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

Nick Schefner

July 22, 2024

Contents

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 does not 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 does not 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 specifically 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 multi threading 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 neglect 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 don't 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 graphics cards [3] and ROCm compute units on AMD graphics, which I will neglect 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] Don't 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:

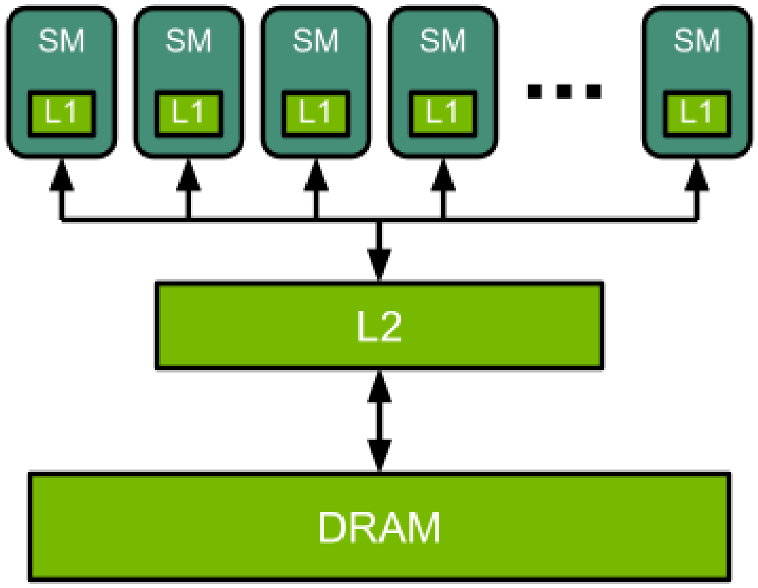

Figure 1: Memory Hierarchy of the GPU [8]

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 reoccurring 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 that's 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 processed 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 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.

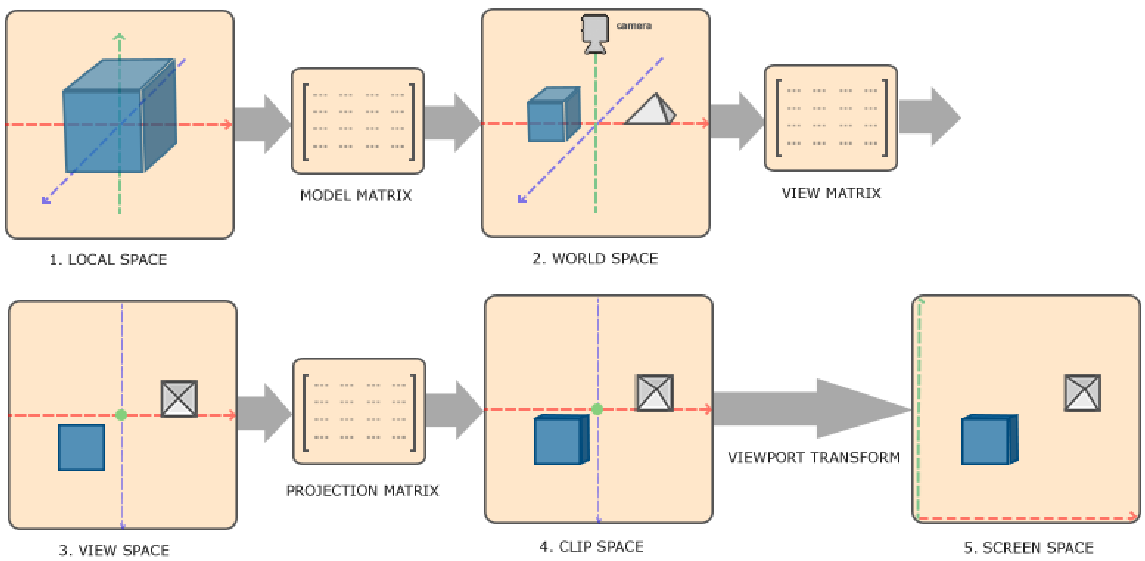

Figure 2: Coordinate Systems [12]

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.

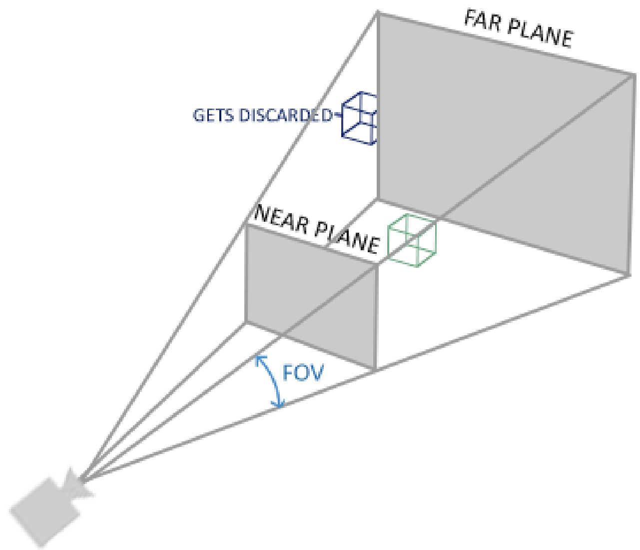

Figure 3: View Frustum [12]

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 Homogeneous Coordinates

When we want to transform a 3D point, we can use a 4x4 matrix to transform the point. But why a 4x4 matrix? Because we need to add a fourth coordinate to the point, which is called the w coordinate. This coordinate will be used to add perspective to the point, so that the point or object will appear smaller when it is further away from the camera. It achieves this by dividing the x, y and z coordinates by the w coordinate.

3.2.2 Matrix Transformations

When we have an box in 3D space, we can represent its position as an array of 4 floats, which are the x, y, z and w coordinates of its center. To move the box, we need to add or subtract a value from the x, y or z coordinate but how can we do that? We can multiply our position with a translation matrix to move the box. Wait how do we multiply 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.

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

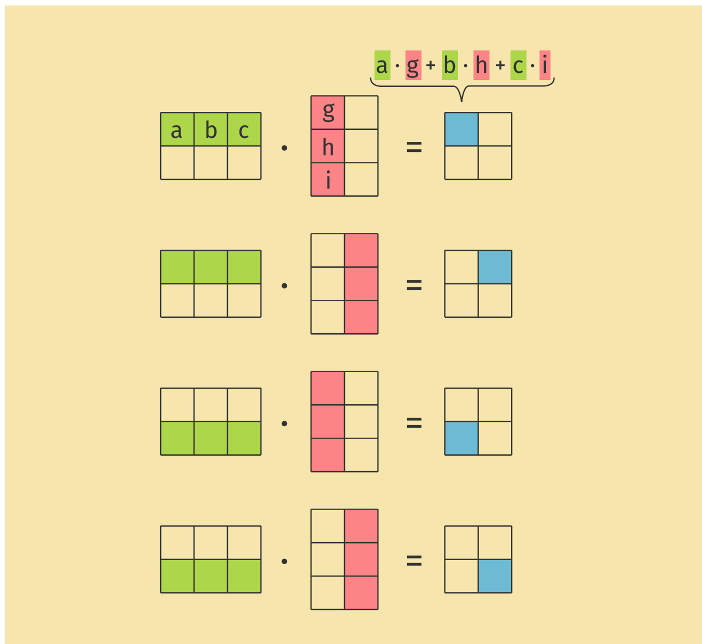

Figure 4: Matrix Multiplication [13]

Therefor we can create a translation that moves the box by {tx, ty, tz}

$$\begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} x \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x+tx*w \\ y+ty*w \\ z+tz*w \\ w \end{bmatrix}$$

Please note that the w coordinate is changing the x, y and z coordinates, to adjust the movement to the perspective of the camera.

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 tackle each corner of the box and move it around the center of the box.

To rotate the box, we can multiply the position of each corner with a rotation matrix.

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 δ degrees, we can apply the following rotation matrices to the points:

x-axis:

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

y-axis:

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

z-axis:

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

Let's rotate the line around the y axis:

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

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

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.

