Vulkan-based Rendering Engine

Bohdan Opyr
Research and Development
Lviv, Ukraine
opyr.pn@ucu.edu.ua

Roman Zaletskyi
Research and Development
Lviv, Ukraine
zaletskyi.pn@ucu.edu.ua

Oles Yarish
Research and Development
Lviv, Ukraine
yarish.pn@ucu.edu.ua

Ostap Trush
Mentorship
Lviv, Ukraine
ostap.trush.2003@gmail.com

***Abstract*—This report describes the exploration of next-gen graphics APIs by developing a rendering engine based on Vulkan.**

***Index Terms*—Vulkan, GPU, computer graphics, dingus**

I. Introduction

Visualizing 3D scenes usually involves using the GPU. Graphics APIs are often used to interface with GPUs. Traditional graphics APIs are generally not built with modern graphics hardware in mind and lack performance and flexibility. Next-gen APIs, on the other hand, are inherently low-level, requiring a lot of boilerplate to perform even the simplest tasks. In this project, we developed a rendering engine – a wrapper around the Vulkan graphics API that abstracts away its low-level nature while retaining its rendering power and allowing one to load and render scenes faster and extend the tool to explore Vulkan as an example of a modern next-gen graphics API.

II. Background

Traditionally, with early GPUs, the whole process of rendering an image was determined and, therefore, heavily restricted by the hardware. Each stage was implemented on the GPU, and different GPUs were pretty similar in what the specific stages were and how they worked – because of the limitations and similarities introduced by the GPUs, traditional graphics APIs, such as OpenGL, introduced the idea of a fixed-function rendering pipeline. The data goes through distinct predefined stages to render a scene, but only some are configurable.

As hardware evolved, the restrictions of such a pipeline slowly faded away. The developer now has significantly more control over the entire process. This newfound flexibility, however, is not accessible through traditional graphics APIs. This limits the developer in terms of, for example, how much graphics code optimization is available. New graphics APIs, such as DirectX 12 and Vulkan, were introduced to solve this. They expose this flexibility while fixing other issues with common graphics APIs, e.g., their synchronous single-threaded nature. They give the developer much more fine-grained control over the resources, such as memory.

As a result, though, these APIs are inherently low-level. The developer handles tasks such as checking GPU compatibility and picking the correct GPU. Different tools exist to reduce this boilerplate. Game engines, for example, include tools that abstract away not just the complexity of a single graphics API but also of using different render backends, allowing one to render the same scene using a variety of APIs.

III. The rendering pipeline

A key concept in this work is the rendering pipeline – the lifecycle of a scene as it goes from an abstract description of three-dimensional geometry (mesh) to an image. An example of a rendering pipeline – the Vulkan one – can be seen in Fig.1. It differs from implementation to implementation, from graphics API to graphics API. However, most graphics APIs provide default implementations for most stages. The four stages we are most interested in are the vertex shader, the rasterizer, the fragment shader, and the color blending stage. The vertex shader is a program that runs for each vertex in the scene. The developer configuring the graphics pipeline writes this program, allowing for arbitrary computation. For example, the vertex shader can simulate ocean waves by calculating the ocean height value and moving each vertex vertically. However, the end goal of the vertex shader is returning the coordinates of each vertex in clip space – a coordinate system that results from applying the perspective transform – see Fig.2 where the x and y coordinates map linearly onto the viewport and the z coordinate, ranging from -1 to 1, defines what is visible and helps specify occlusions. Afterward, the mesh in clip space runs through the rasterizer – a stage that determines which 2d pixels correspond to the 3d geometry. This stage, unlike the shader stages, is much less configurable – though you can configure some of it, such as whether the wireframe outlines of objects should be computed separately, arbitrary computations are not possible. Then, the fragment shader – an arbitrary program similar to the vertex shader – runs for every visible scene pixel. This is where, for example, textures are applied. The last significant step is blending the pixel values. For example, the value computed by the fragment shader of the pixel closest to the camera will end up in the final image for opaque meshes.

