Journal of Theoretical and Applied Information Technology
31st May 2023. Vol.101. No 10

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

REAL-TIME RAY TRACING REFLECTIONS AND SHADOWS IMPLEMENTATION USING DIRECTX RAYTRACING

YOUNGSIK KIM
Dept. of Game and Multimedia Engineering, Tech University of Korea, Republic of Korea
E-mail: kys@tukorea.ac.kr (corresponding author)

ABSTRACT

In traditional 3D games, techniques such as environment mapping and shadow mapping were used to simulate reflections and shadows due to the high computational load of ray tracing. However, recent advancements in technology have made real-time ray tracing possible, allowing for higher quality reflections and shadows compared to traditional methods. This paper proposes DirectX Raytracing (DXR) to achieve high-quality real-time reflections and shadows. To reduce the computational load of real-time ray tracing, we use the G-buffer from deferred rendering to compute only the information required for shadows and reflections, which is then combined to generate the final color. This paper verifies the effectiveness of our approach by comparing the performance of a DXR-based program with images produced using Unreal Engine 4's ray tracing capabilities. The results show that the proposed method provides high-quality graphics while minimizing computational load.

Keywords: *Reflection, Shadow, Real-time Raytracing, DirectX Raytracing, Performance Evaluation*

1. INTRODUCTION

With Recent advancements in computer hardware, particularly in graphics accelerators, have enabled the implementation of higher quality game graphics. Reflection and shadows are essential components for achieving high-quality game graphics, and various techniques have emerged to implement these elements in games. While ray tracing, which involves tracing the movement of light, is a simple and powerful way to implement reflections and shadows, the computational load of ray tracing is currently too high for real-time implementation of these effects using GPU hardware. As a result, other alternative methods have been used to implement reflections and shadows.

In [1], researchers describe using shadow maps to implement real-time shadows. Shadow maps are textures that store information about which parts of a scene are in shadow and which parts are illuminated. By using shadow maps, the computational cost of shadow generation is greatly reduced, making it possible to implement real-time shadows in games. In [2], researchers describe using environment maps to implement real-time reflections in a rendering environment. Environment maps are images that capture the surrounding environment from a single viewpoint, and they can be used to simulate the reflection of light off of surfaces. This technique allows for the creation of realistic reflections in real-time rendering environments. In [3], researchers describe using screen-space reflections to implement real-time reflections. Screen-space reflections involve rendering the scene from the perspective of the camera, and then using that image to determine how light would reflect off of surfaces in the scene. This technique is particularly useful for reflections of dynamic objects, such as characters, as it allows for more accurate reflections in real-time rendering environments.

Recent attempts to implement high-quality game graphics have increased as computer performance, including graphic accelerators, has improved. In particular, reflections and shadows are essential elements in implementing high-quality game graphics, and various techniques have been introduced to implement them in games. Although ray tracing is a simple and powerful method to implement shadows and reflections by tracing the movement of light, real-time implementation of ray tracing for shadows and reflections using existing GPU hardware is challenging due to the high computational load involved. Therefore, several other techniques have been employed to implement shadows and reflections using methods other than ray tracing. However, attempts to apply more straightforward yet powerful real-time ray tracing in rendering have continued. Nvidia has demonstrated improved ray tracing performance by releasing graphics cards with dedicated hardware for ray tracing [4]. In addition, graphics APIs such as

DirectX and Vulkan from MICROSOFT have added APIs for ray tracing, and multiple support for real-time ray tracing has been added.

The paper [5] discusses a hybrid rendering approach that combines rasterization and ray tracing to achieve real-time performance with high-quality visuals. It proposes a two-step process where geometry is first rendered using traditional rasterization techniques, followed by ray tracing for the accurate calculation of shadows, reflections, and global illumination. The authors in [5] also discuss various optimization techniques to improve performance, such as dynamic geometry tessellation and light culling. The paper [5] provides a comprehensive overview of the hybrid rendering approach and demonstrates its effectiveness in real-world applications such as video game development.

