Resource Loading

Loading vertices and indices, and using staging buffers

Loading Vertex Data

- Alter pipeline to accept Vertex data with VkPipelineVertexInputStateCreateInfo
  - Takes descriptions of vertex data layout, as well as shader binding information
- GLSL input similar to how it's done in OpenGL, must have "layout" identifier appended: layout(location = x)
- Vertex data itself is represented with a Buffer and Device Memory, two concepts bound together
- Buffer then bound as vkCmd and executed

Vertex Buffer

- **Buffer** created but does not hold data by itself
- Describes size of data in memory, as well as usage and queue sharing
  - **Usage**: What kind of data buffer will hold, e.g. vertex data, index data, storage data, etc.
  - **Queue Sharing**: If buffer will be used by multiple by multiple queue families simultaneously (for basic programs, exclusive to one queue)

Vertex Buffer (cont.)

- **Device Memory** represents raw data that holds actual vertex data to be used
- Memory is allocated from heap on Physical Device
- Multiple types of memory, first query Buffer for memory requirements, then iterate over available memory types to find compatible type
- Memory types have properties that define usability. Important ones are:
  - **VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT**: Memory is optimised for Device (GPU) usage. Can not be accessed directly by CPU and can only be interacted with via command buffers.
  - **VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT**: Memory is accessible by Host (CPU). Allows us to map data in application to GPU.
  - **VK_MEMORY_PROPERTY_HOST_COHERENT_BIT**: Allows data mapping to bypass caching commands, meaning data mapped does not need to be flushed to memory.

Mapping Memory

- After creating Buffer and Device Memory, bind two together.
- Now need to map Vertex data to memory.
- vkMapMemory(…) allows mapping of GPU memory to chosen point in Host memory
- Create arbitrary point in memory with void pointer
- Use vkMapMemory
- “memcpy” Vertex data to void pointer address
  - memcpy is a C library function for copying memory between addresses
- Unmap with vkUnmapMemory
- The data at the void pointer is now in the buffer!

Using Vertex Buffer

- Very simple to use now, just bind Vertex data before draw command
  - vkCmdBindVertexBuffers
- Use vkCmdDraw with vertex count as number of vertices to draw
- Want to pass more data with each vertex? Just add another attribute to VkPipelineVertexInputStateCreateInfo at Pipeline creation
- And then add new input in shader! No need to modify any more code

Index Buffer

- Most complex objects reuse same vertices, don’t want duplicate code
- Solution: Index Buffer. Use numbers to re-reference a vertex from a list of unique vertices
- How to create? Exactly the same way as Vertex Buffer!
- Usage slightly different:
  - **vkCmdBindIndexBuffer**: Binds a single Index Buffer. Must be done along with Vertex Buffer binding.
  - **vkCmdDrawIndexed**: Draws using indices and takes number of indices to draw. Must be used in place of vkCmdDraw.

Staging Buffers

- Optimisation: Recall memory property
  “VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT”
  - Allows memory to be optimised for GPU usage… but disallows Host access.
- Current method does not use this because vkMapMemory would not work because it is a Host command accessing GPU
- Solution: Create TWO buffers, one that is Host visible, and one that is GPU only…
- … and then use a Command Buffer to copy data from one to the other.
- This is fine because command buffers run on GPU, not Host!

Staging Buffers (cont.)

- Start by creating Host visible buffer + memory. This is the “staging buffer”.
- However, instead of making buffer into Vertex Buffer, set up to be a “Transfer Source” buffer.
  - **VK_BUFFER_USAGE_TRANSFER_SRC_BIT**: Literally the “source” of a transfer, or “the place data is being copied from”.
- vkMapMemory as done previously
- Create second GPU only buffer + memory, set up to be “Transfer Destination” buffer… but ALSO a Vertex Buffer! A Buffer can be multiple types!
  - **VK_BUFFER_USAGE_TRANSFER_DST_BIT**: Literally the “destination” of a transfer, or “the place data is being copied to”.

Staging Buffers (cont.)

- To copy data, need new temporary command buffer
- Command Buffer created same as one used for rendering…
- …but no Render Pass!
- Instead, has a single command: vkCmdCopyBuffer
  - VkCmdCopyBuffer: Command that copies from one buffer to another. Takes source and destination buffers to copy from and to.
  - Also takes a “VkBufferCopy” struct which defines offset to copy from, offset to copy to, and size of data to copy.
- Command Buffer is submitted to queue *immediately* and then command buffer is destroyed.
- Staging Buffer can then also be destroyed.
- Data is now GPU-only! This is much more efficient.

Summary

- Define Vertex data layout with VkPipelineVertexInputStateCreateInfo
- Create Buffer and Device Memory, then bind them
- Copy data to Device Memory with vkMapMemory
- Use by binding vertex buffer in command buffer
- Can also use Index Buffer to minimize vertex data duplicates, created same way as vertex buffer
- Can use Staging Buffers to copy Vertex/Index buffers to optimised memory
- Staging Buffers use temporary command buffer to “transfer” data from one buffer to another

See you next video!