Vulkan Unveiled: A Beginner's Guide to Professional Graphics Programming

Nick Schefner

June 26, 2024

Contents

1 Introduction **2**

2 The GPU **4**

2.1 Why do we need a GPU? 4

2.2 Basic Architecture of the GPU 5

2.3 Summary 6

3 Graphics Rendering Pipeline **7**

3.1 What is the Graphics Pipeline? 7

3.2 The Basics 8

3.2.1 Matrix Transformations 9

3.3 Stages of the Pipeline 11

3.3.1 Draw 12

3.3.2 Input Assembler 12

3.3.3 Vertex Shader 16

Bibliography **18**

1. Introduction

Dear Reader,

Welcome to "Mastering Vulkan: A Professional Guide for Beginners." It is with great pleasure and excitement that I extend my sincerest greetings to you as you embark on this journey into the world of Vulkan API. Before we start, I would like to introduce myself to you.

I began my journey with the Vulkan API about 3 months ago when I embarked on the creation of my own game engine, "Magma". Before that, I have started my programming journey with the well known Unity Game Engine and learned many basics on how games and graphics behave. I also have experience with Glium, which is an OpenGL Wrapper for Rust. For those unfamiliar, Glium further abstracts OpenGL, which made it easier to understand the logic and functionalities of my application. I'm deeply invested on creating a precise representation of the provided topic, which will also be my ground base on further deepening my knowledge and understanding. While researching and writing this book, I anticipate learning a great deal myself. Nonetheless, I am committed to providing you with a comprehensive understanding of visual programming through this book.

Graphical programming is a hard topic with a steep learning curve, I often feel that I missed the dots connecting every aspect of the underlying technologies when first getting started in this field as a game dev. Therefor it might be quite overwhelming at first, to understand how the graphics that we see every day are created and displayed onto our displays. It doesn't help that Vulkan expands on that complexity and provides us with many options to explicitly control certain rendering processes like memory management

or command buffer construction to provide portability and flexibility, whereas older graphic APIs abstracted these processes to simplify the rendering for us developers.

Because Vulkan requires us to explicitly control the hardware resources, you need even more knowledge about the multiple hardware components and rendering possibilities that modern GPUs allow us to twist and tweak.

The main idea off this book is to slowly introduce someone who has had no contact with graphical programming so far, but is excited to dive into an interesting topic that is a must know for every aspiring future game developer or artist that doesn't want to rely on the existing ecosystems. We will first learn to understand the GPU and its usage. I will then focus on explaining pipelines, buffers, swap chains and other components of the render system and explain in detail how they work and are linked together. Besides that I will explain how Vulkan manages each of these components and give an example on how its implementations would look.

I also want to give a special thanks to Brendan Galea for his "Vulkan (c++) Game Engine" series, that helped me out a lot on understanding not just Vulkan as an API but also the basics of graphical programming and its Components.

Lastly, I want to assure you that while learning Vulkan may seem daunting at first, with time and patience, anyone interested in this topic can develop a strong understanding of it.

2. The GPU

2.1 Why do we need a GPU?

The graphics processing unit (GPU) is, like the name implies, a hardware component that is made specificly for processing graphical data. Sure, okay, but why do we need a separate hardware component for that? Why not just let the CPU handle the calculations? The answer is simple: speed and parallelism.

The CPU is great at executing one command after another, but it is not very good at executing multiple commands at the same time. Sure we have multithreading but the CPU is limited to just a few cores. [1] On the other hand, the GPU is mainly build on processing mathematically complex operations like matrix multiplications, that are not just executed once but a multiple of times. [2] Because the GPU focuses on speed and parallelism, it is designed to neglegt the need for complex control structures and caching, which are the main focus of the CPU. [3]

Imagine creating a animation where the color of your full screen changes between red and green. In that case you have a single command, the color change, which is performed on every pixel on the screen. Using a CPU we would have to wait for the CPU to execute the command on every pixel one after another, which would take a lot of time. With the GPU we can execute the command on every pixel simultaneously, which is much faster.

While a CPU usually has 4 to 16 cores, GPUs can have up to almost 16,400 cores. [4] But that doesn't mean that the GPU is 1000 times faster than the CPU. The compute units of the GPU are much slower than the cores of the CPU, but because they can

run in parallel they are more suitable for graphical computations. [3]

The GPU also has its own memory called the video random access memory (VRAM) which is used to store the processed data. I dont want to go into detail about the VRAM because it is not relevant for understanding Vulkan, but it is important to add that the VRAM allows simultaneously reading and writing data, which is a big advantage over the CPU. [5]

It might also be interesting that when the VRAM is full, rendering speed and performance will drop because we need to get the data from a slower memory source like the DRAM. Therefore it is important to manage data efficiently and to not overload the VRAM by loading to many or to detailed textures.

