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

Nick Schefner

August 28, 2024

Contents

1. Introduction

Dear Reader,

Welcome to "Vulkan Unveiled: A Beginner's Guide to Professional Graphics Programming". 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", which is just a very basic implementation of a game engine. 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 in creating a precise representation of the Vulkan API. 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 graphics programming throughout this book.

Graphical programming is a complex 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. Therefore 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 an API that is very low level, whereas older graphic APIs abstract many of the processes, that Vulkan requires us to handle ourself.

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 employ.

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. 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 a possible implementation could look like.

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 multiple times. [2] Because the GPU focuses on speed and parallelism, it is designed to use more of the accessible transistors for processing data. This is beneficial for parallel computations because the GPU can hide memory access latencies with computation. The CPU on the other hand has an additional cache layer and a more complex control logic that makes sequential computations faster. [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 just has a hand full of cores, GPUs have multiple thousands of 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 processed and unprocessed 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, for example, 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 threads 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 a 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 if needed. [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:

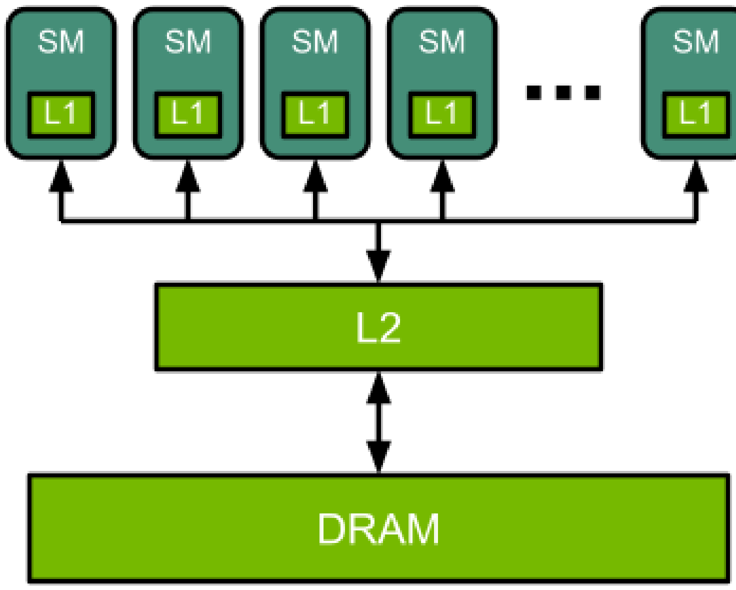

Figure 1: Memory Hierarchy of the GPU [8]

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 of 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. The name pipeline comes from the fact that we pass data through multiple stages that process the data in a specific way. Once the data has passed through all stages, it is ready to be displayed on the screen.

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

There are 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, which starts the data processing. [10]

3.2 The Basics

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

3.2.1 Verices and Indices

Every model that we see is made up of off either triangles, points or isolines. These shapes are called primitives and are created by connecting vertices so that they form the corners of the primitive. For points we only need one vertex. A single vertex can contain multiple attributes like position {x, y, z}, color {r, g, b, a} or texture coordinates {u, v}. [11]

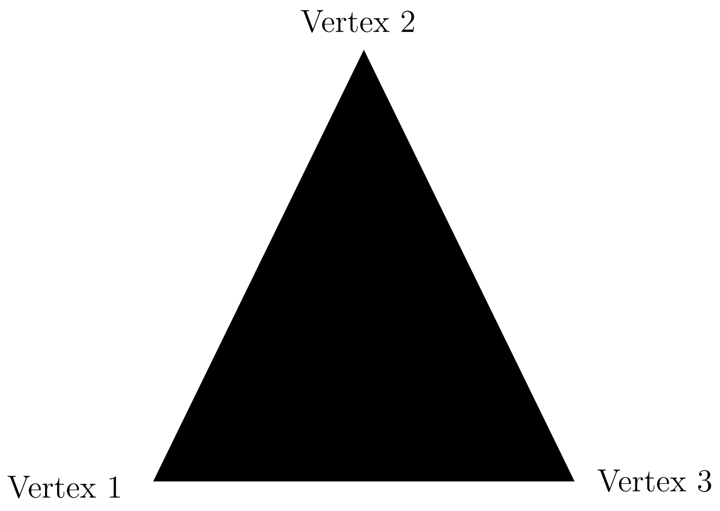

Figure 2: Triangle with vertices

When we want to reuse certain vertices, for example when drawing a square, we can add indices. These indices tell the input assembler what vertices need to be combined to form a triangle. [12]

For example: A square has 4 vertices because it has 4 corners. When we want to draw the square 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}

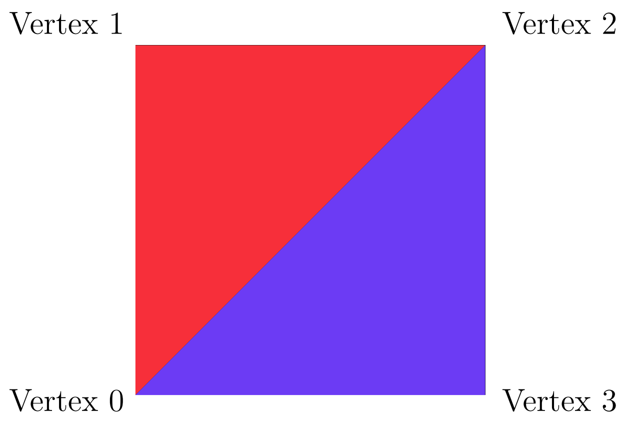

Figure 3: Square with vertices created using indices

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.

3.2.2 Buffers

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. [13]

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. [13]

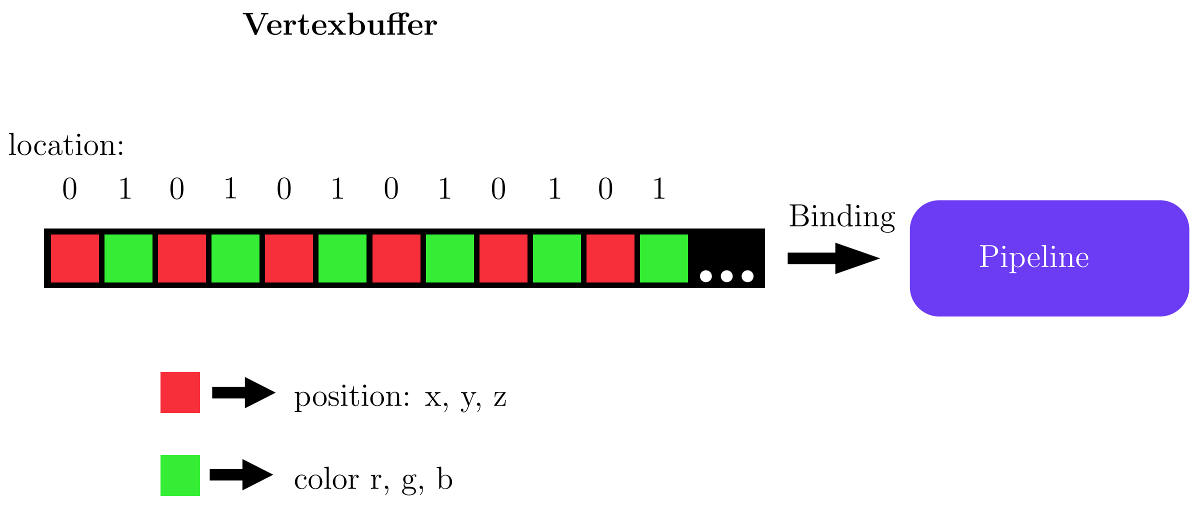

Figure 4: Interleaved Vertex Buffer

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 location is separated. In this case it would be 12 bytes for the color, because the position contains 3 floats, which are 4 bytes each if you use single precision floats and 8 bytes each when you use double precision floats. [13]

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.

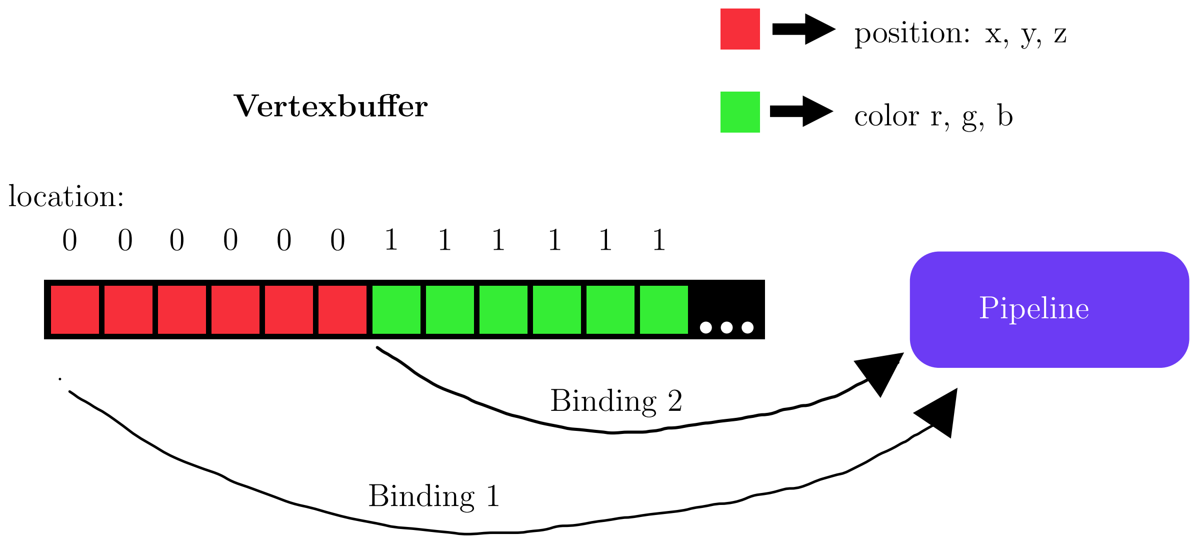

Figure 5: Non-Interleaved Vertex Buffer

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. [13]

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. [12]

3.2.3 Coordinate Systems

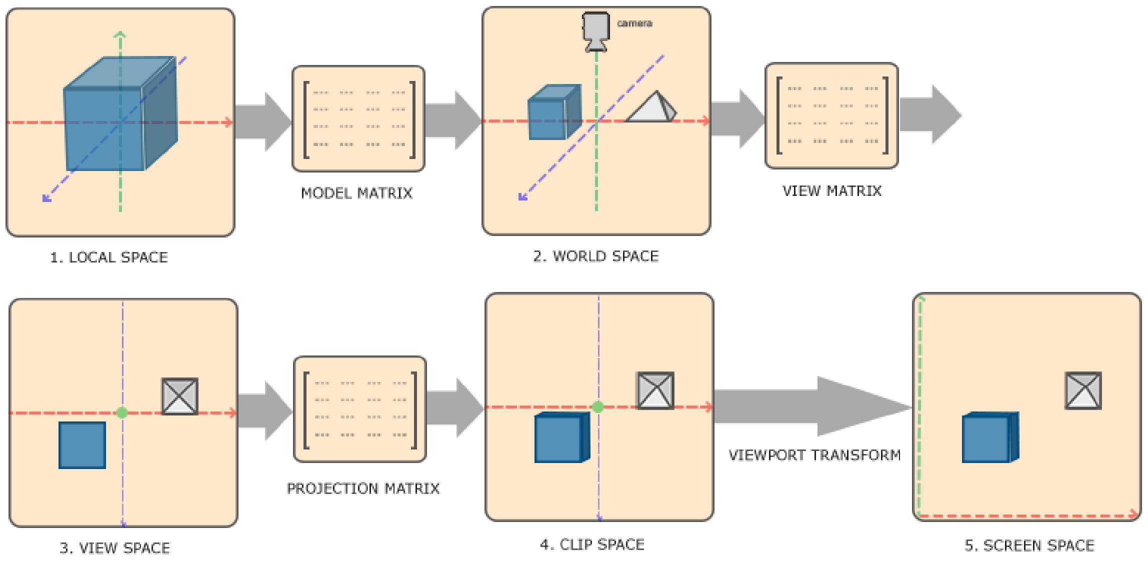

Figure 6: Coordinate Systems [14]

We start with the local space, which is the space that we use to define our objects. Often the objects are centered around the origin of the local space, but it is up to the modeler to decide where the object is located. The local space is then transformed into 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. [14]

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.

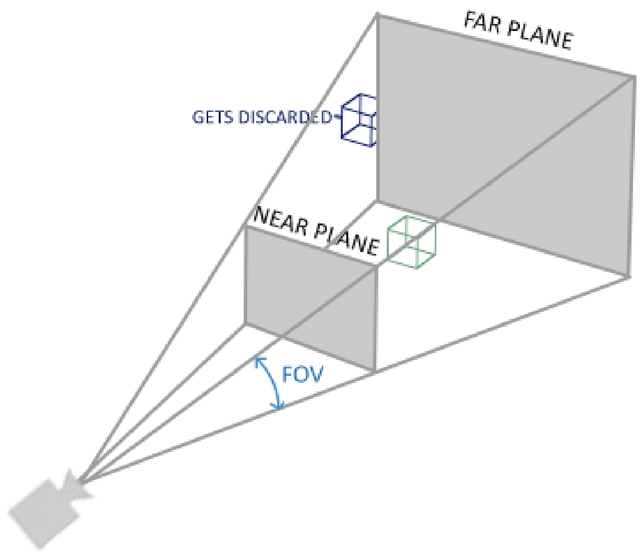

Figure 7: View Frustum [14]

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.

3.2.4 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 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.

For example, when we have two matrices A and B, we multiply them like this:

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

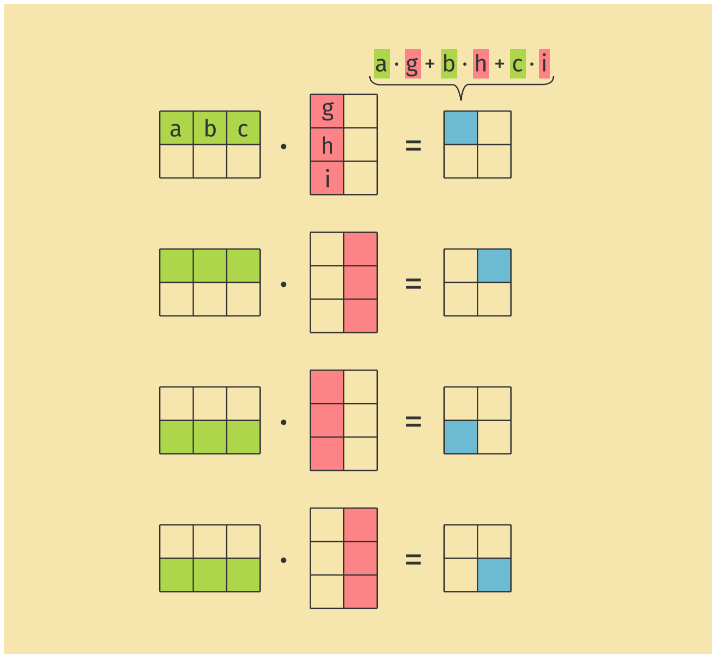

Figure 8: Matrix Multiplication [15]

As long as the number of columns in the first matrix is equal to the number of rows in the second matrix, we can multiply them. The result will have the same number of rows as the first matrix and the same number of columns as the second matrix.

When we have a position {x, y, z} we cannot just move it by {tx, ty, tz} because multiplying the position with the translation vector would just multiply the coordinates. But what we need is to add the wanted translation to the corresponding coordinate. To do so we need to add a 4th coordinate to the position, which is called the homogeneous coordinate. This coordinate is usually set to 1 and is used to set the proportions of the clip space. Don't worry about the clip space for now, we will talk about it later.

Homogeneous coordinates are used to represent the position of a point with 4 coordinates instead of 3. They have the special property that the homogeneous coordinates are a multiple of the cartesian coordinates by the w coordinate.

So when we have a point {1, 2, 3} in 3D space, we can represent it as {1, 2, 3, 1} in homogeneous coordinates but also as {2, 4, 6, 2} or {3, 6, 9, 3} and so on. We will keep the w coordinate at 1, because it is the easiest to work with.

Our position would then be {x, y, z, 1}. Because the w coordinate is the fourth coordinate, we can multiply the position with a 4x4 matrix where the forth column is the translation vector. This will add the translation to the position as long as we keep the diagonal values of the matrix to 1, so that we don't loose the original position in the calculation.

Therefore we can create the translation that moves the box by {tx, ty, tz} like this:

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

Please note that the w coordinate is changing the x, y and z coordinates, to adjust the position to our clip space proportions.

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:

$$P1 = \begin{bmatrix} cos(\delta) & 0 & -sin(\delta) & 0 \\ 0 & 1 & 0 & 0 \\ sin(\delta) & 0 & cos(\delta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$P2 = \begin{bmatrix} cos(\delta) & 0 & -sin(\delta) & 0 \\ 0 & 1 & 0 & 0 \\ sin(\delta) & 0 & cos(\delta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} cos(\delta) \\ 0 \\ sin(\delta) \\ 1 \end{bmatrix}$$

You can see that the origin $P1$ of the line stays the same, but the end point $P2$ 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 an object is a bit simpler than rotation. We just multiply the position of the object with a scaling matrix that looks like this:

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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.

