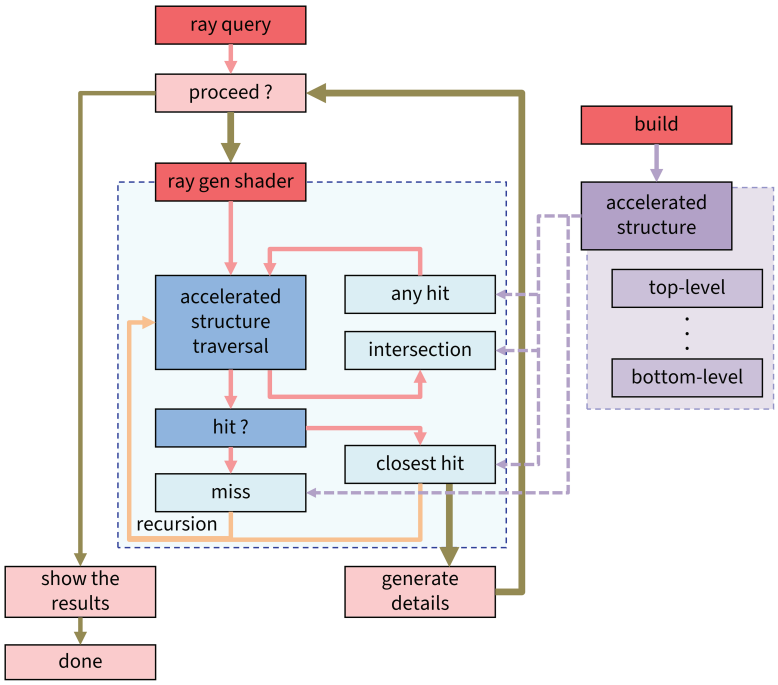


Figure 4: Typical ray tracing pipeline [23]

Many attempts have been made to reduce the computational costs of ray tracing methods. In the iteration loop in Fig. 4, the number of iterations can be limited to speed up the final calculation, although the rendering quality is slightly lower. During the operation process, most computation time is spent on *collision detection* between light rays and target objects. The computation speed can be improved by introducing 3D space partition data structures, such as the *bounding volume hierarchy* [24] and *k-dimensional tree* [25].

Along with the development of semiconductor and graphics processing unit (GPU) technology, the processing speed of ray tracing has dramatically increased. In particular, after the emergence of GPU-based massively parallel computing technology, such as *CUDA* [26–29] and *OpenCL* [30,31], ray tracing acceleration methods, such as *OptiX* [32,33], became available. These acceleration methods typically use general-purpose massively parallel computing hardware with a general-purpose GPU architecture, demonstrating that ray tracing applications can be significantly accelerated.

Modern graphics cards have recently begun to support ray tracing methods directly at the semiconductor chip level. The *Turing* architecture [12] by NVIDIA introduced dedicated ray tracing processors called RT cores, displaying much faster ray tracing processing speed. The second-generation AMD RDNA architecture [13] supports ray tracing-specific hardware. Such dedicated ray tracing hardware can be used, and research efforts for faster ray tracing processing continue.

2.3 Hybrid Rendering Approaches

Hybrid rendering approaches combine the fast processing speed of traditional local shading techniques with high-quality ray tracing techniques. In typical implementations, geometry processing, texture processing, and other methods have been implemented on the traditional rasterization pipeline. In contrast, advanced optics effects, such as light reflection and light refraction, have been

implemented with the ray tracing techniques, combining the computation results to produce the final image.

Various efforts have been made to reduce the computational costs of ray tracing processing. Heuristic hybrid rendering techniques [14,15] have introduced *selection graphs* and heuristic methods, focusing on objects with current visual effects. Today, many 3D rendering applications and commercial game engines have adopted these hybrid rendering techniques [34]. Vardis et al. [35] proposed hybrid rendering system named *DIRT* (deferred image-based ray tracing) to achieve high performance rendering, without modern hardware acceleration supports. Granja and Ao Pereira [36] proposed hybrid rendering system that focused on optimizing *SVGF* (Spatio-temporal Variance-Guided Filtering) to achieve real-time, photo-realistic rendering result.

Low-level graphics APIs, such as Vulkan [16,17], are preferred for practical implementations of hybrid rendering techniques. Vulkan was initially developed to implement the 3D graphics rendering pipeline more precisely and effectively, based on low-level graphics hardware. As the application of Vulkan has expanded, ray tracing extensions [37–40] have been added, allowing the *acceleration structure* [41] required for ray tracing to be set within the API or shader architecture in traditional 3D graphics pipelines to be conveniently implemented for ray tracing and other uses.

This paper suggests which implementation model is more appropriate when creating ray tracing applications with Vulkan and its ray tracing extensions. The traditional rendering pipeline and ray tracing extension can be used simultaneously in the Vulkan environment; therefore, this work designs and implements both models. The following sections present practical implementation models via practical benchmark results.

3 Main Idea

The traditional rendering pipeline and ray tracing techniques should be used simultaneously to implement a hybrid rendering method. In most cases, 3D graphics libraries that provide traditional rendering pipelines do not use ray tracing techniques. Thus, two or more independent libraries or independent underlying hardware systems should be used. As an example, in the hybrid approach, which combines OpenGL [42] and OptiX [32,33], OpenGL provides a traditional rendering pipeline. Additionally, OptiX implements a ray tracing function based on CUDA [26] and provides it as a separate API.

Thus, API-level connections are required in the typical hybrid rendering approach combining OpenGL and OptiX, and the independent calculation results are combined. In practical implementations, using the texture memory space available in OpenGL was standard, whereas communication overhead degrades performance. Additionally, OptiX only works on NVIDIA graphics cards; hence, a ray tracing implementation other than OptiX should be considered to support a wider variety of graphics cards or hardware.

Vulkan [17,19] provides low-level 3D graphics functions to replace the previous 3D API of OpenGL. Later, ray tracing extensions were introduced, integrating the traditional graphics pipeline and ray tracing functions in a single graphics library. From a hybrid rendering viewpoint, ray tracing results can be directly delivered to the frame buffer area in the traditional shading pipeline. Most newly released graphics hardware systems support Vulkan, regardless of the manufacturer; hence, it also has higher portability.

Based on these facts, this work employs the Vulkan library to implement a graphics pipeline with the deferred shading method, performing the necessary ray tracing calculations using ray tracing

extensions. Finally, the implementation directly stores the results in the frame buffer. From a hybrid rendering perspective, the final result can be synthesized using deferred shading, determining whether ray tracing techniques are necessary and activating the ray tracing extension functions only when necessary. The following sections present the design and practical implementation process.