2.2 Basic Architecture of the GPU

I've already mentioned that the GPU can take one instruction and execute it on multiple data at the same time. This is called SIMD (Single Instruction, Multiple Data). [6]

The GPU is made up of thousands of small cores, which are called CUDA compute units on NVIDIA grapahics cards [3] and ROCm compute units on AMD graphics, which I will neglegt to simplify this topic. [7] CUDA stands for "Compute Unified Device Architecture" and is a low level parallel computing platform and programming model that comes with an API for writing programs that can handle the power and advantages of the GPU in an high level language like c or c++. [3] Dont worry, we will not use CUDA or an assembly language in this book, because Vulkan abstracts the usage of the GPU for us, but it is important to understand that the GPU is not just a black box that we can throw commands at and expect it to work. These cores are combined into an array of so called streaming multiprocessors (SM) also known as CUDA blocks, which are the main building blocks of the GPU. Each SM has its own control logic, memory and execution units. [3] These SMs are then combined into a grid called the GPU. When the GPU is invoked by the CPU, a free SM will be assigned to performing the requested tasks and then return the results to the CPU. [3]

The GPU also consists of a memory hierarchy, which is used to store the data that is

processed by the GPU. The memory hierarchy looks like this:

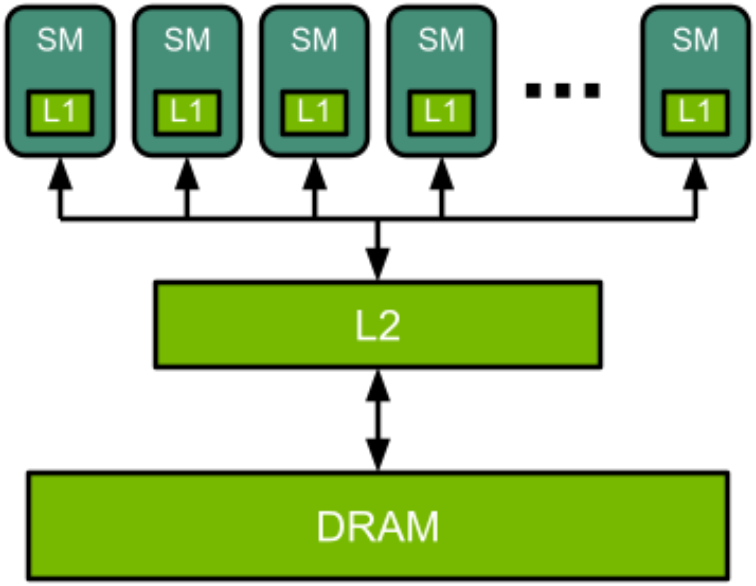

Figure 1: Memory Hierarchy of the GPU [8]

Please note that the VRAM is not part of the memory hierarchy, but is connected to the L2 cache. Caches are used to store frequently used data and to reduce access times to the main memory. While every SM has its own L1 cache, the L2 cache is shared between all SMs. Which means that all cores in one SM are sharing one L1 cache. The CPU on the other hand has one cache and one memory controller per core. [3]

2.3 Summary

Okay that was a lot of information, so let's summarize the really important bits what we have learned so far.

The GPU was created to process graphical data much faster than a CPU could ever do. It allows us to execute the same command on multiple data simultaneously to prevent waiting for a single command to finish, which helps us render reaccuring tasks like drawing a texture or model much faster. It is made up of many cores which are combined into an array of streaming multiprocessors (SM) to allow better control and management of the cores. The VRAM is the memory of the GPU and its important to optimize your data to prevent overloading the VRAM.

3. Graphics Rendering Pipeline

3.1 What is the Graphics Pipeline?

When we talk about the graphics pipeline, I want to make clear that we are not talking about a physical pipeline that is used to transport data from one place to another. I've made that mistake when I first started working with pipelines. My first thought was that the pipeline is a tool to transport data from the CPU to the GPU, but thats totally wrong.

What is it then? Let's start with a simple definition.

The graphics pipeline is an abstraction layer that is located on the GPU and is used to process incoming data to create the image that we see on our screen. [9] This saves us from writing low level code explicitly targeting the GPU, which would be a massive task to do.

Please note that the pipeline is implemented differently in every graphics API, but the basic ideas stay the same. We feed the pipeline with some graphical data, which then performs operations to transform the data into an 2D image that can be displayed on our screen. The proccessed data is then stored in a so called framebuffer, which I will explain later. [9]

There is actually other pipeline types, like the compute or raytracing pipeline, made for different tasks, but we will focus on the graphics pipeline for now.