Scaling of an object is a bit simpler than rotation. We just multiply the position of the object with a scaling matrix that looks like this:

$$\begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

You might see that we simply multiply the x, y and z coordinates with the scaling factor we want to apply to that axis. So when we want to scale an object by 2 on the x axis, we would multiply the x coordinate with 2 and leave the y and z coordinates as they are.

3.2.3 View Space

Now that we know how to transform the world space, we can move the camera and the objects, so that the camera is the origin of the view space.

To do so we simply take the position of the camera, for example $\{1, 2, 3, 1\}$, and subtract it from every object in the world space. This will move the camera to the origin of the view space and keep the objects in the same position relative to the camera.

$$\begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -2 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x - 1 * w \\ y - 2 * w \\ z - 3 * w \\ w \end{bmatrix}$$

Looking at the result, we can see that it doesn't really make sense to keep the w coordinate for the camera, because the camera is not a 3D object. Therefor we can set the w coordinate of the camera to 1. This will make the homogeneous coordinates of the camera {x, y, z, 1} equal to {x, y, z} in 3D space.

Now that the camera is the origin of the view space, we need to rotate everything, so that the camera is looking down the z axis. Lets say we have a camera that is is looking down this direction: {x,y,z}. We can create a rotation matrix that rotates the world space so that the camera is looking down the z axis {0, 0, z}.

But first we need to get the angles of the rotation. We can do this by using the dot product of the camera direction and the z axis. The dot product is simply the cosine of the angle between two vectors. So when we have two vectors {x, y, z} and {0, 0, z}, we can calculate the angle between them by using the dot product.

The dot product looks like this:

$$a \cdot b = |a| * |b| * cos(\theta)$$

$$a \cdot b = a_x * b_x + a_y * b_y + a_z * b_z$$

From here we can divide everything by $|a| * |b|$ to get $cos(\theta) = \frac{a \cdot b}{|a| * |b|}$. Let's calculate the angles we need to rotate the camera to look down the z axis.

We start off by rotating the camera, so that it is looking down the xz plane. This means that the y coordinate of the camera direction is 0. By taking the dot product of the camera direction and the x axis on the same hight, we can calculate the angle we need to rotate the camera around the z axis.

$$\begin{bmatrix} x \\ y \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{x}{\sqrt{x^2 + y^2}} = cos(\theta)$$

Now we just need to rotate the xz plane around the y axis, so that the camera is looking up the z axis. We can calculate the angle by taking the dot product of the camera direction after rotating it around the z axis and the z axis itself

$$\begin{bmatrix} x \\ z \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{z}{\sqrt{x^2 + z^2}} = cos(\theta)$$

Let's calculate the angles for the camera direction $\{1, 1, 1\}$.

$$\theta_z = cos^{-1}(\frac{1}{\sqrt{1^2 + 1^2}}) = cos^{-1}(\frac{1}{\sqrt{2}}) = 45°$$

Now that we have the z angle, we can rotate everything by 45 degrees around the z axis.

$$\begin{bmatrix} cos(\theta_z) & sin(\theta_z) & 0 & 0 \\ -sin(\theta_z) & cos(\theta_z) & -1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} \sqrt{2} \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

Now we calculate the angle for the y axis.

$$\theta_y = cos^{-1}(\frac{1}{\sqrt{\sqrt{2}^2 + 1^2}}) = cos^{-1}(\frac{1}{\sqrt{3}}) = 54.7°$$

$$\begin{bmatrix} cos(\theta_y) & 0 & -sin(\theta_y) & 0 \\ 0 & 1 & 0 & 0 \\ sin(\theta_y) & 0 & cos(\theta_y) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \sqrt{2} \\ 0 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \sqrt{3} \\ 1 \end{bmatrix}$$

Don't forget to perform all of this for every object in the world space. The result will be the view space, where the camera is the origin and the objects are rotated so that the camera is looking down the z axis.

3.2.4 Projection Transformation

The next step is the transformation from the view space to the canonical view volume (CVV). The CVV is a that is used to define what we can see on the screen. It is also called the clip space, because it is used to clip everything that is outside of the view frustum.

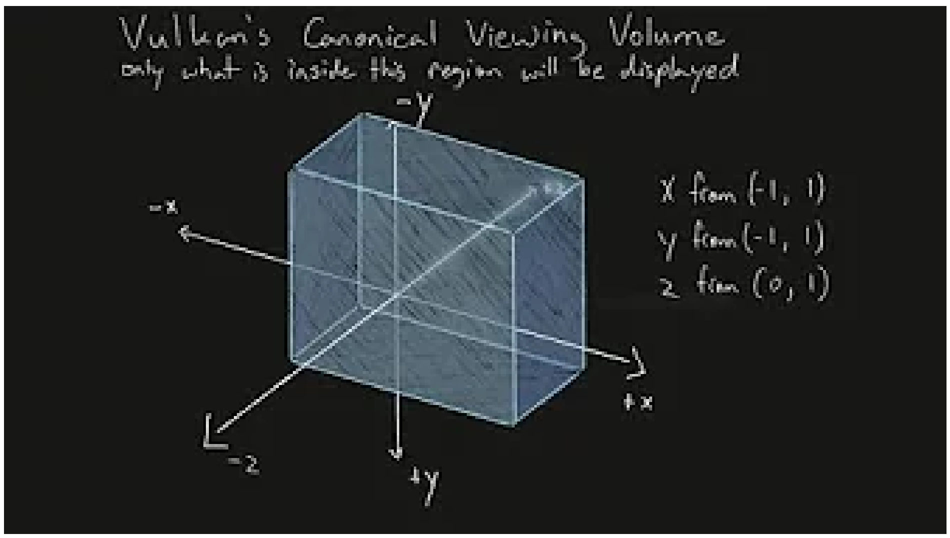

Figure 5: Canonical View Volume [14]

This task involves transforming the view frustum into a orthographic view volume, which is a cube with the size of the near plane and the same depth as the frustum. Then we move the orthographic view volume to the origin and scale it to the size of the cvv.

Let's start with the transformation of the view frustum to the orthographic view volume. We basically need to scale the objects so that the far plane is the same size as the near plane. Normally the near plane is 0.1 units away and the far plane is 100 units away.

When we're looking at the frustum from the side, we can see that the camera and the far plane create a triangle that can be divided into two right triangles at the z axis.

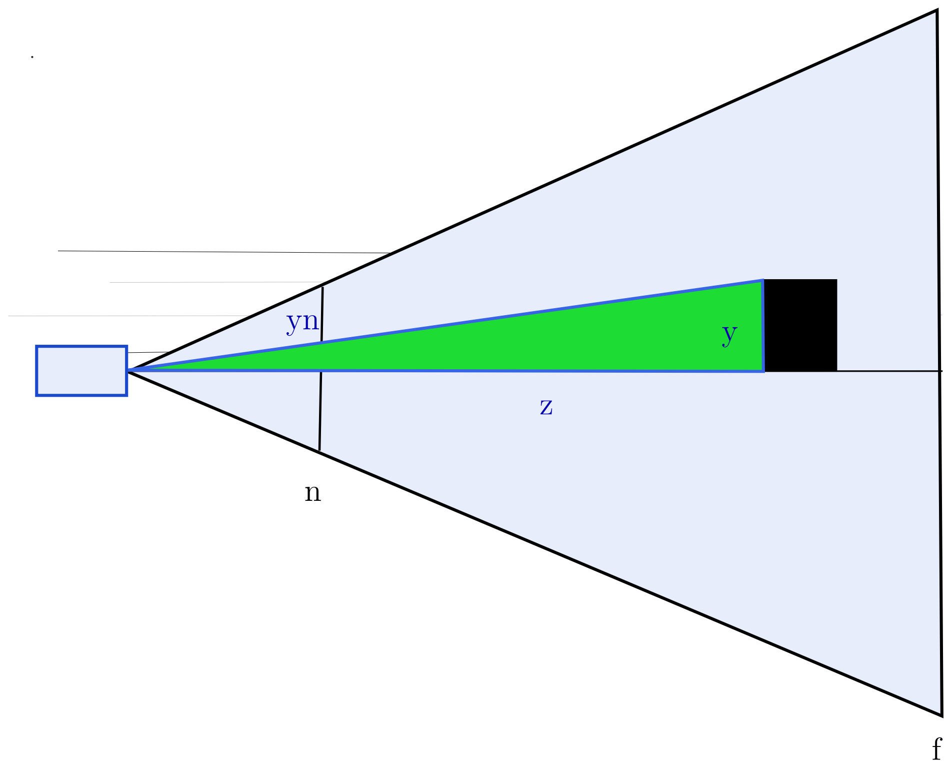

Figure 6: View Frustum

What we need to calculate is yn, which is the new height of the object. To do so we use the green triangle and the near planes distance on the z axis (n) to get the following formula:

$$\frac{yn}{n} = \frac{y}{z}$$

We can use this because the triangles sin angle is the same in the green triangle. When we solve this for yn, we get:

$$yn = \frac{y * n}{z}$$

The same can be applied for the x axis:

$$xn = \frac{x * n}{z}$$

The z axis stays the same.