3.1 Stand-Alone Ray-Tracing Implementation

Vulkan's ray tracing extensions preprocess target 3D objects and store them in the acceleration structure [41], a data structure dedicated to ray tracing processing (Fig. 4). A specific Vulkan function, vkCmdTraceRaysKHR$(\cdots)$, invokes the start of ray tracing processing. The ray tracing process traces all rays corresponding to the entire screen, and the ray tracing pipeline operates to calculate the ray tracing results for each pixel.

The ray tracing pipeline includes several cases for the ray-object intersection results, including *ray-generation*, *ray-hit*, *ray-miss*, and others. Vulkan programmers can implement which processing should be completed for each case using *compute shaders* and register them in advance as a callback function. The ray tracing pipeline traces all rays launched in the vkCmdTraceRaysKHR$(\cdots)$ function, uses the acceleration structure to determine which target object is intersected, and reflects at the intersection point according to optics phenomena, such as reflection and refraction. These reflection and refraction rays are traced recursively, as shown in Fig. 5.

Several kinds of shader programs are pre-registered to process these rays. The **ray-generation shader** works for each pixel to generate the initial ray and defines the initial information for the recursive invocation of the internal shader traceRay operation, as presented in Algorithm 1. Finally, this shader collects all processed results to determine the final pixel color.

Algorithm 1: Pseudocode of the ray generation shader

```
uniform variables: {globally accessible}
    topLevelAS : top-level acceleration structure (for geometry database)

function RAY-GENERATION SHADER MAIN {parallelly executed for each pixel}
    input: (u,v) : pixel coordinate
    output: pixel.color : pixel color for (u,v)
    ray ← generate a ray for (u,v) {initial ray}
    ray.dir ← direction of the ray
    node ← the closest geometry node from topLevelAS, intersecting the ray
    if the ray hit an object then {ray closest hit case}
        launch the ray-closest-hit shader for the ray
        pixel.color ← ray.color from the ray-closest-hit shader
    else {the ray does not hit any object, or ray miss case}
        launch the ray-miss shader for the ray
        pixel.color ← ray.color from the ray-miss shader
    end if
end function
```

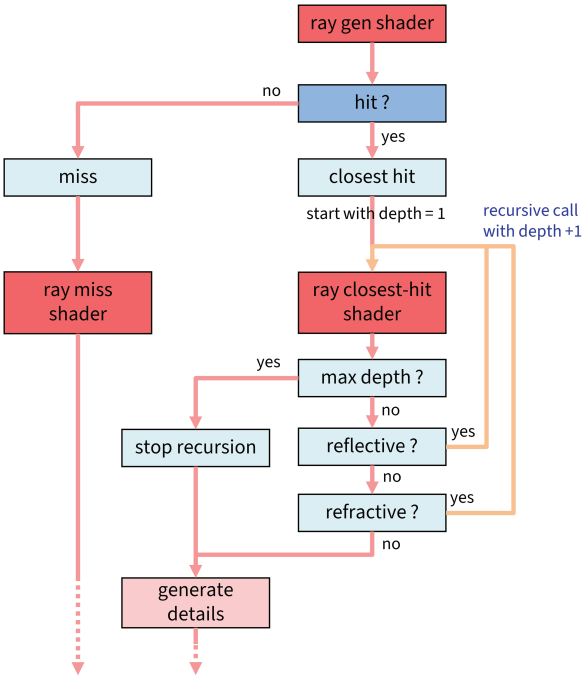


Figure 5: Recursive implementation of the ray tracing pipeline

The **ray-closest-hit** shader is called when a ray collides with an object, calculating the object intersection point and color values based on lighting and material properties and recursively generating additional reflection and refraction rays (see Algorithm 2). When initially invoking the ray tracing pipeline, the *maximum recursion depth value* can be predefined to control the quality of the ray tracing results.

Algorithm 2: Pseudocode of the ray closest hit shader

uniform variables: {globally accessible}
topLevelAS : top-level acceleration structure (for geometry database)
maxDepth : maximum depth for the recursive ray processing

function RAY-CLOSEST-HIT SHADER MAIN {parallelly executed for each ray}
input: *ray* : current ray
output: *ray.color* : color calculated for the ray
{terminate condition}
if *ray.depth* > *maxDepth* **then**
ray.color ← (0, 0, 0) {black color for depth pruning}
return
end if
ray.depth ← *ray.depth* + 1
{find and prepare the target object}
node ← closest geometry node from *topLevelAS*, intersecting the *ray*
object ← retrieve the object associated to the *node*

(Continued)

Algorithm 2 (continued)

```
    retrieve vertex positions, material properties, albedo maps, etc., from the node and object
    {ray tracing for the object}
     if object is opaque then
         for each light source do
             ray.color ← ray.color + the color value calculated with Phong shading
         end for
     end if
     if object is reflective then
        traceRay for the new reflection ray {implies ray-generation shader}
        ray.color ← ray.color + reflection-ray.color
     end if
     if object is refractive then
         traceRay for the new refraction ray {implies ray-generation shader}
         ray.color ← ray.color + refraction-ray.color
     end if
end function
```

The **ray-miss shader** is called when a ray does not hit an object, using the precalculated background colors from the *skybox* [43] or the background cube map [44] (see Algorithm 3). This study also implements the **shadow-closest-hit** shader and **shadow-miss** shader to process shadows. These shaders determine whether a shadow ray collides with an object to determine the shadow situation.

Algorithm 3: Pseudocode of the ray miss shader

```
uniform variables: {globally accessible}
    cubeMap : the background cube map

function RAY-MISS SHADER MAIN {parallelly executed for each ray}
    input: ray : current ray from the ray-generation shader
    output: ray.color : color calculated for the ray
    ray.color ← texture(cubeMap, ray.dir)
end function
```

3.2 Hybrid Rendering Implementation

Hybrid rendering methods are also implemented in the Vulkan environment. The proposed hybrid rendering method integrates the traditional shading and ray tracing pipelines; thus, the process is divided into two passes: the geometry and ray tracing passes. Fig. 6 presents the two-pass architecture.

In the geometry pass, the geometric information, such as vertex positions, normal vectors, albedo values, and material properties, are processed and stored in the G-buffer, as illustrated in Fig. 7. The *albedo* values are the intrinsic diffuse colors of materials, which are typically used to calculate lighting in the ray-tracing passes. These values determine whether an object is reflective or refractive, as well as the values for Phong shading calculations. In Fig. 7d, only the reflective and refractive objects have distinguished values in the material properties, for further ray tracing operations.