The graphics pipeline is invoked by a draw command, while the the compute pipeline is invoked by a dispatch command. [10]

[11]

3.2 The Basics

To understand the graphics pipeline, we first need to understand the how the 2D image on our screen is created from the 3D data that we provide to the pipeline.

We start with the world space, which is the 3D space that we use to define the positions of our objects in the scene. The world space is then transformed into the view space, which is the space that the camera sees. [12]

But wait, camera? If you have worked with 3D graphics before, you know that we need a camera that is used to define the space we can view, aka the view space. The camera is an object that has a view frustum with a near and far plane, that is used to define what we can see.

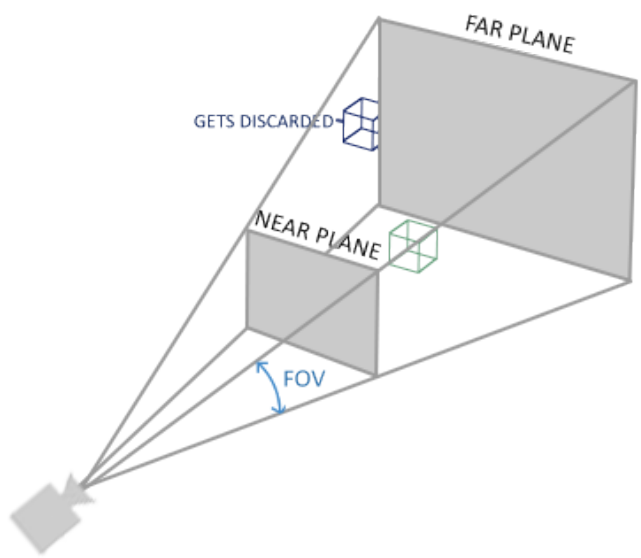

Figure 2: View Frustum [12]

To transform the world space into the view space, we use a view matrix. Let's briefly talk about matrix transformations and how multiplication can be used to move, rotate or scale objects in 3D space.

If you have never touched linear algebra before, I suggest you to go over 2Blue1Brown's Essence of Linear Algebra series on YouTube. It is a great introduction to linear algebra

and will help you understand the further topics.

3.2.1 Matrix Transformations

When we have an box in 3D space, we can represent its position as an array of 3 floats, which are the x, y and z coordinates of its center. To move the box, we can simly add or subtract a value from the x, y or z coordinate. But what if we want to rotate the box? We can't just add or subtract a value from the boxes center to rotate it. We need to takle each corner of the box and move it around the center of the box. This is where matrix transformations come into play.

To rotate the box, we can multiply the position of each corner with a rotation matrix. Wait how do we multipy matrices? It's simple. We multiply the rows of the first matrix with the columns of the second matrix and add them together. The result is the value of the new matrix at the position where the row and column intersect.

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} x \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} = \begin{bmatrix} (aj+bm+cp) & (ak+bn+cq) & (al+bo+cr) \\ (dj+em+fp) & (dk+en+fq) & (dl+eo+fr) \\ (gj+hm+ip) & (gk+hn+iq) & (gl+ho+ir) \end{bmatrix}$$

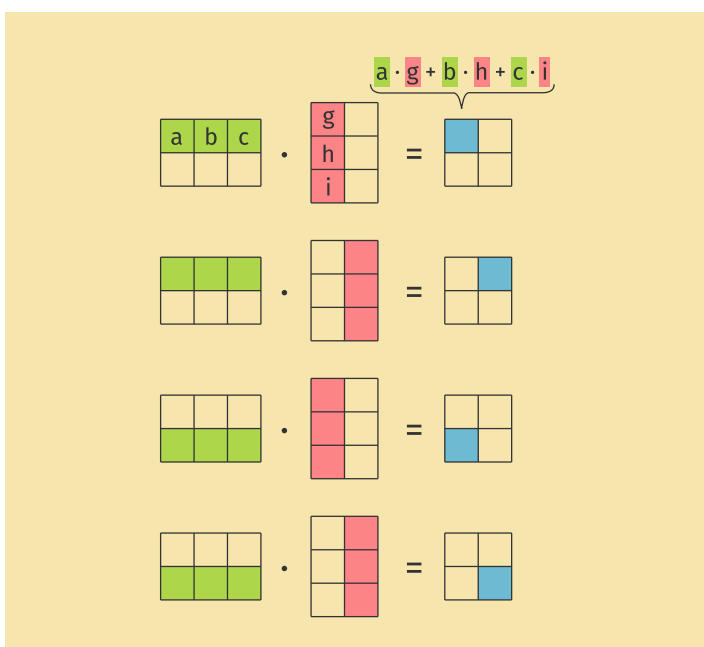

Figure 3: Matrix Multiplication [13]