IV. Key Terms

Below is a short list of key terms extensively used in Vulkan API (with Vk prefix omitted) and thus in our further discussion:

- **Graphics pipeline** – the path a set of vertices (and accompanying data) takes to get drawn on the screen.

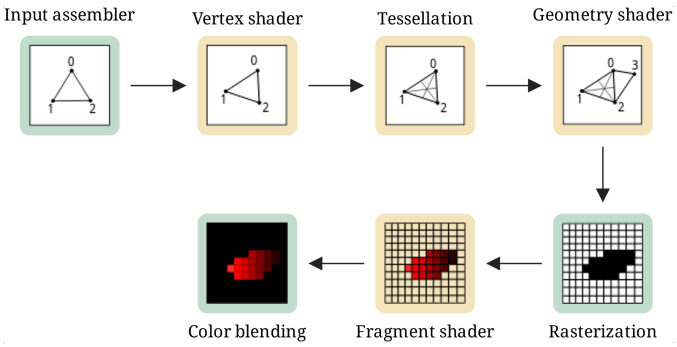

Fig. 1. The Vulkan rendering pipeline (courtesy of Vulkan tutorial).

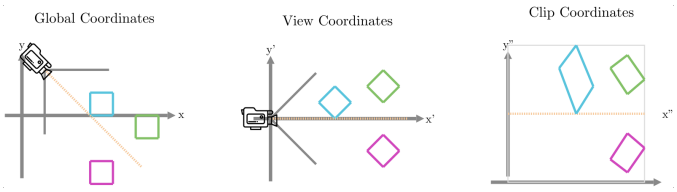

Fig. 2. The different coordinate spaces.

- **Buffer** – an object capable of storing arbitrary data. It is used in almost all major components of the engine and helps to facilitate data transfers between CPU and GPU.
- **Image** – serves the same role as Buffer but implies storage of some pixel data. It is usually accessed through Image views.
- **Device** – a physical device that supports Vulkan and can run its commands.
- **Swapchain** – a queue of images the engine can render and then present to the screen, used to render newer images while older ones are still relevant – e.g., for rendering faster than the framerate.
- **Depth buffer** – a buffer used to determine which polygons overlap by storing the distance of the fragments from the camera.
- **Command buffer** – a tiny object that binds together all information needed to render some vertices: render/transfer commands, resources, etc...
- **Queue** – a unit of execution to which Command buffers are submitted to be executed on the device. Intuitively, it is an independent part of the GPU with everything that entails.
- **Descriptor sets** – describe the data accompanying the vertex data into the shader, such as transformation matrices, constants, etc.

V. Exploring Vulkan

The first step in building the render engine was getting familiar with Vulkan. We followed the well-known Vulkan Tutorial [5], implementing a basic triangle renderer to understand the fundamental concepts. We could outline the journey to a rendered 3d scene through our exploration. As a crude approximation, the steps can roughly be broken into two categories: initialization and rendering a frame.

The first goal of the initialization process is to select the proper device and acquire a way to interface with it. Here, 'device' refers to any physical device that supports Vulkan and can run its commands (e.g., GPU). Unlike older and higher-level APIs, Vulkan does not handle this for us; instead, it lets us choose the device that best suits a specific task. This versatility allows Vulkan to work on more platforms and with more kinds of devices, which is crucial in some applications. In our case, though, all it means is to list the available devices, check whether they meet our criteria, and, if so, choose the most powerful one according to some arbitrary metric. The next step is retrieving two queues to submit commands. Vulkan is inherently asynchronous and multi-threaded. Therefore, all commands to the device are submitted through queues. These commands can be widely separated into several groups. Using different queues explicitly for separate types of operations makes sense, as independent parts of the GPU will likely perform them. An example of two independent types of operations is video encoding and data transfer – a dedicated video encoding chip could handle the first operation, with the second operation being simultaneously performed through DMA. In Vulkan, a set of queues with common properties is called a queue family, and we can get a queue for a specific task by finding the corresponding queue family. We need two types of queues for a rasterization render engine: a graphics and a present queue. The graphics queue sends commands to the GPU describing how to draw the scene, while the present queue instructs the GPU to display the data eventually. It is worth noting that these tasks are often handled by the same queue for most modern GPUs, which, luckily, does not majorly impact how we use them. Once we retrieve the queues, we can pack our commands for the GPU into small packets called command buffers and send them through the queues – a single command buffer is usually used to configure the GPU for the task at hand (e.g., specify rendering commands and resources to use) and then execute the task.