A thing that is important to note is that when applying transformations, we apply them to the entire world space. This means when rotating, scaling or moving an object, we are actually rotating, scaling or moving the entire world, which will also rotate, scale or move the object.

3.2.5 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.

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

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 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

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

Luckily we don't need to calculate the angles by hand, we can use the GLM library to calculate the rotation matrix for us. The GLM library is a header only library that provides us with many useful functions for mathematical operations.

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.

3.2.6 Projection Transformation

The next step is the transformation from the view space to the clip space. The clip space is used to define what we can see on the screen, by clipping everything that is outside of the view frustum.

This task involves transforming the view frustum into an 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 clip space, which is plus and minus the w coordinate on all axes. The w coordinate is almost always set to 1 by the user, so the clip space is a cube with the length of 2 on all axes except the z axis, because the negative z axis is the behind the camera we just have the positive z axis.

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 but that's up to the user to decide.

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.

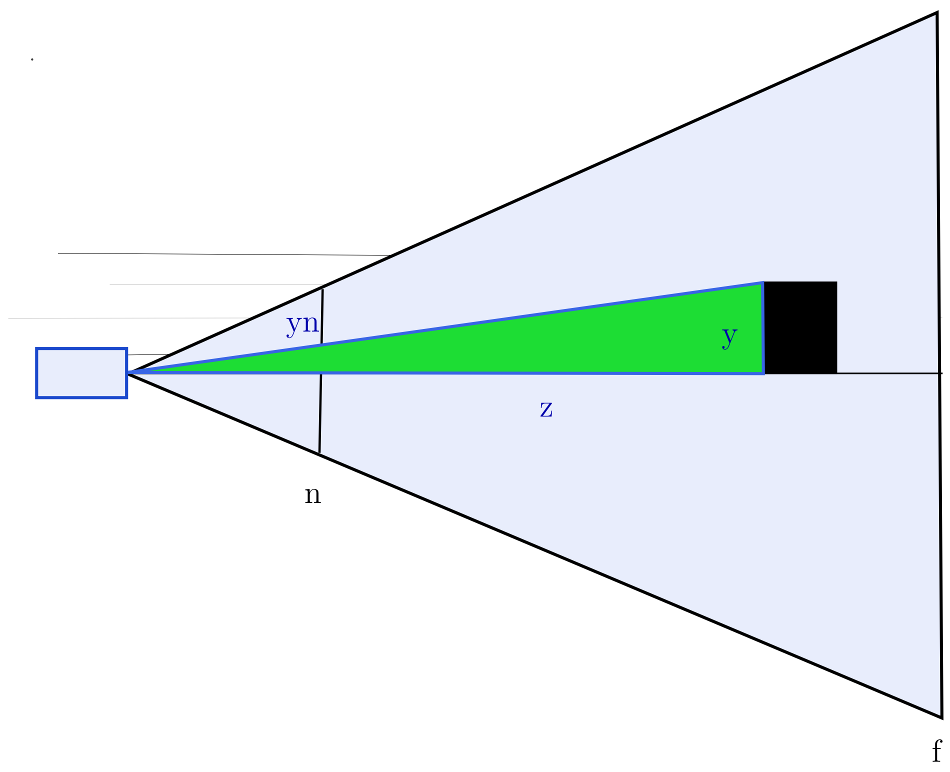

Figure 9: View Frustum

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 aka the width of the object:

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

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. Here the w coordinate saves us again.

We transfer the z coordinate to the 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. This division is called the perspective divide. So after the perspective divide we get $\{\frac{x*n}{z}, \frac{y*n}{z}, z\}$ which are called normalized device coordinates (NDC) or the canonical view volume (CVV).

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.

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

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

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

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

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

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

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

$$m_1 = f + n$$

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

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

$$m_2 = -f * n$$

One thing that has to be looked out for is that the near plane and far plane are not the same because that would result in a division by zero. Also we don't want the near plane to be 0 because that would result in weird behavior and clipping into other objects.

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:

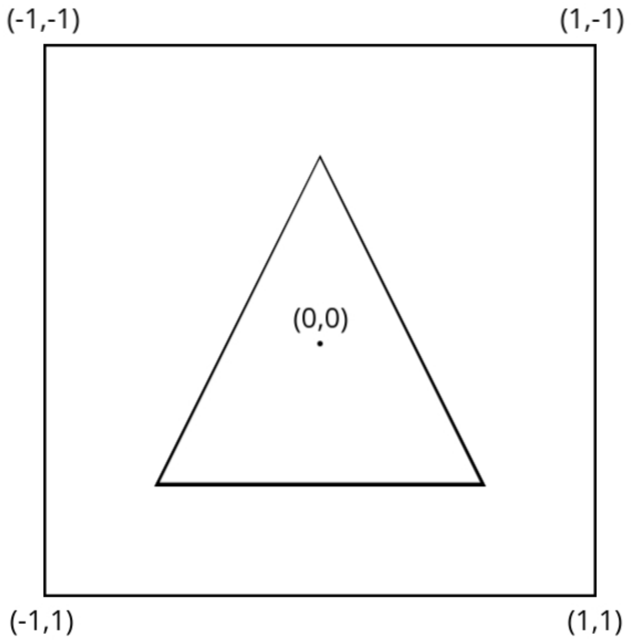

Figure 10: Canonical View Volume Front

It is a coordinate system where the y axis is pointing downwards and 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.

3.2.7 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). Optionally it is possible to use other shading languages like the High Level Shading Language (HLSL). These shaders have to be given to the pipeline as compiled SPIR-V bytecode. [16]

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

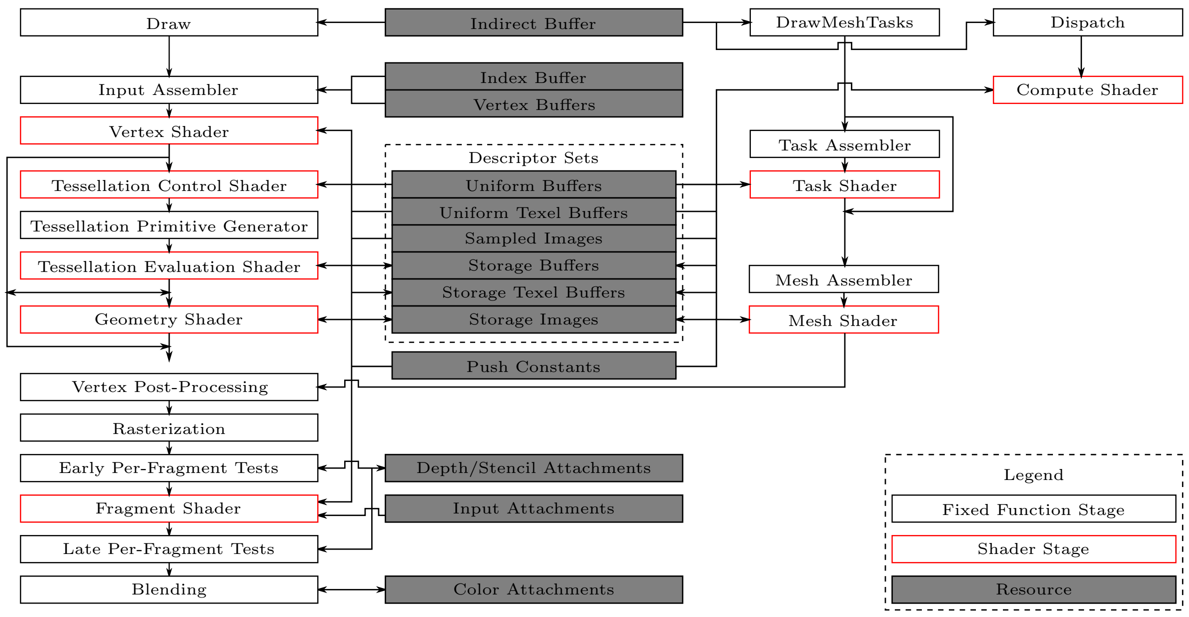

Figure 11: Pipeline Structure [17]

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. [18]

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. [19]

3.3.2 Input Assembler

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

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 adjacency types, so don't worry on understanding them now. [22]

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. [23]

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.

When we want to pass data during the rendering process to the shaders, we have to use resource descriptors. 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. [24]

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. [25]

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.[26]

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. [26]

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. [26]

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 three 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. [26]

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 two 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 three corners it creates the corners of the inner triangle. [26]

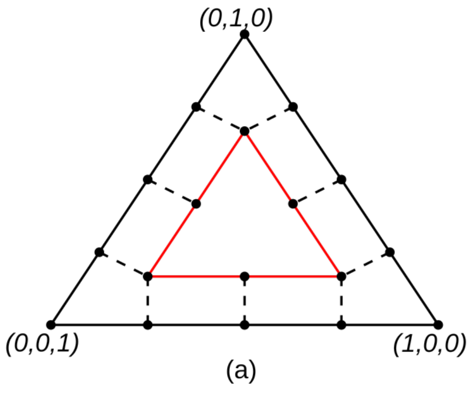

Figure 12: Inner Tessellation with 4 segments
[27]

You can see that the inner tessellation level of four cuts each edge of the outer triangle into four 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 three points where u is always zero 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 two 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. [28]

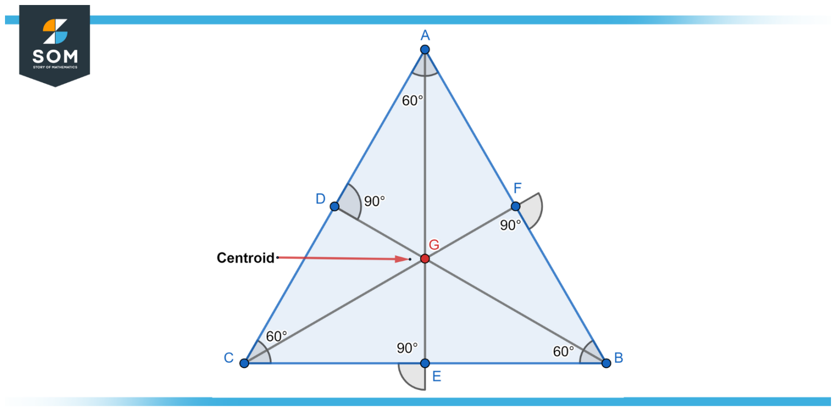

Figure 13: Equilateral Triangle
[28]

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.

$$\begin{aligned}
& f(v) = f(u) \\
I) \quad & \frac{1}{4} - \frac{1}{2} \cdot v = u \\
II) \quad & v = \frac{1}{4} - \frac{1}{2} \cdot u \\
III) \quad & \frac{3}{4} - \frac{1}{2} \cdot v = \frac{3}{4} - \frac{1}{2} \cdot u \\
& v = u \\
I) \quad & \frac{1}{4} - \frac{1}{2} \cdot u = u \\
& \frac{3}{2} \cdot u = \frac{1}{4} \\
& u = \frac{1}{6} \qquad v = \frac{1}{6} \qquad w = \frac{4}{6}
\end{aligned}$$

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 from 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. [26]

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. [26] Unfortunately, considering more complicated tessellation, the order in wich 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. [26]

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. [26]

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. [26]

Quads have 4 outer tessellation levels that subdivide the outer edges and 2 inner tessellation levels that subdivide the area of the square into smaller squares. 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 square with triangles. [26]

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.

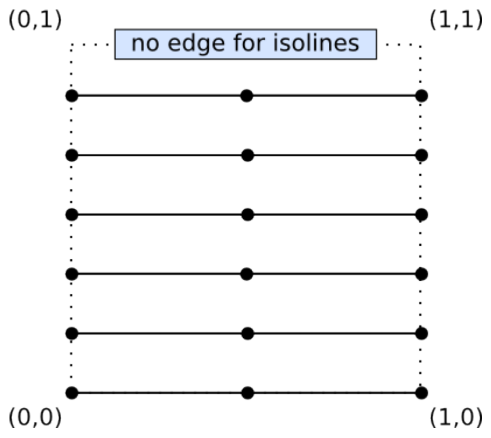

Figure 14: Isoline Tessellation (6, 2)
[29]

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. [26] 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. [26]

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. [26]

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. [26]

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 square we can adjust the vertices by using a 2D texture like a height map to displace the vertices. [26]

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. [26]

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 square around it by emitting 4 new vertices. These will then be combined to 2 triangles by the pipeline, forming a square. [30]

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. [31]

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. [31]

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. [31]

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. [31] 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. [31]

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) [31]

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. [31]

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 aka fragments 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:

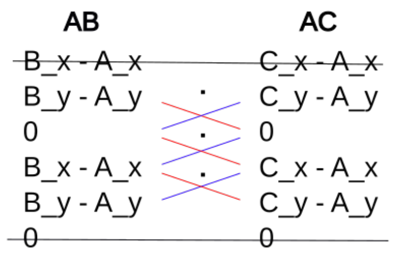

Figure 15: Cross Product

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. [32]

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. [32]

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 fragment if it is inside the triangle, but only the fragments 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 fragments 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 fragments 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 fragment 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 fragment in the triangle to the next. This comes from an intresting property of the barycentric coordinates. When we go from one fragment 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 fragment 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 fragment 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 fragment.

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 fragment 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 fragment 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 fragment is covered by the line and then color the fragment 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 fragment is in front of each other. When we rasterize a fragment we write the z value into an depth buffer and then we check if the z value of the next fragment is smaller than the z value in the depth buffer. If it is we, draw the fragment 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 one 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, which are floats in the range of 0 to 1, with the barycentric coordinates and adding them together.

$$r = 0.25 \cdot A_r + 0.25 \cdot B_r + 0.5 \cdot C_r$$
$$g = 0.25 \cdot A_g + 0.25 \cdot B_g + 0.5 \cdot C_g$$
$$b = 0.25 \cdot A_b + 0.25 \cdot B_b + 0.5 \cdot C_b$$

$$r = 0.25 \cdot 0 + 0.25 \cdot 1 + 0.5 \cdot 0 = 0.5$$
$$g = 0.25 \cdot 1 + 0.25 \cdot 0 + 0.5 \cdot 0 = 0.25$$
$$b = 0.25 \cdot 0 + 0.25 \cdot 0 + 0.5 \cdot 1 = 0.25$$

Which will give us this color: $\{0.5,\ 0.25,\ 0.25\}$. This will then be converted to the color space of our surface, which can be sRGB, displayP3 or some other supported color space. 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. [33]

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. [34]

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. The best implementation is to use Triple Buffering, which uses 3 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. [35]

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

Because the Setup of Vulkan is highly dependent on your operating system and IDE I will only give a brief overview of the setup process.

Let's start by setting up our development environment. First of all we need to to install the Vulkan SDK. You can download it from the LunarG website or use your preferred package manager.

Now verify if your GPU and drivers support Vulkan. To do so go into the Bin folder of the Vulkan SDK and run the *vkcube.exe* program. If it runs without any errors and display a cube you are good to go. If not you have to update your drivers or include the Vulkan runtime in your system.

Next we'll need to install GLFW, a library for creating windows and handling input. We will use it to create a window for our application. You can download the 64-bit binaries from the GLFW website or use your package manager. After downloading the binaries, extract them to a location that is easy to access.

Because Vulkan does not include a library for linear algebra operations we will use the OpenGL Mathematics (GLM) library. Again, download the library and extract it to a location that is easy to access.

Don't forget to install a C++ and GLSL compiler when needed.

Next you will need to either setup your IDE or write a Makefile to compile your program.

This is a very brief overview of the setup process, because explaining the setup for every operating system and IDE would be too much. Please read through this Vulkan Tutorial to get a more detailed explanation.

4.1 Creating a Window

Before we can start rendering we need to create an application window, because vulkan is only a graphics API and doesn't provide a way to create a window. Do not worry about implementing a window from scratch, because there are libraries for creating windows for vulkan. We will use the previously installed open source and multi-platform library GLFW (Graphics Library Framework) which provides us with a simple API that we can use to create a window and handle input. [36]

Let's start by creating a core folder in our source folder. Here all of the core functionalities of our game engine will be stored. In this folder create a new header file called window.hpp.

Inside we first need to include the GLFW header file and the define of GLFW_INCLUDE_VULKAN to include the vulkan header files.

```
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>
```

Next we add a namespace that we want to use for our engine. In my case I will use the namespace *engine*, but feel free to use the actual name of your engine. Inside we create our class Window with an constructor that takes the width, height and name of the window as arguments. We also add a destructor to clean up the window.

```
#include <string>
namespace engine {
  class Window {
  public:
    Window(int width, int height, std::string name);
    ~Window();
  };
}
```

The job of the constructor is to initialize a window with the given width, height and name. So let's create a private helper function that initializes the window. We will call it *initWindow*. Let's also store the width, height and name of the window in private variables. After initializing the window we will need a pointer to the window object, so we can access it later.

```
  class Window {
      public:
        Window(int width, int height, std::string name);
        ~Window();
      private:
        void initWindow();

        const int width;
        const int height;

        std::string name;
        GLFWwindow* window;
    };
```

That's the header file for now. Let's implement the defined functions in the source file. In the Window.cpp file we include the window.hpp file add the namespace and call the initWindow function in the constructor and set the width, height and name of the class.

```
#include "window.hpp"

namespace engine {
  Window::Window(int width, int height, std::string name)
  : width(width), height(height), name(name) {
      initWindow();
    }
}
```

Let's implement the *initWindow* function. First we need to initialize GLFW with the *glfwInit* function. Then we set two window hints with the *glfwWindowHint* function. The first hint is the *GLFW_CLIENT_API* and we set it to *GLFW_NO_API*. This tells GLFW to not create context for the OpenGL API, because it was originaly designed for OpenGL. [37] The second hint is called *GLFW_RESIZABLE* and will be set to *GLFW_FALSE* for now, but will be changed later, when implementing a resizable windows.

