RENDERING IN VULKAN

Melvin Rother, melvin.rother@pm.me

INTRODUCTION

- This presentation serves as a sort of archive of what I learned about the Vulkan rendering API in Block B.

- You can find the project on my GitHub

- This is a sequel to my self-study from Block A: Rendering a Triangle in Vulkan

- As I can only work on this on Fridays, I only aim to learn about Vertex Buffers

- You can find the tutorial I use as a reference here: vulkan-tutorial.com

STEP 0: IN-FLIGHT FRAME SYNCHRONIZATION

- During my self-study in Block A, I purely focused on rendering a Triangle and ignored everything else (such as Buffers etc.) -> only very basic building blocks are implemented

- Because of that, the program always has to wait until that one frame has finished before a new one can be drawn

- Since that is very ineffective, I implemented the ability to have several frames in flight -> which means one frame can be rendered while still waiting for the first one to finish

- This can be done the same way as only using one frame

    - The only requirement is that there need to be individual synchronisation objects (Semaphores & Fences) and Command buffers per frame

    - Thus, I changed my implementation to use vectors of synchronisation objects and Command Buffers that can then easily be indexed into per frame

```cpp
// for each frame in flight
for (int i = 0; i < m_MaxInFlightFrames; i++)
{
    // create Semaphores
    if (vkCreateSemaphore(m_LogicalDevice, &t_SemaphoreCreateInfo, pAllocator:nullptr, &m_ImageAcquiredSemaphores[i]) != VK_SUCCESS ||
        vkCreateSemaphore(m_LogicalDevice, &t_SemaphoreCreateInfo, pAllocator:nullptr, &m_RenderFinishedSemaphores[i]) != VK_SUCCESS)
    {
        throw std::runtime_error(_Message:"Could not create Semaphores!");
    }

    // create Fence
    if (vkCreateFence(m_LogicalDevice, &t_FenceCreateInfo, pAllocator:nullptr, &m_InFlightFences[i]) != VK_SUCCESS)
    {
        throw std::runtime_error(_Message:"Could not create Fence!");
    }
}
```

STEP 1: BUFFER INPUT DESCRIPTIONS

- **To get data from the CPU to the GPU, there are a few steps that we need to take:**

  - ➢ **Create a CPU visible buffer**

  - ➢ **Copy data (in our case, vertex data) into it using memcpy**

  - ➢ **Use a staging buffer to copy the data to high-performance memory**

- To create a CPU visible buffer, we first need to tell Vulkan how to interpret the data after it has been uploaded to the GPU. This can be done using Input Descriptions:

  - A **Binding Description** informs Vulkan about each vertex input binding, the stride between the elements in the buffer and the `VertexInputRate`
    - Binding descriptions are created using the `VkVertexInputBindingDescription` struct
  - **Attribute Descriptions** tell Vulkan about the vertex input attributes, including which shader input the data should be bound to, the binding number the attribute retrieves data from and the format and size of the attribute data
    - Attribute Descriptions are created using the `VkVertexInputAttributeDescription` struct

- Then, the graphics pipeline needs to be told to receive the data that we want to buffer

  - This can be done by modifying the `VkPipelineVertexInputStateCreateInfo` struct that we have to create when creating the Graphics Pipeline
  - To do so, the binding & attribute description count members have to be set, the `pVertexBindingDescriptions` member has to reference the binding description struct, and the `pVertexAttributeDescriptions` member has to be fed the raw data of the attribute descriptions

    - ➢ **! Note that as we have not yet bound a vertex buffer, the validation layers will report a validation error !**

STEP 1: BUFFER INPUT DESCRIPTIONS

- **Creating the binding descriptions works the same as everything in Vulkan**
  - Create a VkVertexInputBindingDescription struct and assign to it the binding number, the size of an individual element of what is being bound and the input rate
    - The input rate can either be VK_VERTEX_INPUT_RATE_VERTEX (meaning it moves to the following data entry after each vertex) or VK_VERTEX_INPUT_RATE_INSTANCE (meaning it moves to the next data entry after each instance)

- **Creating the attribute descriptions follows the same pattern**
  - Create a VkVertexInputAttributeDescription struct and assign its members.
    - The location member references the location directive set in the vertex shader (e.g. where can the data be accessed from in the shader)
    - The format member describes the type of data sent to the GPU (e.g. a glm::vec would be VK_FORMATR32G32B32_SFLOAT because it contains three floats of a size of 32 bits) -> if there are more components defined in the format than are in the shader data type, the additional components will be discarded

- **To apply the binding and attribute descriptions, they need to be passed to the graphics pipeline**
  - In the VkPipelineVertexInputStateCreateInfo struct used to generate the graphics pipeline, add both the binding and attribute descriptions to the corresponding members
  - The Graphics Pipeline is then prepared to receive data from the buffer

STEP 2: VERTEX BUFFER CREATION

- In Vulkan, the concept of a buffer describes regions of memory that can be used to store any data, which the graphics card can then read

- Buffers do not allocate memory for themselves (unlike other objects in Vulkan) -> Vulkan gives the developer complete control of the memory management of Buffers