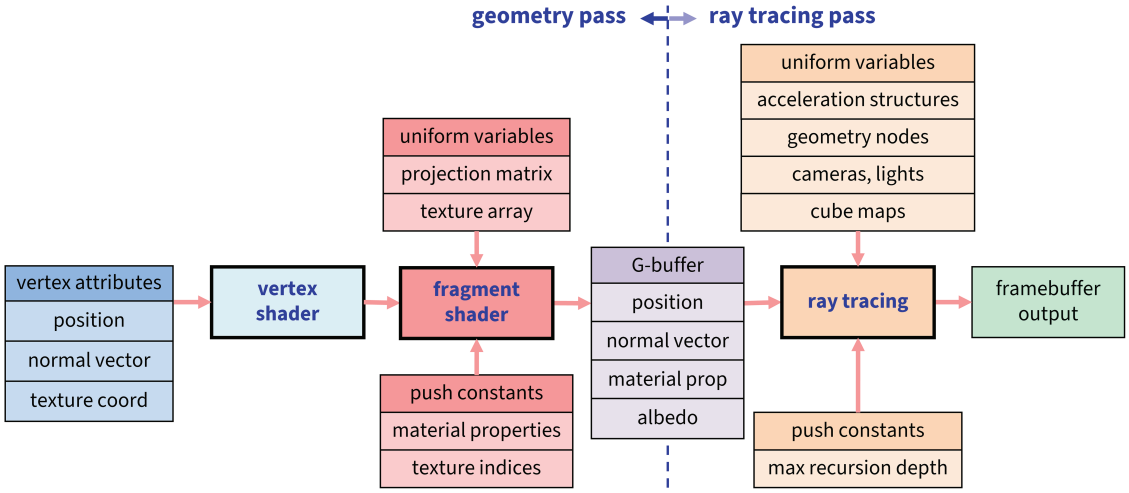
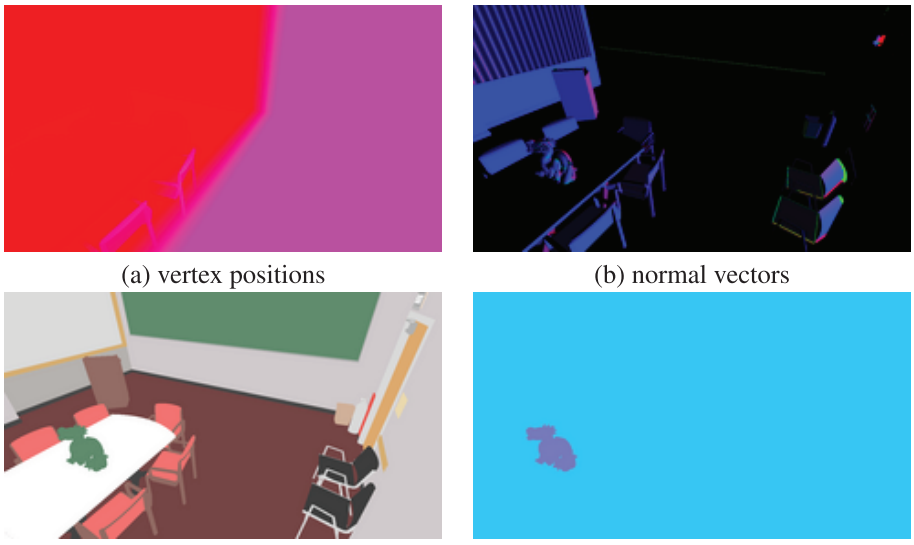


Figure 6: Overall pipeline of hybrid rendering



(a) vertex positions (b) normal vectors

(c) albedo values (d) material properties

Figure 7: Examples of G-buffer stored values

In the ray tracing pass, the geometric values in the G-buffer are retrieved, and depending on those values, the ray tracing operations or the traditional shading operations are selectively executed. More precisely, the internal traceRay operations are executed only when the target objects are reflective or refractive. The recursive ray tracing operations are significantly reduced; thus, considerable performance improvements are expected. In particular, if there are no reflective or refractive objects, traceRay operations are not executed; thus, a substantial speed improvement can be expected.

The geometry pass is implemented for shader programming with a pair of **vertex** and **fragment** shaders, similar to typical shading pipelines. The vertex properties of positions, normal vectors, and texture coordinates are passed in the vertex shader. Typical model, viewing, and projection transforms can be applied in this shader program, as presented in Algorithm 4.

Algorithm 4: Pseudocode of the vertex shader in the geometry pass

uniform variables: {globally accessible}
$\mathbf{M}_{\text{model}}$, $\mathbf{M}_{\text{view}}$, $\mathbf{M}_{\text{proj}}$: model, viewing and projection matrices for geometric transformation

function VERTEX SHADER MAIN {parallelly executed for each vertex}
input: *inPos*, *inNormal*, *inTexCoord*; {vertex position, normal vector, texture coordinate}
output: *outPos*, *outNormal*, *outTexCoord*;
requisite output: *gl_Position*; {built-in device coordinate}
gl_Position $\leftarrow \mathbf{M}_{\text{proj}} \times \mathbf{M}_{\text{view}} \times \mathbf{M}_{\text{model}} \times$ *inPos* {geometric transformation}
outPos $\leftarrow$ *inPos* {by-pass}
outNormal $\leftarrow$ *inNormal* {by-pass}
outTexCoord $\leftarrow$ *inTexCoord* {by-pass}
end function

In the fragment shader, material properties and texture index information are received through *push constants*, and these values are recalculated for each pixel to update the G-buffer (see Algorithm 5). Based on the G-buffer values, the next *ray tracing pass* is executed. The pre-stored G-buffer values can be used; hence, the overall ray tracing operations can be reduced once for all pixels. In the case of reflective and refractive objects, the internal traceRay operations are executed. As explained in the previous section, the typical ray tracing pipeline is activated with the **ray-generation** shader, **ray-closest-hit** shader, **ray-miss** shader, and others.

Algorithm 5: Pseudocode of the fragment shader in the geometry pass

uniform variables: {globally accessible}
$\mathbf{M}_{\text{model}}$, $\mathbf{M}_{\text{view}}$, $\mathbf{M}_{\text{proj}}$: model, viewing and projection matrices for geometric transformation
push constants: {for each object}
matProp : material properties (roughness, index of refraction, alpha value, $\cdots$)
albedoMap : texture map for albedo values

function FRAGMENT SHADER MAIN {parallelly executed for each pixel}
input: *pos*, *normal*, *texCoord*; {passed from vertex shaders}
output: *G-buf*; {G-buffer}
G-buf.pos $\leftarrow$ geometric transform(*pos*)
G-buf.normal $\leftarrow$ geometric transform(*normal*)
G-buf.albedo $\leftarrow$ texture(*albedoMap*, *texCoord*)
G-buf.matProp $\leftarrow$ *matProp* at the position *pos*
end function

The typical deferred rendering operations can be applied for non-reflective and non-refractive objects, and the Phong shading values can be calculated. These Phong-shaded pixel values are the final result for these objects, as in the traditional shading pipelines. The experimental results from the proposed implementations are presented in the next section.