We can use this knowledge to create a rotation matrix. Lets say we have a line with the points {0, 0, 0} and {1, 0, 0}. When we want to rotate this line around an axis by 45 degrees, we can apply the following rotation matrices to the points:

x-axis:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(45) & -sin(45) \\ 0 & sin(45) & cos(45) \end{bmatrix}$$

y-axis:

$$\begin{bmatrix} cos(45) & 0 & sin(45) \\ 0 & 1 & 0 \\ -sin(45) & 0 & cos(45) \end{bmatrix}$$

z-axis:

$$\begin{bmatrix} cos(45) & -sin(45) & 0 \\ sin(45) & cos(45) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Let's rotate the line around the y axis:

$$\begin{bmatrix} cos(45) & 0 & sin(45) \\ 0 & 1 & 0 \\ -sin(45) & 0 & cos(45) \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} cos(45) & 0 & sin(45) \\ 0 & 1 & 0 \\ -sin(45) & 0 & cos(45) \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} cos(45) \\ 0 \\ -sin(45) \end{bmatrix}$$

You can see that the origin of the line stays the same, but the direction of the line has changed. This is because we actually rotate the entire world space and not the object itself. But because the object is in the world space, it will also be rotated.

3.3 Stages of the Pipeline

The Vulkan pipeline consists of multiple stages. Some of them are fixed and can not be changed, while others are programmable to act as we want them to. These changable stages are called shaders and are usually written in the OpenGL Shading Language (GLSL). These shaders have to be given to the pipeline as compiled SPIR-V bytecode. [14]

Lets take a look at the Vulkan pipeline and its stages:

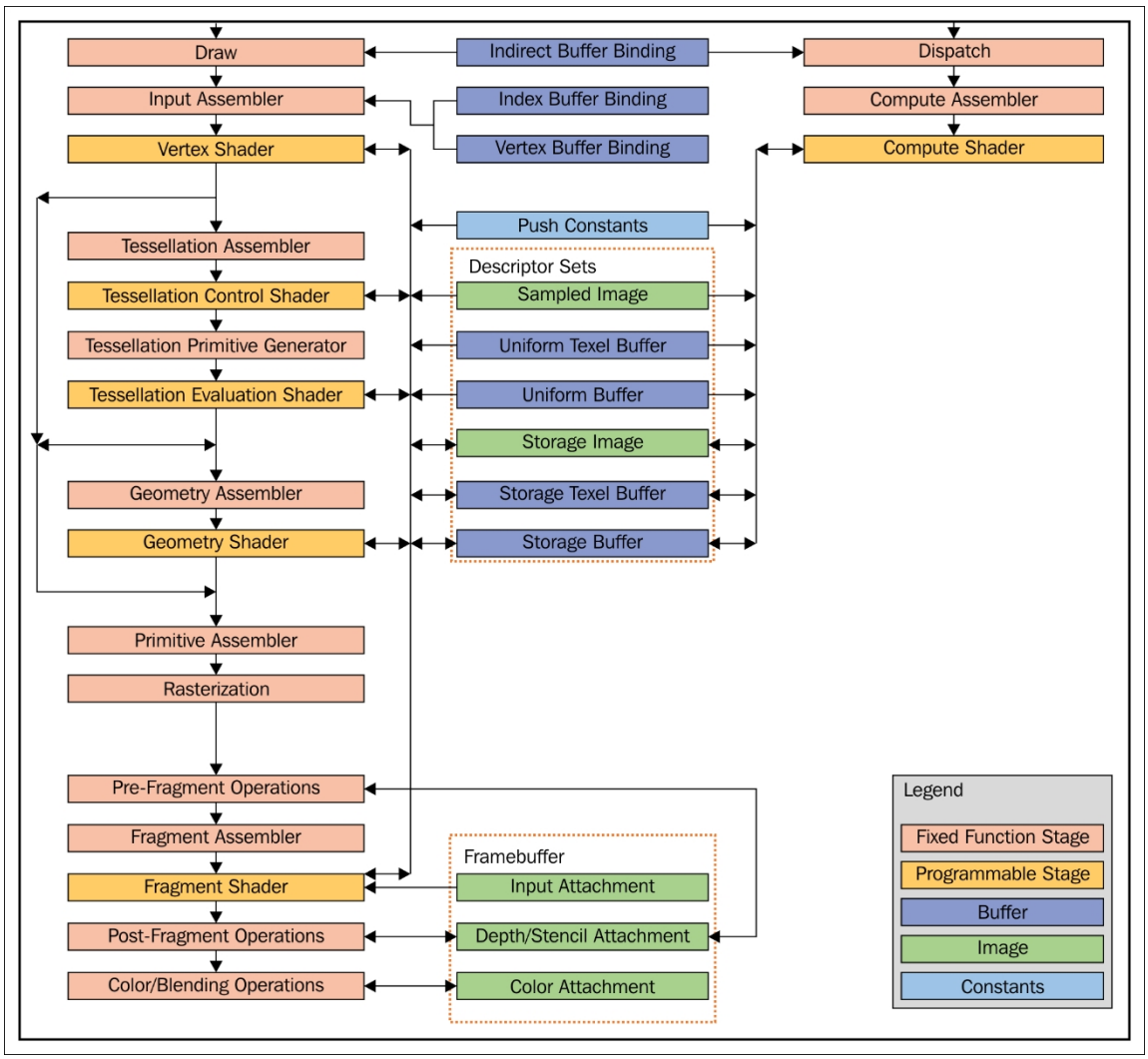

Figure 4: Pipeline Structure [15]

Okay okay, I know thats alot and there is stuff I haven't talked about yet, like these Buffers, Images and the Push Constants. Dont worry I will explain all of these things when going over the stages.

On the left side we have the graphics pipeline and on the right side the compute pipeline. Now it makes sense that there are different pipelines for different tasks, right? If we would use the graphics pipeline but actually just need one shader to edit some data, we would waste a lot of resources and time sending it though the graphics pipeline.

Im going to focus on the graphics pipeline and its most important stages. So some stages will be cut short to better explain the important ones. In the end you will rarely need to use geometry shaders or tessellation shaders, but it is important to know that they exist and what they do.

3.3.1 Draw

The draw "stage" is not really a stage, but the command that starts the rendering process. When we call the vkCmdDraw Command on the GPU, the pipeline will start processing the data that we have given to it. [16]

3.3.2 Input Assembler

The input assembler is the first stage of the graphics pipeline. It takes a vertex and index buffer. [17] Before we talk about the input assembler, lets define what a vertex, index and buffer is.

Basically every model that we see is made up of triangles, and vertices are the corners of these triangles. A single vertex can contain multiple attributes like position {x, y, z}, color {r, g, b, a} or texture coordinates {u, v}. [18]

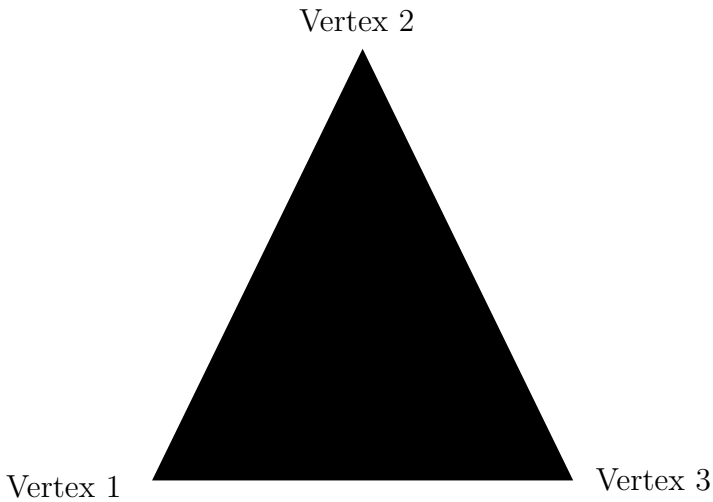

Figure 5: Triangle with vertices

When we want to reuse certain vertices, for example when drawing a quad, we can add indices. These indices tell the input assembler what vertices need to be combined to form a triangle. [19]

For example: A quad has 4 vertices because it has 4 corners. When we want to draw the quad we have to create two triangles, which means we need 2 * 3 indices.

Vertices: {0, 1, 2, 3} Indices: {0, 1, 2 , 0, 2, 3}

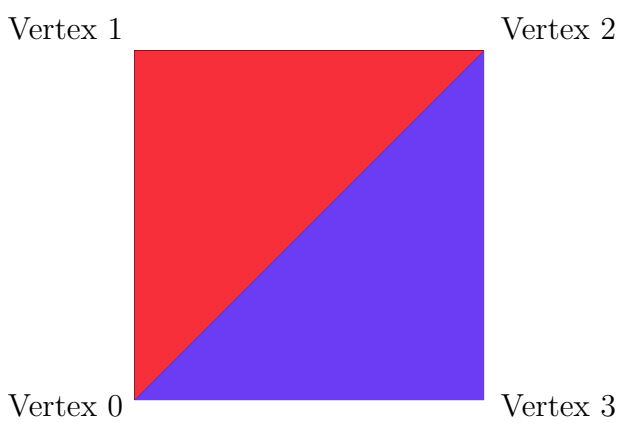

Figure 6: Quad with vertices created using indices

Here you can see that the first triangle is created by using the vertices {0, 1, 2} and the second triangle is created by using the vertices {0, 2, 3}. I've added the colors just to clarify which indices are used to create which triangles.

