The use of data in vulkan

Drawing a triangle is nice, but it is only a first step. We now have to pass data to the GPU, so that is can input data, uniform and constants to the shaders and manipulate the depth buffer. In a future lesson we will also setup more complex data, such as textures and models.

This project will be supported by an other visual studio solution, called VulkanApp_DataInjection. It starts from the state of the VulkanApp project, so you do not have to switch your test solution.

Ressource loading : Vertex input, index buffers and staging buffers

Concepts

We want our pipeline to accept vertex data (currently hardcoded in the vertex shader). We will have to modify it to do so. The GLSL input will be similar to how it is done in OpenGL: we will have a layout identifier `layout(location=x)`. The vertex data itself will be represented with a **Buffer** and **Device Memory**, two concepts bound together.

The **Buffer** is created but do not hold the data itself. It describes how memory should be accessed, and the memory is taken from the physical device. We will set it up by :

- Describing the size of data in memory,
- Deciding **Usage** : what kind of data it will contains (e.g. vertex data, index data, storage data...)
- Defining **Queue sharing** : if the buffer should be used by one or multiple queue families simultaneously

The **Device Memory** represents raw data that holds actual vertex data to be used. The memory is allocated from a heap on Physical Device. There are multiple types of memory ; we will first query the buffer for memory requirements, then iterate over available memory types to a find compatible type. Memory types have properties:

- `vk::MemoryPropertyFlagBits::eDeviceLocal` : memory is optimized for **Device** (GPU) usage. Cannot be accessed directly by CPU and can only be interacted

with via command buffers.
- `vk::MemoryPropertyFlagBits::eHostVisible`: memory is accessible by **Host** (CPU). It allows to map data in application to GPU.
- `vk::MemoryPropertyFlagBits::eHostCoherent`: allows data mapping to bypass caching commands, meaning data mapped does not need to be flushed to memory.

After create the buffer and device memory, we will bind them together.

We need to map vertex data to memory. **Mapping Memory** will be mapping a device memory location to a chosen point of the host memory. We will create an arbitrary point in memory with a void pointer, then use the `vkMapMemory` function to map this memory, then `memcpy` the vertex data to the void pointer address. Once the data is copied, we can unmap memory with `vkUnmapMemory`. The data at the void pointer will be now in the device buffer.

Once the buffer is ready, we just have to bind the vertex data before the draw command with `vkCmdBindVertexBuffers`. We will have to change the vertex count to the right amount of vertices. If we want to pass more data with each vertex, we can add attribute to the `VkPipelineVertexInputStateCreateInfo` at pipeline creation, then add new input in the shader.

When we want to use an index buffer and reuse vertices, we will create it the same way we created the vertex buffer, this time with a `vkCmdBindIndexBuffer` function. The draw will slightly change and we will use `vkCmdDrawIndexed` in place of `vkCmdDraw`.

If we want to optimize with the `vk::MemoryPropertyFlagBits::eDeviceLocal` property, `vkMapMemory` won't work, because it is a host command accessing GPU. The solution will be to create two buffers, one that is host visible, one that is GPU only, and then use a command buffer to copy data from one to the other. Indeed, the command buffers run on GPU.

We will start by creating a host visible buffer plus its memory. This will be the **Staging Buffer**. Instead of making it a vertex buffer, we will set it up to be a **Transfer Source** buffer. Then we will map memory. Then create a second buffer and its memory, set it up to be a **Transfer Destination** buffer and ALSO a vertex buffer. We now need to copy data, we will need a temporary command buffer, but this time with no render pass ; we will just use `vkCmdCopyBuffer`. We will submit it to the queue and destroy it. We can also destroy the staging buffer which is not used anymore. Our data is now GPU only.

NB: usually when we execute command buffers, we bind a pipeline. Here we won't. Actually, people refer to transfer as a stage of a one stage pipeline. Compute stage (for compute shaders) is the same kind of stage.

Vertex input

Prepare the data

Add a Vertex struc in the utilities:

`VulkanUtilities.h`

```
#include <glm/glm.hpp>
...
struct Vertex
{
        glm::vec3 pos;
};
```

Modify the shaders:

`shader.vert`

```
#version 450

// From vertex input stage
layout(location = 0) in vec3 pos;



void main() {
    gl_Position = vec4(pos, 1.0);
}
```

`shader.frag`

```
#version 450

// Final output color, must have location
layout(location = 0) out vec4 outColor;

void main() {
    outColor = vec4(0.8, 1.0, 0.2, 1.0);
}
```

Do not forget to compile them again.

Get the data into the pipeline.

In the `VulkanRenderer::createGraphicsPipeline` function, change the code from the `// Create pipeline` comment to the end of the vertex input stage:

`VulkanRenderer.cpp`

```
...
        // Vertex description
        // -- Binding, data layout
        vk::VertexInputBindingDescription bindingDescription{};
        // Binding position. Can bind multiple streams of data.
        bindingDescription.binding = 0;
        // Size of a single vertex data object, like in OpenGL
        bindingDescription.stride = sizeof(Vertex);
        // How ot move between data after each vertex.
        // vk::VertexInputRate::eVertex: move onto next vertex
        // vk::VertexInputRate::eInstance: move to a vertex for the next instance.
        // Draw each first vertex of each instance, then the next vertex etc.
        bindingDescription.inputRate = vk::VertexInputRate::eVertex;

        // Different attributes
        array<vk::VertexInputAttributeDescription, 1> attributeDescriptions;

        // Position attributes
        // -- Binding of first attribute. Relate to binding description.
        attributeDescriptions[0].binding = 0;
        // Location in shader
        attributeDescriptions[0].location = 0;
        // Format and size of the data(here: vec3)
        attributeDescriptions[0].format = vk::Format::eR32G32B32Sfloat;
        // Offset of data in vertex, like in OpenGL. The offset function automatically find it.
        attributeDescriptions[0].offset = offsetof(Vertex, pos);

        // -- VERTEX INPUT STAGE --
        vk::PipelineVertexInputStateCreateInfo vertexInputCreateInfo{};
        vertexInputCreateInfo.vertexBindingDescriptionCount = 1;
        vertexInputCreateInfo.pVertexBindingDescriptions = &bindingDescription;
        vertexInputCreateInfo.vertexAttributeDescriptionCount =
                static_cast<uint32_t>(attributeDescriptions.size());
        vertexInputCreateInfo.pVertexAttributeDescriptions = attributeDescriptions.data();
...
```

A mesh class

We will store our input data and prepare it into a buffer thanks to a `VulkanMesh` class. Create it.

`VulkanMesh.h`

```
#pragma once
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>

#include <vector>
using std::vector;

#include "VulkanUtilities.h"

class VulkanMesh
{
public:
        VulkanMesh(vk::PhysicalDevice physicalDeviceP,
                vk::Device deviceP, vector<Vertex>* vertices);
        VulkanMesh() = default;
        ~VulkanMesh() = default;

        size_t getVextexCount();
        vk::Buffer getVertexBuffer();
        void destroyBuffers();

private:
        size_t vertexCount;
        vk::Buffer vertexBuffer;
        vk::PhysicalDevice physicalDevice;
        vk::Device device;
        vk::DeviceMemory vertexBufferMemory;

        void createVertexBuffer(vector<Vertex>* vertices);
        uint32_t findMemoryTypeIndex(uint32_t allowedTypes, vk::MemoryPropertyFlags properties);
};
```

The constructor, getter and `destroyBuffers` function are straightforward:

`VulkanMesh.cpp`

```cpp
#include "VulkanMesh.h"

VulkanMesh::VulkanMesh(vk::PhysicalDevice physicalDeviceP,
        vk::Device deviceP, vector<Vertex>* vertices)
        :
        vertexCount(vertices->size()), physicalDevice(physicalDeviceP), device(deviceP)
{
        createVertexBuffer(vertices);
}

size_t VulkanMesh::getVextexCount()
{
        return vertexCount;
}

vk::Buffer VulkanMesh::getVertexBuffer()
{
        return vertexBuffer;
}

void VulkanMesh::destroyBuffers()
{
        device.destroyBuffer(vertexBuffer, nullptr);
        device.freeMemory(vertexBufferMemory, nullptr);
}
...
```