Once we're done with setting up the window we can create it with the *glfwCreateWindow* function. It takes the width, height, title as a char pointer, a *GLFWmonitor* pointer, to create a full screen window and a 5th parameter that is OpenGL specific. Pass in our variables, set the monitor pointer to a *nullptr*, making our window use windowed mode and do the same for the last parameter.

The output will be a pointer to our created *GLFWwindow* that we can store in our **window* variable.

```
void Window::initWindow() {
  glfwInit();

  glfwWindowHint(GLFW_CLIENT_API, GLFW_NO_API);
  glfwWindowHint(GLFW_RESIZABLE, GLFW_FALSE);

  window =
    glfwCreateWindow(width, height, name.c_str(), nullptr, nullptr);
}
```

Finally we implement our destructor. It will destroy the window with *glfwDestroyWindow()* and terminate GLFW with *glfwTerminate()*.

```
Window::~Window() {
  glfwDestroyWindow(window);
  glfwTerminate();
}
```

Let's test our window by creating a application that creates a window. Create a app header and source file in the src folder. In the header file we include the window header file, define a constant *WIDTH* and *HEIGHT* and create a private window object. We also need a public *run* function that initializes the window and runs the main loop.

```
#include "core/window.hpp"

namespace engine {
  class App {
    public:
      static constexpr int WIDTH = 800;
      static constexpr int HEIGHT = 600;

      void run();
    private:
      Window window{WIDTH, HEIGHT, "Vulkan Application"};
  };
}
```

In the source file we include the app header file and check in the *run* function if the window should be closed. While it is not closed we poll the events with *glfwPollEvents()*.

For checking if the window should be closed we create a boolean function *shouldClose* in the window class and set it to the return value of *glfwWindowShouldClose(window)*.

```
#include "app.hpp"

namespace engine {
  void App::run() {
    while (!window.shouldClose()) {
      glfwPollEvents();
    }
  }
}
```

In the window class add the public *shouldClose* function. Because it is very simple we can just implement it in the header file.

```
bool shouldClose() { return glfwWindowShouldClose(window); }
```

Now we can create a main.cpp that runs the application. If it fails we return *EXIT_FAILURE* and the error. If it succeeds we return *EXIT_SUCCESS*. We don't need a header file for the main.cpp.

```
#include "app.hpp"
#include <iostream>

int main() {
  engine::App app{};

  try {
    app.run();
  } catch (const std::exception &e) {
    std::cerr << e.what() << '\n';
    return EXIT_FAILURE;
  }

  return EXIT_SUCCESS;
}
```

When running our program it should show us a non-resizable window with the title "Vulkan Application".

Great! But we should also remove the windows copy constructor and operator. We don't want to copy the window object, because we're working with a pointer to the window object. This means that when we copy the window object we also copy the pointer. When we now delete the original object we also delete the *GLFWwindow*, what will leave our copy with a dangling pointer. To prevent this potential bug we delete the copy constructor and copy operator in the window header file.

```
  Window(const Window &) = delete;
  Window &operator=(const Window &) = delete;
```

4.1.1 Why non-resizable window

Our current window is not resizable. I've made this decision because when we will create our SwapChain later it is bound to the windows width an height. So when we resize or window the SwapChain will not be valid anymore. We will get at it once we have the base of our engine set up.

4.2 Device

Let's go ahead and setup Vulkan. To do so we have to create an Instance, a Surface, pick a physical device aka graphics card, create a logical device and optionally setup validation layers and create a command pool.

What each of these are will be explained further in a second. You might feel disoriented and confused sometimes when going through this code, but don't feel discouraged even if you don't understand anything on the first go. Keep copying the code and come back to this part eventually when you feel more comfortable with the whole system of Vulkan.

4.2.1 Vulkan instance

The first thing we need to do is to create a Vulkan instance. A vulkan instance is the connection between our application and the Vulkan library.

Let's start by creating a new header file in the core folder called *device.hpp* and include the Vulkan header file. In here we will create a constructor, a destructor a private function to create the Vulkan instance and a private variable to store the instance and a reference to the window object. We'll also delete the copy and move operators.

```
#include <vulkan/vulkan.h>
#include "window.hpp"

namespace engine {
  class Device {
      public:
        Device(Window &window);
        ~Device();

        Device(const Device &) = delete;
        Device &operator=(const Device &) = delete;
        Device(const Device &&) = delete;
        Device &operator=(const Device &&) = delete;

      private:
        void createInstance();
        VkInstance instance;
        Window &window;

        VkSurfaceKHR surface_;
    };
}
```

Create the source file and include the header file. In the constructor we call the *createInstance* function.

```
#include "device.hpp"

namespace engine {
  Device::Device(Window &window) : window{window} {
    createInstance();
  }
}
```

To create an instance we need to fill out a struct called *VkInstanceCreateInfo*. This struct holds information about the application and the Vulkan instance. The applications information is stored in a struct called *VkApplicationInfo*. The application info is optional but it is good practice to fill it out. It contains the name and the version of the application, the engine and the version of the Vulkan API.

```

void Device::createInstance() {
  VkApplicationInfo appInfo = {};
  appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
  appInfo.pApplicationName = "Vulkan Application";
  appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
  appInfo.pEngineName = "No Engine";
  appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
  appInfo.apiVersion = VK_API_VERSION_1_3;
}
```

Please note the *sType* member of the struct. This member is used to tell Vulkan what type of struct we are using. Why we need this? Vulkan has another member called *pNext* that is used to extend the struct with additional information. It basically creates a linked list of structs by pointing to the next struct. The porblem is that there is no way to know what type of struct is next in the list. The implementation reads the first 4 bytes of the next struct, which is the *sType* member, to determine the type of the struct and cast the pointer to the correct struct.

Next we create the *VkInstanceCreateInfo* struct and fill it with the application info. We also add the extension data from GLFW. For that we will create a private function called *getRequiredExtensions* that returns the required extensions as a vector of strings.

```
VkInstanceCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
createInfo.pApplicationInfo = &appInfo;

std::vector<const char *> extensions = getRequiredExtensions();
createInfo.enabledExtensionCount = static_cast<uint32_t>(extensions.
   size());
createInfo.ppEnabledExtensionNames = extensions.data();
```

The *getRequiredExtensions* function is used to get the required Vulkan extensions that GLFW needs to create a window with Vulkan. We can get the required extensions with the *glfwGetRequiredInstanceExtensions* function. To return the extensions as a vector of strings we need the size of the extensions and a pointer to the extensions as an array. Then we call the function *glfwGetRequiredInstanceExtensions* which returns the extensions as an array of strings and the size to the pointer we passed as a parameter to the function. We then copy the extensions to a vector of strings and return it.

```
std::vector<const char *> getRequiredExtensions() {
  uint32_t glfwExtensionCount = 0;
  const char **glfwExtensions;
  glfwExtensions = glfwGetRequiredInstanceExtensions(&
   glfwExtensionCount);

  std::vector<const char *> extensions(glfwExtensions, glfwExtensions
   + glfwExtensionCount);

  return extensions;
}
```

Don't forget to add the *getRequiredExtensions* function to the header file.

Finally we create the Vulkan instance with the *vkCreateInstance* function in the *createInstance* function. We pass the *VkInstanceCreateInfo* struct, a pointer to a custom allocator, that we will leave as *nullptr* and a pointer to the instance variable. Custom allocators are used to allocate a range of memory for Vulkan objects. This optimizes memory usage, by not having to allocate memory for each object separately, but allocating a range of memory and then sub-allocating the memory for the objects from this range. But we don't need to worry about this for now.

We also have to check if the instance was created successfully. We can do this by checking if the functions return value is *VK_SUCCESS*. If not we throw an exception with an error message.

```
if (vkCreateInstance(&createInfo, nullptr, &instance) != VK_SUCCESS)
    throw std::runtime_error("Failed to create Vulkan instance");
```

Finally we have to clean up the instance in the destructor. We can do this by calling the *vkDestroyInstance* function with the instance as a parameter.

```
Device::~Device() {
  vkDestroyInstance(instance, nullptr);
}
```

Great! Now our instance is created, but there is one more thing I would like to add to it. Because vulkan is designed to be fast, it doesn't provide much error checking. To enable error checking we can use validation layers. Validation layers hook into the function calls of Vulkan and perform additional checks operation.

4.2.2 Validation Layers

Let's start off by creating a public constant bool enableValidationLayers in the device class header file. If we want to enable validation layers we set the *NDBUG* macro to 0, otherwise to 1.

```
#ifdef NDEBUG
```

```
  const bool enableValidationLayers = false;
#else
  const bool enableValidationLayers = true;
#endif
```

Next we add a private vector of strings called *validationLayers* that contains the names of the validation layers we want to enable. We will use the *VK_LAYER_KHRONOS_validation* layer to use the standard error checking provided by Vulkan.

```
const std::vector<const char *> validationLayers = {
  "VK_LAYER_KHRONOS_validation"
};
```

Now we have to check if the validation layers, we want to enable, are supported by Vulkan. To do so we create a private function called *checkValidationLayerSupport* that returns a boolean. We will use the *vkEnumerateInstanceLayerProperties* function to get the available layers.

First we will create and pass a pointer to a variable that will hold the number of available layers. Then we will create a vector of *VkLayerProperties* that will hold the available layers.

Then we will iterate thorugh the available layers and check if the the validation layer is in the available layers. If it is we return true, otherwise we return false.

```
bool Device::checkValidationLayerSupport() {
  uint32_t layerCount;
  vkEnumerateInstanceLayerProperties(&layerCount, nullptr);

  std::vector<VkLayerProperties> availableLayers(layerCount);
  vkEnumerateInstanceLayerProperties(&layerCount, availableLayers.data
   ());

  for (const char *layerName : validationLayers) {
      bool layerFound = false;

      for (const auto layerProperty : availableLayers) {
          if (strcmp(layerName, layerProperty.layerName) == 0) {
              layerFound = true;
              break;
            }
        }

      if (!layerFound)
      return false;
    }
  return true;
}
```

At the top of the *createInstance* function we will check if the validation layers are supported when the *enableValidationLayers* constant is set to true. If the validation layers are not supported we throw an exception with an error message.

```
if (enableValidationLayers && !checkValidationLayerSupport())
  throw std::runtime_error("Requested validation layers are not
   supported!");
```

GLFW needs another extension to make the validation layers works. This extension is called *VK_EXT_DEBUG_UTILS_EXTENSION_NAME*. So let's add this extension to the *getRequiredExtensions* function.

```
std::vector<const char *> getRequiredExtensions() {
  uint32_t glfwExtensionCount = 0;
  const char **glfwExtensions;
  glfwExtensions = glfwGetRequiredInstanceExtensions(&
   glfwExtensionCount);

  std::vector<const char *> extensions(glfwExtensions, glfwExtensions
   + glfwExtensionCount);

  if (enableValidationLayers)
    extensions.push_back(VK_EXT_DEBUG_UTILS_EXTENSION_NAME);

  return extensions;
}
```

To activate the validation layers we have to add them to the *VkInstanceCreateInfo* struct. We also have to add a debug messenger to get the validation layer messages. To do so we can use the *VkDebugUtilsMessengerCreateInfoEXT* struct that we will use to extend the *VkInstanceCreateInfo* struct. Let's add a private function called *populateDebugMessenger* that takes a *VkDebugUtilsMessengerCreateInfoEXT* struct reference as an argument fills it with the desired values. Then we add the debug messenger to the *VkInstanceCreateInfo* struct.

```
if (enableValidationLayers) {
createInfo.enabledLayerCount = static_cast<uint32_t>(validationLayers.
   size());
createInfo.ppEnabledLayerNames = validationLayers.data();

VkDebugUtilsMessengerCreateInfoEXT debugCreateInfo;
populateDebugMessenger(debugCreateInfo);

createInfo.pNext = (VkDebugUtilsMessengerCreateInfoEXT *) &
   debugCreateInfo;

} else {
createInfo.enabledLayerCount = 0;
createInfo.pNext = nullptr;
}
```

```
void Device::populateDebugMessenger(VkDebugUtilsMessengerCreateInfoEXT
    &createInfo) {
  createInfo = {};
  createInfo.sType =
   VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
  createInfo.messageSeverity =
   VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT |

   VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
  createInfo.messageType = VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT
    |

   VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT |

   VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT;
  createInfo.pfnUserCallback = debugCallback;
  createInfo.pUserData = nullptr;
}
```

The *messageSeverity* member of the *VkDebugUtilsMessengerCreateInfoEXT* struct is used to determine which message severity levels should be used. We basically have 4 levels of severity. Verbose, Info, Warning and Error. Each of them is assigned a bit.

For example the *VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT* is equal to 0x00000100 and the *VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT* is equal to 0x00001000. We can combine these bits with the bitwise OR operator to get the desired severity level 0x00001100. When we get a message it has a severity level assigned to it. This level is then AND combinded to the *messageSeverity* member. The same is done with the *messageType* member.

If the results are not 0 the *pfnUserCallback* function is called that outputs the message. If you want to pass user data to the callback function you can do so with the *pUserData* member.

Let's implement the *debugCallback* function. It takes the severity, type, pCallbackData and pUserData as arguments and returns a *VkBool32* which has to be *VK_FALSE*.

```
static VKAPI_ATTR VkBool32 VKAPI_CALL debugCallback(
VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity,
VkDebugUtilsMessageTypeFlagsEXT messageType,
const VkDebugUtilsMessengerCallbackDataEXT *pCallbackData,
void *pUserData) {
    std::cerr << "Debug callback" << pCallbackData->pMessage << std::
   endl;
    return VK_FALSE;
  }
```

The *VKAPI_ATTR* and *VKAPI_CALL* are enables Vulkan to use the correct calling convention for the function. For now we will just output the message to the console but you can do whatever you want with the message.

Next we have to create the debug messenger with the *vkCreateDebugUtilsMessengerEXT* function. But before we can do that we have to load the function with the *vkGetInstanceProcAddr* function. We'll start off by adding a function in the constructor that setups the debug messenger and then use a generic function to load the function and create the messenger. Let's also add a private variable that stores the debug messenger.

```
//device.hpp
void setupDebugMessenger();
VkDebugUtilsMessengerEXT debugMessenger;
```

```
//device.cpp
Device::Device(Window &window) : window{window} {
  createInstance();
  setupMessenger();
}

void Device::setupDebugMessenger() {
  if (!enableValidationLayers) return;

  VkDebugUtilsMessengerCreateInfoEXT createInfo;
  populateDebugMessenger(createInfo);

  if (CreateDebugUtilsMessengerEXT(instance, &createInfo, nullptr, &
   debugMessenger) != VK_SUCCESS)
    throw std::runtime_error("Failed to set up debug messenger");
}

VkResult CreateDebugUtilsMessengerEXT(
  VkInstance instance, const VkDebugUtilsMessengerCreateInfoEXT *
   pCreateInfo,
  VkAllocationCallbacks *pAllocator, VkDebugUtilsMessengerEXT *
   pDebugMessenger) {
    auto func = (PFN_vkCreateDebugUtilsMessengerEXT)
   vkGetInstanceProcAddr(instance, "vkCreateDebugUtilsMessengerEXT");
    if (func != nullptr)
      return func(instance, pCreateInfo, pAllocator, pDebugMessenger);
    else return VK_ERROR_EXTENSION_NOT_PRESENT;
```

The *CreateDebugUtilsMessengerEXT* function trys to load the *vkCreateDebugUtilsMessengerEXT* function with the *vkGetInstanceProcAddr* function. If the function is loaded successfully we create the debug messenger with that function and return the result. If the function is not loaded we return *VK_ERROR_EXTENSION_NOT_PRESENT*.

And don't forget to destroy the debug messenger in the destructor if the validation layers are enabled. To do so we'll have to load the *vkDestroyDebugUtilsMessengerEXT* function with the same function we used to load the *vkCreateDebugUtilsMessengerEXT* function. Let's add a lokal helper function to the device class that destroys the debug messenger called *destroyDebugUtilsMessengerEXT*.

```cpp
//device.cpp
Device::~Device() {
  if (enableValidationLayers)
    destroyDebugUtilsMessengerEXT(instance, debugMessenger, nullptr);

  vkDestroyInstance(instance, nullptr);
}


void destroyDebugUtilsMessengerEXT(VkInstance instance,
  VkDebugUtilsMessengerEXT debugMessenger,
  const VkAllocationCallbacks *pAllocator) {
    auto func = (PFN_vkDestroyDebugUtilsMessengerEXT)
   vkGetInstanceProcAddr(instance, "vkDestroyDebugUtilsMessengerEXT");
    if (func != nullptr)
      func(instance, debugMessenger, pAllocator);
  }
```

4.2.3 Surface

Now that we have some error checking and a window, we need a surface to render on. GLFW provides us with a function to create a surface. Let's add a private function, in the window class, called *createWindowSurface* that creates a surface with the *glfwCreateWindowSurface* function. It takes the instance and a pointer to the surface as arguments. So add a private variable to store the surface and a *createSurface* function in the device class.

```cpp
//device.cpp
Device::Device(Window &window) : window{window} {
    createInstance();
    setupDebugMessenger();
    createSurface();
  }
```

```
void Device::createSurface() {
  window.createWindowSurface(instance, &surface);
}
```

```
//window.cpp
void Window::createWindowSurface(VkInstance instance, VkSurfaceKHR *
    surface) {
  if (glfwCreateWindowSurface(instance, window, nullptr, surface) !=
    VK_SUCCESS)
     throw std::runtime_error("Surface creation failed!");
}
```

We also have to destroy the surface in the destructor of the device class.

```
Device::~Device() {
  if (enableValidationLayers)
    destroyDebugUtilsMessengerEXT(instance, debugMessenger, nullptr);

  vkDestroySurfaceKHR(instance, surface, nullptr);
  vkDestroyInstance(instance, nullptr);
}
```

4.2.4 Physical Device

The physical device is the actual graphics card that we will use to render our scene. We can use either one or multiple physical devices to render our scene. But for now we will only use one.