Now that we know what vertices and indices are, we can talk about how we provide the GPU with this data. There the buffers come into play. A buffer is basically an array of

data that can be bind to the graphics Pipeline. When the pipeline is created, we need to provide it with the layout of the buffer, which tells the pipeline how the data in the buffer is structured. [20]

In the Vertex Buffer we store the attributes of each vertex as an array. An attribute can be a position, color or any other related data of the Vertex. [20]

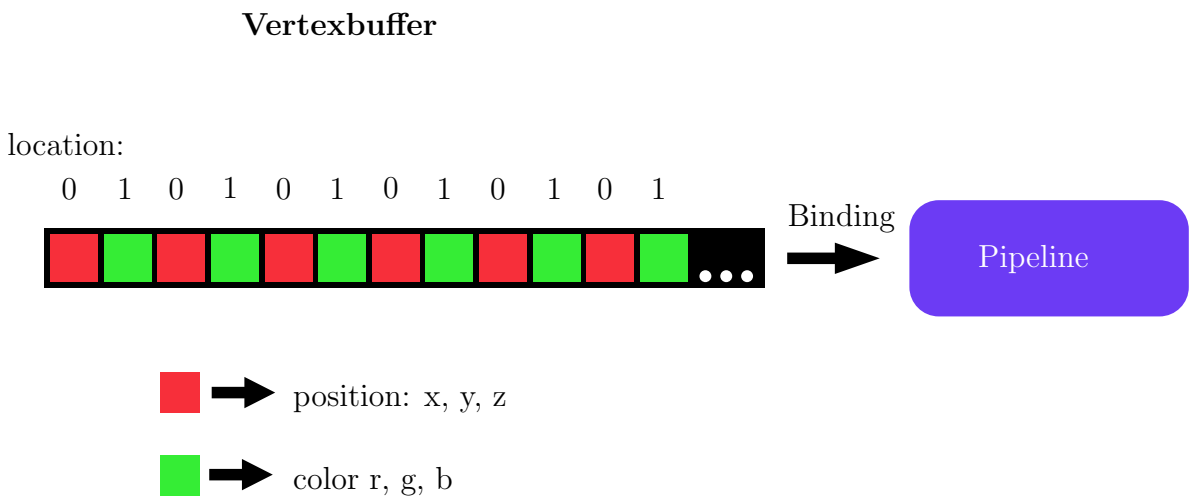

Figure 7: Interleaved Vertex Buffer

You can see that each attribute has a specific location in the buffer. These locations tell the pipeline what bytes belong together. So when accessing the buffers 0. location it gets an vec3(x,y,z). The pipeline also needs to know by what offset, in bytes, each vertex is sperated. In this case it would be 24 bytes, because position and color both contain 3 floats, which are 4 bytes each. [20]

This example shows a single buffer containing all atrributes in an interleaved pattern, but we can also create one buffer for each attribute and then attach them into a single vertexbuffer. This is called a non-interleaved pattern.