The `createVertexBuffer` function implements a part what we have discussed in the concept paragraph:

```
...
void VulkanMesh::createVertexBuffer(vector<Vertex>* vertices)
{
        // -- CREATE VERTEX BUFFER --
        // Buffer info
        vk::BufferCreateInfo bufferInfo{};
        bufferInfo.size = sizeof(Vertex) * vertices->size();
        // Multiple types of buffers
        bufferInfo.usage = vk::BufferUsageFlagBits::eVertexBuffer;
        // Is vertex buffer sharable ? Here: no.
        bufferInfo.sharingMode = vk::SharingMode::eExclusive;

        vertexBuffer = device.createBuffer(bufferInfo);

        // Get buffer memory requirements
        vk::MemoryRequirements memoryRequirements;
        device.getBufferMemoryRequirements(vertexBuffer, &memoryRequirements);

        // Allocate memory to buffer
        vk::MemoryAllocateInfo memoryAllocInfo{};
        memoryAllocInfo.allocationSize = memoryRequirements.size;
        memoryAllocInfo.memoryTypeIndex = findMemoryTypeIndex(
                physicalDevice,
                // Index of memory type on physical device that has requiered bit flags
                memoryRequirements.memoryTypeBits,
                // CPU can interact with memory
                vk::MemoryPropertyFlagBits::eHostVisible |
                // Allows placement of data straight into buffer after mapping
                vk::MemoryPropertyFlagBits::eHostCoherent
        );

        // Allocate memory to vk::DeviceMemory
        auto result = device.allocateMemory(&memoryAllocInfo, nullptr, &vertexBufferMemory);
        if (result != vk::Result::eSuccess)
        {
                throw std::runtime_error("Failed to allocate vertex buffer memory");
        }

        // Allocate memory to given vertex buffer
        device.bindBufferMemory(vertexBuffer, vertexBufferMemory, 0);

        // -- MAP MEMORY TO VERTEX BUFFER --
        // 1. Create pointer to a random point in memory
        void* data;
        // 2. Map the vertex buffer memory to that point
        vkMapMemory(device, vertexBufferMemory, 0, bufferInfo.size, 0, &data);
        // 3. Copy memory from vertices memory to the point
        memcpy(data, vertices->data(), static_cast<size_t>(bufferInfo.size));
        // 4. Unmap the vertex buffer memory
        vkUnmapMemory(device, vertexBufferMemory);
        // Because we have eHostCoherent, we don't have to flush
}
...
```

The `findMemoryTypeIndex` makes a heavy use of bit-wise calculations. Please refer to the comment to understand the code. The goal is the get a memory type with all the flags we need.

```
...
uint32_t VulkanMesh::findMemoryTypeIndex(vk::PhysicalDevice physicalDevice,
        uint32_t allowedTypes, vk::MemoryPropertyFlags properties)
{
        // Get properties of physical device
        vk::PhysicalDeviceMemoryProperties memoryProperties =
                physicalDevice.getMemoryProperties();

        for (uint32_t i = 0; i < memoryProperties.memoryTypeCount; ++i)
        {
                // We iterate through each bit, shifting of 1 (with i) each time.
                // This way we go through each type to check it is allowed.
                if ((allowedTypes & (1 << i)) &&
                // Desired property bit flags are part of memory type's property flags.
                // By checking the equality, we check that all properties are available at the
                // same time, and not only one property is common.
                (memoryProperties.memoryTypes[i].propertyFlags & properties) == properties)
                {
                        // If this type is an allowed type and has the flags we want, then i
                        // is the current index of the memory type we want to use. Return it.
                        return i;
                }
        }
}
```

Using the vertex buffer

First add a mesh in the renderer:

`VulkanRenderer.h`

```
VulkanMesh firstMesh;
```

Change the `init` and `clean` function to use it:

`VulkanRenderer.cpp`

```
int VulkanRenderer::init(GLFWwindow* windowP)
{
        window = windowP;
        try
        {
                // Device
                createInstance();
                setupDebugMessenger();
                surface = createSurface();
                getPhysicalDevice();
                createLogicalDevice();

                // Pipeline
                createSwapchain();
                createRenderPass();
                createGraphicsPipeline();
                createFramebuffers();
                createGraphicsCommandPool();

                // Objects
                vector<Vertex> meshVertices{
                        {{ 0.0, -0.4, 0.0}},
                        {{ 0.4,  0.4, 0.0}},
                        {{-0.4,  0.4, 0.0}}
                };
                firstMesh = VulkanMesh(mainDevice.physicalDevice,
                        mainDevice.logicalDevice, &meshVertices);

    // Commands
                createGraphicsCommandBuffers();
                recordCommands();
                createSynchronisation();
        }
        catch (const std::runtime_error& e)
        {
                printf("ERROR: %s\n", e.what());
                return EXIT_FAILURE;
        }
        return EXIT_SUCCESS;
}
...
void VulkanRenderer::clean()
{
        mainDevice.logicalDevice.waitIdle();

        firstMesh.destroyBuffers();
    ...
}
```

We now need to update the `recordCommand` function to bind the vertex buffer:

```
void VulkanRenderer::recordCommands()
{
        ...
        // Bind pipeline to be used in render pass, you could switch
        // pipelines for different subpasses
        commandBuffers[i].bindPipeline(vk::PipelineBindPoint::eGraphics, graphicsPipeline);

        // -- NEW CODE --

        // Bind vertex buffer
        vk::Buffer vertexBuffers[] = { firstMesh.getVertexBuffer() };
        vk::DeviceSize offsets[] = { 0 };
        commandBuffers[i].bindVertexBuffers(0, vertexBuffers, offsets);

        // Execute pipeline
        commandBuffers[i].draw(static_cast<uint32_t>(firstMesh.getVextexCount()), 1, 0, 0);


  // -- END NEW CODE --

  // End render pass
  ...
}
```

Your triangle should now display from the data you send through the CPU.

Draw a rectangle

To draw a rectangle, add a new triangle in the mesh vertices:

```
vector<Vertex> meshVertices {
    {{ 0.4, -0.4, 0.0}},
    {{ 0.4,  0.4, 0.0}},
    {{-0.4,  0.4, 0.0}},

    {{-0.4,  0.4, 0.0}},
    {{-0.4, -0.4, 0.0}},
    {{ 0.4, -0.4, 0.0}},
};
```

It is actually a square, but there is no projection matrix.

Get colors back

We want the vertex colors from our triangle back on our rectangle. We will use a new location to achieve that. Update the shaders:

`shader.vert`

```
#version 450

// From vertex input stage
layout(location = 0) in vec3 pos;
layout(location = 1) in vec3 col;

// To fragment shader
layout(location = 0) out vec3 fragColor;

void main() {
    gl_Position = vec4(pos, 1.0);
    fragColor = col;
}
```

`shader.frag`

```
#version 450

// Input colors from vertex shader
layout(location = 0) in vec3 fragColor;

// Final output color, must have location
layout(location = 0) out vec4 outColor;

void main() {
    outColor = vec4(fragColor, 1.0);
}
```

Change the Vertex structure:

`VulkanUtilities.h`

```
struct Vertex
{
        glm::vec3 pos;
        glm::vec3 col;
};
```

Do not forget to recompile the shaders.

In the `createGraphicsPipeline` function, change the array of vertex input attribute descriptions:

`VulkanRenderer.cpp`

```
...
        // Different attributes
        array<vk::VertexInputAttributeDescription, 2> attributeDescriptions;

        // Position attributes
        // -- Binding of first attribute. Relate to binding description.
        attributeDescriptions[0].binding = 0;
        // Location in shader
        attributeDescriptions[0].location = 0;
        // Format and size of the data(here: vec3)
        attributeDescriptions[0].format = vk::Format::eR32G32B32Sfloat;
        // Offset of data in vertex, like in OpenGL. The offset function automatically find it.
        attributeDescriptions[0].offset = offsetof(Vertex, pos);

        // Color attributes
        attributeDescriptions[1].binding = 0;
        attributeDescriptions[1].location = 1;
        attributeDescriptions[1].format = vk::Format::eR32G32B32Sfloat;
        attributeDescriptions[1].offset = offsetof(Vertex, col);
...
```

We can now pass the color attributes with the position attributes.

`VulkanRenderer.cpp`

```
...
vector<Vertex> meshVertices{
    {{ 0.4f, -0.4f, 0.0f}, {1.0f, 0.0f, 0.0f}},
    {{ 0.4f,  0.4f, 0.0f}, {0.0f, 1.0f, 0.0f}},
    {{-0.4f,  0.4f, 0.0f}, {0.0f, 0.0f, 1.0f}},

    {{-0.4f,  0.4f, 0.0f}, {0.0f, 0.0f, 1.0f}},
    {{-0.4f, -0.4f, 0.0f}, {1.0f, 1.0f, 0.0f}},
    {{ 0.4f, -0.4f, 0.0f}, {1.0f, 0.0f, 0.0f}}
};
...
```

Our rectangle is now colored!

We still have to use an index buffer to avoid repeating vertices, and use the staging buffers to avoid using a visible host as we did here, because it is not optimal. We will also take profit of next paragraphs to draw multiple objects.

In the example code, the project until now is VulkanApp_Data1.

Staging buffer

Creating buffers with a function

We will first move the buffer creation logics from the mesh to the utilities.