The paper [6] presents a comprehensive overview of the deferred rendering technique and its use in modern real-time graphics pipelines. The paper [6] explains the concept of deferred rendering, its advantages, and limitations, and compares it to other rendering techniques. It also covers various optimization strategies and implementation details such as G-buffer organization, lighting models, and antialiasing methods. The paper [6] concludes with a discussion of the challenges and future directions for deferred rendering in the context of emerging graphics hardware and applications.

The article [7] proposes a technique for deferred shading that allows for rendering multiple light sources with a single pass. The technique involves rendering the scene geometry into multiple render targets and storing additional data about each pixel's depth, normals, and material properties. These multiple render targets are then used in a subsequent pass to compute lighting calculations for each pixel, improving performance and allowing for more realistic lighting effects. The article [7] includes code examples and implementation details for the proposed technique.

The paper [8] discusses the implementation of cinematic rendering using real-time ray tracing and denoising in Unreal Engine 4. The authors in [8] describe how they leveraged the DXR API to enable real-time ray tracing in the engine, and discuss their approach to denoising to improve the visual quality of the rendered images. The authors in [8] also discuss the challenges they faced in implementing these techniques, including performance limitations and the need for specialized hardware. Finally, they present several examples of cinematic rendering using these techniques to demonstrate their effectiveness in creating high-quality, photorealistic images.

The paper [9] presents a new acceleration structure called Ray-Specialized Bounding Volume Hierarchy (RSBVH) for efficient ray tracing of complex scenes. The RSBVH structure partitions the scene into regions that are specific to each ray, enabling faster traversal of the BVH tree by culling irrelevant parts of the scene. The authors in [9] also propose a new scheme for building the RSBVH structure using hierarchical clustering and iterative refinement. The results in [9] show that RSBVH outperforms existing methods for both static and dynamic scenes.

The paper [10] proposes a method to create 3D scenes from 2D images by combining machine learning and ray tracing. To achieve this, a new data structure called Neural Radiance Field (NeRF) is developed. NeRF is a framework for representing complex 3D scenes as continuous functions learned by a neural network. By modeling a scene as a radiance field, NeRF can estimate the appearance of any view of the scene by integrating the radiance along a ray. This allows for highly realistic rendering of novel views and enables applications such as free-viewpoint video, 3D scene reconstruction, and augmented reality. The key challenge in training NeRF is to optimize the network to represent fine-scale details of the scene while maintaining a tractable computational complexity.

The paper [11] proposes a real-time algorithm to compute ambient occlusion for complex scenes using spatial hashing. The approach combines the benefits of ray tracing with the efficiency of spatial hashing to generate high-quality results without sacrificing performance. The algorithm is designed to work on modern hardware, including GPUs, and is capable of handling dynamic scenes in real-time. The paper [11] also presents several optimizations to further improve the performance and quality of the algorithm, making it suitable for use in games and other interactive applications.

The paper [12] presents a new ray-tracing tool, named DELSOL3, which is designed for simulating heliostat fields used in solar thermal power plants. DELSOL3 aims to improve the accuracy and efficiency of heliostat field simulations by incorporating new algorithms for handling the complexity of the heliostat field geometry, such as the introduction of a hierarchical bounding box structure. The authors in [12] demonstrate the improved accuracy and performance of DELSOL3

compared to other ray-tracing tools through a series of simulations and comparisons.

The paper [13] proposes a differentiable Monte Carlo ray tracing method using edge sampling, which allows for the computation of gradients of ray tracing images, enabling a wide range of applications such as optimizing the position of light sources, materials and geometry. By applying the method to the path tracing and photon mapping algorithms, the authors demonstrate that their approach can achieve comparable or even better results compared to traditional non-differentiable methods, while offering the flexibility of end-to-end optimization. The proposed method in [13] is evaluated on a variety of test scenes and is shown to be effective in generating high-quality images with reduced noise.

