VULKAN: COMPUTE SHADERS

Melvin Rother, melvin.rother@pm.me

1. READ AND WRITE TO BUFFERS ON THE GPU

- In rendering, data (like textures) are passed from the CPU to the GPU
- However, PCI-E bandwidth is limited → Updating on and uploading particles from the CPU is very slow
- Using Compute Shaders, particles are only uploaded once and then updated on the GPU
- For this, it is required to know not only how to read data from the GPU but also how to write it
- To manipulate data on the GPU from the GPU, one needs to know two things:
  - Shader Storage Buffer Objects
  - Storage Images

1.1 SHADER STORAGE BUFFER OBJECTS

- SSBOs allow shaders to read from and write to a buffer → similar in usage to UBOs (Uniform Buffer Objects)
  - SSBOs can be arbitrarily large
  - It is possible to alias other buffer types to SSBOs
- Often these objects would be required to be a storage buffer in one pass (typically the compute pass) and another buffer type in another pass (for example a vertex buffer in the graphics pass)
- Using them in such a capacity is very easy if you can already create buffers:
  - Simply create the buffer with both usage flags, e.g. `VK_BUFFER_USAGE_VERTEX_BUFFER_BIT` and `VK_VBUFFER_USAGE_STORAGE_BUFFER_BIT` (of course this will also need the transfer destination bit flag to allow data to be transferred into it)
  - It is also important to specify the buffer as located in GPU memory using the `VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT` flag

1.1 SHADER STORAGE BUFFER OBJECTS

- Accessing the SSBO in the GLSL shader is then trivial:

```
layout (std140, binding = 1) readonly buffer ParticleSSBOIn
{
    Particle particlesIn[];
}

layout (std140, binding = 2) buffer ParticlesSSBOOut
{
    Particle particlesOut[];
}
```

1.2 STORAGE IMAGES

- Storage images are simply images that allow for read-and-write access
- Useful for post-processing etc.
- Creating them is similar to creating normal images
  - The only major difference is the usage flags: `VK_IMAGE_USAGE_SAMPLED_BIT` and `VK_IMAGE_USAGE_STORAGE_BIT`
  - They tell the program to use the image for both sampling in the fragment shader and for storage in the compute shader
- The image can then be accessed by simple GLSL bindings:
  - `layout (binding = 0, rgba8) uniform readonly image2D inImage;`
  - `layout (binding = 1, rgba8) uniform writeonly image2D outImage;`
- The image can then be accessed using the `imageLoad()` and `imageStore()` functions

The GLSL code on this page was taken from: https://vulkan-tutorial.com/Compute_Shader

2. COMPUTE QUEUE FAMILIES

- To use compute shaders, a Queue family needs to be queried that supports it
- Compute uses the `VK_QUEUE_COMPUTE_BIT` queue family property flag
- By default, Vulkan requires an implementation that has at least one queue family that supports both Compute and Graphics operations → That means your operations will always be supported by any Vulkan device
- To use the compute queue family, simply choose the queue family that supports `VK_QUEUE_COMPUTE_BIT`
- A Compute Queue can then be accessed by calling `vkGetDeviceQueue`

3. THE COMPUTE SHADER STAGE

- Compute Shaders can be loaded and accessed similarly to any other shader
- Loading them is the same as vertex and fragment shaders, only the Pipeline Stage is `VK_SHADER_STAGE_COMPUTE_BIT`

3.1 SETTING UP SHADER STORAGE BUFFERS

- Creating Shader Storage Buffers is in principle the same as any other buffer:
  1. Create a Staging Buffer local to the CPU
  2. Map the Buffer Memory, `memcpy` the data into it and unmap the memory
  3. Create the Shader Sorage Buffer and copy the Staging Buffer into it using `vkCmdCopyBuffer`
  4. Free the staging buffer

- To avoid waiting for a frame to finish, it makes sense to create as many Shader Storage Buffers as there are frames in flight
- That way, the CPU and GPU can already continue working without having to wait for the previous frame to finish → avoids idling

3.2 DESCRIPTORS

- Creating Descriptors is the same for the Compute Stage as it is for the Graphics Stage
  1. Create a `VkDescriptorSetLayoutBinding`. Make sure the `stageFlags` member is set to `VK_SHADER_STAGE_COMPUTE_BIT`
     - Note that if you want the Shader Storage Buffer to be accessible from multiple stages, all the stages the buffer should be accessible from need to be specified in the `stageFlags` member
     - Set the `descriptorType` member to `VK_DESCRIPTOR_TYPE_STORAGE_BUFFER`
  2. Request the descriptor types for the shader storage buffers from the descriptor pool

4. COMPUTE PIPELINE

- The Graphics Pipeline and the Compute Pipeline are separate
- Thus we need to set up a Compute Pipeline in addition to the graphics pipeline
- In general, the compute pipeline is much simpler to create as it does not need to define any of the states related to rasterization