`VulkanUtilities.h`

```
#define GLFW_INCLUDE_VULKAN
#include <GLFW/glfw3.h>
...

static uint32_t findMemoryTypeIndex(vk::PhysicalDevice physicalDevice,
        uint32_t allowedTypes, vk::MemoryPropertyFlags properties)
{
        // Get properties of physical device
        vk::PhysicalDeviceMemoryProperties memoryProperties =
                physicalDevice.getMemoryProperties();

        for (uint32_t i = 0; i < memoryProperties.memoryTypeCount; ++i)
        {
                // We iterate through each bit, shifting of 1 (with i) each time.
                // This way we go through each type to check it is allowed.
                if ((allowedTypes & (1 << i)) &&
                // Desired property bit flags are part of memory type's property flags.
                // By checking the equality, we check that all properties are available
                // at the same time, and not only one property is common.
                (memoryProperties.memoryTypes[i].propertyFlags & properties) == properties)
                {
                        // If this type is an allowed type and has the flags we want, then i
                        // is the current index of the memory type we want to use. Return it.
                        return i;
                }
        }
}

static void createBuffer(vk::PhysicalDevice physicalDevice, vk::Device device,
        vk::DeviceSize bufferSize, vk::BufferUsageFlags bufferUsage,
        vk::MemoryPropertyFlags bufferProperties, vk::Buffer* buffer,
        vk::DeviceMemory* bufferMemory)
{
        // Buffer info
        vk::BufferCreateInfo bufferInfo{};
        bufferInfo.size = bufferSize;
        // Multiple types of buffers
        bufferInfo.usage = bufferUsage;
        // Is vertex buffer sharable ? Here: no.
        bufferInfo.sharingMode = vk::SharingMode::eExclusive;

        *buffer = device.createBuffer(bufferInfo);

        // Get buffer memory requirements
        vk::MemoryRequirements memoryRequirements;
        device.getBufferMemoryRequirements(*buffer, &memoryRequirements);

        // Allocate memory to buffer
        vk::MemoryAllocateInfo memoryAllocInfo{};
        memoryAllocInfo.allocationSize = memoryRequirements.size;
        memoryAllocInfo.memoryTypeIndex = findMemoryTypeIndex(
                physicalDevice,
                // Index of memory type on physical device that has required bit flags
                memoryRequirements.memoryTypeBits,
                bufferProperties
        );

        // Allocate memory to VkDeviceMemory
        auto result = device.allocateMemory(&memoryAllocInfo, nullptr, bufferMemory);
        if (result != vk::Result::eSuccess)
        {
                throw std::runtime_error("Failed to allocate vertex buffer memory");
        }

        // Allocate memory to given vertex buffer
        device.bindBufferMemory(*buffer, *bufferMemory, 0);
}
```

We can now parameter the buffer we want thanks to this new function. Let's go back to the mesh and reimplement the buffer creation. Note that we change the mapping/unmapping code to a C++ interface:

`VulkanMesh.cpp`

```
void VulkanMesh::createVertexBuffer(vector<Vertex>* vertices)
{
        vk::DeviceSize offset{};
        vk::DeviceSize bufferSize = sizeof(Vertex) * vertices->size();
        createBuffer(physicalDevice, device, bufferSize, vk::BufferUsageFlagBits::eVertexBuffer,
                vk::MemoryPropertyFlagBits::eHostVisible | vk::MemoryPropertyFlagBits::eHostCohere
                &vertexBuffer, &vertexBufferMemory);

        // -- MAP MEMORY TO VERTEX BUFFER --
        // 1. Create pointer to a random point in memory
        void* data;
        // 2. Map the vertex buffer memory to that point
        device.mapMemory(vertexBufferMemory, {}, bufferSize, {}, &data);
        // 3. Copy memory from vertices memory to the point
        memcpy(data, vertices->data(), static_cast<size_t>(bufferSize));
        // 4. Unmap the vertex buffer memory
        device.unmapMemory(vertexBufferMemory);
        // Because we have eHostCoherent, we don't have to flush
}
```

Create a staging buffer

We can now modify the `VulkanMesh::createVertexBuffer` function to create a staging buffer and create the GPU local buffer that will receive the data from the staging buffer.

`VulkanMesh.cpp`

```
void VulkanMesh::createVertexBuffer(vector<Vertex>* vertices)
{
	vk::DeviceSize bufferSize = sizeof(Vertex) * vertices->size();

	// Temporary buffer to stage vertex data before transfering to GPU
	vk::Buffer stagingBuffer;
	vk::DeviceMemory stagingBufferMemory;

	createBuffer(physicalDevice, device, bufferSize, vk::BufferUsageFlagBits::eTransferSrc,
		vk::MemoryPropertyFlagBits::eHostVisible | vk::MemoryPropertyFlagBits::eHostCohere
		&stagingBuffer, &stagingBufferMemory);

	// Map memory to staging buffer
	void* data;
	device.mapMemory(stagingBufferMemory, {}, bufferSize, {}, &data);
	memcpy(data, vertices->data(), static_cast<size_t>(bufferSize));
	device.unmapMemory(stagingBufferMemory);

	// Create buffer with TRANSFER_DST_BIT to mark as recipient of transfer data
	// Buffer memory need to be DEVICE_LOCAL_BIT meaning memory is on GPU only
	// and not CPU-accessible
	createBuffer(physicalDevice, device, bufferSize,
		vk::BufferUsageFlagBits::eTransferDst | vk::BufferUsageFlagBits::eVertexBuffer,
		vk::MemoryPropertyFlagBits::eDeviceLocal,
		&vertexBuffer, &vertexBufferMemory);
}
```

Transfer from one buffer to the other

First, update the VulkanMesh constructor and the `createVertexBuffer` signatures.

`VulkanMesh.cpp`

```
VulkanMesh::VulkanMesh(vk::PhysicalDevice physicalDeviceP, vk::Device deviceP,
        vk::Queue transferQueue, vk::CommandPool transferCommandPool, vector<Vertex>* vertices):
        vertexCount(vertices->size()), physicalDevice(physicalDeviceP), device(deviceP)
{
        createVertexBuffer(transferQueue, transferCommandPool, vertices);
}
...
void VulkanMesh::createVertexBuffer(vk::Queue transferQueue,
        vk::CommandPool transferCommandPool, vector<Vertex>* vertices)
...
```

Then we need a utility function to copy buffers:

`VulkanUtilities.h`

```
static void copyBuffer(vk::Device device,
        vk::Queue transferQueue, vk::CommandPool transferCommandPool,
        vk::Buffer srcBuffer, vk::Buffer dstBuffer, vk::DeviceSize bufferSize)
{
        // Command buffer to hold transfer commands
        vk::CommandBuffer transferCommandBuffer;

        // Command buffer details
        vk::CommandBufferAllocateInfo allocInfo{};
        allocInfo.level = vk::CommandBufferLevel::ePrimary;
        allocInfo.commandPool = transferCommandPool;
        allocInfo.commandBufferCount = 1;

        // Allocate command buffer from pool
        transferCommandBuffer = device.allocateCommandBuffers(allocInfo).front();

        // Information to begin command buffer record
        vk::CommandBufferBeginInfo beginInfo{};
        // Only using command buffer once, then become unvalid
        beginInfo.flags = vk::CommandBufferUsageFlagBits::eOneTimeSubmit;

        // Begin records transfer commands
        transferCommandBuffer.begin(beginInfo);

        // Region of data to copy from and to
        vk::BufferCopy bufferCopyRegion{};
        bufferCopyRegion.srcOffset = 0;                // From the start of first buffer...
        bufferCopyRegion.dstOffset = 0;                // ...copy to the start of second buffer
        bufferCopyRegion.size = bufferSize;

        // Copy src buffer to dst buffer
        transferCommandBuffer.copyBuffer(srcBuffer, dstBuffer, bufferCopyRegion);

        // End record commands
        transferCommandBuffer.end();

        // Queue submission info
        vk::SubmitInfo submitInfo{};
        submitInfo.commandBufferCount = 1;
        submitInfo.pCommandBuffers = &transferCommandBuffer;

        // Submit transfer commands to transfer queue and wait until it finishes
        transferQueue.submit(submitInfo);
        transferQueue.waitIdle();

        // Free temporary command buffer
        device.freeCommandBuffers(transferCommandPool, transferCommandBuffer);
}
```

We can now use the `copyBuffer` function in the mesh's `createVertexBuffer` function:

`VulkanMesh.cpp`

```
...
void VulkanMesh::createVertexBuffer(vk::Queue transferQueue,
        vk::CommandPool transferCommandPool, vector<Vertex>* vertices)
{
    ...
        // Create buffer with vk::BufferUsageFlagBits::eTransferDst to mark as recipient
        // of transfer data Buffer memory need to be vk::MemoryPropertyFlagBits::eDeviceLocal
        // meaning memory is on GPU only and not CPU-accessible
        createBuffer(physicalDevice, device, bufferSize,
                vk::BufferUsageFlagBits::eTransferDst | vk::BufferUsageFlagBits::eVertexBuffer,
                vk::MemoryPropertyFlagBits::eDeviceLocal,
                &vertexBuffer, &vertexBufferMemory);

        // Copy staging buffer to vertex buffer on GPU
        copyBuffer(device, transferQueue, transferCommandPool,
                stagingBuffer, vertexBuffer, bufferSize);

  // Clean staging buffer
        device.destroyBuffer(stagingBuffer, nullptr);
        device.freeMemory(stagingBufferMemory, nullptr);
}
```

Also change the mesh constructor call in the renderer:

`VulkanRenderer.cpp`

```
int VulkanRenderer::init(GLFWwindow* windowP)
{
...
                firstMesh = VulkanMesh(mainDevice.physicalDevice, mainDevice.logicalDevice,
                        graphicsQueue, graphicsCommandPool, &meshVertices);
...
}
```

Now we have an optimized draw, using full capabilities of the GPU queues.