Unreal Engine 4's ray tracing is a software implementation that runs on both Nvidia and AMD GPUs. It uses the CPU to set up acceleration structures and uses the GPU to trace rays. DirectX Raytracing (DXR) is a hardware-based ray tracing solution that runs on compatible GPUs. It has lower overhead, faster setup times, and can handle more complex scenes. DXR can also be used with Unreal Engine 4 to achieve hardware-accelerated ray tracing.

This paper describes the implementation of real-time shadows and reflections using DXR, a ray tracing API for DirectX, along with traditional rasterization rendering. In Section 2, the paper provides an overview of various terms and concepts related to using DXR. In Section 3, the paper explains the overall rendering process using ray tracing and the methods for implementing shadows and reflections with ray tracing. The paper also describes how the resulting images are combined to create the final image. Section 4 compares the actual implementation and performance differences with the commercial engine Unreal Engine 4, and Section 5 concludes the paper.

2. BACKGROUNDS

2.1 DirectX Raytracing (DXR)

DirectX is a graphics API developed by Microsoft, which provides APIs suitable for real-time ray tracing and enables GPU-accelerated ray tracing. In DXR, an extension of the graphics API DirectX, a data structure called the acceleration structure is provided for efficient ray intersection testing. There are two types of acceleration structures: the bottom-level acceleration structure, which represents geometry such as polygons, and the top-level acceleration structure, which places the geometry represented by the bottom-level acceleration structure into the actual scene. Each acceleration structure is built and managed in an optimized structure through the DXR API, enabling fast ray collision detection during the ray tracing stage.

The bottom-level acceleration structure is a structure that defines shapes using actual vertices or parameter equations, and is slow to create and update but has the advantage of fast collision detection. The top-level acceleration structure is a structure that defines which bottom-level acceleration structure an object uses, which hit group it uses, and where it is located. It is fast to create and update.

In DXR, the ray generation shader is a programmable shader stage that allows for the purposeful generation of rays and the ability to reflect the results of ray collisions onto textures. This shader stage provides the flexibility to generate custom rays to suit specific rendering needs and enables the creation of dynamic lighting and shadow effects in real-time rendering applications. By defining and implementing the ray generation shader in DXR, developers can harness the power of hardware acceleration and real-time ray tracing to achieve stunning visual effects and improved performance in graphical applications.

The hit shader is a shader stage used in DXR that allows programming of how the ray should be processed when it collides with an acceleration structure. Depending on the type of collision, there are three types of hit shaders: closest hit shader that handles the closest collision, any hit shader that handles all objects hit by the ray, and intersection shader that handles intersection with boundaries. Through hit shaders, it is possible to program how the result of the collision should be reflected in the final image.

In DXR, a hit group is a group created by combining hit shaders, and it exists continuously in the shader table. When a ray collision occurs, the object can specify which hit shader to invoke within the same hit group. Hit groups allow developers to organize and manage the invocation of hit shaders efficiently during ray tracing, and they are defined by specifying the closest hit shader, any hit shader, and/or intersection shader. By using hit groups, developers can define and group together multiple shaders that are executed together during the ray tracing process. This enables more efficient

management of resources and control over the execution of shaders.

Miss shader is a shader stage in DXR that allows for programming what happens when a ray does not hit any geometry in the scene. This shader is called after all hit shaders have been executed for a ray and can be used to set the color or transparency of the background, simulate atmospheric effects, or generate procedural patterns. The miss shader can be associated with a miss shader table, which is a set of miss shaders that can be selected by ray tracing programs.

A shader table is a data structure used to pass information on which resources to use for ray generation, hit, and miss shaders to the DXR internally. The actual shader table is an array in GPU memory that contains identifiers to identify the shaders and handles to the resources that will be used by the shaders, as well as GPU memory addresses. In the Dispatch function that executes the ray tracing stage, the size of each element in the table and the starting GPU memory address of the table are passed as parameters, and the shader table to be used for each shader - ray generation, hit group, and miss - can be specified. In particular, for the hit group shader table, the hit group to be used is determined by the hit group number in the top-level acceleration structure, and the offset is specified in the ray generation function of the shader code to specify which hit shader to call.