4 Experimental Results

The Vulkan environment and its ray tracing extensions are used; thus, newly released graphics cards that support NVIDIA RT core features or AMD RDNA 2 features are used. The experimental system was configured as follows:

- CPU: AMD Ryzen 7 5800X 8-Core, 3.70 GHz
- GPU: NVIDIA GeForce RTX 3060
- RAM: 32 GB
- OS: Windows 11

Three computer animation sequences from widely used datasets were employed for the rendering experiments: *living room*, *conference*, and *gallery* [20]. Table 1 lists the geometric configurations for each animation sequence.

Table 1: Number of vertices and faces for each animation sequence

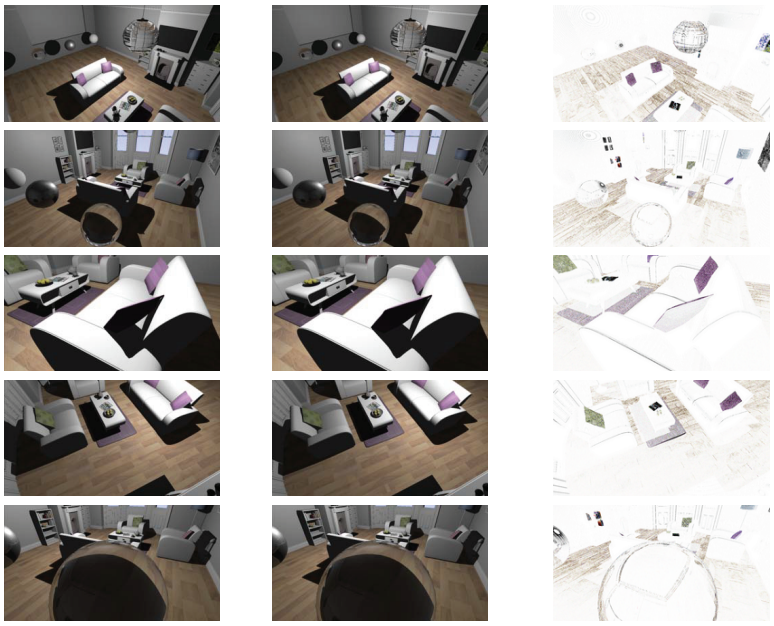
	Number of vertices	Number of faces
Living room	548,894	594,167
Conference	1,338,208	1,139,971
Gallery	648,417	998,831

This work rendered 4000 frames from these three animation sequences with five camera configurations. The stand-alone ray tracing and hybrid rendering methods were applied to produce 1280×720-pixel resolution image sequences. Five camera configurations with various positions or angles were set to vary the ratio of reflection and refraction objects for the scenes.

Figs. 8–10 present the *living room*, *conference*, and *gallery* rendering results, respectively. The final images from the stand-alone ray tracing and hybrid rendering methods were compared to check the correctness of the rendering results. First, the absolute differences in a pixel-by-pixel comparison were calculated to check for visual differences. For a clearer comparison, the colors of the difference images were inverted and multiplied by the factor of 32, as depicted in the rightmost columns of Figs. 8–10.

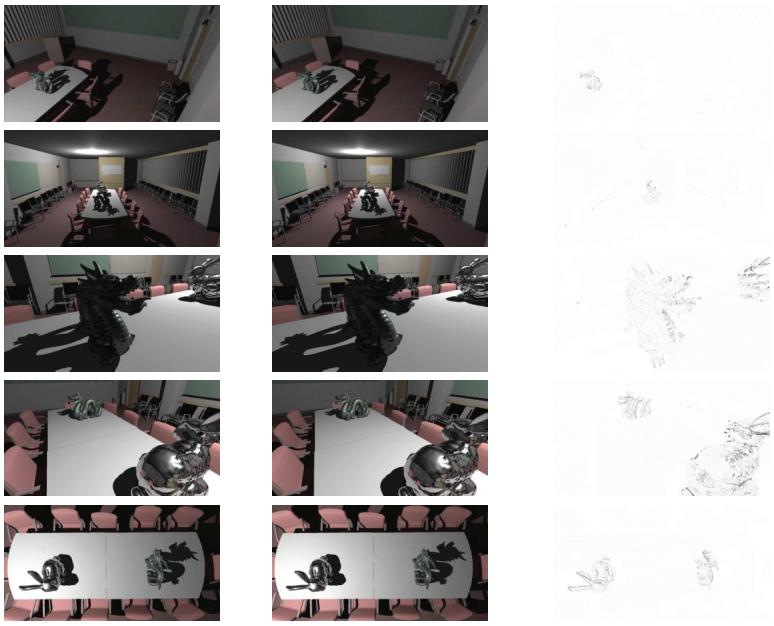
Table 2 presents the PSNR (peak signal-to-noise ratio) [45,46] and SSIM (structural similarity index measure) [47] values from the result images. The images present few structural differences, although pixel-value differences may exist. The PSNR values were calculated to confirm the objective difference between pixels. The PSNR was calculated based on the numerical difference between simple pixels; a larger value indicates higher quality. However, it does not directly reflect human visual perception. The SSIM values were also calculated to check the structural similarity between images. The SSIM values were calculated from brightness, contrast, and structure. A value closer to 1 indicates more similarity between two compared images. The SSIM values were greater than 0.940 in all animation sequences.

Regarding efficiency, the processing times for each frame and the resulting images were compared, determining the average, maximum, and minimum rendering times. The average rendering time was calculated as the average value of each rendering time from all 4000 frames. The maximum and minimum rendering times are also the maximum and minimum rendering times for all 4000 frames. The rendering times were measured using the vkCmdWriteTimeStamp$(\cdots)$ functions in Vulkan at the start and the end of the overall operations. The measured time values were converted to milliseconds for more clarity.



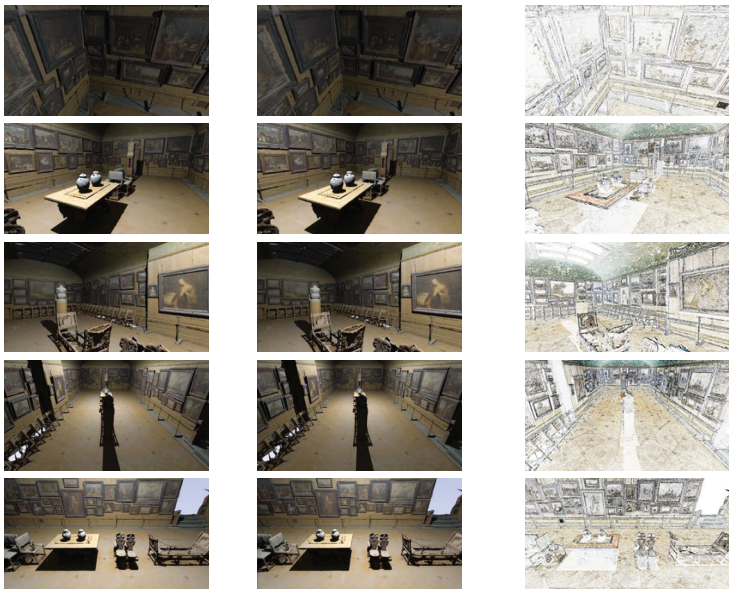
(a) standalone ray tracing (b) hybrid rendering (c) difference(inverted,×32)

