Cross-platform Renderer for Volumetric Bodies with Vulkan API

Roman Naumenko
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
roman.naumenko@ucu.edu.ua

***Abstract*—This project demonstrates the implementation of a cross-platform rendering engine, leveraging the Vulkan API's capabilities to render volumetric bodies efficiently. Building upon a foundational understanding of OpenGL, it explores Vulkan's functionality as a modern, efficient, and low-overhead graphics API. The robust application developed not only serves as a scene renderer but also supports a graphical user interface, showcasing Vulkan's versatility in complex rendering tasks. Utilizing raw Vulkan API for rendering and GLFW for window creation and OS interaction on both Windows and Linux, the project underscores Vulkan's comparative advantages over traditional APIs like OpenGL and highlights the developmental journey and learning curve associated with it.**

To achieve high-fidelity images when rendering volumetric objects like clouds, smoke, and fog, the method of ray marching was employed. This particular method is often utilized in modern game engines for the same purpose, as it not only can produce visuals indistinguishable from reality but also possesses significant potential for optimization and parallelization. This allows it to operate efficiently on GPUs and be executed exclusively in the fragment shader.

***Index Terms*—Vulkan, volumetric clouds, ray marching, rendering framework**

I. Introduction

For a long time, rendering volumetric objects such as clouds, fog, and smoke in real-time applications was an unattainable goal. However, with the advancement in computing capabilities of consumer-level computers, modern GPUs now facilitate the parallelization of previously sequential algorithms. This development has elevated the quality of in-game volumetrics to a level where they can rival the realism of actual ones. As can be seen on Fig. 1.

Concurrently, extensive research in rendering algorithms has been instrumental in this progress. A standout contribution in this field is the work of Guerrilla Games [1], particularly their engine developed for the "Horizon: Zero Dawn" game series.

For this resource-intensive task, Vulkan, regarded as the successor to OpenGL, became the ideal choice due to its flexible and low-overhead API. For learning purposes, the entire rendering system was built from scratch using only the Vulkan SDK, deliberately avoiding any bootstrap libraries to gain a deeper understanding of the underlying processes and Vulkan's capabilities.

Fig. 1. Clouds in Unreal Engine. [2]

II. Ray marching overview

A. In general

Ray marching is a rendering technique used for accurately depicting volumetric phenomena like clouds and fog. It operates by casting rays from the viewer's perspective into the scene and iteratively "marching" these rays forward in small, fixed steps until they hit an object or reach a maximum distance.

At each step, the algorithm evaluates the scene's density function to determine the presence of volumetric material. If material is encountered, the algorithm computes lighting and color contributions at that point, integrating these over the ray's path to produce the final pixel color. The precision of the render is controlled by the step size: smaller steps increase accuracy and visual quality but require more computational resources.

Ray marching excels in GPU environments due to its parallel nature, as each ray can be processed independently. This makes it particularly effective for real-time applications where rendering performance is critical.

B. Math behind the algorithm

One of the main optimization techniques underlies the way volume is contained in the bounding box and then "projected" on its faces. When calculating pixel color, we do not sample points on the ray from the camera all the way to infinity. Indeed, the only points considered for sampling are those

contained inside the bounding volume. These points are chosen on the ray segment between two points of intersection of the camera ray with a box (Fig. 2).

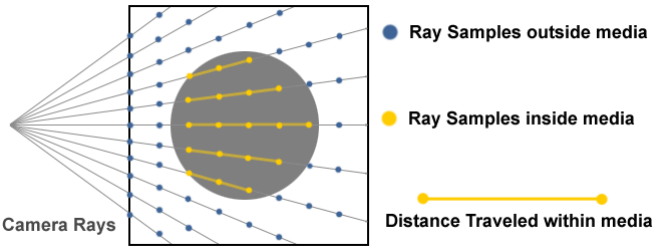

Fig. 2. Ray marching principles [3].

So far, the algorithm describes the opacity as only cloud ray marching. This means that what is called a cloud here is actually just a glowing smoke. To add lighting, apart from sampling the volume along the camera vector, we also need to track the light scattering out of the volume on its way from the sample to the light source (Fig. 3).

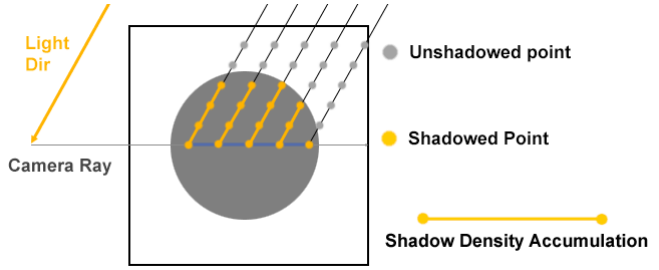

Fig. 3. Shadow tracing [3].