A state object is a data structure in DirectX Raytracing (DXR) that replaces the Pipeline State Object in traditional DirectX. It informs the system which shaders to use and allows for the configuration of various options required for ray tracing. These options include but are not limited to, the acceleration structure used, the shader tables, the depth stencil state, the rasterizer state, and the blend state. The state object is created by specifying the desired configuration and then compiling it into a binary format that can be loaded into the GPU memory. It is then passed to the device during execution to set up the necessary states for the ray tracing process.

Hybrid ray tracing is a technique that combines rasterization and ray tracing in order to handle the entire rendering process. This approach is necessary because the computational demands of rendering everything through ray tracing can be very high. According to reference [5], this approach involves using rasterization for G-buffer rendering and utilizing compute shaders for direct lighting, while using ray tracing to implement shadows, reflections, indirect lighting, ambient occlusion, and transparent objects.

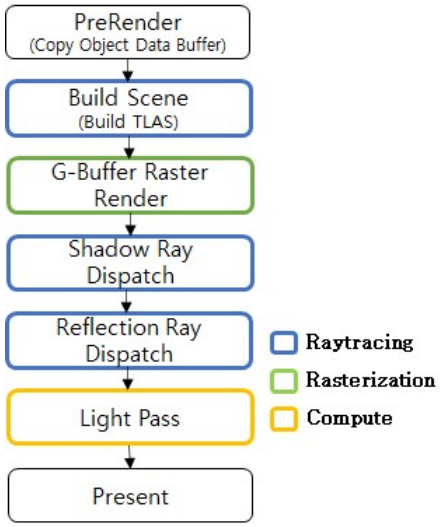


Figure 1. Rendering Flow.

3. IMPLEMENTATION

3.1 Rendering Flow

This paper employs a hybrid rendering method using the previously described hybrid ray tracing technique to render the entire scene. The rendering flow used to apply hybrid ray tracing in this paper is illustrated in Figure 1. The paper utilizes G-buffer rendering with rasterization for deferred rendering, and implements direct lighting and final result computations using compute shaders. Shadow and reflection processing are implemented using ray tracing.

3.2 Preparation

This step involves collecting the resources required for the actual rendering process, such as textures and geometry, and preparing them to be passed to the GPU buffer. The world matrix is calculated using the position, rotation, and size information of each object, and is then copied to the GPU memory to be prepared for use in the rendering process. In addition, global variables, constant buffers, and other resources required for the rasterization and ray tracing stages are prepared and copied to the GPU memory, ready for use in the rendering process.

Journal of Theoretical and Applied Information Technology
31st May 2023. Vol.101. No 10

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

If skin animation is applied, it is necessary to prepare the final vertices after the animation-related operations before building the acceleration structures, so that the geometry structure with animation can be passed to DXR. In this paper, a compute shader stage is used to apply the animation operation to the entire vertex buffer and create a vertex buffer with applied animations. This prepared vertex buffer is then built into a bottom-level acceleration structure in the subsequent scene build stage and passed to DXR for use.

The process of building various resources necessary for ray tracing by traversing all objects in the entire scene, which is performed identically for both hybrid ray tracing and ray tracing only, is called the construction process. This process involves two main steps: building the hit group shader table and building the acceleration structure.

In the first step of building the hit group shader table, elements of the shader table are generated based on the combination of textures, shader code, vertices, and indices used in the object, and their indices are assigned as hit group numbers. Objects using the same combination are assigned the same hit group number. Even if the same original mesh is used, different hit group numbers are assigned if the position of the vertices is different due to animation. In this paper, two types of shaders, one for shadow processing and the other for reflection processing, were placed in the hit group shader table for hybrid ray tracing, as shown in Figure 2.

