Framebuffers, Command Buffers, and Drawing

Setting up a Framebuffer, Command Buffer, and finally drawing something!

Framebuffer

- Framebuffer is a connection between an image (or images) and the Render Pass
- We attach an image or multiple images to a Framebuffer
- The Render Pass then outputs fragment data from a Pipeline's execution, to the images bound to the Framebuffer's attachments
- The Framebuffer images line up 1-to-1 with the attachments in the Render Pass, so make sure you get the order right!

Command Buffer

- Unlike OpenGL where we submit one command to the GPU at a time, Vulkan works by pre-recording a group of commands, and then submitting them all to a queue at once.
- Commands will usually be in the form of:
  - 1. Start a Render Pass
  - 2. Bind a Pipeline
  - 3. Bind Vertex/Index data
  - 4. Bind Descriptor Sets and Push Constants
  - 5. Draw
- You can also submit a command to begin a new subpass, but you will need to bind the appropriate Pipeline again!

Command Pool

- Command Buffers are not created like most objects in Vulkan. They are instead allocated from a pool.
- This is a minor memory management concept: A pool creates a space of memory partitioned into blocks of the same size, without defining their details.
- We can then allocate one of these blocks for a Command Buffer.
- This pool concept is useful for things that are dynamically created and destroyed frequently!

Queues

- Command Buffers can not be executed directly. They need to be submitted to the appropriate queue.
- In our case, we have a Graphics queue so our Command Buffers must be performing graphical actions, such as render passes.
- To execute a Command Buffer, we simply submit it to a queue. We don’t need to do anything else, as the queue is constantly moving and executing Command Buffers.
- Think of the queue as a conveyor belt that’s constantly in motion. The moment we submit a Command Buffer to it, it automatically starts on its way to execution!

Synchronisation

- One of the harder concepts to grasp in Vulkan if you haven't done parallel programming before: Synchronisation.
- This is making sure we don't accidentally try to access something twice.
- For example: We draw to Swapchain Image 1 then begin to present it. For some reason the system lags and the loop returns to drawing to Image 1, and while it's drawing, the presentation system gets over the lag and tries to display the image **whilst it's being drawn to**.
- That is a clash we don't want!
- To fix this, we introduce Semaphores and Fences.

Synchronisation (cont.)

- Semaphores are flags that say if a resource can be accessed. If they are “signalled” (i.e. set to “true”) then the resource is available to use.
- If they are “unsignalled” (i.e. set to “false”) then the resource is in use, and the program must wait.
- In the case of Vulkan, semaphores are used solely on the GPU itself, meaning they only create synchronisation between GPU functions (however, we can set them up on the CPU side, we just can’t control them after creation).
- In our case, we use semaphores to indicate:
  - 1. When an image has become available following presentation (the GPU will not draw until it is available).
  - 2. When an image has finished being rendered and is therefore ready to present (the presentation command will not present until the GPU has finished rendering).

Synchronisation (cont.)

- Fences are similar to Semaphores, except we have the ability to unsignal a Fence and wait on a Fence CPU side (allowing us to block on the CPU).
- The GPU can still signal a Fence to say a resource has become available. It is then up to us to unsignal it when we want to use it.
- We do this with:
  - vkWaitForFences: This will block the CPU code until the GPU signals the Fence.
  - vkResetFences: This will unsignal a Fence until the GPU signals it again.
- We will use Fences to ensure a “frame” is available, so we don’t accidentally flood the queue with too many draw/present commands.

Summary

- Framebuffers contain attachments that are linked up to a Render Pass to draw to.
- Command Buffers are allocated from Command Pools.
- Command Buffers contain a list of pre-recorded commands.
- Command Buffers are then submitted to appropriate queues for execution.
- We need to synchronise our Command Buffer execution.
- We do this with Semaphores and Fences.
  - Semaphores synchronise GPU-GPU actions.
  - Fences synchronise CPU-GPU actions.

See you next video!