Index buffer

Preparing data for indexation

Change the vertex input data to have only 4 vertices:

`VulkanRenderer.cpp`

```
                // Objects
                // -- Vertex data
                vector<Vertex> meshVertices{
                        {{ 0.4f, -0.4f, 0.0f}, {1.0f, 0.0f, 0.0f}},        // 0
                        {{ 0.4f,  0.4f, 0.0f}, {0.0f, 1.0f, 0.0f}},        // 1
                        {{-0.4f,  0.4f, 0.0f}, {0.0f, 0.0f, 1.0f}},        // 2
                        {{-0.4f, -0.4f, 0.0f}, {1.0f, 1.0f, 0.0f}},        // 3
                };

                // -- Index data
                vector<uint32_t> meshIndices{
                        0, 1, 2,
                        2, 3, 0
                };

                firstMesh = VulkanMesh(mainDevice.physicalDevice, mainDevice.logicalDevice,
                        graphicsQueue, graphicsCommandPool, &meshVertices, &meshIndices);
```

Update the mesh

We will need new members in the mesh:

`VulkanMesh.h`

```
...
public:
    ...
    size_t getIndexCount();
                vk::Buffer getIndexBuffer();
    ...
private:
    ...
                size_t indexCount;
                vk::Buffer indexBuffer;
                vk::DeviceMemory indexBufferMemory;
    ...
    void createIndexBuffer(vk::Queue transferQueue,
                        vk::CommandPool transferCommandPool, vector<uint32_t>* indices);
```

We will update the mesh constructor, destroyer and implement the `createIndexBuffer` and the getters functions:

`VulkanMesh.cpp`

```
VulkanMesh::VulkanMesh(vk::PhysicalDevice physicalDeviceP,vk::Device deviceP,
        vk::Queue transferQueue, vk::CommandPool transferCommandPool,
        vector<Vertex>* vertices, vector<uint32_t>* indices)
        :
        vertexCount(vertices->size()), indexCount(indices->size()),
        physicalDevice(physicalDeviceP), device(deviceP)
{
        createVertexBuffer(transferQueue, transferCommandPool, vertices);
        createIndexBuffer(transferQueue, transferCommandPool, indices);
}
...
void VulkanMesh::destroyBuffers()
{
        device.destroyBuffer(vertexBuffer, nullptr);
        device.freeMemory(vertexBufferMemory, nullptr);
        device.destroyBuffer(indexBuffer, nullptr);
        device.freeMemory(indexBufferMemory, nullptr);
}
...
size_t VulkanMesh::getIndexCount()
{
        return indexCount;
}

vk::Buffer VulkanMesh::getIndexBuffer()
{
        return indexBuffer;
}
...
void VulkanMesh::createIndexBuffer(vk::Queue transferQueue,
        vk::CommandPool transferCommandPool, vector<uint32_t>* indices)
{
        vk::DeviceSize bufferSize = sizeof(uint32_t) * indices->size();

        vk::Buffer stagingBuffer;
        vk::DeviceMemory stagingBufferMemory;
        createBuffer(physicalDevice, device, bufferSize, vk::BufferUsageFlagBits::eTransferSrc,
                vk::MemoryPropertyFlagBits::eHostVisible | vk::MemoryPropertyFlagBits::eHostCohere
                &stagingBuffer, &stagingBufferMemory);
        void* data;
        device.mapMemory(stagingBufferMemory, {}, bufferSize, {}, &data);
        memcpy(data, indices->data(), static_cast<size_t>(bufferSize));
        device.unmapMemory(stagingBufferMemory);

        // This time with vk::BufferUsageFlagBits::eIndexBuffer,
        // &indexBuffer and &indexBufferMemory
        createBuffer(physicalDevice, device, bufferSize,
                vk::BufferUsageFlagBits::eTransferDst | vk::BufferUsageFlagBits::eIndexBuffer,
                vk::MemoryPropertyFlagBits::eDeviceLocal,
                &indexBuffer, &indexBufferMemory);

        // Copy to indexBuffer
        copyBuffer(device, transferQueue, transferCommandPool,
                stagingBuffer, indexBuffer, bufferSize);

        device.destroyBuffer(stagingBuffer);
        device.freeMemory(stagingBufferMemory);
}
```

Use the indices

Update to renderer's `recordCommands` function to bind the index buffer in addition to vertex buffers. We will also change the draw command to use indices.

`VulkanRenderer.cpp`

```
...
                // Bind vertex buffer
                vk::Buffer vertexBuffers[] = { firstMesh.getVertexBuffer() };
                vk::DeviceSize offsets[] = { 0 };
                commandBuffers[i].bindVertexBuffers(0, vertexBuffers, offsets);

                // Bind index buffer
                commandBuffers[i].bindIndexBuffer(firstMesh.getIndexBuffer(),
                        0, vk::IndexType::eUint32);

                // Execute pipeline
                commandBuffers[i].drawIndexed(
                        static_cast<uint32_t>(firstMesh.getIndexCount()), 1, 0, 0, 0);
...
```

Your code should now draw using indexes, passed to the GPU through a staging queue.

How to draw multiple objects ?

We could have multiple command buffers, one for each object. But there is simpler method : draw multiple objects in the render pass recorded commands.

Replace the mesh with a vector of meshes.

`VulkanRenderer.h`

```
        vector<VulkanMesh> meshes;
```

Also create new data for meshes:

```
// -- Vertex data
vector<Vertex> meshVertices1{
        {{-0.1f, -0.4f, 0.0f}, {1.0f, 0.0f, 0.0f}},        // 0
        {{-0.1f,  0.4f, 0.0f}, {0.0f, 1.0f, 0.0f}},        // 1
        {{-0.9f,  0.4f, 0.0f}, {0.0f, 0.0f, 1.0f}},        // 2
        {{-0.9f, -0.4f, 0.0f}, {1.0f, 1.0f, 0.0f}},        // 3
};

vector<Vertex> meshVertices2{
        {{ 0.9f, -0.4f, 0.0f}, {1.0f, 0.0f, 0.0f}},        // 0
        {{ 0.9f,  0.4f, 0.0f}, {0.0f, 1.0f, 0.0f}},        // 1
        {{ 0.1f,  0.4f, 0.0f}, {0.0f, 0.0f, 1.0f}},        // 2
        {{ 0.1f, -0.4f, 0.0f}, {1.0f, 1.0f, 0.0f}},        // 3
};

// -- Index data
vector<uint32_t> meshIndices{
        0, 1, 2,
        2, 3, 0
};

VulkanMesh firstMesh = VulkanMesh(mainDevice.physicalDevice,
        mainDevice.logicalDevice,graphicsQueue, graphicsCommandPool,
        &meshVertices1, &meshIndices);
VulkanMesh secondMesh = VulkanMesh(mainDevice.physicalDevice,
        mainDevice.logicalDevice, graphicsQueue, graphicsCommandPool,
        &meshVertices2, &meshIndices);
meshes.push_back(firstMesh);
meshes.push_back(secondMesh);
```

The clean up code should be updated:

```
void VulkanRenderer::clean()
{
        mainDevice.logicalDevice.waitIdle();

        for (auto& mesh : meshes)
        {
                mesh.destroyBuffers();
        }
    ...
}
```

Now we can go back to the commands recording and call multiple draw commands:

`VulkanRenderer.cpp`

```
void VulkanRenderer::recordCommands()
{
        ...
        // Bind pipeline to be used in render pass, you could switch pipelines
        // for different subpasses
        commandBuffers[i].bindPipeline(vk::PipelineBindPoint::eGraphics, graphicsPipeline);

        // Draw all meshes
        for (size_t j = 0; j < meshes.size(); ++j)
        {
                // Bind vertex buffer
                vk::Buffer vertexBuffers[] = {meshes[j].getVertexBuffer()};
                vk::DeviceSize offsets[] = {0};
                commandBuffers[i].bindVertexBuffers(0, vertexBuffers, offsets);

                // Bind index buffer
                commandBuffers[i].bindIndexBuffer(
                        meshes[j].getIndexBuffer(), 0, vk::IndexType::eUint32);

                // Execute pipeline
                commandBuffers[i].drawIndexed(
                        static_cast<uint32_t>(meshes[j].getIndexCount()), 1, 0, 0, 0);
        }

        // End render pass
        commandBuffers[i].endRenderPass();
        ...
```

You draw as much quads you want to! This is not at all optimal though, and we will study better techniques to send data to the GPU.