So when we have a point {x, y, z} in the view space, we need to transform it to $\{\frac{x*n}{z}, \frac{y*n}{z}, z\}$ to get the orthographic view volume. Unfortunately there is no way we can divide the x and y value with z with just 3 dimensions. We need to add a 4th coordinate to the point to be able to divide by z. This is called the perspective divide.

To do so we add a homogeneous variable to the point and transfer the z coordinate to this new w variable. This will allow us to divide the x, y and z coordinate by the z coordinate. So what we want is to transform the point {x, y, z, w} to $\{x*n, y*n, z^2, z\}$. Keep in mind that the homogeneous coordinate is the same as the 3D coordinate after dividing by w.

Okay so what we need is a 4x4 matrix that multiplies the x and y coordinate with n and the z coordinate with itself. We also need to multiply the w coordinate with z, to move the z coordinate to the w coordinate. This works because w is like a scaling factor for the x, y and z coordinate but is often set to 1.

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & m_1 & m_2 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x*n \\ y*n \\ z^2 \\ z \end{bmatrix}$$

Okay we have one last problem. How do we get to z^2 ? First we have the equation $z^2 = m_1 * z + m_2$, considering w is 1. The problem is that there are 2 possible solutions for m1 and m2, which we can not work with. So we need to add 2 constrains to the equation. We can just say that we want to apply this equation to the near and far planes z coordinate. So we will say z = n and z = f. This will give us 2 equations that we can solve for m1 and m2.

$$\begin{bmatrix} n^2 \\ f^2 \end{bmatrix} = \begin{bmatrix} m_1 * n + m_2 \\ m_1 * f + m_2 \end{bmatrix}$$

With this we can now calculate m1 and m2.

$$n^2 = m_1 * n + m_2$$

$$m_2 = n^2 - m_1 * n$$

$$f^2 = m_1 * f + m_2$$

$$f^2 = m_1 * f + n^2 - m_1 * n$$

$$f^2 - n^2 = m_1 * f - m_1 * n$$

$$f^2 - n^2 = m_1 * (f - n)$$

$$m_1 = \frac{f^2 - n^2}{f - n}$$

$$m_1 = f + n$$

$$m_2 = n^2 - (f + n) * n$$

$$m_2 = n^2 - f * n - n^2$$

$$m_2 = -f * n$$

Therefor the final matrix looks like this:

$$\begin{bmatrix} n & 0 & 0 & 0 \\ 0 & n & 0 & 0 \\ 0 & 0 & f+n & -f*n \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

This matrix will actually only give us z^2 on the near and far plane. The Objects in between will have a little offset on the z axis but will remain linearly interpolated between the near and far plane.

Now that we have the orthographic view volume, we need to move it to the origin and scale it to the size of the CVV. The CVV looks like this from the front:

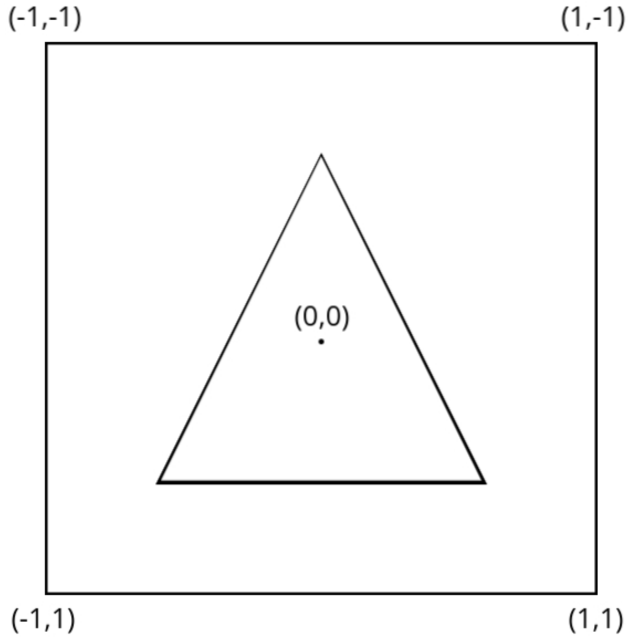

Figure 7: Canonical View Volume Front

It is a coordinate system where the y axis is flipped and therefore the positive of the z axis is pointing away from us. Each axis ranges from -1 to 1 except the z axis, which ranges from 0 to 1. This is because the everything negative on the z axis is behind us and therefor not visible.

Lets first move the orthographic view volumes near planes center to the origin.

$$\begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

When the near plane is 0.1 units away, we need to move the orthographic view volume by 0.1 units on the z axis. So the c_z value is the distance of the near plane on the z axis aka n.

For the x and y value we just have to calculate the center of the near plane. This is done by adding the opposite sides together and then dividing them by 2. So for c_x we get $\frac{l+r}{2}$ and for c_y we get $\frac{t+b}{2}$.

Next we scale the orthographic view volume to the size of the CVV. This is done by multiplying the x, y and z coordinate with the dimension of the canonical view volume over the size of the orthographic view volume. We can get the size from subtracting the right side from the left side for the x axis, the top from the bottom for the y axis and the far from the near plane for the z axis. So the scaling matrix looks like this:

$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{1}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Now we can combine both transformations to get the final projection matrix:

$$\begin{bmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -n \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{1}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{l+r}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{b-t} \\ 0 & 0 & \frac{1}{f-n} & -\frac{n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This is the projection matrix that is used to transform the view space to the clip space. The clip space is then used to determine what is visible on the screen and what is not. Everything that is visible will now be projected onto the 2D screen.

3.2.5 Viewport Transformation

The viewport is the area on the screen where the image is displayed. It is defined by the x and y position, the width and height and the min and max depth and starts at the top left corner of the screen. The viewport transformation is used to transform the NDC to the screen space. This is done by scaling the x and y coordinate to the width and height of the viewport, scaling and moving the z coordinate to the depth range and then adding half of the width to the x coordinate and subtracting half of the height from the y coordinate to move the origin to the top left corner of the screen.

$$\begin{bmatrix} \frac{width}{2} & 0 & 0 & \frac{width}{2} \\ 0 & \frac{height}{2} & 0 & \frac{height}{2} \\ 0 & 0 & maxDepth - minDepth & minDepth \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

This will transform the NDC to the screen space, but we can also add an x and y offset to the viewport to move the image around on the screen. That would mean we need to add the x and y offset when moving the origin to the top left corner of the screen.

$$\begin{bmatrix} \frac{width}{2} & 0 & 0 & \frac{width}{2} + x \\ 0 & \frac{height}{2} & 0 & \frac{height}{2} + y \\ 0 & 0 & maxDepth - minDepth & minDepth \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Wow that was a lot of complicated math, but it's important to understand how the world is projected onto our screen. That's the whole point of 3D graphics after all :P

Don't worry tho if you don't quite understand everything, maybe dive into some other resources to further understand this topic and until then, do what programmers do best. Copy and paste it from somewhere else.