The first thing we need to do is to get the available physical devices. Then we check each of them until we find a suitable device.

Let's begin with defining our helper functions *pickPhysicalDevice* and *isDeviceSuitable* and a private variable to store the physical device.

```
//device.hpp
private:
...
VkPhysicalDevice physicalDevice = VK_NULL_HANDLE;
```

```
VkPhysicalDeviceProperties properties;
void pickPhysicalDevice();
bool isDeviceSuitable(VkPhysicalDevice device);
```

Next we call the *pickPhysicalDevice* function in our constructor. Inside the function we first need to get all available physical devices with the *vkEnumeratePhysicalDevices* function. Then we iterate through the devices and check until we find a suitable device with the *isDeviceSuitable* function. In case the loop ends without finding a suitable device we check if the physical device is *VK_NULL_HANDLE*. If it is we throw an exception with an error message. Finally we get the properties of our device with *vkGetPhysicalDeviceProperties* and store it in a private variable.

```
//device.cpp
Device::Device(Window &window) : window{window} {
  ...
  pickPhysicalDevice();
}

void Device::pickPhysicalDevice(){
  uint32_t deviceCount = 0;
  vkEnumeratePhysicalDevices(instance, &deviceCount, nullptr);

  if(deviceCount == 0)
    throw std::runtime_error("Failed to find a GPU that supports
   Vulkan!");

  std::vector<VkPhysicalDevice> devices(deviceCount);
  vkEnumeratePhysicalDevices(instance, &deviceCount, &devices);

  for(const auto &device : devices){
    if(isDeviceSuitable(device)){
      physicalDevice = device;
      break;
    }
  }

  if(physicalDevice == VK_NULL_HANDLE)
```

```
    throw std::runtime_error("No Device was suitable!");

  vkGetPhysicalDeviceProperties(physicalDevice, &properties);
  \\optional
  std::cout << "Physical Device: " << properties.deviceName << std::
   endl;
}
```

The *isDeviceSuitable* function is used to check if the device is suitable for our application. In here we will check if the device can process graphics, present images on the surface, supports the required extensions and if it is swapchain adequate.

Okay, let's start by checking if the device supports graphics commands. This data is stored in the *VkQueueFamilyProperties* struct and needs to be queried with the *vkGetPhysicalDeviceQueueFamilyProperties* function from the physical device.

Before I explain further I have to explain what a queue and a queue family is. Do you remember when I mentioned Command Buffers? These are used to send commands to the GPU. For example, to run a graphics pipeline we would add the *vkCmdDraw* command to the command buffer. This command buffer then has to be submitted to a queue. A queue determines the order in which the command buffers are executed.

But here is the catch. Not all queues can execute all command buffers. One queue might be able to execute graphics commands, while another queue might be able to execute compute commands and so on. Therefore we have to check if the device supports the required queues, which are graphics for now.

We can get the available queues with the *vkGetPhysicalDeviceQueueFamilyProperties* function. It returns all queues with the same capabilities as a queue family. So when the GPU has 2 queues that can execute graphics commands we get a queue family with a queue count of 2 and a flag that indicates that it can execute graphics commands.

Let's go ahead and add a private function called *findQueueFamilies* that returns a struct that contains the indices of the graphics and present queue family. We will call this struct *QueueFamilyIndices* and it will contain an optional uint32_t for the graphics queue family, an optional uint32_t for the present queue family and a function *isComplete* that returns true if the graphics and present queue family are set.

Inside *findQueueFamilies* we can check if the queue family can execute graphics commands by iterating through each queue family and checking if its *queueFlags* member has the *VK_QUEUE_GRAPHICS_BIT* set. If it is we set the graphics queue family index to the current index. We also check for a queue family that can present images on a given surface with the *vkGetPhysicalDeviceSurfaceSupportKHR* function. If it does we set the present queue family index to the current index.

We will then return the *QueueFamilyIndices* struct from the *findQueueFamilies* function. We will need the indices to create the device.

```
\\device.hpp

struct QueueFamilyIndices {
  std::optional<uint32_t> graphicsFamily;
  std::optional<uint32_t> presentFamily;

  bool isComplete() {
    return graphicsFamily.has_value() && presentFamily.has_value();
  }
};
```

```
\\device.cpp

bool Device::isDeviceSuitable(VkPhysicalDevice device) {
  QueueFamilyIndices indices = findQueueFamilies(device);

  return indices.isComplete();
}

QueueFamilyIndices Device::findQueueFamilies(VkPhysicalDevice device)
   {
  QueueFamilyIndices indices;

  uint32_t queueFamilyCount = 0;
  vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount,
   nullptr);
  std::vector<VkQueueFamilyProperties> queueFamilies(queueFamilyCount)
   ;
  vkGetPhysicalDeviceQueueFamilyProperties(device, &queueFamilyCount,
   queueFamilies.data());

  int i = 0;
  for (const auto &queueFamily : queueFamilies) {
    if (queueFamily.queueCount > 0 && queueFamily.queueFlags &
   VK_QUEUE_GRAPHICS_BIT)
      indices.graphicsFamily = i;

    VkBool32 presentSupport = false;
    vkGetPhysicalDeviceSurfaceSupportKHR(device, i, surface, &
   presentSupport);

    if (queueFamily.queueCount > 0 && presentSupport)
      indices.presentFamily = i;

    if (indices.isComplete())
      break;

    i++;
  }

  return indices;
}
```

Now we check if the device supports the required extensions. For that we will create another helper function called *checkDeviceExtensionSupport* that returns a boolean. We will use the *vkEnumerateDeviceExtensionProperties* to get the available extensions. Then we will create a set of strings that contains the required extensions which will be the *VK_KHR_SWAPCHAIN_EXTENSION_NAME* for now. We will then iterate through the available extensions and remove the required extensions from the set. If the set is empty we return true, otherwise false.

```
//device.hpp

```

```
private:
  bool checkDeviceExtensionSupport(VkPhysicalDevice device);
  const std::vector<const char *> deviceExtensions = {
    VK_KHR_SWAPCHAIN_EXTENSION_NAME
  };
```

```
//device.cpp
bool Device::isDeviceSuitable(VkPhysicalDevice device) {
  QueueFamilyIndices indices = findQueueFamilies(device);
  bool extensionsSupported = checkDeviceExtensionSupport(device);

  return indices.isComplete() && extensionsSupported;
}

bool Device::checkDeviceExtensionSupport(VkPhysicalDevice device) {
  uint32_t extensionCount;
  vkEnumerateDeviceExtensionProperties(device, nullptr, &
   extensionCount, nullptr);

  std::vector<VkExtensionProperties> availableExtensions(
   extensionCount);
  vkEnumerateDeviceExtensionProperties(device, nullptr, &
   extensionCount, availableExtensions.data());

  std::set<std::string> requiredExtensions(deviceExtensions.begin(),
   deviceExtensions.end());

  for (const auto &extension : availableExtensions)
    requiredExtensions.erase(extension.extensionName);

  return requiredExtensions.empty();
}
```

Another thing we have to check is if the device is adequate for a swapchain and simultaneously get some required data for creating the swapchain.

Each device has a set of capabilities, like the minimum and maximum images for a swapchain and the minimum and maximum width and height of the image, that we can get with the *vkGetPhysicalDeviceSurfaceCapabilitiesKHR* function.

We also need the formats and the present modes that the device supports via the *vkGetPhysicalDeviceSurfaceFormatsKHR* and the *vkGetPhysicalDeviceSurfacePresent-ModesKHR* functions.

Let's go ahead and create another struct called *SwapChainSupportDetails* that stores the capabilities, formats and present modes.

```
//device.hpp

struct SwapChainSupportDetails{
  VkSurfaceCapabilitiesKHR capabilities;
  std::vector<VkSurfaceFormatKHR> formats;
  std::vector<VkPresentModeKHR> presentModes;
}
```

Next we create a function called *querySwapChainSupport* that returns our *SwapChain-SupportDetails* struct with the data.

```
//device.cpp

bool Device::isDeviceSuitable(VkPhysicalDevice device){
  ...
  bool swapChainAdequate = false;
  if(extensionsSupported){
    SwapChainSupportDetails swapChainSupport = querySwapChainSupport(
   device);
    swapChainAdequate = !swapChainSupport.formats.empty() &&
                        !swapChainSupport.presentModes.empty();
  }

  return indices.isComplete() && extensionsSupported &&
   swapChainAdequate;
}

SwapChainSupportDetails Device::querySwapChainSupport(VkPhysicalDevice
    device){
  SwapChainSupportDetails details;
  vkGetPhysicalDeviceSurfaceCapabilitiesKHR(device, surface_, &details
   .capabilities);

  uint32_t formatCount;
  vkGetPhysicalDeviceSurfaceFormatsKHR(device, surface_, &formatCount,
    nullptr);

  if(formatCount != 0){
    details.formats.resize(formatCount);
    vkGetPhysicalDeviceSurfaceFormatsKHR(device, surface_, &
   formatCount, details.formats.data());
  }

  uint32_t presentModeCount;
  vkGetPhysicalDeviceSurfacePresentModesKHR(device, surface_, &
   presentModeCount, nullptr);

  if(presentModeCount != 0){
    details.presentModes.resize(presentModeCount);
    vkGetPhysicalDeviceSurfacePresentModesKHR(device, surface_, &
   presentModeCount, details.presentModes.data());
  }

  return details;
}
```

Finally we check if the device supports anisotropic filtering. Anisotropic filtering is a technique that enhances the image quality of textures on surfaces, by reducing blurriness that can occur when rendering texture on a sharp angle. We'll look deeper into this topic later.

We can check if the device supports anisotropic filtering by checking the *VkPhysicalDeviceFeatures* struct that contains the features of the device. *vkGetPhysicalDeviceFeatures* returns the features of the device, which contains the boolean *samplerAnisotropy* member.

```
bool Device::isDeviceSuitable(VkPhysicalDevice device){
  ...
  VkPhysicalDeviceFeatures supportedFeatures;
  vkGetPhysicalDeviceFeatures(device, &supportedFeatures);

  return indices.isComplete() && extensionsSupported &&
   swapChainAdequate && supportedFeatures.samplerAnisotropy;
}
```

That's it we have now picked a physical device that is suitable for our application. But please note that we just take the first suitable device we find, which is not the best practice. You should rather rate the devices by certain factors. For example, you could favor a dedicated GPU over an integrated GPU.

For now we will just take the first suitable device we find.

4.2.5 Logical Device

We currently have a physical device but no interface to interact with it. This interface is provided by a logical device that we will create in the next step. Let's add a private function called *createLogicalDevice* that creates the logical device and a private variable to store the device. We'll also create our queues here and will need a *VkQueue* variable for each queue;

```
//device.hpp
private:
  ...
  VkDevice device_;
  VkQueue graphicsQueue_;
  VkQueue presentQueue_;
  void createLogicalDevice();
```

The logical device is created with the *vkCreateDevice* function. We have to specify the queues that we want to create for the device. We do so by populating the *VkDevice-QueueCreateInfo* struct and add it to the *VkDeviceCreateInfo* struct. We'll need to specify the queue family index, the amount of queues and the priority of the queues. We'll keep the same priority for all queues for now, but you can adjust the priority to your needs. We also have to specify the device features in an *VkPhysicalDeviceFeatures* struct, that we want to enable, which will be sampler anisotropy for now. We also have to specify the extensions that we want to enable, which is the swapchain extension in our *deviceExtensions* vector.

When we create the logical device we also have to check if the creation was successful. If not we throw an exception with an error message. After that we retrieve the queues from the device with the *vkGetDeviceQueue* function.

```
//device.cpp
Device::Device(Window &window) : window{window} {
  ...
  createLogicalDevice();
}

void Device::createLogicalDevice(){
  QueueFamilyIndices indices = findQueueFamilies(physicalDevice);

  std::vector<VkDeviceQueueCreateInfo> queueCreateInfos;
  std::set<uint32_t> uniqueQueueFamilies = {indices.graphicsFamily.
   value(), indices.presentFamily.value()};

  float queuePriority = 1.0f;
  for(uint32_t queueFamily : uniqueQueueFamilies){
    VkDeviceQueueCreateInfo queueCreateInfo = {};
    queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO
   ;
    queueCreateInfo.queueFamilyIndex = queueFamily;
    queueCreateInfo.queueCount = 1;
    queueCreateInfo.pQueuePriorities = &queuePriority;
    queueCreateInfos.push_back(queueCreateInfo);
  }

  VkPhysicalDeviceFeatures deviceFeatures = {};
  deviceFeatures.samplerAnisotropy = VK_TRUE;

  VkDeviceCreateInfo createInfo = {};
  createInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
  createInfo.queueCreateInfoCount = static_cast<uint32_t>(
   queueCreateInfos.size());
  createInfo.pQueueCreateInfos = queueCreateInfos.data();
  createInfo.enabledExtensionCount = static_cast<uint32_t>(
   deviceExtensions.size());
  createInfo.pEnabledFeatures = &deviceFeatures;
  createInfo.ppEnabledExtensionNames = deviceExtensions.data();

  if(vkCreateDevice(physicalDevice, &createInfo, nullptr, &device_) !=
    VK_SUCCESS)
    throw std::runtime_error("Failed to create logical device!");

  vkGetDeviceQueue(device_, indices.graphicsFamily.value(), 0, &
   graphicsQueue_);
  vkGetDeviceQueue(device_, indices.presentFamily.value(), 0, &
   presentQueue_);
}
```

Now we have to make sure that the logical device is destroyed in the cunstructor. We can do this by calling the *vkDestroyDevice* function.

```
Device::~Device() {
  vkDestroyDevice(device_, nullptr);

  if (enableValidationLayers)
    destroyDebugUtilsMessengerEXT(instance, debugMessenger, nullptr);

  vkDestroySurfaceKHR(instance, surface_, nullptr);
  vkDestroyInstance(instance, nullptr);
}
```

4.2.6 Command Pool

The last step that this chapter will cover is the creation of a command pool. A command pool is used to manage the memory for storing buffers. From the command pool we can allocate command buffers that store the commands that we want to send to the GPU.

Let's add a private variable to store the command pool and a private function to create the command pool called *createCommandPool*. Inside the function we have to again populate a struct, which will be the *VkCommandPoolCreateInfo* struct. We have to specify the queue family index that the command pool will create the command buffers for. We will use the graphics queue family index, because thats the commands we want to send to the GPU.

The *VkCommandPoolCreateInfo* struct also has a *flags* member. By setting the member to the *VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT* we can reset command buffers individually with the *vkResetCommandBuffer* function or via the implicit reset that occurs when calling *vkBeginCommandBuffer*.

We then create the command pool with the *vkCreateCommandPool* function and check if the creation was successful. If not we throw an exception with an error message. We also have to destroy the command pool in the destructor.

```
//device.hpp
private:
  ...
  VkCommandPool commandPool;
  void createCommandPool();
```

```
//device.cpp
Device::Device(Window &window) : window{window} {
    ...
    createCommandPool();
  }

void Device::createCommandPool(){
  QueueFamilyIndices indices = findQueueFamilies(physicalDevice);

  VkCommandPoolCreateInfo poolInfo = {};
  poolInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
  poolInfo.queueFamilyIndex = indices.graphicsFamily.value();
  poolInfo.flags = VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;

  if(vkCreateCommandPool(device_, &poolInfo, nullptr, &commandPool) !=
    VK_SUCCESS)
    throw std::runtime_error("Failed to create command pool!");
}

Device::~Device() {
  vkDestroyCommandPool(device_, commandPool, nullptr);
  ...
}
```

And that's it for the device class. Let's recap what we have done so far. We have created a Vulkan instance that allows us to interact with the Vulkan API. We also added validation layers to get additional error checking and a debug messenger to output the error messages. We have created a surface to render on and picked a physical device that is suitable to render our scene for the given surface. Then we created a logical device that is used to interact with the physical device. We also created a command pool to manage the memory for storing command buffers.

You might ask yourself why we didn't split the code into multiple classes, like the instance, the surface, the device and the command pool class. The reason is that all of them rely on each other. For example the surface relies on the instance and to pick the physical device we need to check if the device supports the surface. Having all of them in one class also saves us from writing classes that will only be used once. But you can split the code into multiple classes if you want.

But we are not done yet. We have to instantiate the device class in the application class. We can do that directly in the header file. Let's add a private variable to the application class that stores the device.

```
//app.hpp
private:
  ...
  Device device{window};
```

In the next chapters we will come back to the device class and add functionalities like creating buffers. But for now we will move on to the swapchain.

