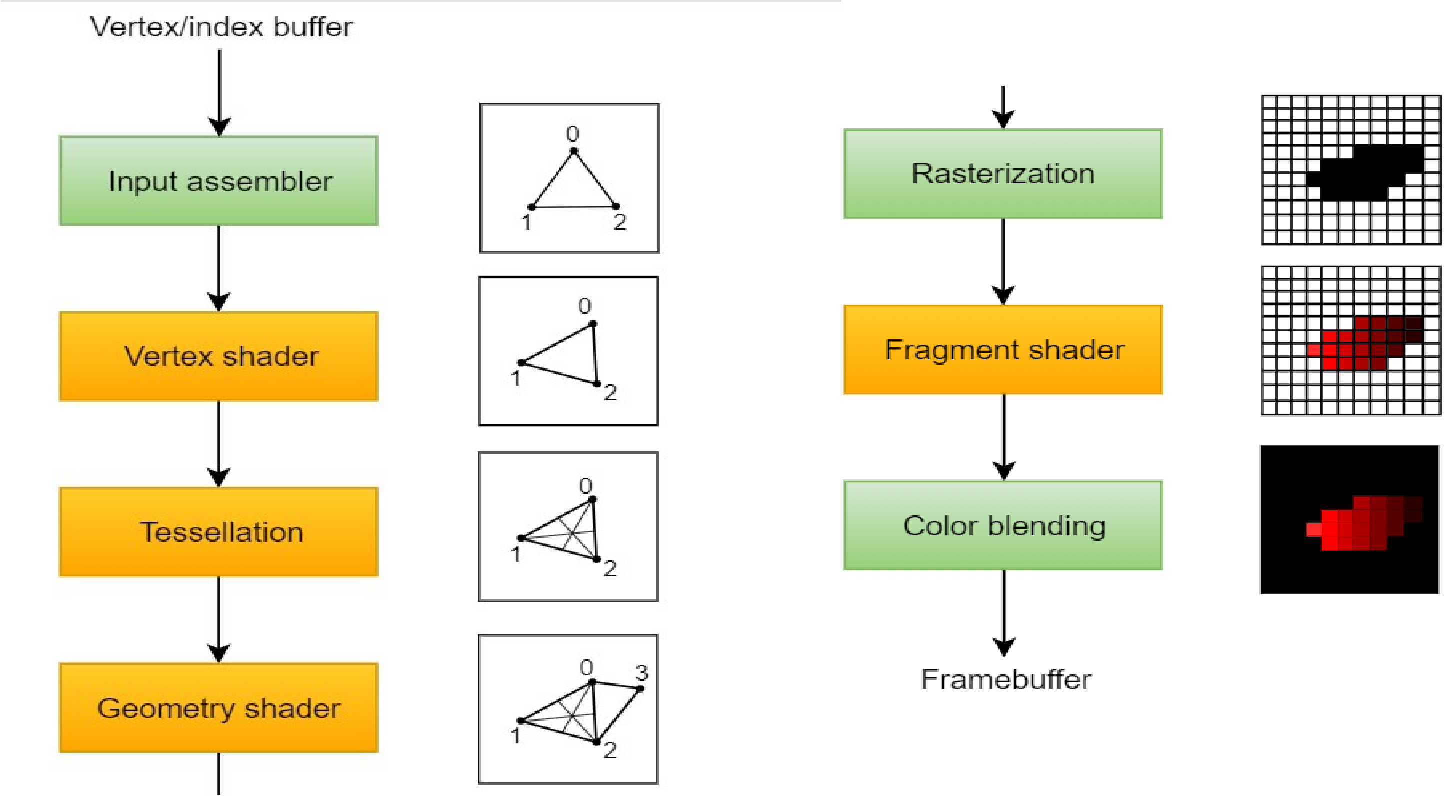
# Graphics Pipeline

Setting up shaders and the Graphics Pipeline

# Vulkan Graphics Pipeline

- The Graphics Pipeline in Vulkan is largely the same as that in OpenGL.
- However, in Vulkan, you must set up each stage individually.
- There is *a lot* of code to set up the Pipeline!
- ...but it's mostly just applying setting to info structs. Nothing complicated!
- Each Pipeline can connect to a "Render Pass" which actually handles rendering. This is the difficult part!