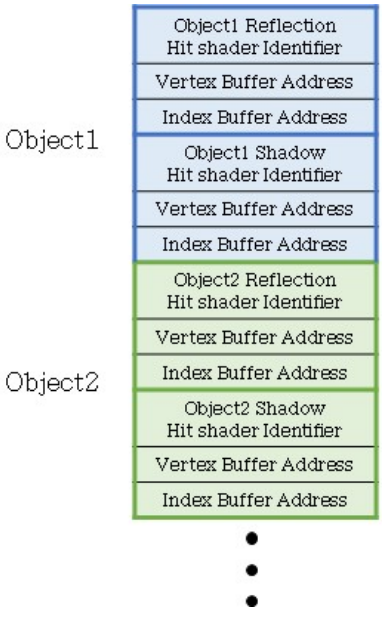


Figure 2. Hit Group Table.

The next step is building the acceleration structure. First, the bottom-level acceleration structure is built. Since the bottom-level acceleration structure only needs to be built once per original geometry, it is determined whether it has been built or not and built if it hasn't been built yet. For objects with animation applied, the position of the vertices changes every frame, so the bottom-level acceleration structure must be built or updated every time. In this paper, the method of rebuilding it every time was used to simplify the implementation.

Once the bottom-level acceleration structure is built, the top-level acceleration structure needs to be built. Using the hit group numbers obtained while building the shader table, the object's world matrix, and the bottom-level acceleration structure to be used, the top-level acceleration structure is built.

3.3 G-Buffer Rendering Stage

The G-Buffer rendering stage, as depicted in Figure 3, is used to store geometry information similar to that used in deferred rendering as discussed in papers [6][7]. In this paper, we use the MRT (Multiple Render Targets) feature of Direct3D 12 to render object material information, normal values, depth values, and other information to three render targets and a depth-stencil buffer. The G-Buffer rendering stage is performed before the ray tracing stage in order to determine whether an object is visible at a particular pixel based on its depth value and to obtain the world position of the pixel to be used as the starting position for the ray in the subsequent ray tracing stage.

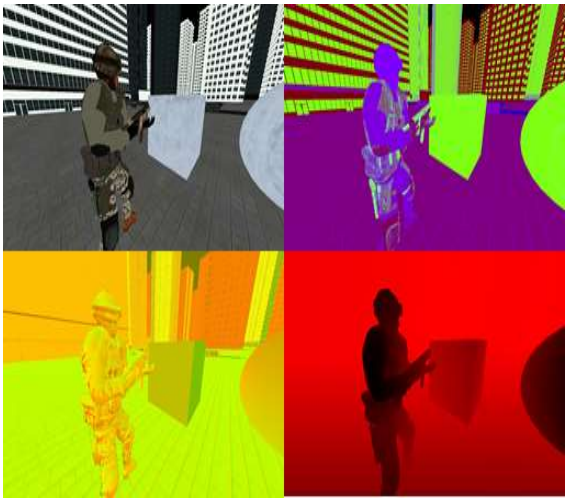


Figure 3. G-Buffer

3.4 Ray Tracing Stage

After the G-Buffer rendering is complete, ray tracing is used to calculate whether each pixel has shadows and its reflection color. In this paper, to reduce the number of rays generated, the depth buffer is used to determine whether there is an actual object visible in that pixel. Only pixels with actual objects are used to generate rays for optimization.

A ray is fired from each pixel's world position towards the light source. The collision detection with the previously constructed top-level acceleration structures is used to determine whether the ray hits an object or not. If the ray hits an object, the hit shader is called; otherwise, the miss shader is called. If the hit shader is called, it means there is an object obstructing the path between the light source and the pixel, resulting in shadows. This paper implements shadows for one directional light in Figure 4. Since the light direction is the same for all pixels, rays are fired in the opposite direction of the light. Additionally, to use the shadow information in subsequent lighting operations, the hit shader records a value of 0, and the miss shader records a value of 1 in a separate texture.