4.3 Swapchain

When we picked our physical device we also queried the swapchain support details. We will use this data to create the swapchain. Let's add a new file called *swapchain.cpp* and *swapchain.hpp* to the project.

Let's start off by creating a new class called *Swapchain* in the *swapchain.hpp* file and adding a constructor that takes a reference to the device and a *VkExtent2D* struct as arguments. The extent is the resolution of the swapchain images. We will also add a function called *createSwapchain* that creates the swapchain. We'll also delete the copy constructor and the copy assignment operator.

Because the *querySwapChainSupport* function can return multiple formats present modes we will add private functions that chooses a format and present mode from the available formats and present modes.

We also need a function that chooses the extent of the swapchain images. You might assume that the extent of the swapchain images matches the extent of the window. But this is not always the case. GLFW measures the resolution in screen coordinates but Vulkan measures the resolution in pixels. This means that our swapchain extent needs to be in pixels as well. The problem is that when using high DPI displays, the extent in screen coordinates will be smaller than the extent in pixels. We can get the extent in pixels with the *glfwGetFramebufferSize* function, but we have the extent already defined in the windowExtent variable. We will use the window extent and then clamp it to the minimum and maximum extent that the device supports. After the creation of the swapchain we will also get the swapchain images from it and store them in a vector of *VkImages*, which contains the pixels and the main memory of the texture, but misses the information about how the image can be read.

```
//swapchain.hpp
namespace engine {
class SwapChain {
public:
  SwapChain (Device &device, VkExtent2D extent);
  ~SwapChain ();
  SwapChain (const Swapchain &) = delete;
  SwapChain &operator=(const Swapchain &) = delete;

private:
  Device &device;
  VkExtent2D windowExtent;

  VkSwapchainKHR swapChain;
  std::vector<VkImage> swapChainImages;

  void createSwapChain();
  VkSurfaceFormatKHR chooseSwapSurfaceFormat(const std::vector<
    VkSurfaceFormatKHR> &availableFormats);
  VkPresentModeKHR chooseSwapPresentMode(const std::vector<
    VkPresentModeKHR> &availablePresentModes);
  VkExtent2D chooseSwapExtent(const VkSurfaceCapabilitiesKHR &
    capabilities);
};
}
```

The constructor calls the *createSwapChain* function. Inside of it we have to populate the *VkSwapchainCreateInfoKHR* struct. It requires the surface, the min image count, the image format and color space, the image extent, the image array layers, the image usage, the sharing mode, the queue family indices, the pre transform, the composite alpha, the present mode, the clipped and the old swapchain.

We get the surface from the device, but currently it is a private variable. We will need a getter function to the device class that returns the surface. We will also get data like the min image count from the swapchain support details. Therefore we will add a public function that calls the *querySwapChainSupport* function. We will do the same for the *findQueueFamilies* function. To create the swapchain we also need the logical device. We could add a getter function to the device class that returns the logical device, but I rather define a conversion operator that returns the device when we access the device instance.

```
//device.hpp
```

```
VkSurfaceKHR surface() { return surface_; }
operator VkDevice() const { return device_; }

SwapChainSupportDetails getSwapChainSupport() { return
   querySwapChainSupport(device_); }
QueueFamilyIndices findQueueFamilies() { return findQueueFamilies(
   physicalDevice); }
```

In the *swapchain.cpp* file we will implement the *createSwapChain* function. We will also implement the *chooseSwapSurfaceFormat*, *chooseSwapPresentMode* and *chooseSwapExtent* functions.

Because they rely on the swapchain support details we will start off by calling the *getSwapChainSupport* function. We will then call the *chooseSwapSurfaceFormat*.

```
//swapchain.cpp
SwapChain::SwapChain(Device &device, VkExtent2D extent) : device{
   device}, windowExtent{extent} {
  createSwapChain();
}

void SwapChain::createSwapChain() {
  SwapChainSupportDetails swapChainSupport = device.
   getSwapChainSupport();

  VkSurfaceFormatKHR surfaceFormat = chooseSwapSurfaceFormat(
   swapChainSupport.formats);
  VkPresentModeKHR presentMode = chooseSwapPresentMode(
   swapChainSupport.presentModes);
  VkExtent2D extent = chooseSwapExtent(swapChainSupport.capabilities);
}
```

4.3.1 Swapchain Surface Format

The *VkSurfaceFormatKHR* struct contains a format and a color space member. The format defines in what order and size the R, G, B and A components are stored. We will use the *VK_FORMAT_B8G8R8A8_SRGB* format, that stores the components in the order B, G, R and A with 8 bits each. The reason we use the *SRGB* format is that it is the most common format and is supported and optimized on most devices. The color space is used to determine if the SRGB color space is supported. We will iterate through the available formats and check if the format and color space are supported. If they are we return the format. If not we return the first format. You could add additional checks to choose the best format but for now we will assume that sRGB is supported.

```
VkSurfaceFormatKHR SwapChain::chooseSwapSurfaceFormat(const std::
    vector<VkSurfaceFormatKHR> &availableFormats) {
  for (const auto &availableFormat : availableFormats) {
    if (availableFormat.format == VK_FORMAT_B8G8R8A8_SRGB &&
    availableFormat.colorSpace == VK_COLOR_SPACE_SRGB_NONLINEAR_KHR)
      return availableFormat;
  }

  return availableFormats[0];
}
```

4.3.2 Swapchain Present Mode

The present mode determines how the images are presented to the screen. To understand this we have to take a deeper look into the swapchain. The swapchain contains multiple framebuffers. A framebuffer is an array of pixel values that are used to render an image. The swapchain contains multiple framebuffers that are used to render multiple images. If we use triple buffering, which means that we have 3 framebuffers, it could look like this: The first framebuffer is the one that is currently being displayed on the screen. The second framebuffer is the one that is being rendered to and the third framebuffer is the one that is waiting to be rendered to.

Let's take a look at the first present mode called
VK_PRESENT_MODE_IMMEDIATE_KHR. This mode will display the image as soon as it is done rendering. The problem is, that each monitor has a refresh rate that defines when the swapchain swaps the framebuffers. The moment the display is refreshed is called a vertical blank. If the vertical blank happens while the image is being rendered

we get a screen tearing effect.

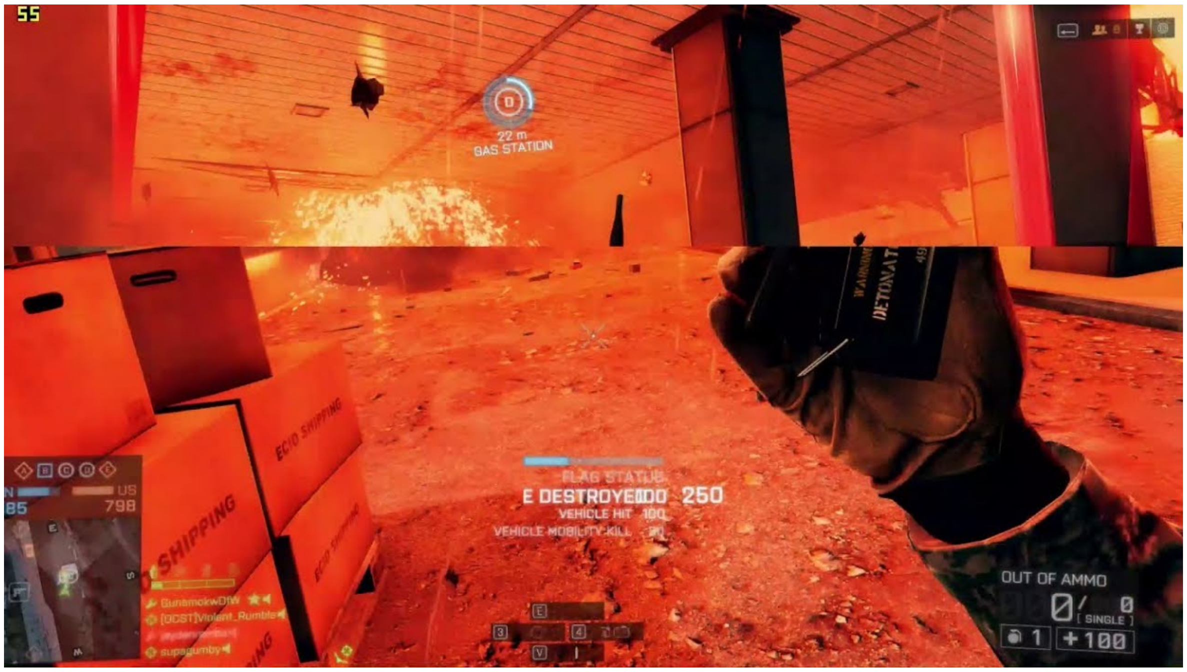

Figure 16: Screen Tearing
[38]

Here you can see that 2 images are displayed at the same time. The top image is the new one that has not managed to fully render yet and the bottom image is the old one that was being displayed before the refresh. Remember that we overwrite the old image with the new one when rendering. Because not every pixel was done rendering when the refresh happened we get the rendered part and the old part displayed at the same time. This is obviously not optimal and we want to avoid this. So let's move on to the next one.

The next present mode is called *VK_PRESENT_MODE_FIFO_KHR*. FIFO stands for first in first out. This mode handles the images in a queue. Each fully rendered image is added to the queue and the image that is at its front is displayed. The *VK_PRESENT_MODE_FIFO_KHR* only swaps the framebuffers when the vertical blank happens and takes the framebuffer's image that is at the front of the queue, which avoids the screen tearing effect. The problem with this mode is that if the queue is full, aka each framebuffer's image is rendered but not displayed yet, the GPU has to wait until the vertical blank happens to continue rendering, which caps the frame rate to the refresh rate of the monitor. This is called V-Sync and causes a delay between the rendering and the displaying of the image, but is guaranteed to be available and has a lower energy consumption than the other modes that have the GPU rendering without a break. Therefore this mode is the most common mode for mobile devices, but is still widely used for desktop applications.

There is a similar version of the *VK_PRESENT_MODE_FIFO_KHR* mode called *VK_PRESENT_MODE_FIFO_RELAXED_KHR*. This mode only differs from the previous mode in the way it handles a empty queue. If the queue is empty, aka no framebuffer's image is rendered yet, the swapchain will swap the framebuffers immediately and display a teared image. This mode is used when we have a long rendering time and want to avoid waiting for the queue to be filled.

Finally we have the
VK_PRESENT_MODE_MAILBOX_KHR mode. This is also the mode that we will be using. This mode is similar to the *VK_PRESENT_MODE_FIFO_KHR* mode, with the difference that the swapchain will not let the GPU idle when the queue is full. Instead it will discard the image that is at the front of the queue and start rendering again. This constant rendering will provide us with the lowest latency but also the highest energy consumption. But that should not be a problem for our application.

We will simply iterate through the available present modes and return the
VK_PRESENT_MODE_MAILBOX_KHR mode if it is available. If not we will return the *VK_PRESENT_MODE_FIFO_KHR* mode.

```
VkPresentModeKHR SwapChain::chooseSwapPresentMode(const std::vector<
    VkPresentModeKHR> &availablePresentModes) {
  for (const auto &availablePresentMode : availablePresentModes) {
      if (availablePresentMode == VK_PRESENT_MODE_MAILBOX_KHR)
      return availablePresentMode;
    }

  return VK_PRESENT_MODE_FIFO_KHR;
}
```

4.3.3 Swapchain Extent

Like mentioned before we have to choose the extent of the swapchain images. We get the extent from the devices capabilities. If the width and height of the extent are set to the maximum value of uint32_t, we are indicated that the extent is defined in screen coordinates and we have to get the extent from our windowExtent variable. We then have to clamp the extent to the minimum and maximum extent that the device supports tho. If the width and height are not set to the maximum value we can use the extent directly because that tells us that the extent is already in pixels.

```
VkExtent2D SwapChain::chooseSwapExtent(const VkSurfaceCapabilitiesKHR
   &capabilities) {
  if (capabilities.currentExtent.width != UINT32_MAX)
    return capabilities.currentExtent;
  else {
    VkExtent2D actualExtent = windowExtent;

    actualExtent.width = std::clamp(actualExtent.width, capabilities.
   minImageExtent.width, capabilities.maxImageExtent.width);
    actualExtent.height = std::clamp(actualExtent.height, capabilities
   .minImageExtent.height, capabilities.maxImageExtent.height);

    return actualExtent;
  }
}
```

4.3.4 Image Count

The final thing we have to do is to set the min image count of the swapchain. The image count is the amount of images that the swapchain will contain aka the amount of framebuffers. We will use one more than the minimum image count that the device supports, for the min image count of the swapchain. This is because we otherwise would have to wait for the driver to complete internal operations before we can acquire the next image. Because we increase the min image count by one we have to check if it is bigger than the max image count. If it is we set the min image count to the max image count. Except if the max image count of the device is 0, which means that there is no maximum image count.

```
uint32_t imageCount = swapChainSupport.capabilities.minImageCount + 1;
if (swapChainSupport.capabilities.maxImageCount > 0 && imageCount >
    swapChainSupport.capabilities.maxImageCount)
  imageCount = swapChainSupport.capabilities.maxImageCount;
```

4.3.5 Swapchain Creation

Now that we have all the data we need we can create the swapchain. We will populate the *VkSwapchainCreateInfoKHR* struct with the data we have and additionally set the image array layers to 1, which is the amount of layers that the image has. This is always 1 unless you are working with stereoscopic 3D applications. We also have to set the image usage to *VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT*, which means that we will render directly to the image.

```
VkSwapchainCreateInfoKHR createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
createInfo.surface = device.surface();
createInfo.minImageCount = imageCount;
createInfo.imageFormat = surfaceFormat.format;
createInfo.imageColorSpace = surfaceFormat.colorSpace;
createInfo.imageExtent = extent;
createInfo.imageArrayLayers = 1;
createInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
```

But that's not all. We also have to specify the sharing mode of the swapchain. The sharing mode determines how the images are shared between multiple queue families. We use 2 queue families, the graphics queue family and the present queue family. Now we have 2 cases. The first case is that the graphics queue family and the present queue family have the same index. In this case we set the sharing mode to exclusive. This means that an image is owned by one queue family at a time and has to be explicitly transferred to the other queue family. In this case the count of queue families and the queue family indices that are used are not relevant. The second case is that the graphics queue family and the present queue family have different indices. In this case we set the sharing mode to concurrent. This means that an image can be used by multiple queue families without explicit ownership transfers. It is important to note that the queue family indices are relevant in this case, because the swapchain needs to know what queue families can access the image.

We'll start off by getting the queue family indices from the device. We will then check if the graphics queue family and the present queue family are different. If they are we set the sharing mode to concurrent and set the queue family indices and their count. If they are the same we set the sharing mode to exclusive.

```
QueueFamilyIndices indices = device.findQueueFamilies();
if (indices.graphicsFamily.value() != indices.presentFamily.value()) {
  createInfo.imageSharingMode = VK_SHARING_MODE_CONCURRENT;
  createInfo.queueFamilyIndexCount = 2;
  uint32_t queueFamilyIndices[] = {indices.graphicsFamily.value(),
   indices.presentFamily.value()};
  createInfo.pQueueFamilyIndices = queueFamilyIndices;
} else {
  createInfo.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
}
```

We're still not done yet! We also have to specify if and what kind of transformation we want to apply to the images. For example we might want to flip or rotate the image. We don't want anything to happen to the image so we set the pre transform to the current transform. We get the current transform from the swapchain support details capabilities. We also have to specify the composite alpha. The composite alpha is used to blend the window with other windows in the system. We want to ignore the alpha channel so we set the composite alpha to the opaque bit. Finally we specify our present mode, the clipped flag and the old swapchain. The clipped flag indicates that we don't need the pixels that are obscured by other windows. If you need the pixels to predict future frames you can set the clipped flag to false but we will keep it true. The old swapchain is used to create a new swapchain from an old one. We will set it to *VK_NULL_HANDLE*, but will come back to this later.

```
createInfo.preTransform = swapChainSupport.capabilities.
   currentTransform;
```

```
createInfo.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
createInfo.presentMode = presentMode;
createInfo.clipped = VK_TRUE;
createInfo.oldSwapchain = VK_NULL_HANDLE;
```

Now that we have the create info struct we can create the swapchain with the *vkCreateSwapchainKHR* function, that takes the device, the create info struct, a custom allocator and a pointer to the swapchain variable that we also have to add to our header. We again will check for an error and throw an runtime error if the creation was not successful.

```
if (vkCreateSwapchainKHR(device, &createInfo, nullptr, &swapChain) !=
    VK_SUCCESS)
  throw std::runtime_error("Failed to create swap chain!");
```

After the creation of the swapchain we need to store the images of the swapchain. We can get the images with the *vkGetSwapchainImagesKHR* function. We have to specify the device, the swapchain, a pointer to the image count and a pointer to the images. We will store the images in the *swapChainImages* vector. Then we should also create a variable for the image format and the extent of the swapchain images and store them in the class.

```
//swapchain.hpp
private:
  ...
  VkSwapchainKHR swapChain;
  VkFormat swapChainImageFormat;
  VkExtent2D swapChainExtent;
```

```
vkGetSwapchainImagesKHR(device, swapChain, &imageCount, nullptr);
swapChainImages.resize(imageCount);
vkGetSwapchainImagesKHR(device, swapChain, &imageCount,
    swapChainImages.data());

swapChainImageFormat = surfaceFormat.format;
swapChainExtent = extent;
```

If you ask yourself why we have to get the image count first and then resize the vector, it is because we have only defined the minimum image count of the swapchain in the imageCount variable. The actual image count could be different.