# Vulkan Graphics Pipeline



Vertex/index buffer

Input assembler

Vertex shader

Tessellation

Geometry shader

Rasterization

Fragment shader

Color blending

Framebuffer

# Shaders and SPIR-V

- Don’t load in raw text and compile it into a shader.
- Instead, pre-compile shader code to intermediate code called SPIR-V, and load it into a shader module.
- SPIR-V stands for “Standard Portable Intermediate Representation – V (Vulkan)”.
- Compiled from GLSL
- SPIR-V can be compiled using tool that comes with LunarG Vulkan SDK: glslangValidator.exe

# Shaders and SPIR-V

- To use: Load in .spv (SPIR-V file) as binary file and create shader module using Vulkan.
- Shader module then passed to info struct.
- Put all shader info structs into list and use in Pipeline Create Info.

# Creating Graphics Pipeline

- **Creating Graphics Pipeline is simple but lengthy. Must define settings for each stage manually:**
  - **Vertex Input:** Defines layout and format of vertex input data.
  - **Input Assembly:** Defines how to assemble vertices to primitives (e.g. Triangles or Lines).
  - **Viewport & Scissor:** How to fit output to image and crop it.
  - **Dynamic States:** Pipelines are static and settings can't be changed at runtime. You need to create a new pipeline to get new settings. However, some settings can be given ability to change at runtime, and they can be set here.
  - **Rasterizer:** How to handle computation of fragments from primitives.
  - **Multisampling:** Multisampling information.
  - **Blending:** How to blend fragments at the end of the pipeline.
  - **Depth Stencil:** How to determine depth + stencil culling and writing.