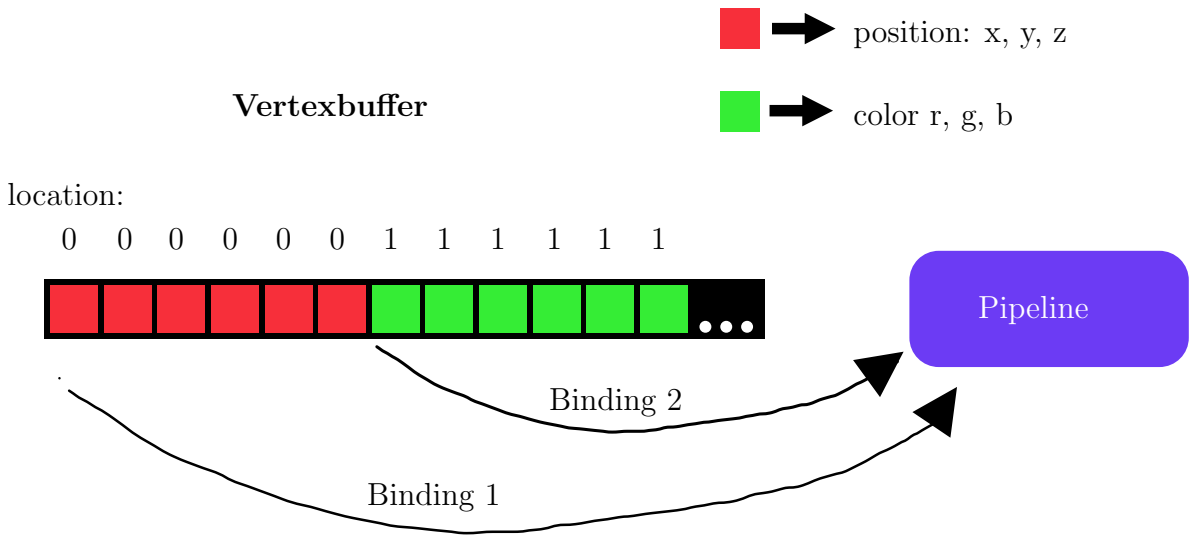

Figure 8: Non-Interleaved Vertex Buffer

Each attribute has has its own binding, to tell the pipeline what location is assigned to the data. So the first binding will always be the position until you get to the next binding, which will be the color in this case. [20]

Usually we will use interleaved buffers, but in some cases, when we need different attributes for different tasks, we can use non-interleaved buffers.

The Indexbuffer on the other hand is just an simple array of integers. [19]

Okay we can now finally talk about the input assembler. The input assembler takes the vertices and indices to create primitives types like points, lines or triangles. [21]

```
// Provided by VK_VERSION_1_0 typedef
enum VkPrimitiveTopology {
  VK_PRIMITIVE_TOPOLOGY_POINT_LIST = 0,
   VK_PRIMITIVE_TOPOLOGY_LINE_LIST
  = 1, VK_PRIMITIVE_TOPOLOGY_LINE_STRIP =
  2, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST =
  3, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP =
  4, VK_PRIMITIVE_TOPOLOGY_TRIANGLE_FAN = 5,
  VK_PRIMITIVE_TOPOLOGY_LINE_LIST_WITH_ADJACENCY = 6,
  VK_PRIMITIVE_TOPOLOGY_LINE_STRIP_WITH_ADJACENCY = 7,
  VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST_WITH_ADJACENCY = 8,
  VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP_WITH_ADJACENCY = 9,
  VK_PRIMITIVE_TOPOLOGY_PATCH_LIST = 10,
} VkPrimitiveTopology;
```

[22]

These are the primitive topology types that the input assembler can create in Vulkan. We have to tell the input assembler what type of topology we want to create, which is usually a triangle strip or list, but we can also create points or lines. The difference between a list and a strip is simple. A list will create a triangle for every 3 indices, while a strip will create a triangle for every 3 indices and then use the last 2 vertices of the last triangle to create the next triangle. The same idea works with lines. The "with adjacency" types are only used when accessing the geometry stage. We will rarely use the adjacany types, so dont worry on understanding them now. [22]

3.3.3 Vertex Shader

The vertex shader is the first programmable stage of the pipeline. It takes the vertices and transforms them by scaling, rotating or translating them into clip coordinates. These coordinates range from -1 to 1 on the x, y and z axis, with the center beginning at {0, 0, 0}.

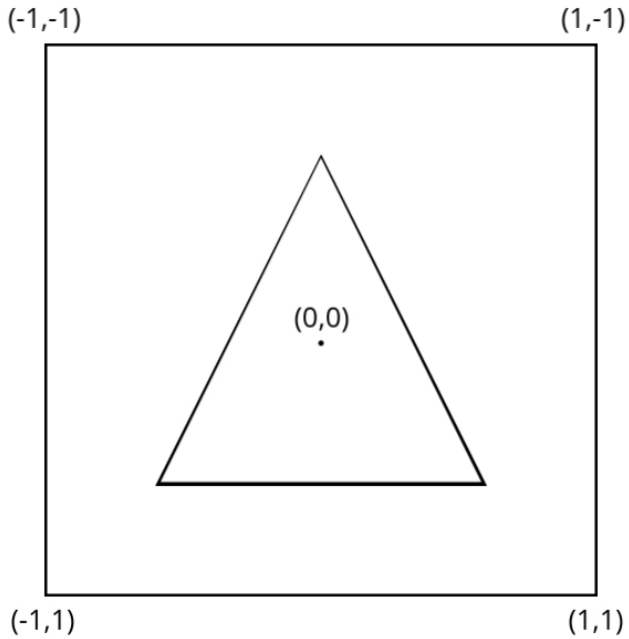

Figure 9: Clip Coordinates in 2D

Please note, that the y axis is inverted in clip space and that there is a z coordinate that can go from 0 to 1, with 0 being directly behind the screen and 1 being the furthest point that will not be clipped off.