Figure 8: Rendering results of the stand-alone ray tracing method, hybrid rendering method, and differences between the two methods for the *living room* animation sequence



(a) standalone ray tracing (b) hybrid rendering (c) difference(inverted,×32)

Figure 9: Rendering results of the stand-alone ray tracing method, hybrid rendering method, and differences between the two methods for the *conference* animation sequence



(a) standalone ray tracing (b) hybrid rendering (c) difference(inverted, $\times 32$)

Figure 10: Rendering results of the stand-alone ray tracing method, hybrid rendering method, and differences between the two methods for the *gallery* animation sequence

Table 2: PSNR and SSIM values from the comparison

Animation seq.	Camera config.	PSNR	SSIM
Living room	cam 1	45.2522	0.9858
	cam 2	45.8928	0.9902
	cam 3	46.6574	0.9892
	cam 4	46.6539	0.9899
	cam 5	47.0252	0.9910
	average	46.2963	0.9892
Conference	cam 1	61.3537	0.9996
	cam 2	60.5829	0.9994
	cam 3	50.9742	0.9954
	cam 4	48.1698	0.9892
	cam 5	54.4251	0.9986
	average	55.1011	0.9964
Gallery	cam 1	42.1382	0.9738
	cam 2	38.4030	0.9484
	cam 3	37.5145	0.9436
	cam 4	37.6100	0.9418
	cam 5	38.0405	0.9529
	average	38.7412	0.9521

Table 3 lists the average rendering times required for the hybrid and stand-alone ray tracing methods. In the hybrid rendering method, the ray tracing pass was accelerated, taking 39% to 64% times the time compared to the stand-alone ray tracing method. In contrast, the geometry pass takes more processing time, and due to this delay, the whole hybrid rendering method takes about 103% to 159% times with respect to the stand-alone ray tracing method.

Table 3: Average rendering times of the hybrid rendering and stand-alone ray tracing methods

(unit: msec)

Animation Sequence	Camera Config.	Standalone Raytracing Method (T)	Hybrid Rendering Method Raytracing Pass (t_a)	Time Ratio (t_a/T)	Geom. Preproc. (t_b)	Overall Proc. ($t = t_a + t_b$)	Time Ratio (t/T)
Living room	cam 1	0.577	0.269	46.6%	0.326	0.595	103.1%
	cam 2	0.621	0.316	50.9%	0.348	0.664	106.9%
	cam 3	0.527	0.207	39.3%	0.411	0.618	117.3%
	cam 4	0.491	0.212	43.2%	0.363	0.575	117.1%
	cam 5	1.047	0.672	64.2%	0.366	1.038	99.1%
	average	0.653	0.335	51.4%	0.363	0.698	107.0%
Conference	cam 1	0.546	0.270	49.5%	0.602	0.872	159.7%
	cam 2	0.530	0.250	47.2%	0.576	0.826	155.8%
	cam 3	0.915	0.589	64.4%	0.756	1.345	147.0%
	cam 4	0.845	0.496	58.7%	0.675	1.171	138.6%
	cam 5	0.708	0.394	55.6%	0.654	1.048	148.0%
	average	0.709	0.400	56.4%	0.653	1.052	148.5%
Gallery	cam 1	0.678	0.392	57.8%	0.576	0.968	142.8%
	cam 2	0.762	0.354	46.5%	0.657	1.011	132.7%
	cam 3	0.781	0.400	51.2%	0.646	1.046	133.9%
	cam 4	0.810	0.350	43.2%	0.699	1.049	129.5%
	cam 5	0.746	0.333	44.6%	0.645	0.978	131.1%
	average	0.755	0.366	48.4%	0.645	1.010	133.8%

Tables 4 and 5 list the minimum and maximum rendering times for the hybrid rendering and stand-alone ray tracing methods. Similar to the average rendering times, the hybrid rendering method takes more processing time than the stand-alone ray tracing method. The hybrid rendering method takes at most 230% times compared to the stand-alone ray tracing method for the maximum rendering time. This outcome reflects that the rendering characteristics of the hybrid rendering method are not entirely stable compared to the stand-alone ray tracing method.

Table 4: Best-case rendering times of the hybrid rendering and stand-alone ray tracing methods

(unit: msec)

Animation Sequence	Camera Config.	Standalone Raytracing Method (T)	Hybrid Rendering Method: Raytracing Pass (t_a)	Time Ratio (t_a/T)	Geom. Preproc. (t_b)	Overall Proc. ($t = t_a + t_b$)	Time Ratio (t/T)
Living room	cam 1	0.570	0.265	46.5%	0.322	0.587	103.0%
	cam 2	0.612	0.308	50.3%	0.345	0.653	106.7%
	cam 3	0.520	0.204	39.2%	0.405	0.609	117.1%
	cam 4	0.484	0.209	43.2%	0.356	0.565	116.7%
	cam 5	1.025	0.653	63.7%	0.362	1.015	99.0%
	average	0.642	0.328	51.0%	0.358	0.686	106.8%
Conference	cam 1	0.539	0.262	48.6%	0.593	0.855	158.6%
	cam 2	0.523	0.242	46.3%	0.571	0.813	155.4%
	cam 3	0.904	0.565	62.5%	0.749	1.314	145.4%
	cam 4	0.834	0.473	56.7%	0.671	1.144	137.2%
	cam 5	0.699	0.370	52.9%	0.649	1.019	145.8%
	average	0.700	0.382	54.6%	0.647	1.029	147.0%
Gallery	cam 1	0.673	0.384	57.1%	0.564	0.948	140.9%
	cam 2	0.754	0.350	46.4%	0.644	0.994	131.8%
	cam 3	0.773	0.385	49.8%	0.627	1.012	130.9%
	cam 4	0.804	0.338	42.0%	0.684	1.022	127.1%
	cam 5	0.740	0.330	44.6%	0.637	0.967	130.7%
	average	0.749	0.357	47.7%	0.631	0.989	132.0%

Table 5: Worst-case rendering times of the hybrid rendering and stand-alone ray tracing methods

(unit: msec)