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 changeable 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. [15]

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

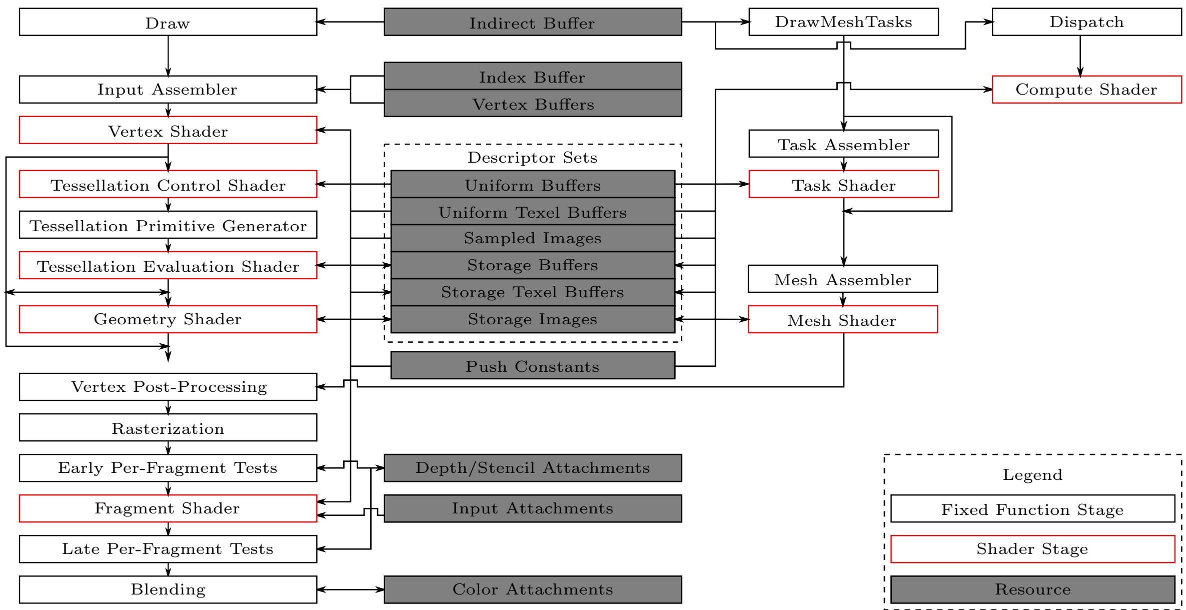

Figure 8: Pipeline Structure [16]

Okay okay, I know that's a lot and there is stuff I haven't talked about yet, like these Buffers, Images and the Push Constants. Don't 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.

I'm 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. [17]

The vkCmdDraw Command is part of the Command Buffer, which is used to record and execute commands on the GPU. This Command Buffer is then submitted to a VkQueue, which will execute the commands on the GPU in the order they were recorded. [18]

3.3.2 Input Assembler

The input assembler is the first stage of the graphics pipeline. It takes a vertex and index buffer. [19] 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}. [20]

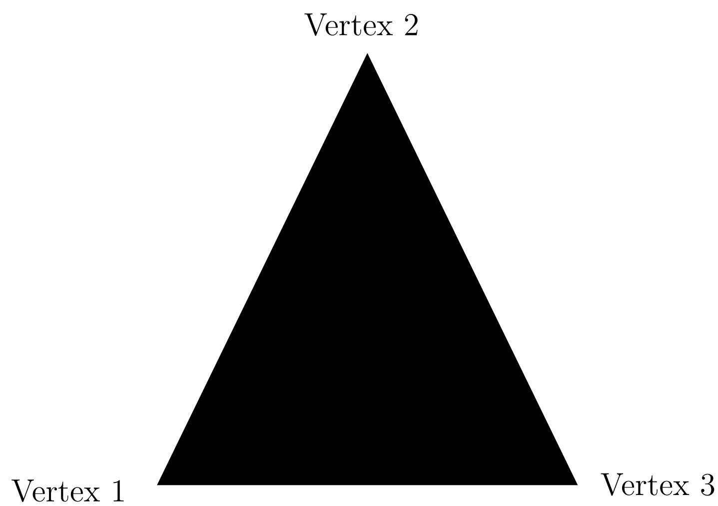

Figure 9: Triangle with vertices

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. [21]

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}

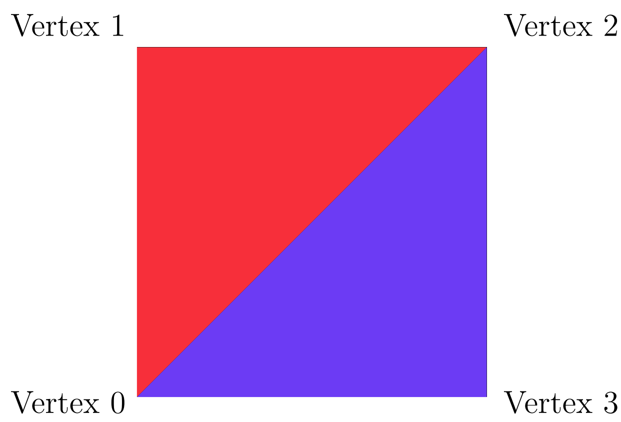

Figure 10: Quad with vertices created using indices

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. [22]

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. [22]

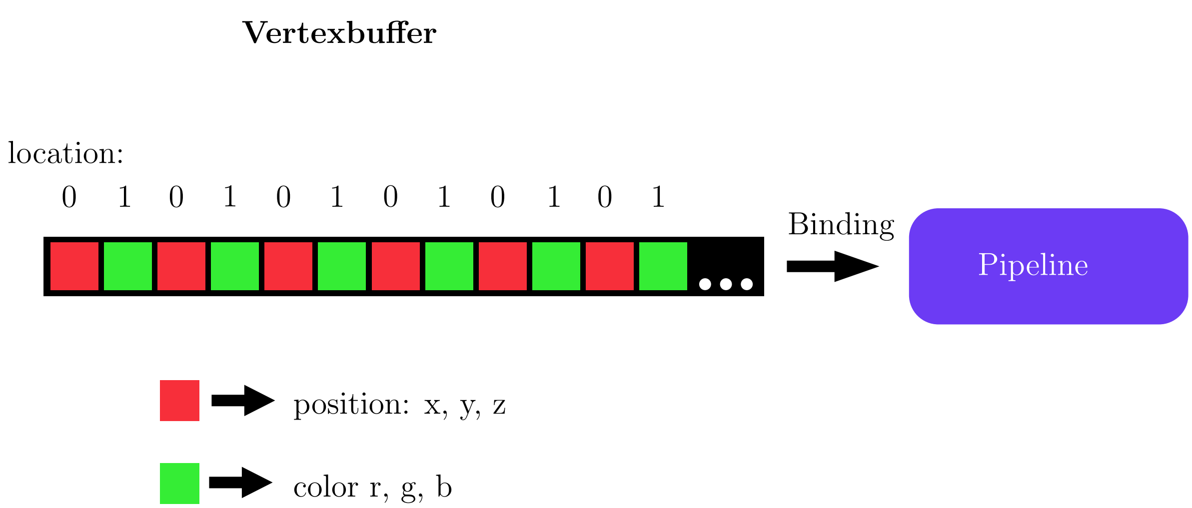

Figure 11: Interleaved Vertex Buffer

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 0st location it gets an vec3(x,y,z). The pipeline also needs to know by what offset, in bytes, each vertex is separated. In this case it would be 24 bytes, because position and color both contain 3 floats, which are 4 bytes each. [22]

This example shows a single buffer containing all attributes 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.

