TP04
CPU - GPU transfer

Robin Faury

01/22/20

Abstract

In this practical work, we will see how to fill a buffer on the device and transfer this buffer back to the host.

1 Staging buffer

On the last practical we see how to create a buffer dedicated to storage on the device, how to create its memory and how to allocate it. For this practical we will call this buffer the GPU buffer or device buffer. Actually we cannot transfer directly the host buffer to this memory area. We need to send our data on a memory area shared by the host and the device.
Create another buffer named stageBuffer. The usage of this buffer should be set to VK_BUFFER_USAGE_TRANSFER_SRC_BIT and the memory we are looking is VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT. On your Buffer class, add these parameters to the constructor:

- VkBufferUsageFlags usage
- VkMemoryPropertyFlags properties

Use those two new parameters to create the stageBuffer.

1.1 memory mapping

Every buffers visible from the host can be mapped for a data transfer. Firstm we need to ask Vulkan to send us a pointer for a memcpy.

```
void* driverData;
vkMapMemory(
        logicalDevice,
        stageBuffer.getBufferMemory(),
        0,
        bufferSize,
        0,
        &driverData);
```

We are now synchronized with the GPU. This kind of situation should be avoided! If the GPU hasn't finishing a computation, the CPU is waiting.
After call a memcpy to fill the driverData pointer returned by vkMapMemory, we must unmap the memory.

```
vkUnmapMemory(logicalDevice, buffer.getBufferMemory());
```

To conclude, we have right now two buffers, the stageBuffer and its memory filled by our data and the GPU buffer and its memory unfilled. We should now transfer the content of the stage buffer to the GPU buffer. The stage buffer is set to a transfer src usage. That means this buffer can be sent to an other. The GPU buffer usage was set to storage That means this buffer can store data. We simply need to add this usage to the GPU buffer to allow it to receive data:

```
VK_BUFFER_USAGE_TRANSFER_DST_BIT | VK_BUFFER_USAGE_STORAGE_BUFFER_BIT
```

2 Copy buffer

Until now, we simply ask Vulkan GPU properties and allocate object on it. The aim here is to call commands to ask Vulkan to transfer the staging buffer to the GPU buffer. Create a function named copyBuffer with the source buffer, the destination buffer, the logical device and the queue family id as parameters.

2.1 Command pool

To execute a process on the device, we need to send every command on an FIFO buffer. Each command will be executed asynchronous from the host. This buffer is called the command buffer and it needs to be allocated on the GPU. The VkCommandPool manage this memory. Inside the copyBuffer function, create this handle:

```
VkCommandPool commandPool;
```

We will populate the handle so we need a create info object:

```
VkCommandPoolCreateInfo commandPoolCreateInfo = {};
commandPoolCreateInfo.sType = VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
commandPoolCreateInfo.queueFamilyIndex = /*The index of the queue family*/;
```

The last member is flags. We will set its value to VK_COMMAND_POOL_CREATE_TRANSIENT_BIT. This value means the command buffer allocated from the pool will be short-lived. This is the case here because we will destroy the pool after the copy.
We can now use vkCreateCommandPool to fill the command pool handle and throw an error if the function doesn't return VK_SUCCESS. Don't forget to destroy the command pool (vkDestroyCommandPool) at the end of the function scope.

2.2 Command buffer

The command buffer don't need to be created, but just allocated. Create a VkCommandBufferAllocateInfo with the sType VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO. The level member can be set to "primary" or "secondary". The primary flag means the command buffer can be submitted to a queue and the secondary flag mean the command buffer can be called by a primary command buffer. In this case, we need to set the level to VK_COMMAND_BUFFER_LEVEL_PRIMARY. For the commandPool member, put the command pool. We need just one command buffer.

```
VkCommandBuffer commandBuffer;
vkAllocateCommandBuffers(
        logicalDevice, &commandBufferAllocationInfo, &commandBuffer);
```

We can now start filling the command buffer to tell Vulkan to transfer the buffer. For that we need to scope our commands by the vkBeginCommandBuffer and the vkEndCommandBuffer.

```
VkCommandBufferBeginInfo commandBufferBeginInfo = {};
commandBufferBeginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
commandBufferBeginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

vkBeginCommandBuffer(commandBuffer, &commandBufferBeginInfo);

// transfer command

vkEndCommandBuffer(commandBuffer);
```

2.3 transfer command

In this situation, our command buffer isn't huge. We just need one command. Every Vulkan command start by vkCmd. We will use the vkCmdCopyBuffer to copy the stage buffer to the GPU buffer. Create a VkBufferCopy object to tell to the command which part of the buffer you want to copy. In our situation the all buffer.

```
VkBufferCopy bufferCopy = {};
bufferCopy.srcOffset = 0; // Optional
bufferCopy.dstOffset = 0; // Optional
bufferCopy.size = size;
vkCmdCopyBuffer(commandBuffer, stageBuffer, GPUBuffer, 1, &bufferCopy);
```

2.4 Submit the command buffer

We have an object who describe how a buffer should be transferred. We want now submit this command buffer. To submit a queue we need a submit info:

```
VkSubmitInfo submitInfo = {};
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
submitInfo.commandBufferCount = 1;
submitInfo.pCommandBuffers = &commandBuffer;
```

We are all set to transfer out buffer. Call the vkQueueSubmit function on your queue using the submitInfo. In this case, we will don't use fence for synchronization, use VK_NULL_HANDLE for the last parameter. At this moment the GPU start to transfer our buffer to the GPU buffer. We mustn't destroy our handle until the process is done. We should wait the queue.

```
vkQueueWaitIdle(queue);
```

We can now free the command buffer (vkFreeCommandBuffers) and destroy the command pool.

3 Send data

Create a function called sendData. This function must instantiate a stage buffer host visible and with its buffer flag set to transfer source. Map this buffer, copy your data and unmap it. Transfer the stage buffer to the GPU buffer using the copyBuffer function. And finally, destroy the stage buffer and free its memory.

4 Get data

Create a function called getData. This function must instantiate a stage buffer host visible and with its buffer flag set to transfer destination. Transfer the GPU buffer to the stage buffer using the copyBuffer function. Map the stage buffer, copy its data and unmap it. And finally, destroy the stage buffer and free its memory.