Now we have to make sure that the swapchain is destroyed in the destructor. We can do this by calling the *vkDestroySwapchainKHR* function.

```
SwapChain::~SwapChain() {
  vkDestroySwapchainKHR(device, swapChain, nullptr);
}
```

4.3.6 Image Views

The Swapchain in now created and we have the images stored in the *swapChainImages* vector. But like mentioned before, we cannot read the images directly. It only stores data about the pixels and the main memory of the texture and misses information like the format or the mipmapping level. To get the full image data we have to create an image view, that stores the missing information. We will need a new function called *createImageViews* that will be called after the creation of the swapchain. We will also need a vector to store the image views, that we will size to the amount of swapchain images in the function.

For Each image in the *swapChainImages* vector we will create an image view and define the image that we view into, the desired format, the view type, the subresource range with the aspect mask, the base mip level, the level count, the base array layer and the layer count. The view type defines how many dimensions the image has. We will use the 2D view type because our pipeline returns a orthogonal 2D image. The aspect mask defines what the image view will be used for. You might ask yourself why we have to define the aspect mask in the image view, when we already defined the image usage in the swapchain creation. The reason for that is that we can use one image for multiple purposes. For example we can use one image for color and depth attachments. The view might only need the color information, so we have to define the aspect mask in the image view to reduce the amount of data that is stored in the view. We will use the color aspect mask for now. We will also keep the same format as the swapchain image format, but for some use cases you might want to use a different format that is compatible with the surface format.

Before we go further I'd like to explain what mipmapping is and what a mip level is. Mipmapping is a technique that is similar to LOD (Level of Detail), but is used for textures. It reduces the textures resolution when the texture is far away from the camera. This fixes an interesting problem that occurs when rendering textures. Would we keep the same resolution for the texture when the texture is far away from the camera, the texture create a so called moiré pattern. It occures because we try to map a high resolution texture on a area that is to small.

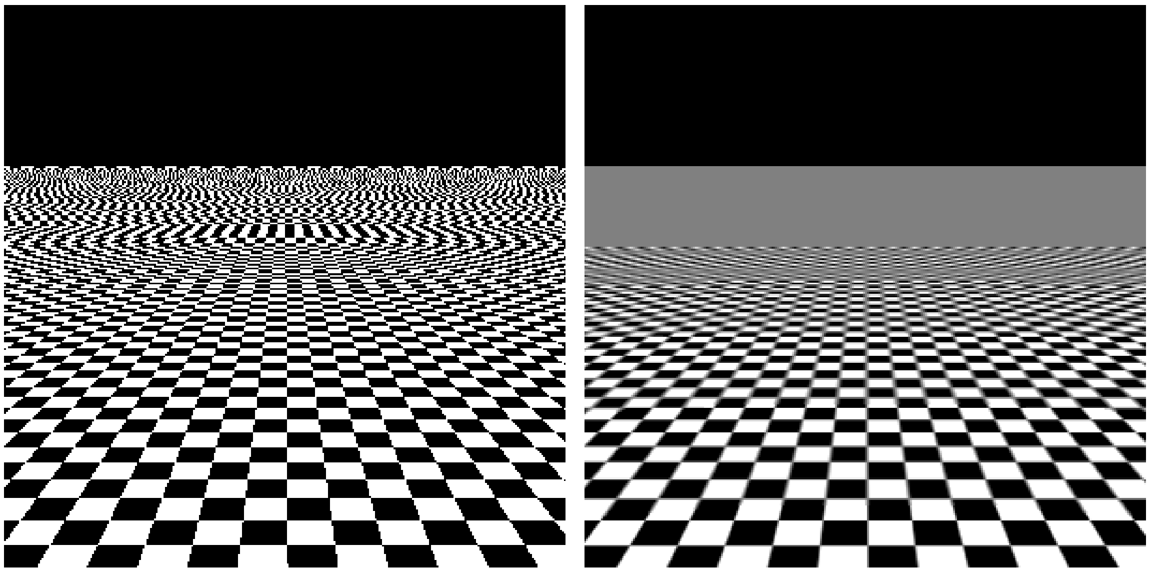

Figure 17: Mipmapping(left: Moiré pattern, right: Mipmapping) [39]

By reducing the resolution of the texture we can avoid this problem. When we reduce the resolution of the texture we create a new texture that is half the size of the original texture. So when we have a texture with a resolution of 256x256 we create a new texture with a resolution of 128x128 and so on, until we reach a resolution of 1x1. The different textures are called mip levels.

Now that we now that we can set the base mip level that basically sets the first mip level that is used. We will set it to 0 we get the original resolution of the texture. The level count defines how many mip levels we want to use. We will set it to 1 for now. The base array layer defines the first layer that is used. We will set it to 0 because we will only use one layer. The layer count defines how many layers we want to use. We will keep it at 1 for now.

```
SwapChain::SwapChain(Device &device, VkExtent2D extent) : device{
    device}, windowExtent{extent} {
  createSwapChain();
  createImageViews();
}

void SwapChain::createImageViews() {
  swapChainImageViews.resize(swapChainImages.size());

  for (size_t i = 0; i < swapChainImages.size(); i++) {
    VkImageViewCreateInfo viewInfo = {};
    viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    viewInfo.image = swapChainImages[i];
    viewInfo.format = swapChainImageFormat;
    viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
    viewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    viewInfo.subresourceRange.baseMipLevel = 0;
    viewInfo.subresourceRange.levelCount = 1;
    viewInfo.subresourceRange.baseArrayLayer = 0;
    viewInfo.subresourceRange.layerCount = 1;

    if (vkCreateImageView(device, &viewInfo, nullptr, &
   swapChainImageViews[i]) != VK_SUCCESS)
      throw std::runtime_error("Failed to create image views!");
  }
}
```

Great! Let's try to run the application. To do so we have to add the swapchain to the application class. Because the swapchain takes the device and the extent as arguments we have to add a getter function to the surface class that returns the extent. We will also add a conversion operator to the device class that returns the device.

```
//device.hpp
public:
  ...
  //returns device_ when we access the device instance
  operator VkDevice() const { return device_; }
```

```
//surface.hpp
public:
  ...
  VkExtent2D getExtent() { return{static_cast<uint32_t>(width),
   static_cast<uint32_t>(height)}; }
```

```
//app.hpp
private:
  ...
  SwapChain swapChain{device, window.getExtent()};
```

If everything is set up correctly so far the code should run.

Okay when closing the application we get an error that tells us to destroy the image views before the device. This is important because we have to destroy all child objects before we destroy the parent object.

So let's destroy the image views in the destructor of the swapchain class. We can do this by calling the *vkDestroyImageView* function for each image view in the *swapChain-ImageViews* vector.

```
SwapChain::~SwapChain() {
  for (const VkImageView &imageView : swapChainImageViews)
    vkDestroyImageView(device, imageView, nullptr);

  vkDestroySwapchainKHR(device, swapChain, nullptr);
}
```

We don't have to destroy the swapchain images because they are destroyed with the swapchain.

4.3.7 Render Pass

Before we can render anything we need to create a render pass. A render pass contains information about the amount of color and depth buffers, how many samples are used for them and how the data is handled during rendering.

Let's add a new function called *createRenderPass* to the swapchain class.

```
SwapChain::SwapChain(Device &device, VkExtent2D extent) : device{
   device}, windowExtent{extent} {
  createSwapChain();
  createImageViews();
  createRenderPass();
}

void SwapChain::createRenderPass() {}
```

Inside we will need to create 2 attachments. One for the color buffer and one for the depth buffer. Let's start with the color attachment. We will have to populate the *VkAttachmentDescription* struct with the format, the sample count, the load and store operation, the stencil load and store operation, the initial and the final layout. For the color attachment we will use the swapchain image format. The sample count defines how MSAA (Multisample Anti-Aliasing) is handled. Let's rewind shortly how MSAA works.

When we render a polygon we rasterize it into fragments. Each fragment is a pixel that is covered by the polygon. The graphics pipeline determines which pixel to color by checking if the polygon covers the center of the pixel. But this returns a jagged edge. To fix this we can use MSAA. MSAA works by sampling multiple points in the pixel and then averaging the color of the points. This creates a smoother edge. We can define how many samples we want to use in each fragment.

Okay back to the color attachment. We will use 1 sample for now. The load operation defines what to do with the data in the attachment before rendering. We will clear the frame buffer before rendering, because we don't need it. That means we have to set it to *VK_ATTACHMENT_LOAD_OP_CLEAR*. When you need it to be preserved you can set it to *VK_ATTACHMENT_LOAD_OP_LOAD*. The store operation defines what to do with the data in the attachment after rendering. We will store the data in the frame buffer after rendering, with the *VK_ATTACHMENT_STORE_OP_STORE* value. It is also possible to say that you don't care about the data in the attachment before or after rendering by setting the store or load operation to *VK_ATTACHMENT_STORE/LOAD*

_OP_DONT_CARE.

The stencil load and store operation are used for the stencil buffer. Wait what is a stencil buffer? A stencil buffer is another buffer that allows us to further manipulate pixels combined with the depth buffer. For example we can create outlines around objects by creating another object with a slightly bigger size and add a stencil buffer to it with a reference value. When rendering the outline we can perform a stencil test that checks if the pixels reference value is equal to 1 and only render the pixels that don't pass the test. The stencil load and store operation work the same way as the load and store operation We don't really care about the stencil buffer for now so we will set the operations to *VK_ATTACHMENT_STORE/LOAD _OP_DONT_CARE*. Finally the initial and final layout define how the image is handled before and after the render pass. The initial layout is set to *VK_IMAGE_LAYOUT_UNDEFINED* because we don't care about the image before the render pass. The final layout is set to *VK_IMAGE_LAYOUT_PRESENT_SRC_KHR*, which means that the image will be presented to the swapchain.

```
void SwapChain::createRenderPass() {
  VkAttachmentDescription colorAttachment = {};
  colorAttachment.format = swapChainImageFormat;
  colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
  colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
  colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
  colorAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
  colorAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
  colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
  colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
}
```

After that we need to create a *VkAttachmentReference* that stores the index and layout type. The color attachment will be our first attachment and has the *VK_IMAGE_LAYOUT_COLO*

```
VkAttachmentReference colorAttachmentRef = {};
colorAttachmentRef.attachment = 0;
colorAttachmentRef.layout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
```

Now we do the same for the depth attachment. For the format we will need a helper function that finds us a suitable depth format called *findDepthFormat*. This function calls the *findSupportedFormat* function that we will need to implement in the device class. The *findSupportedFormat* function takes a vector of formats, how the image is stored aka the tiling and the features.

Inside the *findSupportedFormat* function we will iterate through the formats, get their properties with the *vkGetPhysicalDeviceFormatProperties* function and check if the properties tiling features are equal to the features that we passed to the function. If it is we will return the format. We will check for 2 tilings, the linear and the optimal tiling. The linear tiling mode stores the image in a linear order, which means that the image is stored in a row like a 2D array. The optimal tiling mode stores the image in an oder that is optimized for GPU access. We will use the optimal tiling mode for the depth format. We will check the following formats: *VK_FORMAT_D32_SFLOAT*, *VK_FORMAT_D32_SFLOAT_S8_UINT* and *VK_FORMAT_D24_UNORM_S8_UINT*.

The first format is a 32 bit float format that is used for depth buffers. The second format is a two channel format with a 32 bit float for the depth and a 8 bit unsigned integer for the stencil buffer. The third format is a 24 bit float format for the depth and a 8 bit unsigned integer for the stencil buffer. We will parse in all 3 of them but if you need a stencil buffer you should use the second or third format. For the features we will use the *VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT*.

```
//swapchain.cpp
void SwapChain::createRenderPass(){
  ...
  VkAttachmentDescription depthAttachment = {};
  depthAttachment.format = findDepthFormat();
  depthAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
  depthAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
  depthAttachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
  depthAttachment.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
  depthAttachment.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
  depthAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
  depthAttachment.finalLayout =
   VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
  VkAttachmentReference depthAttachmentRef = {};
  depthAttachmentRef.attachment = 1;
  depthAttachmentRef.layout =
   VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
}

VkFormat SwapChain::findDepthFormat() {
  return device.findSupportedFormat(
    {VK_FORMAT_D32_SFLOAT, VK_FORMAT_D32_SFLOAT_S8_UINT,
   VK_FORMAT_D24_UNORM_S8_UINT},
    VK_IMAGE_TILING_OPTIMAL,
    VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT
  );
}
```

```
//device.cpp
VkFormat Device::findSupportedFormat(const std::vector<VkFormat> &
   candidates, VkImageTiling tiling, VkFormatFeatureFlags features) {
  for (VkFormat format : candidates) {
    VkFormatProperties props;
    vkGetPhysicalDeviceFormatProperties(physicalDevice, format, &props
   );

    if (tiling == VK_IMAGE_TILING_LINEAR && (props.
   linearTilingFeatures & features) == features)
      return format;
    else if (tiling == VK_IMAGE_TILING_OPTIMAL && (props.
   optimalTilingFeatures & features) == features)
      return format;
  }

  throw std::runtime_error("Failed to find supported format!");
}
```

We don't care about the depth data after rendering so we set the store operation to *VK_ATTACHMENT_STORE_OP_DONT_CARE*.

A renderpass can consist of multiple subpasses. A subpass is a rendering operation that takes the output of the previous subpass as input. We will only use one subpass for now, but using multiple subpasses can be useful for post processing effects. We will create a *VkSubpassDescription* struct that contains the pipeline bind point, the amount of color attachments, a pointer to the color attachments and a pointer to the depth stencil attachment. There is more data that we can add to the subpass description, but don't need for now. The pipeline bind point defines if the subpass is used for graphics or compute operations. We will use the graphics pipeline bind point.

```
void SwapChain::createRenderPass() {
  ...
  VkSubpassDescription subpass = {};
  subpass.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
  subpass.colorAttachmentCount = 1;
  subpass.pColorAttachments = &colorAttachmentRef;
  subpass.pDepthStencilAttachment = &depthAttachmentRef;
}
```

There is one problem with the subpass description. When we have multiple subpasses we might get a case where one subpass writes to an attachment and the next subpass reads from it, while the first subpass is not finished. This turns out to be a race condition and is falsifying the results. To avoid this we can use subpass dependencies. A subpass dependency tells the render pass when to start and end the subpass. We will create a *VkSubpassDependency* struct that contains the indices of the previous and the next subpass, the stage mask and the access mask. The *srcSubpass* defines where the dependency starts and the *dstSubpass* defines where the dependency ends. In our case our dependency starts outside of the subpass, which means that we have to set the *srcSubpass* to *VK_SUBPASS_EXTERNAL*. Because we only have one subpass we can set the *dstSubpass* to 0. The stage mask defines what stages of the pipeline the dependency is waiting for. We will wait for the color attachment output stage, which refers to any operations that render to the color attachment, and the early fragment test stage, which refers to the depth and stencil tests. The access mask defines what kind of operations the dependency will wait for. The source will not wait for any operations, so we set it to 0. The destination will wait until the color attachment is written to and the depth and stencil buffer is written to. We will set it to *VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT* and *VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT*.

```
void SwapChain::createRenderPass() {
  ...
  VkSubpassDependency dependency = {};
  dependency.srcSubpass = VK_SUBPASS_EXTERNAL;
  dependency.dstSubpass = 0;
  dependency.srcStageMask =
   VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT |
   VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT;
  dependency.srcAccessMask = 0;
  dependency.dstStageMask =
   VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT |
   VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT;
  dependency.dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT |
   VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
}
```

Now that we know how the attachments and subpasses are connected we can create the render pass. We will create a *VkRenderPass* variable in the header of the swapchain class. Then we will populate the *VkRenderPassCreateInfo* struct with the color and depth attachment, the subpass, the dependency and the attachment count. Let's first go ahead and create an array of attachments that contains the color and depth attachment. Then we add the sType and the other data to the render pass create info, create the render pass and throw an error if the creation was not successful.

```
//swapchain.hpp
private:
  ...
  VkRenderPass renderPass;
```

```
void SwapChain::createRenderPass() {
  ...
  std::array<VkAttachmentDescription, 2> attachments = {
   colorAttachment, depthAttachment};

  VkRenderPassCreateInfo renderPassInfo = {};
  renderPassInfo.sType = VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
  renderPassInfo.attachmentCount = static_cast<uint32_t>(attachments.
   size());
  renderPassInfo.pAttachments = attachments.data();
  renderPassInfo.subpassCount = 1;
  renderPassInfo.pSubpasses = &subpass;
  renderPassInfo.dependencyCount = 1;
  renderPassInfo.pDependencies = &dependency;

  if (vkCreateRenderPass(device, &renderPassInfo, nullptr, &renderPass
   ) != VK_SUCCESS)
    throw std::runtime_error("Failed to create render pass!");
}
```

Finally we have to make sure to destroy the render pass.

```
SwapChain::~SwapChain() {
  ...
  vkDestroyRenderPass(device, renderPass, nullptr);
}
```