The code of the Vulkan application so far is given in the VulkanApp_Data2 project.

Descriptor sets and push constants

Concepts

Setup

Until now we have passed per-vertex data. But what if we need per-model data, like projection or view matrices, lighting data etc. ? This kind of data was called uniform in OpenGL. Vulkan actually gives us multiple way to it. All those ways will rely on the **PipelineLayout**, set during pipeline creation.

A **Descriptor Set** is a group of **Descriptors**, which are a piece of data shared across draw operations. We can use descriptor sets to describe multiple values being passed into a pipeline. There are several kind of descriptor sets : images, samples or "uniform" descriptor set. A **Uniform** is just a piece of data that can be read.

Before getting a descriptor set, we will need a **Layout**, to describe how set connects up to a pipeline. A pipeline can take multiple layouts to connect up multiple descriptor sets. Descriptor sets are allocated from a pool, not created. We will need to know the number of elements of the pool.

We will also need a buffer to hold the descriptor set data. Principle will be similar to the vertex buffer creation. If the data will change frequently, such as with a model matrix, the buffer should be host visible : creating and destroying staging buffers every time data updates would creates a lot of overhead. If data is largely static, such as in a texture, we will continue using staging buffers.

When everything is prepared, we will allocate the descriptor set from **Descriptor Pool**. We will then bond the descriptor set to a buffer. We will require a `VkWriteDescriptorSet` (which part of the descriptor set to bind) and a `VkDescriptorBufferInfo` that will describes which buffer to bind to the descriptor set. The `vkUpdateDescriptorSets` functions will execute binding.

To use the descriptor set, we'll use `vkCmdBindDescriptorSets` in the command buffer, that will bind descriptor sets to the pipeline. Shaders will now have access to bound descriptor sets.

Using descriptor sets in shaders

Unlike OpenGL, Vulkan uniforms can only be passed across as **Uniform Buffer Objects**, not as individual values.

```
layout(set = 0, binding = 0) uniform exampleObject {
        mat4 exampleValue
} exempleName;
```

This is quite similar to C struct. The keyword `uniform` defines the uniform value. The `binding` must match the binding in `VkWriteDescriptorSet`. The `set` allow us to choose between multiple sets ; `set = 0` is the default value, if the set is not indicated.

There is a issue though : UBO are limited in count. We can only create a small amount before Vulkan restricts creation. The **Dynamic Uniform Buffers** will allow us holding multiple uniform buffers in a large block of memory. We will iterate over data with each draw. We can use one dynamic buffer for multiple objects. E.g. all of model matrices of all objects in the same dynamic uniform.

Be careful though: dynamic uniform iteration works using minimum data offset. We will need to align data to this value. We will use bitwise operations to find out if our data is smaller than the minimal alignment.

- If it is smaller, we will use the minimul alignment;
- If it is larger, we will use the minimal alignment times the number of blocks of minimum alignment the data covers.

The creation of a dynamic uniform buffer is largely the same as a regular uniform buffer, except that memory must be created with this new alignment. We will use the C function `_aligned_malloc` and `_aligned_free`, which allocate/free a block of memory and also align the memory blocks. Then, when copying data to buffer, it will need to be done by offseting memory adresses with the previously calculated alignment.

(For linux, you can use `aligned_alloc` and `free`.)

The buffer type will be VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER_DYNAMIC. Binding the Dynamic Uniform Buffer requieres usage of the last two arguments in `vkCmdBindDescriptorSets` which describe the offsets of each uniform value in the buffer.

Push constants

Dynamic uniform buffers are good for sending a lot of data across multiple draws (e.g. transforms of hundreds of spaceships), but if we have to update this data every frame, the number of memory operations will slow the rendering. The solution are **Push Constants**.

Push constants are value that are pushed to the pipeline directly by the command buffer. The pushes are quick : we don't require buffers or memory operations. Nevertheless, they are limited in size, usually 128 bytes. It is fine for model matrices though. Other limit : each shader can have only a single push constant. Worse : we will need to re-record the command buffer every frame, which is fortunatly very quick.

Push constants are easy to setup:

- Create a **Push Constant Range** to describe the data layout
- Pass the push constant range to the pipeline layout
- In the command buffer, pass the desired values to the pipeline with `vkCmdPushConstants`

Summary: pass data to draw

We can use different way to pass draw data:

- **Uniform buffers**: rarely changing values that are the same across all draw operations. E.g. projection matrix ;
- **Dynamic uniform buffers**: rarely changing values that differ across each draw operations ;
- **Push constants**: frequently changing data, same or different across each draw operations.

Choosing the right one for the right operation is one Vulkan way to optimize GPU usage.

There are other types of descriptor sets, like the texture sampler, but we will start with those three.

Description sets : uniform buffers

For now our program displays two squares, that look like rectangles because there is no projection matrix.

Data preparation

We will create a MVP struct in the renderer. Also create a MVP field in the VulkanRenderer class.

`VulkanRenderer.h`

```
struct MVP {
        glm::mat4 projection;
        glm::mat4 view;
        glm::mat4 model;
};
...
        MVP mvp;
```

Change the vertex shader to add uniform buffer object support:

`shader.vert`

```
#version 450

// From vertex input stage
layout(location = 0) in vec3 pos;
layout(location = 1) in vec3 col;

// Uniform Buffer Object
// We pass a full object, not just a value like in OpenGL
layout(binding = 0) uniform MVP {
    mat4 projection;
    mat4 view;
    mat4 model;
} mvp;

// To fragment shader
layout(location = 0) out vec3 fragColor;

void main() {
    gl_Position = mvp.projection * mvp.view * mvp.model * vec4(pos, 1.0);
    fragColor = col;
}
```

Descriptor set layout

We need to describe the data layout of the descriptor set. Create a new function and a new member variable:

`VulkanRenderer.h`

```
        vk::DescriptorSetLayout descriptorSetLayout;
        void createDescriptorSetLayout();
```

This function will be called between `createRenderPass` and createGraphicsPipeline.

```
        ...
        createRenderPass();
        createDescriptorSetLayout();
        createGraphicsPipeline();
        ...
```

Do not forget to clean the future descriptor set:

```
void VulkanRenderer::clean()
{
        mainDevice.logicalDevice.waitIdle();

        mainDevice.logicalDevice.destroyDescriptorSetLayout(descriptorSetLayout);
        ...
```

We can now implement the creation of the descriptor set layout:

```
void VulkanRenderer::createDescriptorSetLayout()
{
        // MVP binding information
        vk::DescriptorSetLayoutBinding mvpLayoutBinding;
        // Binding number in shader
        mvpLayoutBinding.binding = 0;
        // Type of descriptor (uniform, dynamic uniform, samples...)
        mvpLayoutBinding.descriptorType = vk::DescriptorType::eUniformBuffer;
        // Number of descriptors for binding
        mvpLayoutBinding.descriptorCount = 1;
        // Shader stage to bind to (here: vertex shader)
        mvpLayoutBinding.stageFlags = vk::ShaderStageFlagBits::eVertex;
        // For textures : can make sample data un changeable
        mvpLayoutBinding.pImmutableSamplers = nullptr;

        // Descriptor set layout with given binding
        vk::DescriptorSetLayoutCreateInfo layoutCreateInfo{};
        layoutCreateInfo.bindingCount = 1;
        layoutCreateInfo.pBindings = &mvpLayoutBinding;

        // Create descriptor set layout
        descriptorSetLayout = mainDevice.logicalDevice.createDescriptorSetLayout(layoutCreateInfo)
}
```

Connecting descriptor set to the pipeline

Now we will connect the descriptor set layout to the pipeline. Go to `createGraphicsPipeline` and take care of the TODO:

```
        // -- PIPELINE LAYOUT --
        // TODO: apply future descriptorset layout
        vk::PipelineLayoutCreateInfo pipelineLayoutCreateInfo{};
        pipelineLayoutCreateInfo.setLayoutCount = 1;
        pipelineLayoutCreateInfo.pSetLayouts = &descriptorSetLayout;
```

Buffers for the uniforms

Create a vector of buffer member variable. We need a buffer because the same time we will bind our vertex buffer, we will bind our descriptor set. To avoid collisions, we will have one uniform buffer for each swapchain image. We will also need memory, and a function to create the uniform buffers.