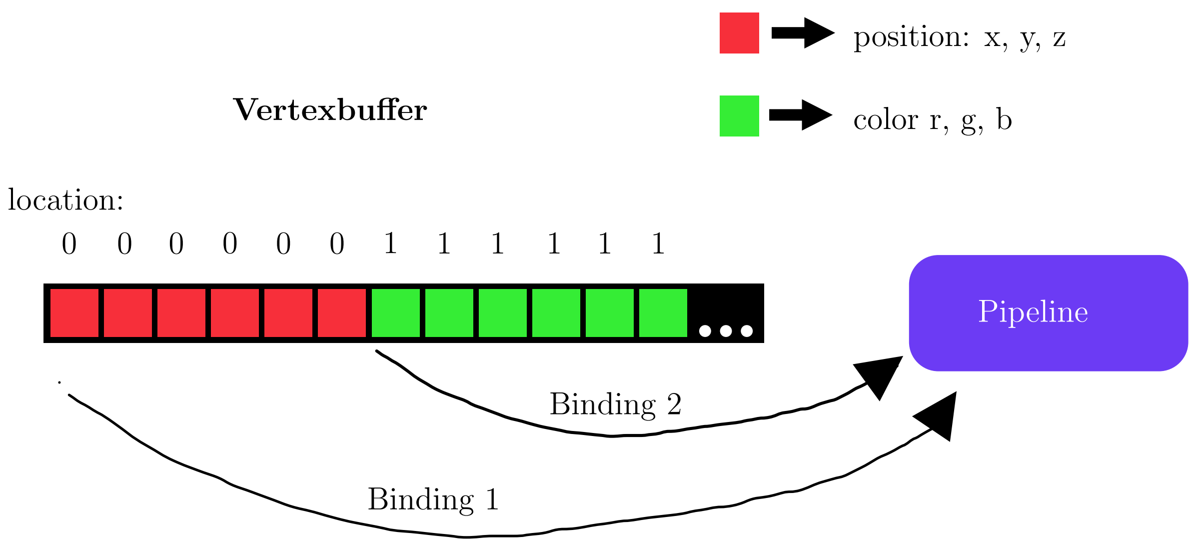

Figure 12: Non-Interleaved Vertex Buffer

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. [22]

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. [21]

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. [23]

```
// 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;
```

[24]

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 adjacency types, so don't worry on understanding them now. [24]

3.3.3 Vertex Shader

The vertex shader is the first programmable stage of the pipeline. Here we transform the world to our clip space. The vertex shader takes the vertex data from the input assembler and transforms each vertex to it's clip space position. [25]

But we need to program the transformation ourselves. We can use our new knowledge of matrices to transform the vertices. To do so we pass the model, view and projection matrix to the vertex shader via resource descriptors.

Resource descriptors are used to pass data to the shaders during the rendering process. There are multiple types of descriptors, but we will focus on the uniform buffer descriptor (UBO) which passes data to the shaders in the form of a buffer.

But first we have to create a descriptor layout that tells the pipeline what type of data we are passing to which shader. Then we create a descriptor pool that holds multiple descriptor sets. The descriptor sets are used to bind the buffers to the shaders and are created from the descriptor pool. [26]

Descriptors are good to pass constant data to the shaders, but when we want to pass data that changes frequently we can use push constants. Push constants are a small amount of data that is stored in the command buffer. We can assign a minimum of 128 bytes to the push constants, which is enough to pass 2 4x4 matrices to the shaders. [27]

After that the pipeline performs the perspective divide and returns the normalized device coordinates.

3.3.4 Tessellation

The tessellation stage is actually divided in 3 stages. The tessellation control shader, tessellation primitive generator (tessellator) and the tessellation evaluation shader. Please note that the Tessellation stage is optional and will only be executed if both shaders are defined in the pipeline.[28]

If you are not familiar with tessellation, it is a technique used to subdivide a polygon, a shape, into smaller polygons without leaving any gaps in between. This can be used to create more detailed models without having to pass all the vertex data from the CPU to the GPU. [28]

The tessellation control shader is used to define how many times our primitive should be subdivided, by defining inner and outer tessellation levels. It also passes the vertex positions to the evaluation shader. [28]

After setting the tessellation levels, the tessellator subdivides a patch, which is a new primitive type, into smaller patches. This can be done in 3 ways: Triangles, Quads or Isolines. Lets first talk about the tessellation of a triangle.

Vulkan uses barycentric coordinates to subdivide the triangle. Barycentric coordinates consist, in the case of a triangle, of 3 variables (u,v,w) that sum up to 1. Each variable represents the relative position of a vertex in the triangle with 1 being directly on the vertex. So when we have a triangle with vertices A, B and C, the barycentric coordinates could be {0, 0, 1} for A, {0, 1, 0} for B and {1, 0, 0} for C, depending on the selected origin of the triangle. [28]

A point in the triangle ABC, like the center, could be calculated like this:

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \div 3 = \begin{bmatrix} \frac{1}{3} \\ \frac{1}{3} \\ \frac{1}{3} \end{bmatrix}$$

Because the center is equally distant from all vertices, the barycentric coordinates need to be equal for u, v and w and when we sum them up they need to equal 1.

First the inner tessellation levels subdivides the patches edges into the desired amount of segments and creates temporary vertices on the edges of the patch. These vertices are then used to create an inner triangle by taking the 2 neighboring vertices of the outer triangles corners and extending perpendicular lines to the edge of the outer triangle. Where these two lines intersect, a new vertex is created. When you do this for all 3 corners it creates the corners of the inner triangle. [28]

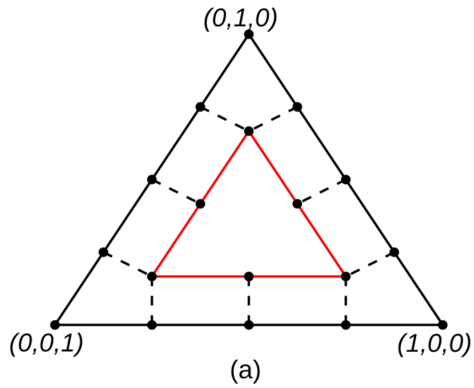

Figure 13: Inner Tessellation with 4 segments
[29]

You can see that the inner tessellation level of 4 cuts each edge of the outer triangle into 4 segments. This is done by calculating the barycentric coordinates of the temporary vertices on the edges. Let's calculate the barycentric coordinates of the first vertex on the left edge AB.

We need 3 points where u is always 0 because the vertices are on the opposite edge of the corner 1,0,0 To get the distance of one segment we divide the edge length by the desired segments length.

So the length of one segment is $\frac{1}{4}$. For the first vertex we then get $\{0, \frac{1}{4}, \frac{3}{4}\}$, for the second vertex $\{0, \frac{2}{4}, \frac{2}{4}\}$ and for the third vertex $\{0, \frac{3}{4}, \frac{1}{4}\}$.

We do the same for each edge and then calculate the intersection of the perpendicular lines to get the inner triangles corner vertices.

For the bottom left corner of the inner triangle we take the 2 vertices of the outer triangle $\{0, \frac{1}{4}, \frac{3}{4}\}$ and $\{\frac{1}{4}, 0, \frac{3}{4}\}$.

Now we need to get the direction of the perpendicular line. This is way easier than it sounds, because our barycentric triangle is equilateral. That means that when we draw a line to display the height of the triangle and take it's direction, we get the direction of the perpendicular line to the lower edge. Luckily, because the triangle stays the same no matter how we rotate it, we can just use the direction of the height line to get the direction of the perpendicular lines for each triangle. [30]

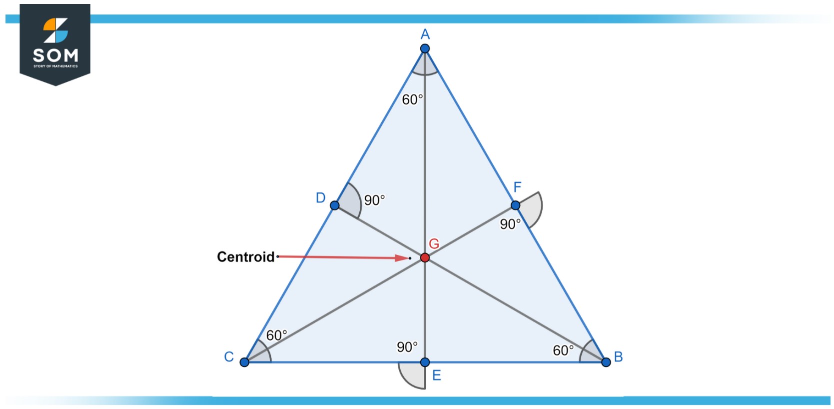

Figure 14: Equilateral Triangle
[30]