The second primary goal is configuring the GPU for rendering – for example, constructing and binding an image instance to the GPU, moving the scene data to the GPU memory, creating the buffers to render to, and preparing graphics pipelines. In Vulkan, to optimize the performance, you configure each graphics pipeline you might use just once and then bind the configuration to the GPU to use it. Before drawing a single frame, you allocate the buffers that will later store the data that the shaders operate on – the input and output textures, vertex buffers, lights etc.

Once the initialization is done, rendering a single frame is relatively straightforward. It consists of preparing all the per-frame data and submitting two command buffers: one to the graphics queue to draw the entire scene and one to the presentation queue to display the image. For each pipeline that renders to this frame, we configure the GPU to use that pipeline, bind all of the relevant descriptor sets – references to buffers we created during the initialization step – and issue a

draw call, the call to render. To enable depth-based blending, we attach another single buffer to each of the pipelines to store the depth of each pixel.

This is, of course, an oversimplification. For example, the GPU can render multiple frames simultaneously, allowing for lower frame times. Similarly, it is possible to fully render a frame without waiting for the previous one to be displayed, leading to such a previous frame never being shown. Vulkan offers tools for synchronizing these operations, such as a swap chain, a collection of texture buffers responsible for handling asynchronous rendering and displaying.

Due to Vulkan's low-level nature, even rendering a single triangle requires hundreds of lines of boilerplate code, with the majority of the concepts – for example, manual memory management – not present in higher-level graphics APIs. This exploration revealed which Vulkan concepts should be exposed to the engine user and which should be abstracted away. While explicit control over shaders and pipelines is often helpful, manual management of command buffers or synchronization primitives rarely is. Similarly, while configuring memory allocation strategies can be crucial for performance, the actual allocation code is pure boilerplate. This understanding led us to design an engine that exposes high-level graphics concepts while internally handling Vulkan's complexity.

VI. Our rendering engine

When designing our rendering engine, we had to choose an appropriate level of abstraction over Vulkan. Some engines, such as EEVEE, commonly used for CGI, abstract away nearly all graphics-related concepts, leaving the developer with only a handful of shader parameters, providing their abstraction (in the case of EEVEE, the material node graph) over the shader. While this approach is versatile for creative use, it does not give the user much control, and consequently, the educational value is limited. A similar node-based system can also suit a high-complexity, high-performance environment. This is the case with the Unreal Engine material nodes, which are helpful for rendering and post-processing. Other engines instead expose an API that is a bit more verbose, giving the user far more explicit control over the rendering process. This is the approach we went with, choosing a level of abstraction high enough to where rendering a simple scene takes just a few lines of code while still leaving a lot available through the API and allowing one to extend the engine, exploring more of Vulkan's vast functionality. These are the key abstractions that take us from nothing to a fully rendered scene:

A. The Engine Abstraction

As the name implies, this abstraction represents the entire running render engine. It's responsible for directly managing the application – for example, initializing Vulkan, picking the correct graphics device, creating the window, processing user input, building pipelines, and starting renders.

B. A render

Rendering a new frame involves retrieving an available command buffer, recording the commands needed to render it to this buffer, and submitting this buffer to the graphics queue. With our render engine, this is done by requesting a new render from the engine, which begins the recording of a command buffer, then by one-by-one specifying whatever objects we want to render, recording the appropriate draw calls, and then instructing the engine to finish this render – to submit the command buffer, with all the appropriate synchronization, and to initialize the transfer of our result to the window surface.