`VulkanRenderer.h`

```
        vector<vk::Buffer> uniformBuffer;
        vector<vk::DeviceMemory> uniformBufferMemory;
        void createUniformBuffers();
```

Use this function just before the command buffers creation:

`VulkanRenderer.cpp`

```
int VulkanRenderer::init(GLFWwindow* windowP)
{
        ...
        // Data
        createUniformBuffers();

        // Commands
        createGraphicsCommandBuffers();
        ...
}
```

We can now implement the creation function. Do not forget to clean the buffers we create.

`VulkanRenderer.cpp`

```
void VulkanRenderer::clean()
{
        mainDevice.logicalDevice.waitIdle();

        mainDevice.logicalDevice.destroyDescriptorSetLayout(descriptorSetLayout);
        for (size_t i = 0; i < uniformBuffer.size(); ++i)
        {
                mainDevice.logicalDevice.destroyBuffer(uniformBuffer[i]);
                mainDevice.logicalDevice.freeMemory(uniformBufferMemory[i]);
        }
        ...
}
...

void VulkanRenderer::createUniformBuffers()
{
        // Buffer size will be size of all 3 variables
        vk::DeviceSize bufferSize = sizeof(MVP);

        // One uniform buffer for each image / each command buffer
        uniformBuffer.resize(swapchainImages.size());
        uniformBufferMemory.resize(swapchainImages.size());

        // Create uniform buffers
        for (size_t i = 0; i < swapchainImages.size(); ++i)
        {
                createBuffer(mainDevice.physicalDevice, mainDevice.logicalDevice, bufferSize,
                vk::BufferUsageFlagBits::eUniformBuffer,
                vk::MemoryPropertyFlagBits::eHostVisible | vk::MemoryPropertyFlagBits::eHostCohere
                &uniformBuffer[i], &uniformBufferMemory[i]);
        }
}
```

Pool for the descriptor set

Now we need a pool for our descriptor set data.

`VulkanRenderer.h`

```
        vk::DescriptorPool descriptorPool;
        void createDescriptorPool();
```

`VulkanRenderer.cpp`

```
int VulkanRenderer::init(GLFWwindow* windowP)
{
        ...
        // Data
        createUniformBuffers();
        createDescriptorPool();

        // Commands
        createGraphicsCommandBuffers();
        ...
}
```

`VulkanRenderer.cpp`

```
void VulkanRenderer::clean()
{
        mainDevice.logicalDevice.waitIdle();

        mainDevice.logicalDevice.destroyDescriptorPool(descriptorPool);
        ...
}

void VulkanRenderer::createDescriptorPool()
{
        // One descriptor in the pool for each image
        vk::DescriptorPoolSize poolSize{};
        poolSize.descriptorCount = static_cast<uint32_t>(uniformBuffer.size());

        // One descriptor set that contains one descriptor
        vk::DescriptorPoolCreateInfo poolCreateInfo{};
        poolCreateInfo.maxSets = static_cast<uint32_t>(uniformBuffer.size());
        poolCreateInfo.poolSizeCount = 1;
        poolCreateInfo.pPoolSizes = &poolSize;

        // Create pool
        descriptorPool = mainDevice.logicalDevice.createDescriptorPool(poolCreateInfo);
}
```

The descriptor set

We now have the info to allocate descriptor sets.

`VulkanRenderer.h`

```
        vector<vk::DescriptorSet> descriptorSets;
        void createDescriptorSets();
```

`VulkanRenderer.cpp`

```
void VulkanRenderer::createDescriptorSets()
{
        // One descriptor set for every image/buffer
        descriptorSets.resize(uniformBuffer.size());

        // We want the same layout for the right number of descriptor sets
        vector<vk::DescriptorSetLayout> setLayouts(uniformBuffer.size(), descriptorSetLayout);

        // Allocation from the pool
        vk::DescriptorSetAllocateInfo setAllocInfo{};
        setAllocInfo.descriptorPool = descriptorPool;
        setAllocInfo.descriptorSetCount = static_cast<uint32_t>(uniformBuffer.size());
        setAllocInfo.pSetLayouts = setLayouts.data();

        // Allocate multiple descriptor sets
        mainDevice.logicalDevice.allocateDescriptorSets(&setAllocInfo, descriptorSets.data());

        // We have a connection between descriptor set layouts and descriptor sets,
        // but we don't know how link descriptor sets and the uniform buffers.

        // Update all of descriptor set buffer bindings
        for (size_t i = 0; i < uniformBuffer.size(); ++i)
        {
                // Description of the buffer and data offset
                vk::DescriptorBufferInfo mvpBufferInfo{};
                // Buffer to get data from
                mvpBufferInfo.buffer = uniformBuffer[i];
                // We bind the whole data
                mvpBufferInfo.offset = 0;
                // Size of data
                mvpBufferInfo.range = sizeof(MVP);

                // Data about connection between binding and buffer
                vk::WriteDescriptorSet mvpSetWrite{};
                // Descriptor sets to update
                mvpSetWrite.dstSet = descriptorSets[i];
                // Binding to update (matches with shader binding)
                mvpSetWrite.dstBinding = 0;
                // Index in array to update
                mvpSetWrite.dstArrayElement = 0;
                mvpSetWrite.descriptorType = vk::DescriptorType::eUniformBuffer;
                // Amount of descriptor sets to update
                mvpSetWrite.descriptorCount = 1;
                // Information about buffer data to bind
                mvpSetWrite.pBufferInfo = &mvpBufferInfo;

                // Update descriptor set with new buffer/binding info
                mainDevice.logicalDevice.updateDescriptorSets(1, & mvpSetWrite, 0, nullptr);
        }
}
```

Do not forget to call the function in `init`:

```
                // Data
                createUniformBuffers();
                createDescriptorPool();
                createDescriptorSets();
                ...
```

Use descriptor sets when we record commands

```
void VulkanRenderer::recordCommands()
{
        ...
        // Bind index buffer
        commandBuffers[i].bindIndexBuffer(
                meshes[j].getIndexBuffer(), 0, vk::IndexType::eUint32);

        // Bind descriptor sets
        commandBuffers[i].bindDescriptorSets(vk::PipelineBindPoint::eGraphics,
                pipelineLayout, 0, 1, &descriptorSets[i], 0, nullptr);

        // Execute pipeline
        commandBuffers[i].drawIndexed(
                static_cast<uint32_t>(meshes[j].getIndexCount()), 1, 0, 0, 0);
        ...
}
```

Set values for the shaders

We need to send the matrices to the shaders, now everything is connected.

Include this in the VulkanRenderer:

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

Before creating the mesh, let's create our matrices.

`VulkanRenderer.cpp`

```
int VulkanRenderer::init(GLFWwindow* windowP)
{
        ...
        // Objects
        float aspectRatio = static_cast<float>(swapchainExtent.width) /
                        static_cast<float>(swapchainExtent.height);
        mvp.projection = glm::perspective(glm::radians(45.0f), aspectRatio, 0.1f, 100.0f);
        mvp.view = glm::lookAt(glm::vec3(3.0f, 1.0f, 2.0f),
                glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
        mvp.model = glm::mat4(1.0f);

        // In vulkan, y is downward, and for glm it is upward
        mvp.projection[1][1] *= -1;

        // -- Vertex data
        ...
}
```

Because we have inverted projection axies for vulkan, we have to update the rasterizer configuration in `createGraphicsPipeline`:

```
...
        // Widing to know the front face of a polygon
        rasterizerCreateInfo.frontFace = vk::FrontFace::eCounterClockwise;
...
```

We want to update the uniform buffers for each draw. Let's use a new `updateUniformBuffer` function.

```
void VulkanRenderer::updateUniformBuffer(uint32_t imageIndex)
{
        void* data;
        mainDevice.logicalDevice.mapMemory(
                uniformBufferMemory[imageIndex], {}, sizeof(MVP), {}, &data);
        memcpy(data, &mvp, sizeof(MVP));
        mainDevice.logicalDevice.unmapMemory(uniformBufferMemory[imageIndex]);
}
```

We will call this function in the `draw` function, between the 1. and 2. steps:

```
void VulkanRenderer::draw()
{
        ...
        // 1. Get next available image to draw and set a semaphore to signal
        // when we're finished with the image.
        uint32_t imageToBeDrawnIndex =
                (mainDevice.logicalDevice.acquireNextImageKHR(swapchain,
                std::numeric_limits<uint32_t>::max(),
                imageAvailable[currentFrame], VK_NULL_HANDLE)
        ).value;

        updateUniformBuffer(imageToBeDrawnIndex);

        // 2. Submit command buffer to queue for execution, make sure it waits
        // for the image to be signaled as available before drawing, and
        // signals when it has finished rendering.
        ...
```

Now your program should show your two quads from a side view. If you change the view matrix, it should change the view angle. (Do not forget to recompile your shaders!)