1. Create a Pipeline Layout for the Compute Pipeline
   - This is the same as for the graphics pipeline
2. Create a `VkComputePipelineCreateInfo` struct
3. Fill its members
4. Call vkCreateComputePipeline

5. COMPUTE SPACE

- The execution model for compute workloads is special:

1. Work Groups
   - Define how Compute work is processed by the hardware
   - Work group dimensions are set at command buffer time with a dispatch command (`vkCmdDispatch`)
   - The maximum count of work groups differs for each implementation → Check for it in `VkPhysicalDeviceLimits`
2. Invocations
   - Work Groups aggregate Invocations
   - Each Invocation in a Work Group executes the same Compute Shader
   - Invocations can run in parallel
   - Dimensions are set in the compute shader
   - Invocations in the same Work Group can access shared memory

- The dimensions of Work Groups and Invocations depend on the structure of the input data

6. COMPUTE SHADERS

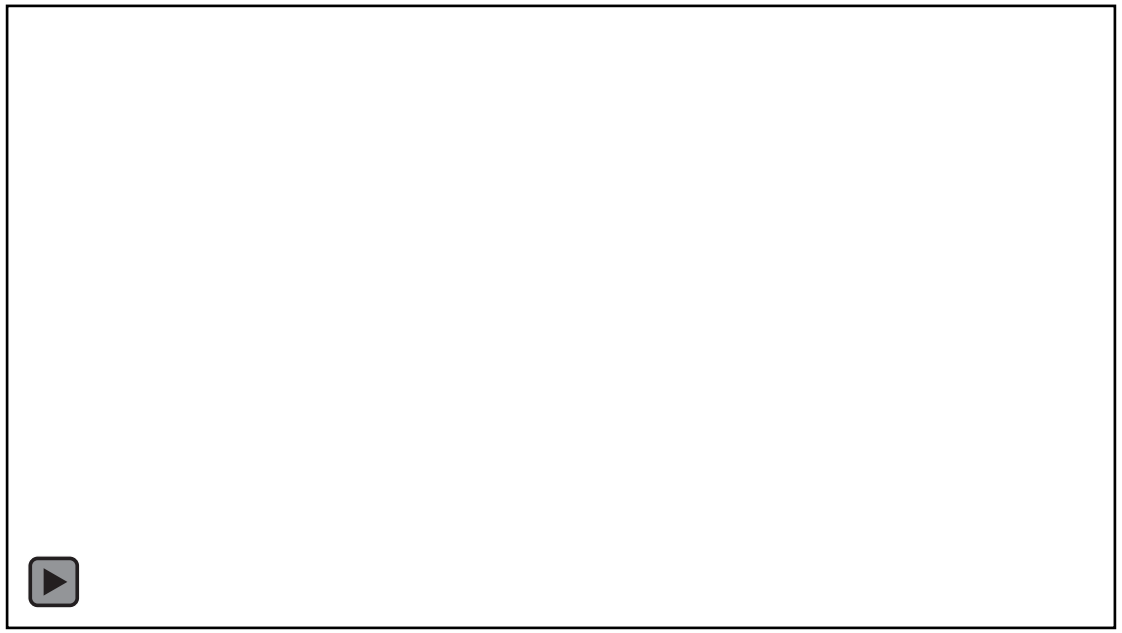
- The basics for setting up Compute Shaders in GLSL are the same as for every other shader, with a few notable exceptions:
  - The number of Invocations needs to be defined: `layout (local_size_x = 100, local_size_y = 1, local_size_z = 1) in;`
  - The actual dimensions depend on the input data, e.g. if the input is a one-dimensional array, only the x-dimension needs to be defined
  - The `local_` prefix defines it as a local part of compute space
  - To identify which invocation is currently being processed users can use `gl_GlobalInvocationID` → useful to index into arrays

6. RUNNING COMPUTE COMMANDS

1. Dispatching Commands
   - Call `vkCmdDispatch` in a Command Buffer
   - Make sure to set the correct amount of work groups, this usually takes some tweaking (for each dimension
2. Submitting Work
   - Submitting compute work is the same as graphics work: simply call `vkQueueSubmit` for the compute queue (make sure to do so before submitting any graphics pipelines that rely on the compute data)
3. Synchronization
   - It is important to properly synchronize when using both compute and graphics
   - For example, If the vertex stage starts reading data before the compute stage has finished work, issues arise
   - This can be done using the usual methods: Fences and Semaphores

RESULTS

The system's Compute Shader is responsible for using the GPU to update all the particles. The initial values, such as the particles' position, velocity, and colour, are received only once by the Compute Pipeline. The Compute Shader then automatically updates each particle's position and stores it in the Shader Storage Buffer, which is aliased as a Vertex Buffer in the Graphics Pipeline. This allows for it to be used for read/write access in the Compute Pipeline and as a Vertex Buffer in the Vertex Shader in the Graphics Pipeline. By doing so, the positions computed in the Compute Shader can be utilised in the Vertex Shader of the Graphics Pipeline, which results in the GPU drawing a point for each particle.



THANK YOU

SOMEONE@EXAMPLE.COM