The camera vector is used with the normal value from the G-Buffer to calculate the reflection vector. A ray is then fired from each pixel's world position in the direction of the reflection vector. If the ray hits an object, the hit shader is called; otherwise, the miss shader is called. Unlike shadows, in reflections, the hit shader needs to identify the intersection point of the ray and the polygon, and then interpolate the adjacent vertex values. Once the intersection point's variables have been interpolated, lighting operations can be performed to calculate the final color of the intersection point. To determine the shadow information of this intersection point, additional shadow rays can be generated. Additionally, this paper did not use it, but to reflect again from the reflected intersection point, another reflection ray can be generated to composite the results. To reduce the number of rays generated, this paper generated only shadow rays in a single reflection ray, and additional reflection processing is performed using the environment map. This process is performed for all pixels, and the results are recorded in a separate texture as shown in Figure 5.

3.5 Direct Lighting Computation and Final Result

The final lighting computation is performed using the G-Buffer, shadow texture generated from ray tracing, reflection texture, and lighting information of the scene using the compute shader stage of DirectX.

The basic lighting calculation process is similar to deferred rendering, where for each pixel of the screen, the material information, normal value, and world coordinate are calculated from the G-Buffer and passed to the lighting calculation function to calculate direct lighting. To determine the shadow of the direct lighting, the previously calculated shadow texture is used to determine whether a pixel is in shadow, and the ratio of direct lighting is adjusted accordingly to implement shadows. In addition, the reflectivity of the material for each pixel is calculated, and the reflection texture previously calculated is combined according to the reflectivity to calculate the final image with reflections and shadows applied.

Figure 4. Shadow.

Figure 5. Reflection.

Figure 6. Final Scene.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

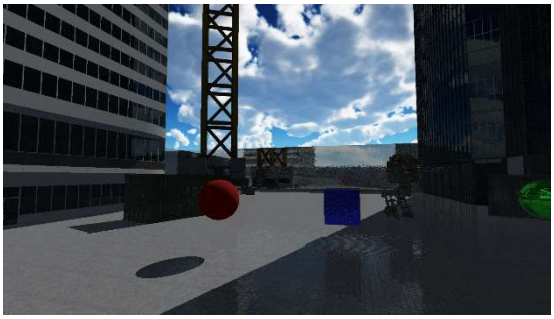
4. PERFORMANCE EVALUATION

4.1 Experimental Environment

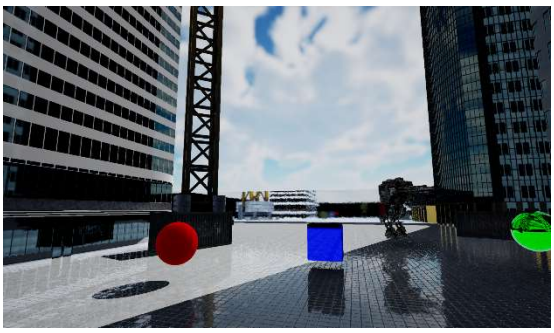
The study compared the rendering performance of self-made scenes and similar scenes implemented using Unreal Engine 4 with ray tracing. The experiment was conducted on a computer with an AMD Ryzen 5 2600X CPU @ 3.6GHz processor, 32GB memory, Windows 11 64-bit operating system, and NVIDIA Geforce RTX 2070 SUPER graphics card. The device screen resolution was set to 1920x1080 for both Unreal Engine 4 and the self-made program. Each scene was implemented using DirectX12's DXR, and factors other than rendering that could significantly affect performance and output were set and measured identically. As shown in Figure 8 and Table 1, the study compared the difference between the output of the self-made program and the output of Unreal Engine 4, as well as their performance measured in frames per second, using three models with similar scenes, namely A, B, and C.