- The process of creating a buffer is the same as always: make a `VkBufferCreateInfo` and fill in the members

  - The `size` member describes the size of the data to be buffered in bytes

  - The `usage` member describes how the buffer is used. Several purposes can be defined using the bitwise or operator

  - The `sharingMode` member fulfils the same function as the member of the same name in the `VkSwapChainCreateInfoKHR`. Buffers can be owned by one queue family or several at the same time, and this member describes how this will be handled

- Then the buffer can be created with a call to `vkCreateBuffer`

STEP 3: ASSIGNING MEMORY TO THE VERTEX BUFFER

- Before memory can be assigned to a buffer, the memory requirements need to be queried using `VkGetBufferMemoryRequirements` that stores the returned information into a `VkMemoryRequirements` struct

- Then, the right type of memory to be allocated from can be queried from the types of memory the graphics card offers

- Afterwards, memory can be allocated by filling a `VkMemoryAllocateInfo` struct with the the type and size of memory to be allocated and then calling `vkAllocateMemory`

- The memory can then be bound to the buffer using `vkBindBufferMemory`

- Then the buffer can be filled with the desired data. This can be done by calling `vkMapMemory`, using `memcpy` to copy the desired data to the location the resulting pointer points to and calling `vkUnmapMemory`.

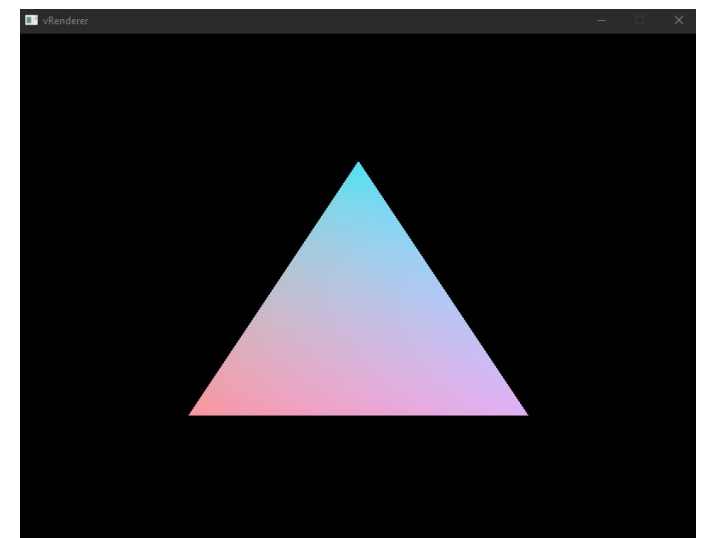# STEP 4: BINDING THE VERTEX BUFFER

- Binding a Buffer is then straightforward. When recording the command buffer, follow these steps:

  1. Add all of the (Vertex) Buffers into an array of VkBuffers

  2. Define all of the Offsets in an array of VkDeviceSize objects

  3. Call vkCmdBindVertexBuffers using the array of buffers and array of offsets

  4. If your buffer is a vertex buffer, also modify the vkCmdDraw to take in the correct size of the vertices buffered

  5. …

  6. Profit!
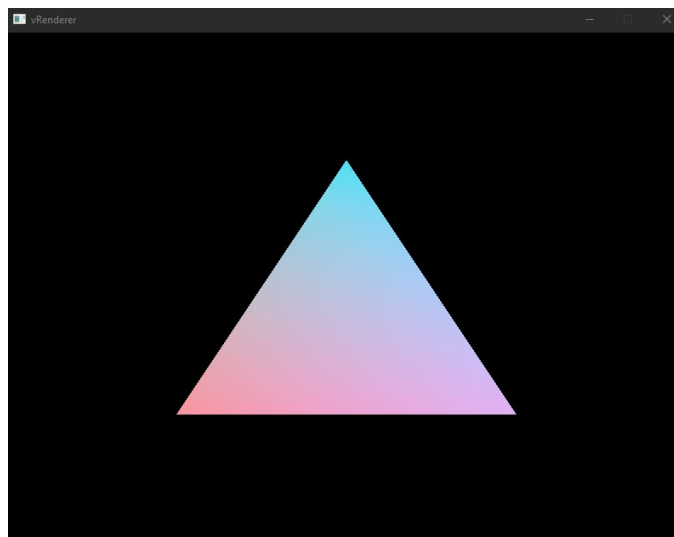
SUMMARY

The Steps to creating and using a buffer in Vulkan follow the general structure of creating anything else in the API:

1. Create the Vertex Binding and Attribute descriptions. These tell Vulkan how to interpret the data uploaded to the GPU

2. Create the Vertex Buffer (this follows the same pattern as any other object in Vulkan: make a BufferCreateInfo struct and feed it to the vkCreateBuffer function)

3. Assign memory to it by

    1. Allocating memory for it

    2. Binding said memory

    3. Mapping the memory, copying data into the buffer using memcpy and then unmapping it

4. Finally, bind the buffer to the Command Buffer using vkCmdBindVertexBuffers

THANK YOU

SOMEONE@EXAMPLE.COM