We will add some movement.

Updating the model matrix

Add a `VulkanRenderer::updateModel(glm::mat4 modelP)` public function.

```
void VulkanRenderer::updateModel(glm::mat4 modelP)
{
        mvp.model = modelP;
}
```

Call it from the main:

`Main.cpp`

```
int main()
{
        initWindow();
        if(vulkanRenderer.init(window) == EXIT_FAILURE) return EXIT_FAILURE;

        float angle = 0.0f;
        float deltaTime = 0.0f;
        float lastTime = 0.0f;

        while (!glfwWindowShouldClose(window))
        {
                glfwPollEvents();

                float now = glfwGetTime();
                deltaTime = now - lastTime;
                lastTime = now;

                angle += 10.0 * deltaTime;
                if (angle > 360.0f) { angle -= 360.0f; }

                vulkanRenderer.updateModel(glm::rotate(glm::mat4(1.0f),
                        glm::radians(angle), glm::vec3(0.0f, 0.0f, 1.0f)));

                vulkanRenderer.draw();
        }

        clean();
        return 0;
}
```

Now the quads will rotate. This project is available in the VulkanApp_Data3 folder.

The problem is quads rotate as a whole, and not individually. It is because we use the same descriptor set, which is based on the image, on the command buffer, and not on the object. We could create a lot of descriptor sets, but descriptor set number has a limit. If you add a breakpoint after the physical device retrivial and go to the limit / maxDescriptorSetUniformBuffer, you will see this maximum number.

It is too low to use a uniform buffer for each object in real game situation. That is why we will have to use dynamic uniform buffers.

Description sets : dynamic uniform buffers

Data preparation

Each frame, the model matrix will change. We'll extract it from the MVP struct and put it into the mesh class.

`VulkanMesh.h`

```
struct Model {
	glm::mat4 model;
};

class VulkanMesh
{
public:
	...
	Model getModel() const { return model; }
	void setModel(const glm::mat4& modelP) { model.model = modelP; }
	...
private:
	...
	Model model;
	...
```

Also set `model.model = glm::mat4(1.0f);` in the mesh constructor.

We can change the MVP data structure to:

`VulkanRenderer.h`

```
struct ViewProjection
{
        glm::mat4 projection;
        glm::mat4 view;
};
...
        ViewProjection viewProjection;
```

Update the names and remove the lines that modify the model and have no meaning anymore.

Update the vertex shader:

`shader1.vert`

```
#version 450

// From vertex input stage
layout(location = 0) in vec3 pos;
layout(location = 1) in vec3 col;

// Uniform Buffer Object
layout(binding = 0) uniform ViewProjection {
    mat4 projection;
    mat4 view;
} viewProjection;

// Dynamic uniform buffer
layout(binding = 1) uniform Model {
    mat4 model;
} model;

// To fragment shader
layout(location = 0) out vec3 fragColor;

void main() {
    gl_Position = viewProjection.projection * viewProjection.view *
                model.model * vec4(pos, 1.0);
    fragColor = col;
}
```

Models alignement in memory

We will need a big raw array of model matrices. As stated in the introduction, dynamic uniform buffers have to respect a certain alignment.

To check your alignement, temporarily comment out a line in `checkDeviceSuitable` and put a breakpoint here:

`VulkanRenderer.cpp`

```
bool VulkanRenderer::checkDeviceSuitable(VkPhysicalDevice device)
{
        // Information about the device itself (ID, name, type, vendor, etc.)
        vk::PhysicalDeviceProperties deviceProperties = device.getProperties();


        // Information about what the device can do (geom shader, tesselation, wide lines...)
        //vk::PhysicalDeviceFeatures deviceFeatures = device.getFeatures();
                // COMMENTED OUT

        // For now we do nothing with this info

        QueueFamilyIndices indices = getQueueFamilies(device);                // HERE
        ...
```

...and check the `deviceProperties.minUniformBufferOffsetAlignment` value. In this course, I will choose 64, which is the value on my computer.

We will store the required values for alignment in the VulkanRenderer class.

`VulkanRenderer.h`

```
        VkDeviceSize minUniformBufferOffet;
        size_t modelUniformAlignement;
        UboModel* modelTransferSpace;
```

Update `getPhysicalDevice` to store the minimum alignement property:

`VulkanRenderer.cpp`

```
void VulkanRenderer::getPhysicalDevice()
{
        ...
        // Get properties of our new device to know some values
        vk::PhysicalDeviceProperties deviceProperties = mainDevice.physicalDevice.getProperties();
        minUniformBufferOffet = deviceProperties.limits.minUniformBufferOffsetAlignment;
}
```

We will use a new function to calculate the other value.

Because we want to allocate memory in function of the size of Model multiplied by a certain number of objects, we have to setup this max number of objects.

`VulkanRenderer.h`

```
const int MAX_OBJECTS = 2;
```

We can now find the `modelUniformAlignement` and allocate memory.

`VulkanRenderer.cpp`

```
void VulkanRenderer::allocateDynamicBufferTransferSpace()
{
        // modelUniformAlignement = sizeof(Model) & ~(minUniformBufferOffet - 1);

        // We take the size of Model and we compare its size to a mask.
        // ~(minUniformBufferOffet - 1) is the inverse of minUniformBufferOffet
        // Example with a 16bits alignment coded on 8 bits:
        //   00010000 - 1  == 00001111
        // ~(00010000 - 1) == 11110000 which is our mask.
        // If we imagine our UboModel is 64 bits (01000000)
        // and the minUniformBufferOffet 16 bits (00010000),
        // (01000000) & ~(00010000 - 1) == 01000000 & 11110000 == 01000000
        // Our alignment will need to be 64 bits.

        // However this calculation is not perfect.

        // Let's now imagine our UboModel is 66 bits : 01000010.
        // The above calculation would give us a 64 bits alignment,
        // whereas we would need a 80 bits (01010000 = 64 + 16) alignment.

        // We need to add to the size minUniformBufferOffet - 1 to shield against this effect
        modelUniformAlignement =
                (sizeof(Model) + minUniformBufferOffet - 1) & ~(minUniformBufferOffet - 1);

        // We will now allocate memory for models.
        modelTransferSpace = (Model*)_aligned_malloc(
                modelUniformAlignement * MAX_OBJECTS, modelUniformAlignement
        );
}
```

We now have to free the allocated memory in the cleaup function.

```
void VulkanRenderer::clean()
{
        mainDevice.logicalDevice.waitIdle();

        _aligned_free(modelTransferSpace);
        ...
}
```

Updating the descriptor set layout

First change `mvpLayoutBinding` to `vpLayoutBinding` in `VulkanRenderer::createDescriptorSetLayout`, then add a second binding:

```
void VulkanRenderer::createDescriptorSetLayout()
{
        // ViewProjection binding information
        vk::DescriptorSetLayoutBinding vpLayoutBinding;
        // Binding number in shader
        vpLayoutBinding.binding = 0;
        // Type of descriptor (uniform, dynamic uniform, samples...)
        vpLayoutBinding.descriptorType = vk::DescriptorType::eUniformBuffer;
        // Number of descriptors for binding
        vpLayoutBinding.descriptorCount = 1;
        // Shader stage to bind to (here: vertex shader)
        vpLayoutBinding.stageFlags = vk::ShaderStageFlagBits::eVertex;
        // For textures : can make sample data un changeable
        vpLayoutBinding.pImmutableSamplers = nullptr;

        // Model
        vk::DescriptorSetLayoutBinding modelLayoutBinding;
        modelLayoutBinding.binding = 1;
        modelLayoutBinding.descriptorType = vk::DescriptorType::eUniformBufferDynamic;
        modelLayoutBinding.descriptorCount = 1;
        modelLayoutBinding.stageFlags = vk::ShaderStageFlagBits::eVertex;
        modelLayoutBinding.pImmutableSamplers = nullptr;

        vector<vk::DescriptorSetLayoutBinding> layoutBindings {
                vpLayoutBinding, modelLayoutBinding
        };

        // Descriptor set layout with given binding
        vk::DescriptorSetLayoutCreateInfo layoutCreateInfo{};
        layoutCreateInfo.bindingCount = static_cast<uint32_t>(layoutBindings.size());
        layoutCreateInfo.pBindings = layoutBindings.data();

        // Create descriptor set layout
        ...
}
```