4.2 Comparing Output Results

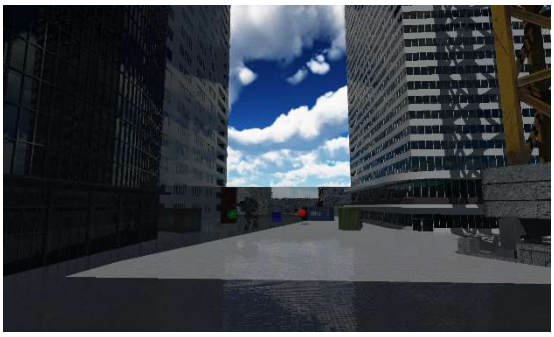
Regarding the difference in output, as shown in Figure 7 for the entire model, the self-made



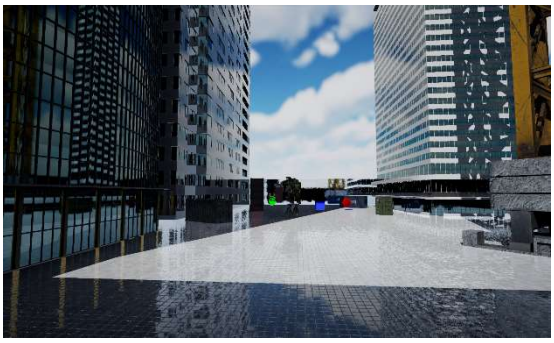
(a) Model A in DXR



(b) Model A in UE4



(c) Model B in DXR



(d) Model B in UE4

(e) Model C in DXR

(f) Model C in UE4

Figure 7 Screen Shots for Various Simulation Models

program's output was darker than that of Unreal Engine 4. To achieve the same output with the same resources, we implemented realistic lighting based on PBR, which is also used in Unreal Engine 4. However, differences in detailed lighting formulas, texture compression and loading methods, and the method of tone mapping HDR results to LDR caused differences in brightness and color.

The most significant difference in terms of ray tracing was the result of the glossy plane. Figure 8 shows the reflection calculation results of each program when the roughness value was 0.25. As shown in the figure, the self-made program's result shows a clear reflection of the surrounding environment, while the reflection in Unreal Engine 4's result appears blurred. In a typical rendering environment, for materials with a roughness value other than 0, which means specular reflection rather than mirror reflection, several hundred rays are fired in multiple directions for sampling. In this paper, we linearly interpolated the results sampled from a single ray and a sky map blurred cube map sampled with blurring processing according to the roughness value to handle specular reflection processing in real-time rendering environments. As a result, even materials with low roughness values can show the surrounding environment like mirror reflection. In contrast, Unreal Engine 4 handles specular reflection by dividing it into two categories: low roughness and high roughness. When the roughness is low, it uses one ray with noise reduction to sample the reflection, while for high roughness, it uses many rays to sample the reflection in multiple directions. This approach is more accurate but requires more computational resources [8].

(a) In DXR (Roughness 0.25)

(b) In UE4 (Roughness 0.25)

Figure 8. Difference between Reflection Results in Same Roughness.

4.3 Rendering Speeds

Table 1 compares the rendering speeds (in FPS: Frames per second) measured in three scene models, A, B, and C. Despite using the same materials and geometry in similar scenes, DXR-based content shows significantly better performance, ranging from 45.0% to 118.3% compared to UE4-based content. Although Unreal Engine 4 uses several techniques such as multithreading to utilize GPU usage up to 100%, it seems to fall short in terms of frame rate compared to a self-made program that uses only a single thread and lacks optimization techniques, resulting in only 50% GPU usage. This could be due to Unreal Engine 4 performing several processes in addition to ray tracing, such as applying noise filters to handle specular reflections and using multiple rays per pixel, causing lower performance compared to a self-made program that uses simple linear interpolation and single rays per pixel.

Table 1. Performance Results in Rendering Speeds (FPS).