4.3.8 Depth resources

Just like the color image we have to create an depth image and an depth image view. We will create a new function called *createDepthResources* that will be called after the creation of the render pass. We will also need a new variable in the header of the swapchain class that stores the depth images and the depth image views. We will need one depth image and one depth image view for each swapchain image. The problem is that when we create the depth image on the CPU we need to allocate memory on the GPU and then bind the image to that memory, because we need the data there. For that we will need a new function in the device class called *createImageWithInfo*. This function will take an image info struct, a memory property flag, a pointer to the image and a pointer to the image memory.

```
//swapchain.hpp
private:
  ...
  void createDepthResources();

  std::vector<VkImage> depthImages;
  std::vector<VkDeviceMemory> depthImageMemories;
  std::vector<VkImageView> depthImageViews;
```

```
SwapChain::SwapChain(Device &device, VkExtent2D extent) : device{
    device}, windowExtent{extent} {
  createSwapChain();
  createImageViews();
  createRenderPass();
  createDepthResources();
}

void SwapChain::createDepthResources() {
  depthImages.resize(swapChainImages.size());
  depthImageMemories.resize(swapChainImages.size());
  depthImageViews.resize(swapChainImages.size());

  VkFormat depthFormat = findDepthFormat();
  swapChainDepthFormat = depthFormat;

  for (size_t i = 0; i < swapChainImages.size(); i++) {
    VkImageCreateInfo imageInfo = {};
    imageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    imageInfo.imageType = VK_IMAGE_TYPE_2D;
    imageInfo.extent.width = swapChainExtent.width;
    imageInfo.extent.height = swapChainExtent.height;
    imageInfo.extent.depth = 1;
    imageInfo.mipLevels = 1;
    imageInfo.arrayLayers = 1;
    imageInfo.format = depthFormat;
    imageInfo.tiling = VK_IMAGE_TILING_OPTIMAL;
    imageInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
    imageInfo.usage = VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT;
    imageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
    imageInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
    imageInfo.flags = 0;
    device.createImageWithInfo(imageInfo,
   VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, depthImages[i],
   depthImageMemories[i]);

    VkImageViewCreateInfo viewInfo = {};
    viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    viewInfo.image = depthImages[i];
    viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
    viewInfo.format = depthFormat;
    viewInfo.subresourceRange.aspectMask = VK_IMAGE_ASPECT_DEPTH_BIT;
    viewInfo.subresourceRange.baseMipLevel = 0;
    viewInfo.subresourceRange.levelCount = 1;
    viewInfo.subresourceRange.baseArrayLayer = 0;
    viewInfo.subresourceRange.layerCount = 1;

    if (vkCreateImageView(device, &viewInfo, nullptr, &depthImageViews
   [i]) != VK_SUCCESS)
      throw std::runtime_error("Failed to create image views!");
  }
}
```

The *createImageWithInfo* function first creates the image with the *vkCreateImage* function. Then it gets the memory requirements of the image with the *vkGetImageMemoryRequirements* function. We will then populate the *VkMemoryAllocateInfo* struct with the sType, allocation size and the memory type index. We will get the memory type index with the *findMemoryType* function that we will implement next. After that we allocate the memory with the *vkAllocateMemory* function and bind the memory to the image with the *vkBindImageMemory* function.

```
//device.cpp
void Device::createImageWithInfo(const VkImageCreateInfo &imageInfo,
                                 VkMemoryPropertyFlags properties,
                                 VkImage &image,
                                 VkDeviceMemory &imageMemory) {
  if (vkCreateImage(device_, &imageInfo, nullptr, &image) !=
   VK_SUCCESS)
```

```
    throw std::runtime_error("Failed to create image!");

  VkMemoryRequirements memRequirements;
  vkGetImageMemoryRequirements(device_, image, &memRequirements);

  VkMemoryAllocateInfo allocInfo = {};
  allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
  allocInfo.allocationSize = memRequirements.size;
  allocInfo.memoryTypeIndex = findMemoryType(memRequirements.
   memoryTypeBits, properties);

  if (vkAllocateMemory(device_, &allocInfo, nullptr, &imageMemory) !=
   VK_SUCCESS)
    throw std::runtime_error("Failed to allocate image memory!");

  if (vkBindImageMemory(device_, image, imageMemory, 0) != VK_SUCCESS)
    throw std::runtime_error("Failed to bind image memory!");
}
```

The *findMemoryType* function will iterate through the memory types of the physical device and check if the properties are supported. If it is it will return the memory type index.

```
//device.cpp
uint32_t Device::findMemoryType(uint32_t typeFilter,
   VkMemoryPropertyFlags properties) {
  VkPhysicalDeviceMemoryProperties memProperties;
  vkGetPhysicalDeviceMemoryProperties(physicalDevice, &memProperties);

  for (uint32_t i = 0; i < memProperties.memoryTypeCount; i++) {
    if ((typeFilter & (1 << i)) && (memProperties.memoryTypes[i].
   propertyFlags & properties) == properties)
      return i;
  }

  throw std::runtime_error("Failed to find suitable memory type!");
}
```

In the if statement we first filter the types by going through the bits of the type filter. Then we check if the properties are supported by the memory type. If it is we return the memory type index.

Now that we have the depth resources we have to destroy them and clear the GPU memory in the destructor of the swapchain class.

```
SwapChain::~SwapChain() {
  ...
  for (size_t i = 0; i < depthImages.size(); i++) {
      vkDestroyImageView(device, depthImageViews[i], nullptr);
      vkDestroyImage(device, depthImages[i], nullptr);
      vkFreeMemory(device, depthImageMemories[i], nullptr);
    }
}
```

4.3.9 Framebuffers

Now that we have all our resources we can create the framebuffers. A framebuffer consists of the attachments and depth resources that are used in the render pass. We will create a new function called *createFramebuffers* that will be called after the creation of the depth resources. We will also need a new variable in the header of the swapchain class that stores the framebuffers.

```
//swapchain.hpp
private:
  ...
  void createFramebuffers();

  std::vector<VkFramebuffer> framebuffers;
```

To create the framebuffers we will iterate through the image views and create a framebuffer for each image view. The create information for the framebuffer takes the render pass, the amount of attachments, the attachments, the width and the height of the framebuffer and the layers count the framebuffer is used for. We will set the layers count to 1 and the width and height to the width and height of the swapchain extent.

```
SwapChain::SwapChain(Device &device, VkExtent2D extent) : device{
   device}, windowExtent{extent} {
  createSwapChain();
  createImageViews();
  createRenderPass();
  createDepthResources();
  createFramebuffers();
}

void SwapChain::createFramebuffers() {
  framebuffers.resize(swapChainImages.size());

  for (size_t i = 0; i < swapChainImages.size(); i++) {
    std::array<VkImageView, 2> attachments = {
      swapChainImageViews[i],
      depthImageViews[i]
    };

    VkFramebufferCreateInfo framebufferInfo = {};
    framebufferInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
    framebufferInfo.renderPass = renderPass;
    framebufferInfo.attachmentCount = static_cast<uint32_t>(
   attachments.size());
    framebufferInfo.pAttachments = attachments.data();
    framebufferInfo.width = swapChainExtent.width;
    framebufferInfo.height = swapChainExtent.height;
    framebufferInfo.layers = 1;

    if (vkCreateFramebuffer(device, &framebufferInfo, nullptr, &
   framebuffers[i]) != VK_SUCCESS)
      throw std::runtime_error("Failed to create framebuffer!");
  }
}
```

After that we have to destroy the framebuffers in the destructor of the swapchain class.

```
SwapChain::~SwapChain() {
  ...
  for (auto framebuffer : framebuffers) {
    vkDestroyFramebuffer(device, framebuffer, nullptr);
  }
}
```

That's it! We now have all the objects that we need to render something. Let's shortly go through the steps that we have done so far.

We began by creating the swapchain, which created the images and defined how the images are presented to the surface. Then we defined how the images are viewed with the image views. After that we created a render pass, that stores the color and depth attachments and ensures that the subpasses are not out of sync. We then created the depth resources that include the depth image and the depth image view. Finally we created the framebuffers that store the attachments and depth resources that are used in the render pass.

4.4 Pipeline

We can now start with the pipeline creation. Let's go ahead and create a new class called *Pipeline* that will contain the pipeline creation. We will need a new function called *createGraphicsPipeline* that will be called in the constructor. This function takes the path to the vertex and fragment shader and a PipelineConfigInfo struct that we will define in the header of the pipeline class. We will also need a variable that stores the graphics pipeline, the device and the shader modules.

```
//pipeline.hpp
struct PipelineConfigInfo {
  PipelineConfigInfo() = default;
  PipelineConfigInfo(const PipelineConfigInfo &) = delete;
  PipelineConfigInfo &operator=(const PipelineConfigInfo &) = delete;

  VkPipelineViewportStateCreateInfo viewportInfo;
  VkPipelineInputAssemblyStateCreateInfo inputAssemblyInfo;
  VkPipelineRasterizationStateCreateInfo rasterizationInfo;
  VkPipelineMultisampleStateCreateInfo multisampleInfo;
  VkPipelineColorBlendAttachmentState colorBlendAttachment;
  VkPipelineColorBlendStateCreateInfo colorBlendInfo;
  VkPipelineDepthStencilStateCreateInfo depthStencilInfo;
  VkPipelineDynamicStateCreateInfo dynamicStateInfo;
  VkPipelineLayout pipelineLayout = nullptr;
  VkRenderPass renderPass = nullptr;
  uint32_t subpass = 0;
};

class Pipeline {
public:
  Pipeline(Device &device, const std::string &vertPath, const std::
   string &fragPath, const PipelineConfig &configInfo);
  ~Pipeline();

  Pipeline(const Pipeline &) = delete;
  Pipeline operator=(const Pipeline &) = delete;

private:
  Device &device;
  VkPipeline graphicsPipeline;
  VkShaderModule vertShaderModule;
  VkShaderModule fragShaderModule;
};
```

The PipelineConfigInfo struct contains all the information that we need to create the pipeline, like the create informations for the states, the render pass, the subpass and the pipeline layout.

The constructor of the pipeline class will call the *createGraphicsPipeline* function and set our device variable.

In the *createGraphicsPipeline* function we will first check if the config info has a valid pipeline layout and render pass. Then we will have to read the shader files and create the shader modules. Let's start off by adding a helper function called *readFile* that reads the shader files and returns the content as a vector of chars.

```
//pipeline.cpp
std::vector<char> Pipeline::readFile(const std::string &path) {
  std::ifstream file{path, std::ios::ate | std::ios::binary};

  if (!file.is_open())
    throw std::runtime_error("Failed to open file: " + path);

  size_t fileSize = static_cast<size_t>(file.tellg());
  std::vector<char> buffer(fileSize);

  file.seekg(0);
  file.read(buffer.data(), fileSize);

  file.close();
  return buffer;
}
```

This function opens a file. The *std::ios::ate* flag sets the position of the file pointer to the end of the file and the *std::ios::binary* flag reads the file as binary. We have binary files because we have to compile our shaders to SPIR-V. We go to the end of the file at the beginning because we want to create a buffer with the size of the file. Then we read the file into the buffer and return the buffer.

We will do this for both the vertex and the fragment shader. We will then create the shader modules with another helper function. The *createShaderModule* function takes a vector of chars and a pointer to the shader module variable.

```
//pipeline.cpp
Pipeline::Pipeline(Device &device, const std::string &vertPath, const
    std::string &fragPath, const PipelineConfigInfo &configInfo)
: device{device} {
    createGraphicsPipeline(vertPath, fragPath, configInfo);
  }

void Pipeline::createGraphicsPipeline(const std::string &vertPath,
    const std::string &fragPath, const PipelineConfigInfo &configInfo)
    {
  assert(configInfo.pipelineLayout != VK_NULL_HANDLE && "Cannot create
    graphics pipeline. Invalid pipeline layout!");
  assert(configInfo.renderPass != VK_NULL_HANDLE && "Cannot create
    graphics pipeline. Invalid render pass!");

  std::vector<char> vertCode = readFile(vertPath);
  std::vector<char> fragCode = readFile(fragPath);

  createShaderModule(vertCode, &vertShaderModule);
  createShaderModule(fragCode, &fragShaderModule);
}

void Pipeline::createShaderModule(const std::vector<char> &code,
    VkShaderModule *shaderModule) {
  VkShaderModuleCreateInfo createInfo = {};
  createInfo.sType = VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
  createInfo.codeSize = code.size();
  createInfo.pCode = reinterpret_cast<const uint32_t *>(code.data());

  if (vkCreateShaderModule(device, &createInfo, nullptr, shaderModule)
      != VK_SUCCESS)
    throw std::runtime_error("Failed to create shader module!");
}
```

The *createShaderModule* function creates a shader module with the *vkCreateShader-Module* function. The *VkShaderModuleCreateInfo* struct contains the size of the code and the code itself. We have to cast the code to a *const uint32_t* pointer because the code is a vector of chars but is actually binary data. This binary data is what the *vkCreateShaderModule* function expects.

4.4.1 Shader Stages

Okay before we go further into the pipeline creation we have to actually write and compile the shaders. We will use the GLSL language for the shaders. Let's create a new folder in our src folder called shaders. In this folder we will create a *shader.vert* and a *shader.frag* file.

We will start off with a simple vertex shaders that basically just pass the vertex position and the color to the fragment shader. GLSL looks very similar to C++ but has some differences. For example we have to define the version of the GLSL language at the beginning of the shader. We will use version 460. Next we have a layout in qualifier that defines the location of the data that is passed to the shader. We will use location 0 for the vertex position and location 1 for the color. Then we have a void main function in which we will pass the color to the fragment shader. We achive this by adding a layout out qualifier that will pass the data to the next stage. Why only the color? Because the fragment shader will only calculate the color of the pixel and has no need for the vertex position.

For the fragment shader we will also define the version at the beginning. We will then have a layout in qualifier that takes the previous data from the vertex shader and a layout out qualifier that passes the color to the next stage. We will then have a void main function that will set the out color to the color that we passed from the vertex shader.

```
//shader.vert
#version 460
layout(location = 0) in vec3 inPosition;
layout(location = 1) in vec3 inColor;

layout(location = 0) out vec3 fragColor;

void main() {
  gl_Position = vec4(inPosition, 1.0);
  fragColor = inColor;
}
```

You can see that we have this mysterious *gl_Position* variable. This is a built-in variable that takes the vertex position as a 4D vector. The 4th component is the w component that is used for perspective division. This *gl_Position* variable is a predefined out variable used to calculate the position of the vertex on the screen in later stages of the pipeline.

```
//shader.frag
#version 460
layout(location = 0) in vec3 fragColor;

layout(location = 0) out vec4 outColor;

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

Here we set the *outColor* to the color that we passed from the vertex shader. The *outColor* is a 4D vector that represents the color of the pixel. The 4th component is the alpha value of the color.

Now that we have them we have to compile them to SPIR-V. We will simply add a compile.sh script that compiles the shaders. We will use the *glslc* compiler that comes with the Vulkan SDK. The *glslc* compiler takes the shader file and then compiles them to a specified output file. We will compile the vertex shader to *shader.vert.spv* and the fragment shader to *shader.frag.spv*.

```
/usr/local/bin/glslc src/shaders/shader.vert -o src/shaders/shader.
    vert.spv
/usr/local/bin/glslc src/shaders/shader.frag -o src/shaders/shader.
    frag.spv
```

We will execute this script in our makefile before we compile the vulkan application. Simply add the following line to the makefile, before the compile line.

sh src/shaders/compile.sh

Perfect! Now we have the shaders that the pipeline will read and use. Later you might add more shaders for more complex rendering, but the vertex and fragment shader are mandatory.

We now have to create the actual stages and will begin with the shader stages. If you're wondering what the difference between a shader module and a shader stage is, the shader module is the binary data that is loaded into the GPU and the shader stage is the stage in the pipeline that the shader module is used in. Let's go ahead and create the shader stages.

We will use the *VkPipelineShaderStageCreateInfo* struct to create the shader stages. This struct contains the sType, the type of stage, the name and the shader module.

```
//pipeline.cpp
void Pipeline::createGraphicsPipeline(const std::string &vertPath,
   const std::string &fragPath, const PipelineConfigInfo &configInfo)
   {
  ...
  VkPipelineShaderStageCreateInfo shaderStageInfos[2];
  shaderStageInfos[0].sType =
   VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
  shaderStageInfos[0].stage = VK_SHADER_STAGE_VERTEX_BIT;
  shaderStageInfos[0].module = vertShaderModule;
  shaderStageInfos[0].pName = "main";

  shaderStageInfos[1].sType =
   VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO;
  shaderStageInfos[1].stage = VK_SHADER_STAGE_FRAGMENT_BIT;
  shaderStageInfos[1].module = fragShaderModule;
  shaderStageInfos[1].pName = "main";
}
```

4.4.2 Vertex Input

Next we have to define the format of the vertex data that is passed to the vertex shader. To do so we have to create a *VkPipelineVertexInputStateCreateInfo* struct that contains the sType, the vertex binding description and the vertex attribute descriptions. We'll come back to this once we have the vertex and index buffers. For now we will leave it empty, because we have to get the vertex data from models that we will load later.

Because we have the vertex input state info in the config info struct we will use that to create the vertex input state info.