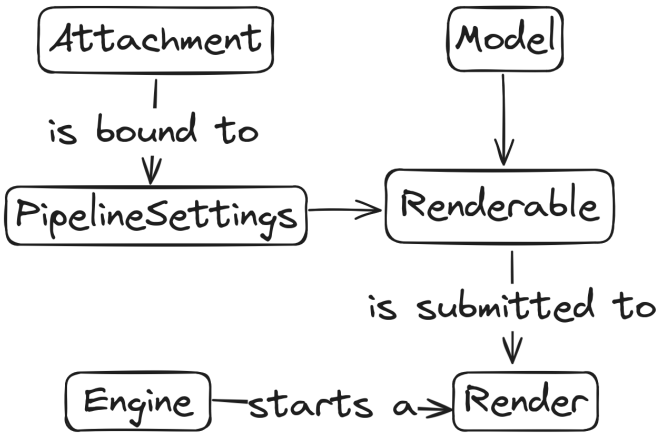

Fig. 3. Core top-level abstractions.

C. A renderable object

This abstraction represents something – anything – that can be rendered. It manages the rendering pipelines and all related buffer bindings, etc. Renderables are submitted to the render every frame. This way, the scene is dynamic in that entire layers of objects, represented by different renderables, may be turned on or off at any point. We only have one kind of renderable – a collection of render passes, instanced 3d meshes (see section VI-G) linked to a specific graphics pipeline along with all of the descriptors, etc. However, you can quite easily extend the render engine to support more complex renderables, such as post-process passes. The project's goal is not to create a versatile rendering tool but to build a solid platform for exploring Vulkan.

D. An attachment

We often need to bind global data to a shader. For example, when displaying a textured mesh, the entire texture should be available in the fragment shader, as the calculations for which pixel on the screen corresponds to which pixel in the mesh are done in the fragment shader itself. We wrapped each of these pieces of data in what we called an attachment. The attachments are then used to create a render pass to specify the layout of the descriptor set associated with the data passed to the rendering pipeline and, optionally, on each render, to write the per-frame data into the relevant buffers.

E. Model Loading

Loading arbitrary 3D scenes is a crucial feature for any rendering engine. While various 3D file formats exist, we

chose the Graphics Library Transmission Format (glTF) as our primary format. glTF is a widely used open standard offering efficient transmission and loading of 3D scenes. We chose it for its simplicity and our existing experience working with it.

Our current implementation focuses on the fundamental geometric aspects of glTF:

- Triangle meshes – the basic building blocks of 3D surfaces
- Vertex attributes (positions, texture coordinates, normal vectors)
- Basic transformations (translation, rotation, scale)

glTF is relatively complex, with supporting features like animations, skeletal rigs, and advanced materials. Still, as supporting most of these is out of the project's scope, we focused on the core geometric data needed for basic rendering.

F. Texture management

There are multiple approaches to passing textures into the pipeline. One of the more prominent techniques is passing them as a single buffer, allowing the same shader to be used with different materials, accessing the textures by index – though you certainly can still use the same shader even when binding the textures to more than one descriptor. The performance impact of using one shader is often positive since you can reduce the number of draw calls – and a high number of draw calls is quite often the cause of low performance. This is the approach we chose, passing a single texture sampler array. We created another entity, a texture manager, responsible for collecting all the textures throughout the model loading process and later initializing the texture attachment.

G. Instanced Rendering

Another optimization we implemented is instanced rendering, resulting in a significant speedup in rendering multiple copies of a single object. With instanced rendering, only one draw call is issued instead of one per copy. Instead, another buffer is attached to the vertex shader, containing the information about instances. Then, the vertex shader duplicates each vertex, applying the per-instance transforms, allowing for significantly less data transfer between the GPU and the CPU and requiring only one draw call.