Animation Sequence	Camera Config.	Standalone Raytracing Method (T)	Hybrid Rendering Method: Raytracing Pass (t_a)	Time Ratio (t_a/T)	Geom. Preproc. (t_b)	Overall Proc. ($t = t_a + t_b$)	Time Ratio (t/T)
Living room	cam 1	0.622	0.291	46.8%	0.347	0.638	102.6%
	cam 2	0.667	0.343	51.4%	0.372	0.715	107.2%
	cam 3	0.572	0.225	39.3%	0.450	0.675	118.0%
	cam 4	0.534	0.230	43.1%	0.388	0.618	115.7%
	cam 5	1.505	1.082	71.9%	0.390	1.472	97.8%
	average	0.780	0.434	55.7%	0.389	0.824	105.6%

(Continued)

Table 5 (continued)

(unit: msec)

Animation Sequence	Camera Config.	Standalone Raytracing Method (T)	Hybrid Rendering Method Raytracing Pass (t_a)	Time Ratio (t_a/T)	Geom. Preproc. (t_b)	Overall Proc. ($t = t_a + t_b$)	Time Ratio (t/T)
Conference	cam 1	0.598	0.292	48.8%	0.650	0.942	157.5%
	cam 2	0.580	0.271	46.7%	0.618	0.889	153.3%
	cam 3	0.978	1.018	104.1%	0.810	1.828	186.9%
	cam 4	0.917	0.929	101.3%	0.723	1.652	180.2%
	cam 5	0.773	0.817	105.7%	0.703	1.520	196.6%
	average	0.769	0.665	86.5%	0.701	1.366	177.6%
Gallery	cam 1	0.683	0.845	123.7%	0.737	1.582	231.6%
	cam 2	0.815	0.795	97.5%	0.699	1.494	183.3%
	cam 3	0.801	0.833	104.0%	0.694	1.527	190.6%
	cam 4	0.845	0.722	85.4%	0.695	1.417	167.7%
	cam 5	0.779	0.795	102.1%	0.691	1.486	190.8%
	average	0.785	0.798	101.7%	0.703	1.501	191.3%

We originally expected the hybrid method to be accelerated, and also the experimental results show sufficient accelerations in the ray tracing pass. However, the geometry pass, which was used as preprocessing, unexpectedly took a considerable amount of time, resulting in not much faster overall processing time. The implementation results demonstrate that the hybrid rendering method can accelerate ray tracing, but preprocessing may took an unexpectedly long time. In our future works, we highly expect that those preprocessing parts will be accelerated to significantly improve the overall processing time.

We also have additional experimental results for more wide range of graphics cards. Since the Vulkan ray-tracing extensions are supported by modern NVIDIA and AMD graphics cards, we choose NVIDIA GeForce RTX 2080 and AMD RX 6600, as additional target graphics cards. Tables 6 and 7 show the average rendering times for both of these graphics cards, with the same animation sequences and also the same camera configurations. As shown in these tables, the rendering times for the standalone raytracing and our hybrid method show the similar situation.

Table 8 presents the comparisons on the average rendering time for different graphics cards. Regarding their performance differences, instead of simple comparisons on the absolute rendering times, we focused on the ratio of processing time for both methods. For all three graphics cards, the execution time (t_a) of the raytracing pass is approximately 40% to 60% of the execution time (T) of the standalone raytracing method. On the other hand, the geometric preprocessing time (t_b) is higher in relatively-new graphics cards, while it is lower, about 25% to 30%, in the relatively-old AMD RX 6600.

Considering that there has been no significant technical improvements for the geometric preprocessing operations, it can be reversely interpreted. Although previous graphics cards required significantly more time for ray tracing operations than for geometric preprocessing, recent technical

improvements in ray tracing implementations have clearly reduced its processing time. Thus, the advantage of the hybrid rendering method is reduced, as the ray-tracing implementations are improved. The final conclusions are followed in the next section.

Table 6: Average rendering times for NVIDIA GeForce RTX 2080

(unit: msec)

Animation Sequence	Camera Config.	Standalone Raytracing Method (T)	Hybrid Rendering Method: Raytracing Pass (t_a)	Time Ratio (t_a/T)	Geom. Preproc. (t_b)	Overall Proc. ($t = t_a + t_b$)	Time ratio (t/T)
Living room	cam 1	0.529	0.261	49.3%	0.296	0.557	105.3%
	cam 2	0.588	0.310	52.6%	0.299	0.609	103.5%
	cam 3	0.439	0.189	43.0%	0.376	0.565	128.7%
	cam 4	0.430	0.195	45.4%	0.326	0.521	121.4%
	cam 5	0.912	0.591	64.8%	0.318	0.909	99.7%
	average	0.580	0.309	53.3%	0.323	0.632	109.1%
Conference	cam 1	0.505	0.279	55.3%	0.502	0.782	154.6%
	cam 2	0.488	0.256	52.5%	0.473	0.729	149.5%
	cam 3	0.989	0.682	69.0%	0.559	1.241	125.5%
	cam 4	0.797	0.531	66.6%	0.564	1.095	137.3%
	cam 5	0.683	0.405	59.3%	0.557	0.962	140.9%
	average	0.692	0.431	62.2%	0.531	0.962	138.9%
Gallery	cam 1	0.593	0.331	55.9%	0.409	0.740	124.8%
	cam 2	0.792	0.317	40.0%	0.476	0.793	100.1%
	cam 3	0.806	0.343	42.5%	0.471	0.813	100.9%
	cam 4	0.885	0.319	36.0%	0.489	0.808	91.3%
	cam 5	0.777	0.302	38.8%	0.470	0.772	99.4%
	average	0.771	0.322	41.8%	0.463	0.785	101.9%

Table 7: Average rendering times for AMD RX 6600

(unit: msec)

Animation Sequence	Camera Config.	Standalone Raytracing Method (T)	Hybrid Rendering Method: Raytracing Pass (t_a)	Time Ratio (t_a/T)	Geom. Preproc. (t_b)	Overall Proc. ($t = t_a + t_b$)	Time Ratio (t/T)
Living room	cam 1	2.079	0.924	44.5%	0.515	1.439	69.2%
	cam 2	2.123	1.069	50.3%	0.507	1.576	74.2%
	cam 3	1.421	0.590	41.5%	0.561	1.151	81.0%
	cam 4	1.447	0.612	42.3%	0.536	1.149	79.4%
	cam 5	3.133	2.085	66.6%	0.497	2.582	82.4%
	average	2.041	1.056	51.8%	0.523	1.579	77.4%
Conference	cam 1	1.955	0.922	47.1%	0.797	1.719	87.9%
	cam 2	1.901	0.800	42.1%	0.793	1.592	83.8%
	cam 3	3.843	2.238	58.2%	0.874	3.112	81.0%
	cam 4	3.209	1.819	56.7%	0.865	2.684	83.7%
	cam 5	2.584	1.299	50.2%	0.833	2.132	82.5%
	average	2.698	1.415	52.5%	0.832	2.248	83.3%
Gallery	cam 1	2.313	1.110	48.0%	0.659	1.769	76.5%
	cam 2	2.910	1.193	41.0%	0.933	2.126	73.1%
	cam 3	3.016	1.262	41.8%	0.919	2.180	72.3%
	cam 4	3.157	1.231	39.0%	0.972	2.203	69.8%
	cam 5	2.866	1.126	39.3%	0.900	2.026	70.7%
	average	2.852	1.184	41.5%	0.876	2.061	72.2%