	Scene A	Scene B	Scene C
DXR	87 ~ 93 (FPS)	97 ~ 102 (FPS)	125 ~ 131 (FPS)
Unreal Engine 4	58 ~ 60 (FPS)	59 ~ 61 (FPS)	60 ~ 62 (FPS)
Difference	45.0 ~ 60.3 (%)	59.0 ~ 72.9 (%)	101.6~118.3(%)

5. CONCLUSION

In this paper, real-time ray tracing reflections and shadows were implemented using DXR, a ray tracing API, and compared with Unreal Engine 4's real-time ray tracing in 3D games. Generating rays and performing collision detection with scene objects is not suitable for computation on the GPU, so special hardware such as NVIDIA's RTX graphics card and APIs such as DXR were used to efficiently implement ray tracing and collision detection in a real-time rendering environment. Additionally, to reduce the heavy workload, primary rays were rendered using the existing rasterization method, and shadow rays and reflection rays were created based on the results, enhancing rendering efficiency. Real-time ray tracing requires special and high-performance hardware, making it difficult to serve a wide range of users. However, as graphics card performance continues to improve significantly, more users are likely to possess such hardware, allowing for more intuitive implementation and realistic graphic effects.

In terms of future research directions, there are several areas that could be explored. First, while this study focused on real-time ray tracing for reflections and shadows, there are other applications of ray tracing that could be investigated, such as global illumination and caustics. Second, while the study used DXR as the ray tracing API, other APIs such as Vulkan and OpenGL could also be examined for their performance and capabilities. Third, the study primarily used NVIDIA's RTX graphics cards, but other hardware configurations and manufacturers could be tested to evaluate their performance and compatibility with real-time ray tracing. Finally, the study examined the performance of real-time ray tracing on a limited number of scenes and objects, so further research could explore the scalability and adaptability of ray tracing to more complex and dynamic environments.

REFERENCES:

[1] Do-Hyoung Kim, “Real Time Shadow Processing Techniques in 3D Game Graphics Engine”, The Magazine of the IEIE, Vol. 34, No. 10, pp. 54-62, 2007.10.

[2] Ned Greenel. "Environment mapping and other applications of world projections.", IEEE computer graphics and Applications, 6.11, pp. 21-29, 1986.

[3] McGuire, Morgan, and Michael Mara. "Efficient GPU screen-space ray tracing." Journal of Computer Graphics Techniques (JCGT) 3.4, pp. 73-85, 2014.

[4] Sanzharov, V. V., et al. "Examination of the Nvidia RTX." Proceedings of the 29th International Conference on Computer Graphics and Vision. Vol. 2485. 2019.

[5] Colin Barré-Brisebois, et al. "Hybrid rendering for real-time ray tracing." Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs, Haines E., Akenine-Möller T.,(Eds.). pp. 437-473, 2019.

[6] Andrew Lauritzen. "Deferred rendering for current and future rendering pipelines". SIGGRAPH Course: Beyond Programmable Shading, 2010.

[7] Nicolas Thibieroz. "Deferred shading with multiple render targets." Shader X 2 pp. 251-251, 2004.

[8] Liu, Edward, et al. "Cinematic rendering in UE4 with real-time ray tracing and denoising." Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs, Haines E., Akenine-Möller T.,(Eds.). pp. 289-319, 2019.

[9] Hunt, Warren, and William R. Mark. "Ray-specialized acceleration structures for ray tracing." 2008 IEEE Symposium on Interactive Ray Tracing. IEEE, 2008.

[10] Mildenhall, B., Srinivasan, P. P., Tancik, M., Barron, J. T., Ramamoorthi, R., & Ng, R, “Nerf: Representing scenes as neural radiance fields for view synthesis”’ Communications of the ACM, 65(1), pp. 99-106, 2021.

[11] Gautron, Pascal. "Real-time ray-traced ambient occlusion of complex scenes using spatial hashing." Special Interest Group on Computer Graphics and Interactive Techniques Conference Talks. 2020.

[12] Belhomme, Boris, et al. "A new fast ray tracing tool for high-precision simulation of heliostat fields." Journal of Solar Energy Engineering 131.3, 2009.

[13] Li, Tzu-Mao, et al. "Differentiable monte carlo ray tracing through edge sampling." ACM Transactions on Graphics (TOG) 37.6 (2018): 1-11.