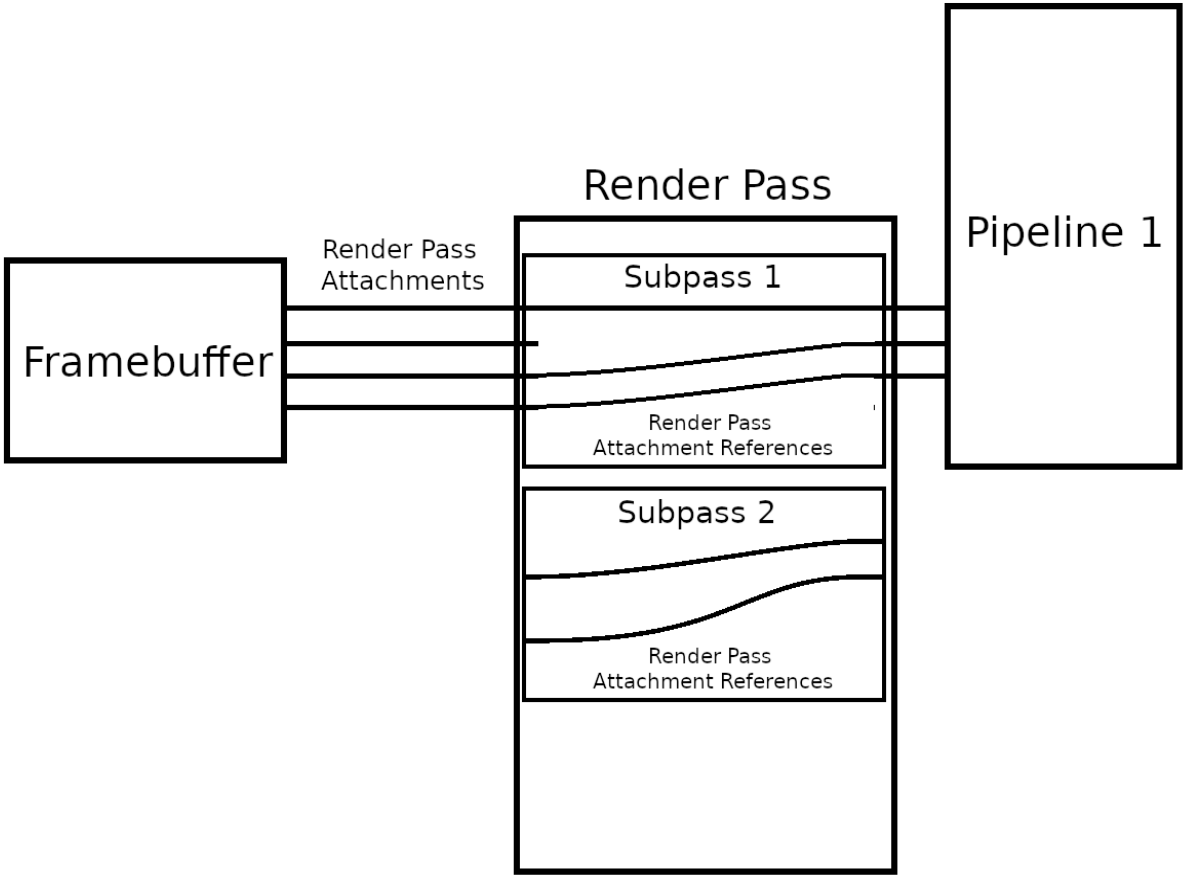
# Pipeline Layout

- Pipeline Layout is the layout of data being given directly to the pipeline for a single draw operation (as opposed to for each vertex/fragment).
- Defines layout of data for “Descriptor Sets”. We’ll learn about these later. They’re similar to uniform buffers in OpenGL.
- Also defines range of “Push Constants”. These are like simpler, smaller Descriptor Sets that pass values directly instead of holding them in dedicated memory.
- For now, we will leave these blank and return to them in a later lesson.