Table 8: Comparing average rendering times for different graphics cards

(unit: msec)

Graphics Card	Animation Sequence	Standalone Raytracing method (T) Ratio	Hybrid Rendering Method Raytracing pass (t_a) (t_a/T)	Geom. Preproc. (t_b) (t_b/T)	Overall Proc. ($t = t_a + t_b$) (t/T)
NVIDIA	*Living room*	0.653	0.335	0.363	0.698
GeForce		ratio	51.36%	55.59%	106.96%
RTX	*Conference*	0.709	0.400	0.653	1.052
3060		ratio	56.41%	92.07%	148.48%
	Gallery	0.755	0.366	0.645	1.010
		ratio	48.42%	85.33%	133.76%
NVIDIA	*Living room*	0.580	0.309	0.323	0.632
GeForce		ratio	53.31%	55.77%	109.08%
RTX	*Conference*	0.692	0.431	0.531	0.962
2080		ratio	62.20%	76.71%	138.90%
	Gallery	0.771	0.322	0.463	0.785
		ratio	41.83%	60.07%	101.90%
AMD	*Living room*	2.041	1.056	0.523	1.579
RX		ratio	51.76%	25.63%	77.40%
6600	*Conference*	2.698	1.415	0.832	2.248
		ratio	52.45%	30.85%	83.30%
	Gallery	2.852	1.184	0.876	2.061
		ratio	41.52%	30.73%	72.24%

5 Conclusions and Future Work

The current ray tracing implementations may require practical implementation guidelines and experimental results. This work proposes a hybrid rendering method using the Vulkan library and its ray tracing extensions. This study compared the rendering results and performance measures for the stand-alone ray tracing and hybrid rendering methods.

As shown in our experimental results, we achieved that the ray tracing pass itself is sufficiently accelerated. In most cases, the rendering performance for the ray tracing part was improved to be 39% to 64% of the original processing time. In contrast, the extra preprocessing of the geometry pass takes an unexpectedly long time. We found that the latest NVIDIA RTX graphics cards are assumed to support increased ray tracing accelerations at the hardware level. The resources dedicated to traditional local shading pipelines may be reduced to achieve this acceleration and act as a bottleneck, which is expected to be improved in our future works.

The Vulkan ray tracing extensions are available only on the latest graphics cards; hence, the ray tracing method is expected to be slower on most mobile devices and low-tier graphics cards. Depending on the technological evolutions and the manufacturers of the specific graphics card, its performance results may vary. In particular, relatively old or low-tier graphics cards may show that the hybrid rendering method is still expected to perform more effectively. In contrast, the hardware raytracing implementations are rapidly improving, and it should be carefully checked whether the latest raytracing hardware implementations clearly show better performances than the hybrid methods.

The experiments found that the number of traceRay operations is a significant factor in the overall rendering time. In addition, a considerable performance improvement can be achieved in ray tracing by appropriately adjusting the recursive depth using various heuristics. In the future, more effective heuristics are expected based on the presented experimental results for hybrid rendering methods. We can also expect to achieve performance gains through geometric preprocessing steps, including *frustum culling* and *occlusion culling*. Those optimizations from a geometric point of view would also be future work.

Acknowledgement: Not applicable.

Funding Statement: This work was supported by the IITP (Institute of Information & Communications Technology Planning & Evaluation)-ITRC (Information Technology Research Center) grant funded by the Korea government (Ministry of Science and ICT) (IITP-2025-RS-2024-00437756, 90%). This study was supported by the BK21 FOUR project (AI-Driven Convergence Software Education Research Program) funded by the Ministry of Education, School of Computer Science and Engineering, Kyungpook National University, Republic of Korea (41202420214871, 10%).

Author Contributions: The authors confirm contribution to the paper as follows: study conception and design: Mingyu Kim, Nakhoon Baek; data collection: Mingyu Kim; analysis and interpretation of results: Mingyu Kim, Nakhoon Baek; draft manuscript preparation: Mingyu Kim, Nakhoon Baek. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: The datasets generated or analyzed during the current study are available from the corresponding author on reasonable request.

Ethics Approval: Not applicable.

Conflicts of Interest: The authors declare no conflicts of interest to report regarding the present study.

References

1. Gregory J. Game engine architecture. 3rd ed. Boca Raton, FL, USA: CRC Press; 2018.
2. Juliani A, Berges V, Vckay E, Gao Y, Henry H, Mattar M, et al. Unity: a general platform for intelligent agents. arXiv:1809.02627. 2018.
3. Laine S, Karras T. High-performance software rasterization on GPUs. In: Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics (HPG '11). New York, NY, USA: ACM; 2011. p. 79–88.
4. Baek N, Kim K. Design and implementation of OpenGL SC 2.0 rendering pipeline. Cluster Comput. 2019;22:931–6. doi:10.1007/s10586-017-1111-1.
5. Kenzel M, Kerbl B, Schmalstieg D, Steinberger M. A high-performance software graphics pipeline architecture for the GPU. ACM Trans Graph. 2018;37:140:1–15. doi:10.1145/3197517.3201374.