As can be seen from an example of rendering a scene containing 1600 dreadnought models [1] 5 4, employment of instanced rendering frees up CPU resources and yields a frame rate higher by 57% and reduces the frame time by 36%.

H. The lifetime of a scene

The overall process of using our render engine is as follows:

- Initialize the render engine – this creates the rendering target window, initializes Vulkan, etc.
- Create a texture manager
- Load the models, adding any associated textures to the texture manager
- Duplicate, properly position and merge models – our engine simplifies this by, for example, merging different models that use the same mesh into one instanced mesh

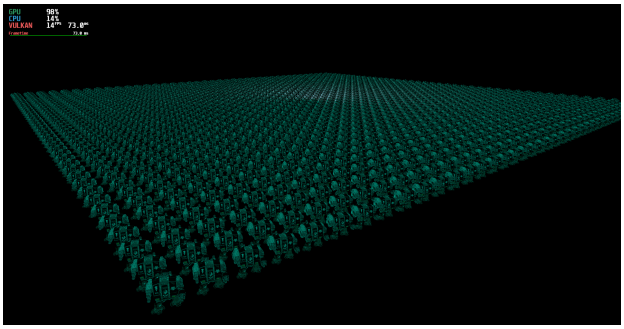

Fig. 4. Scene rendered with no instancing

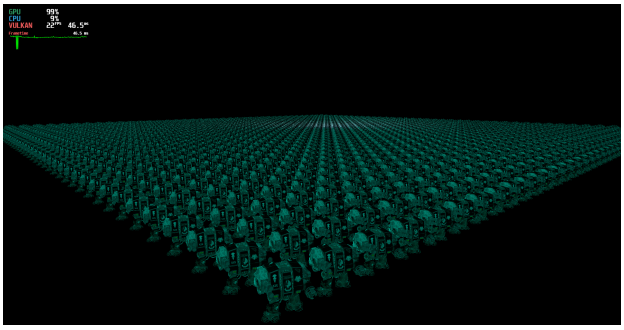

Fig. 5. Scene rendered with enabled instancing

- Get a texture attachment from the texture manager
- Configure rendering pipelines – specify the shaders and the attachments
- Create the pipelines, pairing them up with the models, resulting in renderables
- Repeatedly start a render, submit the renderables to it, and finish it

I. Lighting

Another key detail is the scene shading, handled by the fragment shader. Lighting is one example of its responsibilities. Developing performant yet visually appealing shaders is beyond this project's scope and is far from easy. However, implementing a simple lighting system, such as that with the Blinn-Phong shading model, is far more achievable. This is what we did. Lighting information is passed to the fragment shader through another attachment, containing a structure encoding the information about two types of lights: multiple point lights and one directional light. The point light represents a spherical light source with a defined radius and intensity. On the other hand, directional light is perfect for simulating omnipresent light sources, such as the Sun, that illuminate all objects in a particular direction.

VII. Comparison with other engines

A. Shift

This subsection compares our engine with Shift, the engine designed by our mentor for his bachelor thesis. Performance is measured for the Sponza [3] scene, which features 68 textures and thousands of vertices 6 7.