To calculate the intensity of the light transmitted in a certain direction from a given point in the volume, denoted by α, we apply the Beer-Lambert law. This involves taking the inverse of the exponent of the distance traveled through the volume d, multiplied by the media's thickness at that point t:

$$\alpha = e^{-td} \tag{1}$$

Then, we blend the incoming light C_{in} with the media's own radiance at the sampled point S_c, using the coefficient α, to obtain the final color of the light C_{out}:

$$C_{out} = \alpha C_{in} + (1 - \alpha) S_c \tag{2}$$

III. Render Engine Technical Details

A. Vulkan API multhreading

Although the software engineering involved in the rendering system is complex and cannot be fully detailed in this report's format, several crucial points merit mention.

In contrast to OpenGL and DirectX 11, the Vulkan API is known for being extremely explicit and flexible, to the point that it even supports multiple CPU threads for interaction with its infrastructure. Therefore, it is possible to submit command buffers, which are essentially lists of commands, to the command queue from different threads [4] (Fig. 4); This feature was not fully utilized in this project due to the lack of necessity. But the rendering framework was designed in way so it would be easy to add support for it when needed.

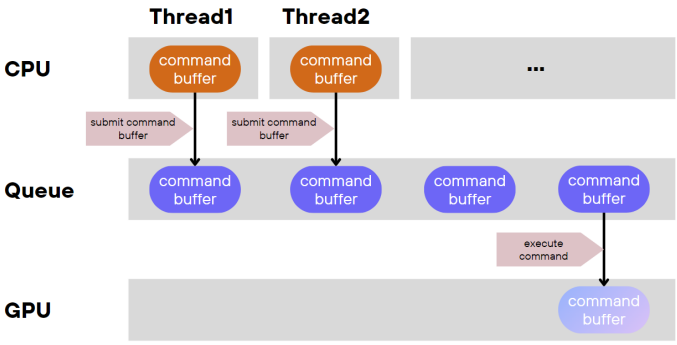

Fig. 4. Multithreaded command submission.

B. Synchronization techniques

When the GPU finishes rendering the frame, it can stream it to the video port to present it on screen. However, all modern display have their own refresh rate, usually 60Hz, which means that a full cycle of image presentation takes around 0.0167s, and a big chunk of that time is allocated for updating the pixels' colors. Which leaves a place for visual artifacts like screen tearing [5] when a new frame has finished rendering before the previous one is fully shown on the display.

To address this issue, numerous presentation modes have been developed that enable vertical synchronization through multiple buffering [6]. While these techniques eliminate visual artifacts, they require the renderer to process multiple frames concurrently to avoid pipeline stalls. In Vulkan, nearly all commands processed by the GPU are asynchronous, with only the order execution start guaranteed by the specification [7] as they are submitted on the CPU. To safeguard critical sections of data that must be processed sequentially, Vulkan provides special synchronization primitives called semaphores. These are utilized by passing handles to consecutive API calls, where the first call signals the semaphore and the second waits for this signal. This method allows for organizing a graphics pipeline into logical sections: acquiring the image, rendering the frame, and sending it for presentation, as shown in Fig. 5

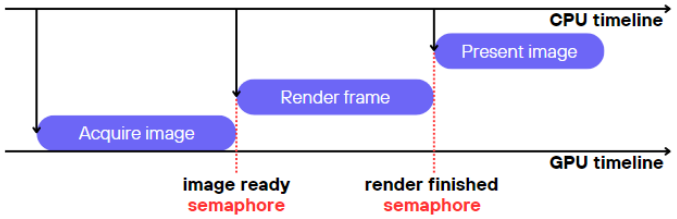

Fig. 5. Pipeline synchronization.

IV. Conclusion

The primary objective of this project – to learn Vulkan's principles and develop a universal rendering framework – was successfully met. However, it's important to note that while Vulkan is a powerful tool for creating efficient, low-overhead rendering and computing applications, it also presents a high level of complexity and can be unnecessarily sophisticated for tasks like those tested in this project. The code of the project is available at https://github.com/Raspy-Py/VolumetricRenderer.

References

[1] A. Schneider, "The real-time volumetric cloudscapes of horizon - zero dawn," 2015. [Online]. Available: https://advances.realtimerendering.com

[2] "Volumetric clouds," 2021. [Online]. Available: https://docs.unrealengine.com/4.26/en-US/BuildingWorlds/LightingAndShadows/VolumetricClouds/

[3] "Creating volumetric ray marcher." [Online]. Available: https://shaderbits.com/blog/creating-volumetric-ray-marcher

[4] "Drawing a triangle - command buffers." [Online]. Available: https://vulkan-tutorial.com/Drawing_a_triangle/Drawing/Command_buffers

[5] "Screen tearing." [Online]. Available: https://en.wikipedia.org/wiki/Screen_tearing

[6] "Multiple buffering." [Online]. Available: https://en.wikipedia.org/wiki/Multiple_buffering

[7] "Vulkan 1.3 specification. synchronization and cache control." [Online]. Available: https://registry.khronos.org/vulkan/specs/1.3/html/chap7.html