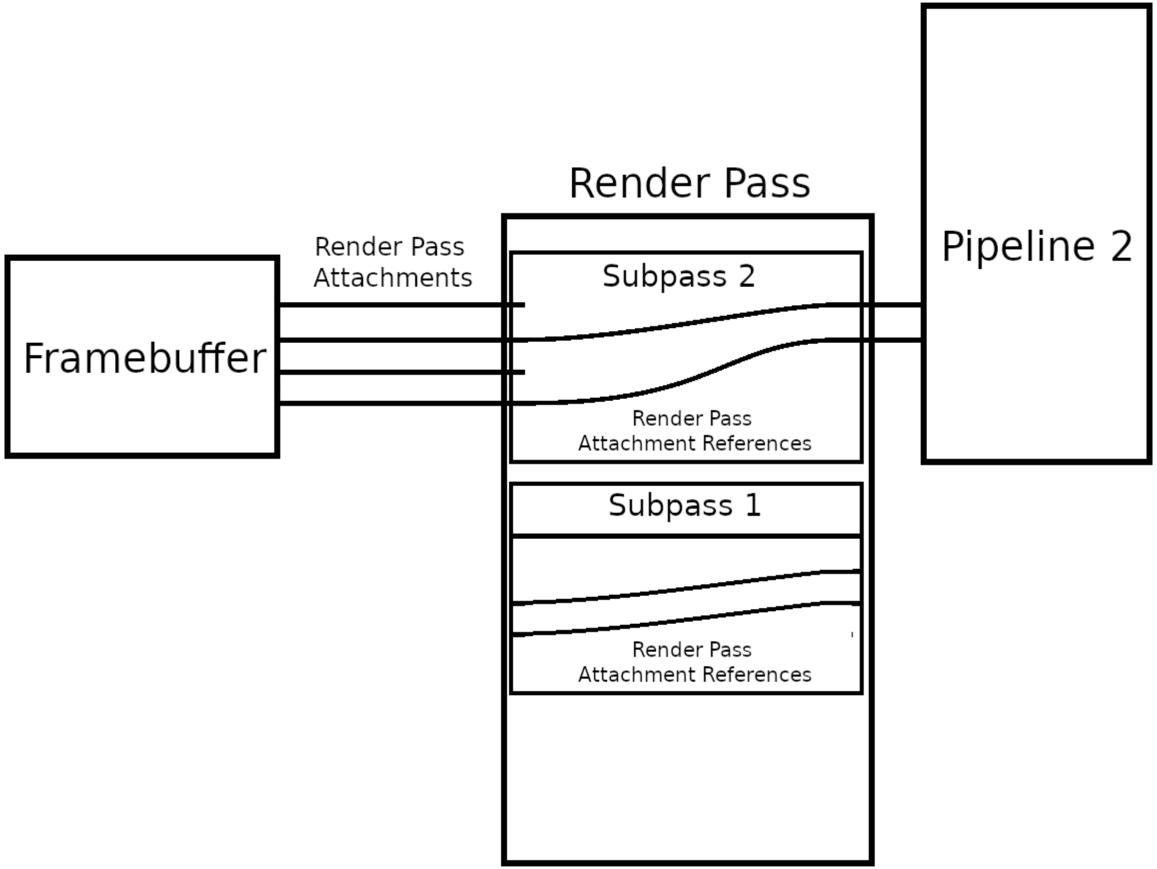
# Render Pass

- Lastly, a Pipeline needs a “Render Pass”.
- A Render Pass is the larger operation that handles the execution and outputs of a Pipeline.
- Can have multiple smaller Subpasses inside it that each use a different Pipeline, so you can combine draws together.
- Render Passes contain multiple “Attachments” to all of the possible outputs (e.g. Colour output, Depth output…)
- Subpasses inside a Render Pass then reference these attachments for each draw operation.

# Render Pass


Render Pass
Pipeline 1
Render Pass
Attachments
Subpass 1
Framebuffer
Render Pass
Attachment References
Subpass 2
Render Pass
Attachment References

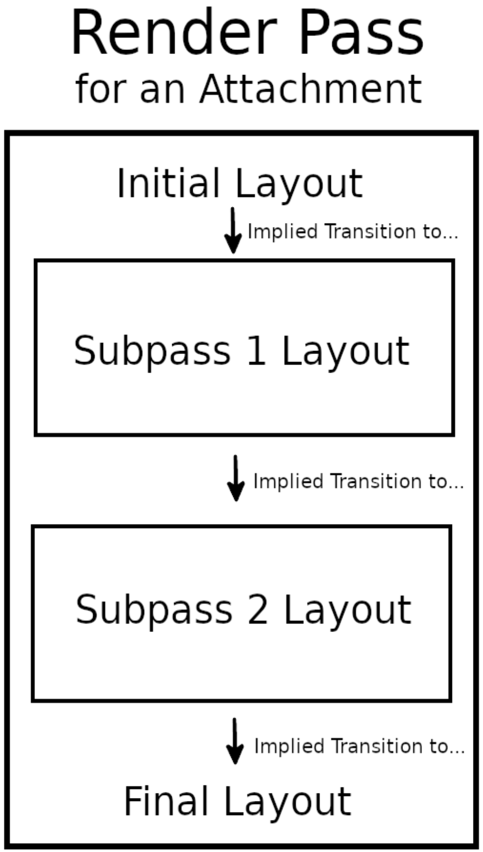
# Render Pass


Render Pass
Render Pass
Attachments
Framebuffer
Subpass 2
Render Pass
Attachment References
Subpass 1
Render Pass
Attachment References
Pipeline 2

# Subpass Dependencies

- Subpasses rely on strict ordering to ensure data is in the right format at the right time.
- For example: Our Swapchain image being written to will need to be in a writable format at the stage of the subpass that writes to the image (Logical!)
- However when we present it, it will need to be in a presentable format.
- Subpass Dependencies define stages in a pipeline where transitions need to occur.
- These are “implicit” transitions since we do not explicitly state the transition that needs to take place. We only say when a transition should occur.
- Vulkan knows what transitions to make because we define the layout at each stage of the subpass when we set it up.

# Subpass Dependencies



# Summary

- Vulkan Pipeline is identical to OpenGL pipeline.
- Must define details of each stage manually.
- Shaders are compiled to SPIR-V format and loaded into modules.
- Pipeline Layout defines layout of uniform inputs (Descriptor Sets + Push Constants)
- Render Pass describes entire render process and outputs from each Pipeline to a Framebuffer.
- Render Pass has multiple attachments referenced by subpasses
- Layouts of attachments change between subpasses and at the end of a Render Pass. These changes are organised by Subpass Dependencies.

## See you next video!