The clip spaced is used to determine if a vertex is visible or not, by checking if the vertex is inbetween the -1 and 1 range. [23]

Bibliography

[1] CDW: *CPU vs. GPU: What's the Difference?* 2022. URL: https://www.cdw.com/content/cdw/en/articles/hardware/cpu-vs-gpu.html#4 (visited on 02/23/2024).

[2] B. Caulfield: *What's the Difference Between a CPU and a GPU?* 2009. URL: https://blogs.nvidia.com/blog/whats-the-difference-between-a-cpu-and-a-gpu/ (visited on 02/23/2024).

[3] NVIDIA: *CUDA C++ Programming Guide*. 2024. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (visited on 02/21/2024).

[4] NVIDIA: *GeForce RTX 4090*. 2022. URL: https://www.nvidia.com/de-de/geforce/graphics-cards/40-series/rtx-4090/ (visited on 02/21/2024).

[5] J. (Hub): *Which is Better – VRAM or RAM?* 2023. URL: https://www.electronicshub.org/vram-or-ram/ (visited on 02/21/2024).

[6] M. Levinas: *GPU Architecture Explained: Everything You Need to Know and How It Has Evolved*. 2021. URL: https://www.cherryservers.com/blog/everything-you-need-to-know-about-gpu-architecture (visited on 02/21/2024).

[7] AMD: *AMD ROCm™ Software*. 2024. URL: https://www.amd.com/en/products/software/rocm.html (visited on 02/21/2024).

[8] NVIDIA: *GPU Performance Background User's Guide*. 2023. URL: https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html (visited on 02/26/2024).

[9] V. Blanco: *The graphics pipeline*. 2020. URL: https://vkguide.dev/docs/new_chapter_3/render_pipeline/ (visited on 02/22/2024).

[10] V. Blanco: *The graphics pipeline*. 2020. URL: https://vkguide.dev/docs/new_chapter_3/building_pipeline/ (visited on 02/25/2024).

[11] A. Overvoorde: *Vulkan Tutorial. Compute Shader*. 2023. URL: https://vulkan-tutorial.com/Compute_Shader (visited on 02/25/2024).

[12] J. de Vries: *Getting started. Coordinate Systems*. 2014. URL: https://learnopengl.com/Getting-started/Coordinate-Systems (visited on 06/25/2024).

[13] S. Richter: *Matrix-Matrix-Multiplikation*. URL: https://www.sofatutor.com/mathematik/videos/matrix-matrix-multiplikation (visited on 06/26/2024).

[14] V. Blanco: *Running code on the GPU*. 2020. URL: https://vkguide.dev/docs/new_chapter_2/vulkan_shader_drawing/ (visited on 02/25/2024).

[15] P. Singh: *Learning Vulkan*. 2016. ISBN: 9781786469809.

[16] K. Group: *Vulkan Specification. 21. Drawing Commands*. 2024. URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/html/chap21.html (visited on 02/26/2024).

[17] K. Group: *Vulkan Specification. 10. Pipelines*. 2024. URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/html/chap10.html (visited on 03/01/2024).

[18] K. Group: *Vulkan Guide. Vertex Input Data Processing*. 2023. URL: https://docs.vulkan.org/guide/latest/vertex_input_data_processing.html (visited on 03/01/2024).

[19] A. Overvoorde: *Vulkan Tutorial. Index Buffer*. 2023. URL: https://vulkan-tutorial.com/Vertex_buffers/Index_buffer (visited on 03/01/2024).

[20] A. Overvoorde: *Vulkan Tutorial. Vertex Input Description*. 2023. URL: https://vulkan-tutorial.com/Vertex_buffers/Vertex_input_description (visited on 03/03/2024).

[21] M. W. Steven White: *Universal Windows Platform. Input Assembler (IA) Stage*. 2022. URL: https://learn.microsoft.com/en-us/windows/uwp/graphics-concepts/input-assembler-stage--ia- (visited on 03/03/2024).

[22] K. Group: *Vulkan Specification. VkPrimitiveTopology(3) Manual Page*. 2024. URL: https://registry.khronos.org/vulkan/specs/1.3-extensions/man/html/VkPrimitiveTopology.html (visited on 03/03/2024).

[23] A. Overvoorde: *Vulkan Tutorial. Shader Modules*. 2023. URL: https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Shader_modules (visited on 06/24/2024).

List of Figures

1 Memory Hierarchy of the GPU [8] 6

2 View Frustum [12] 8

3 Matrix Multiplication [13] 9

4 Pipeline Structure [15] 11

5 Triangle with vertices 13

6 Quad with vertices created using indices 13

7 Interleaved Vertex Buffer 14

8 Non-Interleaved Vertex Buffer 15

9 Clip Coordinates in 2D 17