```
void Pipeline::createGraphicsPipeline(const std::string &vertFilepath,
                                      const std::string &fragFilepath,
                                      const PipelineConfigInfo &
   configInfo) {
  ...
  VkPipelineVertexInputStateCreateInfo vertexInputInfo = {};
  vertexInputInfo.sType =
   VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO;
  vertexInputInfo.vertexBindingDescriptionCount = 0;
  vertexInputInfo.pVertexBindingDescriptions = nullptr;
  vertexInputInfo.vertexAttributeDescriptionCount = 0;
  vertexInputInfo.pVertexAttributeDescriptions = nullptr;
}
```

4.4.3 Dynamic States

While most of the stages are baked into an immutable pipeline object, there are some states that can be changed without recreating the pipeline. These states are called dynamic states. We will use dynamic states for the viewport and the scissor.

The viewport is the region of the framebuffer, that will be rendered to. Or in other words, it describes what proportions the image will be displayed to the framebuffer. For example you might have a FullHD screen and use that width and height for the swapchain. That means you will render in FullHD. Despite that you can define the viewport with half the height, squishing the full image, making the image fit in the defined viewport proportions.

The viewport defines the top-left corner with it's x and y coordinate, the width and height and it's min and max depth.

The scissor on the other hand cuts the image off. It defines a rectangle, again with an offset and an extent, that defines the part of the visible Image.

We will not need any fancy transformations now, but you might want to change the viewport extent when creating cutscenes that have the well known black bars called letterboxes. Or you might use scissors to create game effects where the visible screen gets smaller and smaller over time.

The pipeline config info struct contains a vector of dynamic states that we will use to create the dynamic state info struct. Let's go ahead and add a public static function to the pipeline class called *defaultPipelineConfig* that will populate a PipelineConfigInfo struct with default values.

```
void Pipeline::defaultPipelineConfig(PipelineConfigInfo &configInfo) {
  std::vector<VkDynamicState> dynamicStates = {
    VK_DYNAMIC_STATE_VIEWPORT,
    VK_DYNAMIC_STATE_SCISSOR
  };

  configInfo.dynamicStateInfo.sType =
   VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
  configInfo.dynamicStateInfo.dynamicStateCount = static_cast<uint32_t
   >(dynamicStates.size());
  configInfo.dynamicStateInfo.pDynamicStates = dynamicStates.data();
}
```

Because we have the viewport and scissor set to dynamic we only need to specify the viewport and scissor count in the *VkPipelineViewportStateCreateInfo* struct. We will set the viewport and scissor count to 1.

```
void Pipeline::defaultPipelineConfig(PipelineConfigInfo &configInfo) {
    ...
    configInfo.viewportInfo.sType =
   VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO;
    configInfo.viewportInfo.viewportCount = 1;
    configInfo.viewportInfo.scissorCount = 1;
  }
```

We will create the actual viewport and scissor later at drawing time. If you don't want to use dynamic states you have to create it first and then reference it in the viewport info struct.

4.4.4 Input Assembly

Now we will define the input assembly state. The input assembly state defines how the vertices are assembled into primitives. We will use the *VK_PRIMITIVE_TOPOLOGY_TRIANGLE* topology for now. This means that every 3 vertices that we pass to the pipeline will be assembled into one triangle. There is another member in the input assembly state struct called the primitive restart enable. This is used to break up lines that are created with any strip topology. We don't need this for now, so we will set it to VK_FALSE.

```
void Pipeline::defaultPipelineConfig(PipelineConfigInfo &configInfo) {
    ...
    configInfo.inputAssemblyInfo.sType =
   VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO;
    configInfo.inputAssemblyInfo.topology =
   VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
    configInfo.inputAssemblyInfo.primitiveRestartEnable = VK_FALSE;
  }
```

4.4.5 Rasterizer

Next we have to setup the rasterizer. The rasterizer takes the geometry that is outputted by the vertex shader. Or if you use the tessellation stage the tessellation stage and the geometry shader. The rasterizer then turns the geometry into fragments. It also performs depth testing, face culling and the scissor test. Let's go over the rasterizer state members.

The depth clamp enable is used to clamp the depth values of the fragments that are outside of the near and far planes. This means everything that is closer than the near plane will be clamped to the near plane and everything that is further than the far plane will be clamped to the far plane. This is useful for shadow mapping, but we don't need it for now.

The rasterizer discard enable is used to discard all the fragments that are outputted by the rasterizer. This might be useful when you don't need to output any fragments, but we don't need it for now.

The polygon mode defines how the rasterizer fills the polygons. We will use the fill mode for now, but you can also use the line or point mode. The line mode will draw the edges of the polygons and the point mode will draw the vertices of the polygons. Any mode other than the fill mode will need you to activate a GPU feature like the *VK_EXT_line_rasterization* feature.

The line width is self explanatory. It defines the width of the lines in fragments that are drawn. Setting this value to 1.0 will draw the lines with the width of one pixel and anything greater than 1.0 will need us to activate the wideLines feature.

The cull mode defines which faces of the polygons are culled. Each surface has two sides, the front and the back side. By cutting away aka not rendering the back side of the polygons we can save some performance. We will cull the back side of the polygons for now.

The front face defines which side of the polygons is the front side. We determine the front side by looking at the vertices of the polygon. When we set the front face to clockwise the front side is the side, where the vertices are ordered clockwise.

The last member is the depth bias. We can choose to enable it and set a constant factor to our depth value, a slope factor and clamp the depth bias. This is not something that we will use for now so we will disable it.

```
void Pipeline::defaultPipelineConfig(PipelineConfigInfo &configInfo) {
    ...
    configInfo.rasterizationInfo.sType =
   VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO;
    configInfo.rasterizationInfo.depthClampEnable = VK_FALSE;
    configInfo.rasterizationInfo.rasterizerDiscardEnable = VK_FALSE;
    configInfo.rasterizationInfo.polygonMode = VK_POLYGON_MODE_FILL;
    configInfo.rasterizationInfo.lineWidth = 1.0f;
    configInfo.rasterizationInfo.cullMode = VK_CULL_MODE_BACK_BIT;
    configInfo.rasterizationInfo.frontFace = VK_FRONT_FACE_CLOCKWISE;
    configInfo.rasterizationInfo.depthBiasEnable = VK_FALSE;
  }
```

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] 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).

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

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

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

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

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

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

[18] 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).

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

[20] 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).

[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).

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

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

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

[27] 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).

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

[29] 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).

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

[31] 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).

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

[33] 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).

[34] 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).

[35] 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).

[36] GLFW: *GLFW*. 2024. URL: https://www.glfw.org/ (visited on 07/23/2024).

[37] GLFW: *Window Guide*. 2024. URL: https://www.glfw.org/docs/3.3/window_guide.html#GLFW_CLIENT_API_hint (visited on 07/24/2024).

[38] ViolentRumble: *Battlefield 4 PC Gameplay Screen Tearing*. 2013. URL: https://i.ytimg.com/vi/jVAFuUAKPMc/maxresdefault.jpg (visited on 08/08/2024).

[39] R. F. Tagaro: *Anti-Aliasing Problem and Mipmapping*. URL: https://texturingraphics.wordpress.com/what-is-texture-mapping/anti-aliasing-problem-and-mipmapping/ (visited on 08/20/2024).

4.4.6 Multisampling

Multi sampling is a technique that is used to reduce aliasing. Aliasing is the stair stepping effect that you see on the edges of objects. To avoid this we can take multiple samples inside each fragment and then average the color of the samples. This will make the edges of the objects look smoother. We will use 1 sample for now and turn of sample shading. Later when rendering more complex scenes we will revisit this.

```
void Pipeline::defaultPipelineConfig(PipelineConfigInfo &configInfo) {
    ...
    configInfo.multisampleInfo.sType =
   VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
    configInfo.multisampleInfo.sampleShadingEnable = VK_FALSE;
    configInfo.multisampleInfo.rasterizationSamples =
   VK_SAMPLE_COUNT_1_BIT;
  }
```

4.4.7 Depth and Stencil Testing

When using depth testing we can discard fragments that are behind other fragments. To do that we have to populate the *VkPipelineDepthStencilStateCreateInfo* struct. The depth test enable is used to enable the depth test. The depth write enable is used to enable writing to the depth buffer. The depth compare op is used to compare the depth of the fragments. We will check if the depth of the fragment is less than the depth of the fragment that is already in the depth buffer. If it is we will write the fragment to the depth buffer. We can also set the depth bounds test enable to enable the depth bounds test. This is used to discard fragments that are outside of the depth bounds. We will not need this so we can set it to VK_FALSE and ignore the depth bound members. Finally we can also enable the stencil test, which we will keep disabled.

```
void Pipeline::defaultPipelineConfig(PipelineConfigInfo &configInfo) {
    ...
    configInfo.depthStencilInfo.sType =
   VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
    configInfo.depthStencilInfo.depthTestEnable = VK_TRUE;
    configInfo.depthStencilInfo.depthWriteEnable = VK_TRUE;
    configInfo.depthStencilInfo.depthCompareOp = VK_COMPARE_OP_LESS;
    configInfo.depthStencilInfo.depthBoundsTestEnable = VK_FALSE;
    configInfo.depthStencilInfo.stencilTestEnable = VK_FALSE;
  }
```

4.4.8 Color Blending

After the fragment colors are calculated by the fragment shader, we have to determine how they are combined with the colors that are already in the framebuffer. There are 2 different structures that we have to populate. The first one is a per framebuffer struct called *VkPipelineColorBlendAttachmentState*. This struct contains the color write mask and the blend enable. If we would enable blending we could set a blend factor for color and alpha values and set an mathematical operation to combine the colors. We just want the fragment color to be written to the framebuffer so we will disable blending and set the color write mask to the RGBA bits.

The second struct is the *VkPipelineColorBlendStateCreateInfo* struct. This struct stores the color blend attachments and defines if and what kind of logical operation is used. We will not apply any logical operations so we will disable it. There are many logical operations that you can use, like the copy operation that just copies the source color to the destination color, or the and operation that takes the bitwise and of the source and destination color. By disabling the logical operation we will just copy the source color to the destination color, so we don't have to set the logical operation. Optionally we can set the blend constants that are used in blending operations. We will not need this so we will not set it.

```
void Pipeline::defaultPipelineConfig(PipelineConfigInfo &configInfo) {
    ...
    configInfo.colorBlendAttachment.colorWriteMask =
   VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_G_BIT |
   VK_COLOR_COMPONENT_B_BIT | VK_COLOR_COMPONENT_A_BIT;
    configInfo.colorBlendAttachment.blendEnable = VK_FALSE;

    configInfo.colorBlendInfo.sType =
   VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
    configInfo.colorBlendInfo.logicOpEnable = VK_FALSE;
    configInfo.colorBlendInfo.attachmentCount = 1;
    configInfo.colorBlendInfo.pAttachments = &configInfo.
   colorBlendAttachment;
  }
```

4.4.9 Pipeline layout and renderering system

Now that we have all states set up that we need for the pipeline we can create the pipeline layout. The pipeline layout specifies the uniform variables, like descriptor sets, that are used in the shaders. These uniform variables, stored in the member variables *pSetLayouts* and *setLayoutCount*, are later used to pass data like the transformation matrix to the shaders. It also specifies the push constants that are used to pass smaller bits of data to the shaders. We won't use uniform variables or push constants for now, so we will just have to set the sType.

The thing is that we have to create the pipeline layout before we create the graphics pipeline. So we will add a new class called *RenderSystem*. This class will do the following:

- Create the pipeline layout

- Populate the pipeline config info struct
- Create the graphics pipeline

We will need two functions in the render system class. The first function will be the *createPipelineLayout* function and the second function will be the *createPipeline* function. The *createPipelineLayout* function will take a descriptor set layout and create the pipeline layout. The *createPipeline* function will take the render pass and create the graphics pipeline. We will also need a variable that stores the pipeline layout, the pipeline and the device. The render system's constructor takes the device reference, render pass and the descriptor set layout and first creates the pipeline layout and then the graphics pipeline.

```
//render_system.hpp
class RenderSystem {
public:
  RenderSystem(Device &device, VkRenderPass renderPass,
   VkDescriptorSetLayout descriptorSetLayout);
  ~RenderSystem();

  RenderSystem(const RenderSystem &) = delete;
  RenderSystem operator=(const RenderSystem &) = delete;

private:
  Device &device;
  VkPipelineLayout pipelineLayout;
  std::unique_ptr<Pipeline> pipeline;

  void createPipelineLayout(VkDescriptorSetLayout descriptorSetLayout)
   ;
  void createPipeline(VkRenderPass renderPass);
};
```

For the *VkPipelineLayoutCreateInfo* struct we will set the sType and the descriptor set layout and ignore the push constant range members for now. For the case that we won't have a descriptor set layout, we will add the *descriptorSetLayout* parameter to a new vector. Doing it like this we can simply determine the size and the data of the vector.

```
//render_system.cpp
RenderSystem::RenderSystem(Device &device, VkRenderPass renderPass,
    VkDescriptorSetLayout descriptorSetLayout) : device{device} {
  createPipelineLayout(descriptorSetLayout);
  createPipeline(renderPass);
}

void RenderSystem::createPipelineLayout(VkDescriptorSetLayout
    descriptorSetLayout) {
  std::vector<VkDescriptorSetLayout> layouts{descriptorSetLayout};

  VkPipelineLayoutCreateInfo pipelineLayoutInfo = {};
  pipelineLayoutInfo.sType =
    VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
  pipelineLayoutInfo.setLayoutCount = static_cast<uint32_t>(layouts.
    size());
  pipelineLayoutInfo.pSetLayouts = layouts.data();

  if (vkCreatePipelineLayout(device, &pipelineLayoutInfo, nullptr, &
    pipelineLayout) != VK_SUCCESS)
     throw std::runtime_error("Failed to create pipeline layout!");
}
```

The *createPipeline* function will populate the pipeline config info struct with the default pipeline config and then instantiate our pipeline class with the vertex and fragment shader paths and the pipeline config info struct. Before we do that, we have to make sure that we have a valid pipeline layout.

```
void RenderSystem::createPipeline(VkRenderPass renderPass) {
  assert(pipelineLayout != VK_NULL_HANDLE && "Cannot create pipeline
    before pipeline layout!");

  PipelineConfigInfo pipelineConfigInfo = {};
  Pipeline::defaultPipelineConfig(pipelineConfigInfo);
  pipelineConfigInfo.renderPass = renderPass;
  pipelineConfigInfo.pipelineLayout = pipelineLayout;

  pipeline = std::make_unique<Pipeline>(device, "src/shaders/shader.
    vert.spv", "src/shaders/shader.frag.spv", pipelineConfigInfo);
}
```

Finally we need to destory the pipeline layout in the destructor of the render system class.

```
RenderSystem::~RenderSystem() {
  vkDestroyPipelineLayout(device, pipelineLayout, nullptr);
}
```

4.4.10 Pipeline creation

Now that we get the render pass and pipeline layout from the render system we can create the graphics pipeline. We will simply populate the *VkGraphicsPipelineCreateInfo* struct with the data that we have and create the graphics pipeline with the *vkCreateGraphicsPipelines* function.

```
void Pipeline::createGraphicsPipeline(const std::string &vertPath,
    const std::string &fragPath, const PipelineConfigInfo &configInfo)
    {
  ...
  VkGraphicsPipelineCreateInfo pipelineInfo = {};
  pipelineInfo.sType = VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO
    ;
  pipelineInfo.stageCount = 2;
  pipelineInfo.pStages = shaderStages;
  pipelineInfo.pVertexInputState = &vertexInputInfo;
  pipelineInfo.pInputAssemblyState = &configInfo.inputAssemblyInfo;
  pipelineInfo.pViewportState = &configInfo.viewportInfo;
  pipelineInfo.pRasterizationState = &configInfo.rasterizationInfo;
  pipelineInfo.pMultisampleState = &configInfo.multisampleInfo;
  pipelineInfo.pDepthStencilState = &configInfo.depthStencilInfo;
  pipelineInfo.pColorBlendState = &configInfo.colorBlendInfo;
  pipelineInfo.pDynamicState = &configInfo.dynamicStateInfo;

  pipelineInfo.layout = configInfo.pipelineLayout;
  pipelineInfo.renderPass = configInfo.renderPass;
  pipelineInfo.subpass = configInfo.subpass;

  pipelineInfo.basePipelineHandle = VK_NULL_HANDLE;
  pipelineInfo.basePipelineIndex = -1;

  if (vkCreateGraphicsPipelines(device, VK_NULL_HANDLE, 1, &
   pipelineInfo, nullptr, &graphicsPipeline) != VK_SUCCESS)
    throw std::runtime_error("Failed to create graphics pipeline!");
}
```

You can see that we also set the base pipeline handle and index to *VK_NULL_HANDLE* and -1. This is used for creating a new pipeline that is based on an existing pipeline. We will not use this for now.

And again we have to clean up our mess. Including the shader modules and the graphics pipeline in the pipeline destructor.

```
Pipeline::~Pipeline() {
  vkDestroyShaderModule(device, vertShaderModule, nullptr);
  vkDestroyShaderModule(device, fragShaderModule, nullptr);
  vkDestroyPipeline(device, graphicsPipeline, nullptr);
}
```

That's it for now, let's go ahead and instantiate the render system in our application class. We will also need to add a getter function to the swapchain class that returns the render pass.

```
//swapchain.hpp
VkRenderPass getRenderPass() { return renderPass; }

//app.cpp
App::App() {
  RenderSystem renderSystem{device, swapChain.getRenderPass(), nullptr
   };
  while (!window.shouldClose()) {
    glfwPollEvents();
  }
}
```

List of Figures