6. Kajiya JT. The rendering equation. In: Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques. New York, NY, USA: ACM; 1986. p. 143–50.
7. Phong BT. Illumination for computer generated pictures. Commun ACM. 1975;18(6):311–7. doi:10.1145/360825.360839.
8. Deering M, Winner S, Schediwy B, Duffy C, Hunt N. The triangle processor and normal vector shader: a VLSI system for high performance graphics. SIGGRAPH Comput Graph. 1988;22(4):21–30. doi:10.1145/378456.378468.
9. Whitted T. An improved illumination model for shaded display. Commun ACM. 1980;23(6):343–9. doi:10.1145/358876.358882.
10. Shirley P, Morley RK. Realistic ray tracing. 2nd ed. Boca Raton, FL, USA: Taylor & Francis; 2008.
11. Goral CM, Torrance KE, Greenberg DP, Battaile B. Modeling the interaction of light between diffuse surfaces. In: Proceeding of the 11th Annual Conference on Computer Graphics and Interactive Techniques. SIGGRAPH '84. New York, NY, USA: ACM; 1984. p. 213–22.
12. NVIDIA. NVIDIA turing GPU arhictecture. 2788 San Tomas Expressway, Santa Clara, CA, USA: NVIDIA; 2018.
13. Advanced Micro Devices. AMD RDNA performance guide. 2024 [cited 2025 Oct 12]. Available from: https://gpuopen.com/learn/rdna-performance-guide/.
14. Walewski P, Gałaj T, Szajerman D. Heuristic based real-time hybrid rendering with the use of rasterization and ray tracing method. Open Phys. 2019;17(1):527–44. doi:10.1515/phys-2019-0055.
15. Sabino TL, Andrade P, Clua EWG, Montenegro A, Pagliosa P. A hybrid GPU rasterized and ray traced rendering pipeline for real time rendering of per pixel effects. In: Herrlich M, Malaka R, Masuch M, editors. Entertainment computing-ICEC 2012. Berlin/Heidelberg, Germany: Springer Berlin Heidelberg; 2012. p. 292–305. doi:10.1007/978-3-642-33542-6_25.
16. The Khronos Vulkan Working Group. Vulkan guide. Beavertgon, OR, USA: Khronos Group; 2022.
17. The Khronos Vulkan Working Group. Vulkan—a specification, version 1.4.325. Beavertgon, OR, USA: Khronos Group; 2025.
18. Kenwright B. Getting started with computer graphics and the vulkan API. In: SA '17: SIGGRAPH Asia 2017 Courses. New York, NY, USA: ACM; 2017. doi:10.1145/3134472.3136556.
19. Bailey M. What educators need to know about where openGL is and where it is going. J Comput Sci Coll. 2015;31(1):161–6.
20. McGuire M. Computer graphics archive. 2025 [cited 2025 Oct 12]. Available from: https://casual-effects.com/data.
21. NVIDIA. Unreal engine 5.4 raytracing guide. 2788 San Tomas Expressway, Santa Clara, CA, USA: NVIDIA; 2024.
22. Ye K, Hou Q, Zhou K. 3D gaussian splatting with deferred reflection. In: ACM SIGGRAPH 2024 Conference Papers. SIGGRAPH '24. New York, NY, USA: ACM; 2024. doi:10.1145/3641519.3657456.
23. Khronos Blog. Ray tracing in Vulkan; 2025 [cited 2025 Oct 12]. Available from: https://www.khronos.org/blog/ray-tracing-in-vulkan.
24. Wang C, Wang Y, Li Y, Huang J, Huang J, Wang CX. An improved ray tracing acceleration algorithm based on bounding volume hierarchies. In: 2022 IEEE 96th Vehicular Technology Conference (VTC2022-Fall). Piscataway, NJ, USA: IEEE; 2022. p. 1–6.
25. Zhou J, Wen D. Research on ray tracing algorithm and acceleration techniques using KD-tree. In: 2021 6th International Conference on Intelligent Computing and Signal Processing (ICSP). Piscataway, NJ, USA: IEEE; 2021. p. 107–10.
26. NVIDIA. CUDA toolkit documentation, version 13.0. 2788 San Tomas Expressway, Santa Clara, CA, USA: NVIDIA; 2025.

27. Kirk D. NVIDIA CUDA software and GPU parallel computing architecture. In: Proceeding of the 6th International Symposium on Memory Management (ISMM '07). New York, NY, USA: ACM; 2007. p. 103–4.
28. Baek N. A prototype implementation of a CUDA-based customized rasterizer. Int J Adv Comput Sci Appl. 2022;13(8):776–81. doi:10.14569/ijacsa.2022.0130889.
29. Shin W, Baek N. Solar irradiance prediction model with recurrent neural networks and computer graphics methods. Rev Int Métodos Numér Cálc Diseño Ing. 2024;40(4):52.
30. Khronos OpenCL Working Group. The OpenCL Specification, version 3.0. Beavertgon, OR, USA: Khronos Group; 2022.
31. Stone JE, Gohara D, Shi G. OpenCL: a parallel programming standard for heterogene-ous computing systems. Comput Sci Eng. 2010;12(3):66–73. doi:10.1109/mcse.2010.69.
32. Parker SG, Bigler J, Dietrich A, Friedrich H, Hoberock J, Luebke D, et al. OptiX: a general purpose ray tracing engine. ACM Trans Graph. 2010;29(4):1–13.
33. Ludvigsen H, Elster AC. Real-time ray tracing using NVIDIA OptiX. In: Proceeding of the Eurographics '10; 2010 Jul 2–4; Madrid, Spain. p. 65–8.
34. Park H, Baek N. Design and implementation of style-transfer operations in a game engine. Int J Adv Comput Sci Appl. 2024;15(8):1265–73. doi:10.14569/ijacsa.2024.01508123.
35. Vardis K, Vasilakis AA, Papaioannou G. DIRT: deferred image-based ray tracing. In: Proceedings of High Performance Graphics. HPG '16. Goslar, Germany: Eurographics Association; 2016. p. 63–73.
36. Granja P, Ao Pereira J. Hybrid-rendering techniques in GPU. arXiv:2312.06827. 2023.
37. The Khronos Vulkan Working Group. Vulkan KHR ray tracing pipeline-device extension. Beavertgon, OR, USA: Khronos Group; 2020.
38. The Khronos Vulkan Working Group. Vulkan KHR ray query-device extension. Beavertgon, OR, USA: Khronos Group; 2020.
39. The Khronos Vulkan Working Group. Vulkan KHR pipeline library-device extension. Beavertgon, OR, USA: Khronos Group; 2020.
40. The Khronos Vulkan Working Group. Vulkan KHR deferred host operations-device extension. Beavertgon, OR, USA: Khronos Group; 2020.
41. The Khronos Vulkan Working Group. Vulkan KHR acceleration structure-device extension. Beavertgon, OR, USA: Khronos Group; 2021.
42. Segal M, Akeley K. The OpenGL graphics system: a specification, version 4.6. Beavertgon, OR, USA: Khronos Group; 2019.
43. Fernando R, Kilgard MJ. The Cg tutorial: the definitive guide to programmable real-time graphics. Boston, MA, USA: Addison-Wesley; 2003.
44. Greene N. Environment mapping and other applications of world projections. IEEE Comput Graph Applicat. 1986;6:21–9.
45. Horé A, Ziou D. Image quality metrics: PSNR vs. SSIM. In: 2010 20th International Conference on Pattern Recognition; 2010 Aug 23–26; Istanbul, Turkey. p. 2366–9.
46. Huynh-Thu Q, Ghanbari M. Scope of validity of PSNR in image/video quality assessment. Electr Lett. 2008;44(13):800–1. doi:10.1049/el:20080522.
47. Wang Z, Bovik AC, Sheikh HR, Simoncelli EP. Image quality assessment: from error visibility to structural similarity. IEEE Trans Image Process. 2004;13(4):600–12. doi:10.1109/tip.2003.819861.