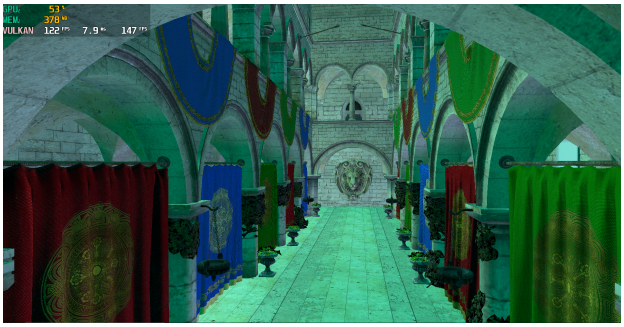

Fig. 6. Sponza scene rendered with our engine

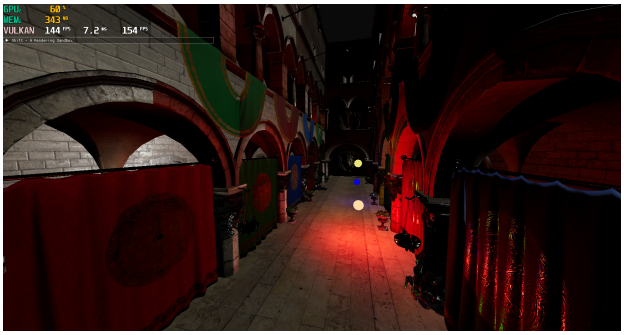

Fig. 7. Sponza scene rendered with Shift

The tests demonstrate that, even though the image is appealing for both engines, it takes more time for our engine to render a frame while almost the same amount of resources is consumed - an indication that more sophisticated performance optimizations must be implemented.

On the other hand, from the point of view of a user, a frame rate of 122 frames is still a praiseworthy result.

VIII. Results

Throughout this project, we have explored Vulkan, an example of a next-gen graphics API. We implemented our abstractions over Vulkan, allowing for easy rendering of simple scenes while leaving much flexibility to the developer. We developed a render engine that allows loading arbitrary scenes, including textured ones, instancing them, and shading them with a user-specified fragment shader. Fig.8 showcases this capability, showing two models running: the Sponza [3] test model running with Blinn-Phong lighting and a cube shaded in such a way to display a volume contained within it, rendered through a simple raymarcher. Both models receive the same lighting and camera position information, allowing for the display of both as a single scene. This demo is available in our GitHub repository [6].

IX. Future work

Our render engine has a lot of room for growth. One example of what could be explored is more advanced rendering techniques, from shadow mapping to using the Vulkan ray tracing extension to model effects such as global illumination. Another example is the optimizations that could be made, such as using descriptor sets more carefully, like binding some globally and some per pipeline, or using virtual texturing and GPU-based culling to optimize memory usage and performance. Since the primary goal of this project is exploring next-gen graphics APIs, the ultimate next step would be implementing a render hardware interface (RHI) - an abstraction layer over several graphics APIs, providing the same interface to all of them.

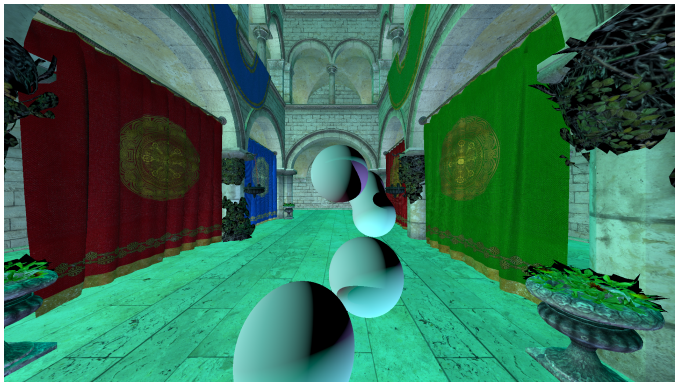

Fig. 8. Raymarching shader.

X. References

References

[1] "Dreadnought model," accessed: 2024-12-10. [Online]. Available: https://sketchfab.com/3d-models/dreadnought-raven-guard-warhammer-40000-9b8a4bb0e94949e9a5a42b08c888c8ed

[2] "Learn opengl," accessed: 2024-12-10. [Online]. Available: https://learnopengl.com/

[3] "Sponza model," accessed: 2024-12-10. [Online]. Available: https://github.com/Breush/lava-assets/blob/master/models/sponza.glb

[4] "Vulkan specification," accessed: 2024-12-10. [Online]. Available: https://registry.khronos.org/vulkan/specs/latest/html/vkspec.html

[5] "Vulkan tutorial," accessed: 2024-12-10. [Online]. Available: https://vulkan-tutorial.com/

[6] R. Z. Bohdan Opyr and O. Yarish, "Github repository." [Online]. Available: https://github.com/triffois/vulkan