Vulkan

Paolo Bettelini

Contents

1 Vulkan

Vulkan is a multi-platform low-level graphical interface (API) for GPU rendering.

The GPU is able to perform computation on a lot of data simultaneously (SIMD, Single Instruction stream, Multiple Data stream).

2 Instance

A Vulkan application starts by setting up the API through a vulkan instance (**VkInstance**).

3 Physical Device

We can query the API for Vulkan supported hardware and select one o more physical devices (**VkPhysicalDevice**) to use for our operations. We can query for different capabilities, VRAM size, etc.

4 Logical Device

Given a phyical device we can create a logical device (**VkDevice**). The logical device represents an open channel with the physical device. it also describes which features (**VkPhysicalDeviceFeatures**) we will be using.

5 Queue

The GPU is able to run multiple operations in parallel. The equivalent of a CPU thread is a queue (**VkQueue**).

6 Queue family

Queues are grouped by queue families.

Whenever we want the device to perform an operation, we have to submit this operation to a specific queue under a family. Some families support only graphical operations, some others support only compute operations, and some others support both and or other operations.

7 Buffers

When we need the GPU to read or write data in memory, we need to use a **buffer** (or an image, later section).

8 Command buffers

To execute an operation we need to create a command buffer containg a list of commands to execute.

To submit a command buffer we need to synchronize with the GPU. We can also tell the GPU to send back a signal, call **fence** when the operation is done.

8.1 Primary command buffers

They can contain any command. They are the only type of command buffer that can be submitted to a queue.

8.2 Secondary command buffers

They allow you to store functionality that you can reuse multiple times in primary command buffers.

9 Compute pipelines

In order to ask the GPU to perform an operation on some data, we need to write a program for it. A program that runs on the GPU is called a shader.

Shaders are written in a shading language (hlsl, glsl, wgpu, rust-gpu, ...) which is then compiled into an intermediate bytecode called **SPIR-V**.

10 Descriptors

When we create a compute pipeline for a shader we must bound it to a **descriptor**. A descriptor can contain a buffer to access, buffer viewes, images, samples images etc.

Descriptors are grouped by **descriptor sets**. The shader will declare a specific descriptor from a specific set (both indexed from 0).

11 Dispatch

To execute a compute pipeline we need to create a command buffer to do so. This is called **Dispatch**.

12 Images

Another way to make the GPU interact with memory is with an **image**. Images are often 2-dimensional array of pixels (there also also 3-dimensional images). The pixels of the image are often referred to as **texels**.

Each pixel can have up to four components.

You can't directly change the content of an image as you would for a buffer. For example, we could create a command buffer to tell the GPU to fill an image with white pixels.

13 Graphics pipelines

A graphics pipeline is the same as a compute pipeline but for graphical operations. Graphics pipelines are more restrictive than compute operations, but they're also much faster.

The purpose of this pipeline is to draw a shape on an image.

Both types of pipelines are represented by **VkPipeline**. It represents the state of the GPU, such as viewport, depth buffer, shaders and so on.

13.1 Vertex shaders

The first step when executing a graphics operation is the **vertex shader**. This shader is executed for each vertex of the shape.

The vertex shader must output the final position (`gl_Position`) for the shape. The vertex struct usually only contains the position of the vertex, so that the shader can set `gl_Position` accordingly. However, a vertex shader can have multiple inputs and output. For example, we could pass `position` and `color` for each vertex.

Listing 1: Vertex Shader

```
#version 450

layout(location = 0) in vec2 position;
layout(location = 1) in vec3 color;

// to frag shader
layout(location = 0) out vec3 fragColor;

void main() {
    gl_Position = vec4(position, 0.0, 1.0);
    fragColor = color;
}
```

13.2 Fragment shaders

The GPU will compute which pixels are in the given final shape, and will execute for each of them the **fragment shader**. Each output of the vertex shader will be available to the fragment shader and will have a smooth interpolation between all the vertex positions. For example, the last vertex shader will call the fragment shader for each pixel within the shape, also passing for each pixel an interpolated color between the colors of the other vertices.

The fragment shader will output the color for that pixel.

Listing 2: Fragment Shader

```
#version 450

layout(location = 0) out vec4 f_color;
layout(location = 0) in vec3 fragColor;

void main() {
    f_color = vec4(fragColor, 1.0);
}
```

13.3 Vertex buffer

Vertex buffer is the name given the the buffer containing the vertices of the shape (and other attributes) passed to the vertex shader. The shape is made out of triangles (3 vertices each).

13.4 Render passes

Before executing a graphics operation we must tell the GPU to enter a ”rendering mode”, by entering what is called a **render pass**. The render pass describes the type of images that are used during rendering

operations, how they are used and how they should be treated.

A render pass is made of **attachments** and **passes**.

14 Windows

Rendering to a window.

14.1 Surface

A handle to a surface object (**VkSurfaceKHR**).

14.2 Swapchain

A swapchain is a collection of one or mutliple images. It is used to draw onto the screen images only when they're done rendering. A swapchain may have one, two (*double buffering*) or most commonly three (*triple buffering*) swapchain images.

The swapchain will provide us with an image to draw to when needed.