Considering that we can calculate the direction of the perpendicular line:

$$E(\frac{1}{2}, 0, \frac{1}{2})$$
$$A(0, 1, 0)$$
$$A - E = (0 - \frac{1}{2}, 1 - 0, 0 - \frac{1}{2}) = (-\frac{1}{2}, 1, -\frac{1}{2})$$

We can now add this direction to the vertex $\{\frac{1}{4}, 0, \frac{3}{4}\}$ to get the perpendicular lines function.

$$f(v) = \{\frac{1}{4} - \frac{1}{2}v, 0 + v, \frac{3}{4} - \frac{1}{2}v\}$$

It's the same for the other points, but we need to change the direction of the perpendicular line.

$$f(u) = \{0 + u, \frac{1}{4} - \frac{1}{2}u, \frac{3}{4} - \frac{1}{2}u\}$$

To get the intersection we simply set the functions equal to each other and solve it.

$$
\begin{aligned}
& & f(v) &= f(u) \\
I) & & \frac{1}{4} - \frac{1}{2} \cdot v &= u \\
II) & & v &= \frac{1}{4} - \frac{1}{2} \cdot u \\
III) & & \frac{3}{4} - \frac{1}{2} \cdot v &= \frac{3}{4} - \frac{1}{2} \cdot u \\
& & v &= u \\
I) & & \frac{1}{4} - \frac{1}{2} \cdot u &= u \\
& & \frac{3}{2} \cdot u &= \frac{1}{4}
\end{aligned}
$$

$$
u = \frac{1}{6} \qquad v = \frac{1}{6} \qquad w = \frac{4}{6}
$$

Voilà, that's how we get the corners of the inner triangle. Now vulkan subdivides the inner triangles edges by n-2 with n being the inner tessellation level. Because our tessellation level is 4 we subdivide the first inner triangles edges into 2 pieces. The vertex dividing the inner edge is then calculated by projecting a perpendicular line form the outer edge to the inner edge. We repeat this until n is smaller than 3, but when we start with n = 2 we will have a tessellation vertex in the center of the triangle. [28]

When we have calculated all inner tessellation vertices we fill the area of the concentric triangles with triangles by connecting the vertices in a way that ensures that the triangles are not overlapping. If the inner tessellation level is 2 and no outer tessellation level is set, the tessellator will connect the corners of the triangle with the center vertex. [28] Unfortunately, considering more complicated tessellation, the order in witch the vertices are created and connected is implementation dependent, which means that it is up to the GPU to decide how the vertices are connected. [28]

After the inner patches are filled, we discard the outer triangles temporary vertices and subdivide each edge by it's outer tessellation level. The tessellator now fills the area of the outer triangle with the outermost inner triangle. Now that all the triangles are created, the tessellation primitive generator assigns the vertices their barycentric coordinates (u,v,w) and passes them to the tessellation evaluation shader. [28]

For Quads and isolines the process is similar, but the patch only has 2 barycentric coordinates. So we can just calculate the barycentric coordinates as if we were in an x,y coordinate system. [28]

Quads have 4 outer tessellation levels that subdivide the outer edges and 2 inner tessellation levels that subdivide the area of the quad into smaller quads. The first inner tessellation level divides the columns and the second inner tessellation level divides the rows. At the end the tessellator fills the area of the quad with triangles. [28]

Isolines on the other hand is a set of horizontal lines that have a length of 1 on the u axis and are equally spaced on the v axis. They only have 2 outer tessellation levels that define how many single lines are created and in how many segments they are divided.

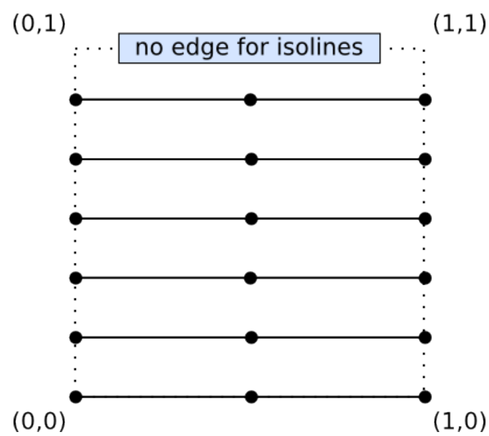

Figure 15: Isoline Tessellation (6, 2)
[31]

These were the possible tessellation types that the tessellator can work with but there are a few more things to consider for each of them. For example, we can change the tessellator spacing, which defines the spacing of the outer tessellation levels. The tessellator spacing can be either equal, fractional even or fractional odd, which allow us to have a tessellation level of 2.5 for example. [28] We can also control the vertex winding order, which defines which side of the triangle is the front and which is the back. This is important for culling and lighting calculations. [28]

Please note that all of this is already handled by vulkan and all we have to do is to set the tessellation levels and optionally spacing and winding order in the tessellation control shader and pass it over to the tessellation primitive generator. [28]

After the tessellator created all of the new triangles the tessellation evaluation shader is called to transform the new vertices to the clip space. The tessellation evaluation shader takes each barycentric coordinate and the position of the original triangles corner vertices, that we have forwarded in the control shader, to calculate the position of the new vertices in clip space. [28]

To do so we can multiply the barycentric coordinates with the corner vertices and add them together to get the new vertex position. For a triangle with vertices A, B and C and barycentric coordinates {u, v, w} the new vertex position would be calculated like this:

$$x = u \cdot A_x + v \cdot B_x + w \cdot C_x$$
$$y = u \cdot A_y + v \cdot B_y + w \cdot C_y$$
$$z = u \cdot A_z + v \cdot B_z + w \cdot C_z$$

After that we can transform them further, adding curvature or displacement to the vertices. When using a quad we can adjust the vertices by using a 2D texture like a height map to displace the vertices. [28]

One last thing to mention is that when we use a tessellation shader we actually skip the perspective projection in the vertex shader and instead do it in the tessellation evaluation shader. This is because the newly created vertice have to be projected to the clip space along with the original vertices. [28]

3.3.5 Geometry Shader

The next stage is the geometry shader (GS). The GS is used to create new primitives and is great at procedural generation of geometry. That includes creating particles, grass or fur. It takes all the vertices of a primitive and generates new vertices that are then combined to either points, line strips or triangle strips. For example, we could take a single point and create a quad around it by emitting 4 new vertices. These will then be combined to 2 triangles by the pipeline, forming a quad. [32]

I will not go further into explaining the geometry shader because its implementation is very dependent on the use case. It is enough to know some use cases and that we basically can create new primitives with it.

3.3.6 Vertex Post Processing

After all of the previous stages we can do some post processing on the vertices. This is done by the vertex post processing stage. Here we have many sub stages that I will go over briefly.

Transform Feedback is used to capture the output of the previous stages and store it in a feedback buffer. This can be used to call a new draw command with the stored data. [33]

Viewport Swizzle is used to change the orientation of the viewport. For example, we can flip the y axis to display the image upside down. To achieve this we can define the swizzle for each axis via the VkViewportSwizzleNV struct. [33]

Flat shading is used to color the whole primitive with one color. This is done by calculating the color of the first vertex and then using this color for the whole primitive. [33]

Primitive Clipping is used to discard primitives that are outside of the clip space. This is done by checking if the primitive is completely outside of the clip space and then discarding it. [33] When only a part of the primitive is outside of the clip space, the pipeline will discard that part and create new vertices on the edge of the clip space, where the primitive would intersects the clip space. [33]

Clipping is done by checking if the vertex's x, y and z coordinates are in between -w and w or z_min and w for z. If they are not, the vertex is outside of the clip space and will be discarded.

Clipping Shader Outputs is used to discard the vertex output attributes that are outside of the clip space.

Controlling Viewport W Scaling can be used to change the w coordinate of the vertex. This can help to adjust the depth of an object.

Coordinate Transformation is the step where our clip coordinates (x,y,z,w) are transformed to (x,y,z) by dividing x, y and z by w. This step is the perspective divide and provides us with normalized device coordinates (NDC) [33]

Render Pass Transform can be enabled to rotate the viewport on the xy plane. This can be used to rotate the image on the screen by either 90, 180 or 270 degrees, using the VK_SURFACE_TRANSFORM_ROTATE_degree_BIT_KHR attribute. [33]

Controlling the Viewport is a stage where vulkan scales the NDC to the dimensions of the viewport. It is also possible to create scissors to only render a part of the viewport.

3.3.7 Rasterization

This stage of the pipeline converts our primitives into a 2D image. It takes our viewport and checks which pixels are covered by the primitives and then fills them with the color of the primitive.

Let's take a look on how this is done. Again, there are multiple primitives that can be rasterized: points, lines and triangles.

We'll start with triangles, because they will be needed for the other types too.

The first step of rasterization is determining if the triangle is front or back facing, which is done by checking if the area of the triangle is positive or negative. We can calculate the area of a triangle using the following formula:

$$a = -\frac{1}{2}\sum_{i=0}^{n-1}(x_i \cdot y_{i\oplus 1} - x_{i\oplus 1} \cdot y_i)$$

This probably looks confusing so let me explain it a bit further. n is the number of vertices in our triangle, which is 3, while i is the current vertex's index and $\oplus$ is basically performing $i + 1$ mod n. So when we have a triangle with vertices A, B and C the sum could look like this:

$$A_x \cdot B_y - B_x \cdot A_y + B_x \cdot C_y - C_x \cdot B_y + C_x \cdot A_y - A_x \cdot C_y$$

or like this:

$$A_x \cdot C_y - C_x \cdot A_y + C_x \cdot B_y - B_x \cdot C_y + B_x \cdot A_y - A_x \cdot B_y$$

depending on the order of the vertices. If your familiar with vector math you might recognize this as the cross product of two side vectors that originate from the same vertex. The cross product of these two vectors gives us a vector that is perpendicular to the triangle, which can either point towards us or away from us. This is because our 2D display can be seen as a 3D space with the z axis pointing out of the screen.

For example we can take the two vectors $\vec{AB} = B - A$ and $\vec{AC} = C - A$ and calculate the cross product of them to get the normal of the triangle.

To calculate the cross product we can look at this visual representation:

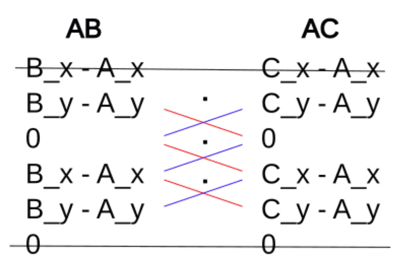

Figure 16: Cross Product

We simply write each vector 2 times under each other, cut out the top and bottom row and then multiply diagonally, subtract the blue result from the red result and add each "cross" result together.

In this case we would have:

$$(B_y - A_y) \cdot 0 - 0 \cdot (C_x - A_x) = 0$$

$$0 \cdot (C_x - A_x) - 0 \cdot (B_x - A_x) = 0$$

$$(B_x - A_x) \cdot (C_y - A_y) - (C_x - A_x) \cdot (B_y - A_y) =$$

$$B_x \cdot C_y - B_x \cdot A_y - A_x \cdot C_y + A_x \cdot A_y - C_x \cdot B_y + C_x \cdot A_y + A_x \cdot B_y - A_x \cdot A_y =$$

$$B_x \cdot C_y - B_x \cdot A_y - A_x \cdot C_y - C_x \cdot B_y + C_x \cdot A_y + A_x \cdot B_y$$

This last therm comes out to our previous formula. Now that we have a perpendicular vector to the triangle we can calculate the area of the triangle, because the area of the triangle is half of the cross product. That's why we multiply the cross product by $\frac{1}{2}$. But wait a minute, why is there a minus sign in front of the formula? This comes from the fact that our z axis is flipped, which means that a positive area means that the triangle is facing away from us which is not intuitive. So we flip the sign to get a positive vector.

To actually tell the pipeline what negation is considered front facing we can set the frontFace attribute in the VkPipelineRasterizationStateCreateInfo to either VK_FRONT_FACE_CLOCKWISE or VK_FRONT_FACE_COUNTER_CLOCKWISE. Clockwise means that the triangle is front facing if the area is negative and counter clockwise then covers the opposite. [34]

Next they are culled according to the cullMode attribute in the VkPipelineRasterizationStateCreateInfo. This can be set to either VK_CULL_MODE_NONE, VK_CULL_MODE_FRONT, VK_CULL_MODE_BACK or VK_CULL_MODE_FRONT_AND_BACK. This determines if the front facing, back facing, both or none of the triangles are discarded. [34]

Next we take the vertices of each triangle and determine its bounding box. A bounding box is basically the smallest rectangle that contains the whole triangle. By doing this we don't have to check every pixel if it is inside the triangle, but only the pixels that are inside the bounding box. This is done by taking the minimum and maximum x and y coordinates of the triangle's vertices.

Next we iterate through every pixels center inside the bounding box, and check if it is inside the triangle. A point is inside the triangle if it is to the right of all the triangles edge vectors when it is clockwise oriented and to the left when it is counter clockwise oriented. We can check this by calculating the cross product of the edge vectors and the vector from the first vertex to the pixels position. If the cross product is positive the point is to the right of the edge, if it is negative it is to the left. If the cross product is 0 the point is on the edge.

Okay but what if the point is on an edge of adjacent triangles? How do we know which triangle should be drawn? For this case graphics APIs usually use the top-left rule. This rule states that left and top edges are considered to be inside the triangle while right and bottom edges are considered to be outside. How do we know which edge is which?

A top edge is an edge where the y coordinate is flat and pointing to the right, so where the y coordinate doesn't change and the x coordinate is increasing when the winding order is clockwise or else is decreasing when it is counter clockwise.

A left edge is an edge where the y coordinate is decreasing (clockwise), which means that it is pointing upwards, because the y axis is flipped.

Is the edge a top or left edge? Then the point is inside the triangle, otherwise an offset is added to move the point outside of the triangle.

Some of you might not like the idea of checking every pixel inside the bounding box, because it is very inefficient. There is actually a way to rasterize a triangle without a bounding box. To do so we have to take a look at a fact that comes from using the barycentric coordinates. When we go through the triangle and calculate one of the barycentric coordinates we can see that the step size we take is always the same when we go from one pixel in the triangle to the next. This comes from an intresting property of the barycentric coordinates. When we go from one pixel to the next the barycentric coordinates change by a constant amount. This means that we can rasterize a triangle by only checking the barycentric coordinates of the first pixel and then adding a constant amount to the barycentric coordinates to get the barycentric coordinates. If the barycentric coordinates are inside the triangle we draw the pixel, if not we skip it.

We can calculate the constant amount by calculating the barycentric coordinates of the next pixel and then subtracting the barycentric coordinates of the current pixel from them. This will give us the constant amount we have to add to the barycentric coordinates to get the barycentric coordinates of the next pixel.

For lines and points we use the same methods, we just have to draw a box around the line or point and then check if the pixel is inside this box. The box drawn is not the bounding box but the thickness of the line or point.

We have two problems left.

First, lines and edges will not look smooth because we only interpolate the color of the vertices. They will look chunky because we fully color the pixel if the center is inside the box. To fix this we can use anti-aliasing. Anti-aliasing is a technique used to smooth the edges of a line or point. We can do this by checking how much of the pixel is covered by the line and then color the pixel accordingly.

Second, what if we have two triangles that overlap? We can't just draw the second triangle over the first one. To fix this we can use a depth buffer. You might have asked yourself why we need a z coordinate for the vertices. This is because we can use the z coordinate to determine which pixel is in front of each other. When we rasterize a pixel we write the z value into an depth buffer and then we check if the z value of the next pixel is smaller than the z value in the depth buffer. If it is we draw the pixel and write the new depth into the depth buffer, if not we skip it.

3.3.8 Fragment Shader

The fragment shader is the last programmable stage of the pipeline. It takes the fragment data from the rasterization stage and performs operations on each fragment. This is where we can calculate the color of the fragment.

To do so we need to interpolate the vertex colors. Fortunately, we already know a way on getting the relative position of a point to a triangles vertices, the barycentric coordinates. We can use the barycentric coordinates to interpolate the color of a vertex to get the color of the pixel.

But how do we get the baricentric coordinates of a pixel? We can use the same method as before, by calculating the area of the triangle between to vertices and the pixel. So we take the cross product of on side vector and the vector from the first vertex to the pixel and divide it by 2 to get its area. We do this for all 3 sides and then divide them by the area of the full triangle to get the barycentric coordinates.

For example when we have green vertex A{1,2}, red vertex B{2,3} and blue vertex C{3,2} and the coordinates of the pixel are {2,2.5} we can calculate the barycentric coordinates like this:

$$\begin{aligned}
\text{Area of the triangle} * 2 &= 2 \\
\text{Area of the triangle ABP} * 2 &= \frac{|0.5 - 1|}{2} = 0.25 \\
\text{Area of the triangle BCP} * 2 &= \frac{|(-0.5)|}{2} = 0.25 \\
\text{Area of the triangle CAP} * 2 &= \frac{|(-2) \cdot 0.5|}{2} = 0.5
\end{aligned}$$

So the barycentric coordinates are {0.25, 0.25, 0.5}. If you're confused on why I'm using the area times 2, it's because we can safe the computation of the actual area because we are only caring about the relative position of the pixel to the vertices, so as long as everything stays at the same scale it will give us the same answer.

Next we interpolate the colors of the vertices linearly to get the color of the pixel. This is done by multiplying the color of each vertex with the barycentric coordinates and adding them together.

$$r = 0.25 \cdot A_r + 0.25 \cdot B_r + 0.5 \cdot C_r$$

$$g = 0.25 \cdot A_g + 0.25 \cdot B_g + 0.5 \cdot C_g$$

$$b = 0.25 \cdot A_b + 0.25 \cdot B_b + 0.5 \cdot C_b$$

$$r = 0.25 \cdot 0 + 0.25 \cdot 255 + 0.5 \cdot 0 = 63.75$$

$$g = 0.25 \cdot 255 + 0.25 \cdot 0 + 0.5 \cdot 0 = 63.75$$

$$b = 0.25 \cdot 0 + 0.25 \cdot 0 + 0.5 \cdot 255 = 127.5$$

Which will give us this color: $\{63.75, 63.75, 127.5\}$. We repeat this for every fragment inside the triangle and then we have our rasterized image.

But we can do more than just interpolate the color of the vertices. We can also apply lighting, textures and more.

3.3.9 Blending

The last stage of the pipeline is blending. Blending is used to create transparency and some other graphical effects. There isn't much to say about blending. It gives us many options to blend the color of the fragment but all of the operations are performed by the GPU, so we don't have to worry about it. When blending is complete we have our final framebuffer that we can display.

3.4 Drawing Process

3.4.1 Framebuffers

A Framebuffer consists of multiple attachments. An attachment is the buffer that holds either the color or depth of each fragment. These buffers are written to by the pipeline and then presented to the screen. The color attachment is used to store the color of each fragment, while the depth attachment is used to store the depth of each fragment. The depth attachment is used to save the z coordinate of each fragment to prevent drawing over fragments if they are not visible. [35]

3.4.2 Render Pass

A render pass holds the information about how many color and depth attachments we have and how they are used in the pipeline. It can consist of multiple subpasses, which are used to render the scene in multiple steps. For example, we can render the scene in one subpass and then apply post processing in another subpass. [36]

3.4.3 Swapchain

The swapchain is used to present the final framebuffer to the screen. It consists of usually 2 (Double Buffering) or more framebuffers. On a swapchain we can present one framebuffer while we render to another framebuffer. The presented framebuffer is called the front buffer and the rendered framebuffer is called the back buffer. Every time a new frame is rendered the front and back buffers are swapped. This is done to prevent screen tearing. You know when the screen is cut into 2 different images? That's screen tearing. It happens when the GPU is rendering to the framebuffer while the screen is reading from it. But that only happens when we have V-Sync disabled.

There are two problems with Double Buffering. It can cause input lag. This is because the GPU has to wait for the swapchain to swap the framebuffers before it can render the next frame. To fix this we can use Triple Buffering. The other problem occures when the GPU takes longer than one frame. This will cause the swapchain to wait for the GPU to finish rendering the frame before it can swap the framebuffers. This will cause the frame to drop and the game to stutter. To fix this we can use a third buffer to render to while the GPU is rendering the frame. This way the GPU can swap the framebuffers without waiting for the GPU to finish rendering the frame. [37]

3.5 Conclusion

This was a brief overview of the Vulkan pipeline and its rendering. Now that we have a basic understanding of the pipeline we can start programming our first Vulkan application. In the next chapter we will set up the Vulkan environment and create a window to render to.

4. Setting up Vulkan

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] B. Galea: *The Math behind (most) 3D games - Perspective Projection*. 2021. URL: https://www.youtube.com/watch?v=U0_ONQQ5ZNM (visited on 07/01/2024).

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

[16] K. Group: *Vulkan Guide. Pipelines*. 2023. URL: https://docs.vulkan.org/spec/latest/chapters/pipelines.html (visited on 08/07/2024).

[17] 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).

[18] V. Blanco: *Executing Vulkan Commands*. 2020. URL: https://vkguide.dev/docs/chapter-1/vulkan_command_flow/ (visited on 07/08/2024).

[19] 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).

[20] 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).

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

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

[23] 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).

[24] 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).

[25] 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).

[26] A. Overvoorde: *Vulkan Tutorial. Uniform Buffers*. 2023. URL: https://vulkan-tutorial.com/Uniform_buffers (visited on 07/08/2024).

[27] V. Blanco: *Push Constants*. 2020. URL: https://vkguide.dev/docs/chapter-3/push_constants/ (visited on 07/08/2024).

[28] K. Group: *Vulkan Specification. Tessellation*. 2024. URL: https://docs.vulkan.org/spec/latest/chapters/tessellation.html (visited on 07/09/2024).

[29] K. Group: *Vulkan Specification. Tessellation*. 2024. URL: https://docs.vulkan.org/spec/latest/chapters/tessellation.html#img-tessellation-topology-ul (visited on 07/09/2024).

[30] W. Smith: *Equilateral Triangle*. 2023. URL: https://www.storyofmathematics.com/equilateral-triangle/ (visited on 07/10/2024).

[31] K. Group: *Vulkan Specification. File:Tessellation isoline 6 2.png*. 2024. URL: https://www.khronos.org/opengl/wiki/File:Tessellation_isoline_6_2.png (visited on 07/14/2024).

[32] J. de Vries: *Advanced OpenGL. Geometry Shader*. 2014. URL: https://learnopengl.com/Advanced-OpenGL/Geometry-Shader (visited on 07/15/2024).

[33] K. Group: *Vulkan Specification. Fixed-Function Vertex Post-Processing*. 2024. URL: https://docs.vulkan.org/spec/latest/chapters/vertexpostproc.html (visited on 07/15/2024).

[34] K. Group: *Vulkan Specification. Rasterization*. 2024. URL: https://docs.vulkan.org/spec/latest/chapters/primsrast.html (visited on 07/22/2024).

[35] I. LunarG: *Create the Framebuffers. The Vulkan Framebuffer*. 2016. URL: https://vulkan.lunarg.com/doc/view/latest/windows/tutorial/html/12-init_frame_buffers.html (visited on 07/22/2024).

[36] A. Overvoorde: *Vulkan Tutorial. Render Passes*. 2023. URL: https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Render_passes (visited on 07/22/2024).

[37] B. Galea: *Swap Chain Overview - Vulkan Game Engine Tutorial 05 part 1*. 2021. URL: https://www.youtube.com/watch?v=IUYH74MqxOA (visited on 07/22/2024).

List of Figures