Updating the buffers

We will now have two buffers one uniform buffer and one dynamic uniform buffer. Rename `uniformBuffer` and `uniformBufferMemory` to `vpUniformBuffer` and `vpUniformBufferMemory`. Also add the buffer and memory for the dynamic buffer:

`VulkanRenderer.h`

```
        vector<vk::Buffer> modelUniformBufferDynamic;
        vector<vk::DeviceMemory> modelUniformBufferMemoryDynamic;
```

`VulkanRenderer.cpp`

```
void VulkanRenderer::createUniformBuffers()
{
        // Buffer size will be size of all 3 variables
        vk::DeviceSize vpBufferSize = sizeof(ViewProjection);

        // Model buffer size
        vk::DeviceSize modelBufferSize = modelUniformAlignement * MAX_OBJECTS;

        // One uniform buffer for each image / each command buffer
        vpUniformBuffer.resize(swapchainImages.size());
        vpUniformBufferMemory.resize(swapchainImages.size());
        modelUniformBufferDynamic.resize(swapchainImages.size());
        modelUniformBufferMemoryDynamic.resize(swapchainImages.size());

        // Create uniform buffers
        for (size_t i = 0; i < swapchainImages.size(); ++i)
        {
                createBuffer(mainDevice.physicalDevice, mainDevice.logicalDevice, vpBufferSize,
                        vk::BufferUsageFlagBits::eUniformBuffer,
                        vk::MemoryPropertyFlagBits::eHostVisible
                                | vk::MemoryPropertyFlagBits::eHostCoherent,
                        &vpUniformBuffer[i], &vpUniformBufferMemory[i]);

                createBuffer(mainDevice.physicalDevice, mainDevice.logicalDevice, modelBufferSize,
                        vk::BufferUsageFlagBits::eUniformBuffer,
                        vk::MemoryPropertyFlagBits::eHostVisible
                                | vk::MemoryPropertyFlagBits::eHostCoherent,
                        &modelUniformBufferDynamic[i], &modelUniformBufferMemoryDynamic[i]);
        }
}
```

Updating the descriptor pool

`VulkanRenderer.cpp`

```
void VulkanRenderer::createDescriptorPool()
{
        // One descriptor in the pool for each image

        // View projection pool
        vk::DescriptorPoolSize vpPoolSize{};
        vpPoolSize.descriptorCount = static_cast<uint32_t>(vpUniformBuffer.size());

        // Model pool
        vk::DescriptorPoolSize modelPoolSize{};
        modelPoolSize.descriptorCount = static_cast<uint32_t>(modelUniformBufferDynamic.size());

        vector<vk::DescriptorPoolSize> poolSizes{ vpPoolSize, modelPoolSize };

        // One descriptor set that contains one descriptor
        vk::DescriptorPoolCreateInfo poolCreateInfo{};
        poolCreateInfo.maxSets = static_cast<uint32_t>(swapchainImages.size());
        poolCreateInfo.poolSizeCount = static_cast<uint32_t>(poolSizes.size());
        poolCreateInfo.pPoolSizes = poolSizes.data();

        // Create pool
        descriptorPool = mainDevice.logicalDevice.createDescriptorPool(poolCreateInfo);
}
```

Updating the descriptor sets creation

`VulkanRenderer.cpp`

```
void VulkanRenderer::createDescriptorSets()
{
        // One descriptor set for every image/buffer
        descriptorSets.resize(swapchainImages.size());

        // We want the same layout for the right number of descriptor sets
        vector<vk::DescriptorSetLayout> setLayouts(swapchainImages.size(), descriptorSetLayout);

        // Allocation from the pool
        vk::DescriptorSetAllocateInfo setAllocInfo{};
        setAllocInfo.descriptorPool = descriptorPool;
        setAllocInfo.descriptorSetCount = static_cast<uint32_t>(swapchainImages.size());
        setAllocInfo.pSetLayouts = setLayouts.data();

        // Allocate multiple descriptor sets
        mainDevice.logicalDevice.allocateDescriptorSets(&setAllocInfo, descriptorSets.data());

        // We have a connection between descriptor set layouts and descriptor sets,
        // but we don't know how link descriptor sets and the uniform buffers.

        // Update all of descriptor set buffer bindings
        for (size_t i = 0; i < vpUniformBuffer.size(); ++i)
        {
                // -- VIEW PROJECTION DESCRIPTOR --
                // Description of the buffer and data offset
                vk::DescriptorBufferInfo vpBufferInfo{};
                // Buffer to get data from
                vpBufferInfo.buffer = vpUniformBuffer[i];
                // We bind the whole data
                vpBufferInfo.offset = 0;
                // Size of data
                vpBufferInfo.range = sizeof(ViewProjection);

                // Data about connection between binding and buffer
                vk::WriteDescriptorSet vpSetWrite{};
                // Descriptor sets to update
                vpSetWrite.dstSet = descriptorSets[i];
                // Binding to update (matches with shader binding)
                vpSetWrite.dstBinding = 0;
```

```
                  // Index in array to update
                  vpSetWrite.dstArrayElement = 0;
                  vpSetWrite.descriptorType = vk::DescriptorType::eUniformBuffer;
                  // Amount of descriptor sets to update
                  vpSetWrite.descriptorCount = 1;
                  // Information about buffer data to bind
                  vpSetWrite.pBufferInfo = &vpBufferInfo;

                  // -- MODEL DESCRIPTOR --
                  // Description of the buffer and data offset
                  vk::DescriptorBufferInfo modelBufferInfo{};
                  modelBufferInfo.buffer = modelUniformBufferDynamic[i];
                  modelBufferInfo.offset = 0;
                  modelBufferInfo.range = modelUniformAlignement;

                  // Data about connection between binding and buffer
                  vk::WriteDescriptorSet modelSetWrite{};
                  modelSetWrite.dstSet = descriptorSets[i];
                  modelSetWrite.dstBinding = 1;
                  modelSetWrite.dstArrayElement = 0;
                  modelSetWrite.descriptorType = vk::DescriptorType::eUniformBufferDynamic;
                  modelSetWrite.descriptorCount = 1;
                  modelSetWrite.pBufferInfo = &modelBufferInfo;

                  // Descriptor set writes vector
                  vector<vk::WriteDescriptorSet> setWrites{ vpSetWrite, modelSetWrite };

                  // Update descriptor set with new buffer/binding info
                  mainDevice.logicalDevice.updateDescriptorSets(
                          static_cast<uint32_t>(setWrites.size()), setWrites.data(), 0, nullptr);
        }
}
```

Updating record commands

We have to state we will use a dynamic uniform buffer.

`VulkanRenderer.cpp`

```
void VulkanRenderer::recordCommands()
{
        ...
        // Dynamic offet amount
        uint32_t dynamicOffset = static_cast<uint32_t>(modelUniformAlignement) * j;

        // Bind descriptor sets
        commandBuffers[i].bindDescriptorSets(vk::PipelineBindPoint::eGraphics,
                pipelineLayout, 0, 1, &descriptorSets[i], 1, &dynamicOffset);

        // Execute pipeline
...
}
```

Updating uniform buffers

Change the `updateUniformBuffer` function name to `updateUniformBuffers`.

`VulkanRenderer.cpp`

```
void VulkanRenderer::updateUniformBuffers(uint32_t imageIndex)
{
        // Copy view projection data
        void* data;
        mainDevice.logicalDevice.mapMemory(
                vpUniformBufferMemory[imageIndex], {},
                sizeof(ViewProjection), {}, &data);
        memcpy(data, &viewProjection, sizeof(ViewProjection));
        mainDevice.logicalDevice.unmapMemory(vpUniformBufferMemory[imageIndex]);

        // Copy model data
        for (size_t i = 0; i < meshes.size(); ++i)
        {
                // Get a pointer to model in modelTransferSpace
                Model* model =
                        (Model*)((uint64_t)modelTransferSpace + i * modelUniformAlignement);
                // Copy data at this adress
                *model = meshes[i].getModel();
        }
        mainDevice.logicalDevice.mapMemory(
                modelUniformBufferMemoryDynamic[imageIndex], {},
                modelUniformAlignement * meshes.size(), {}, &data);
        memcpy(data, &modelTransferSpace, modelUniformAlignement * meshes.size());
        mainDevice.logicalDevice.unmapMemory(modelUniformBufferMemoryDynamic[imageIndex]);
}
```

Updating the clean up

`VulkanRenderer.cpp`

```
void VulkanRenderer::clean()
{
	mainDevice.logicalDevice.waitIdle();

	_aligned_free(modelTransferSpace);
	mainDevice.logicalDevice.destroyDescriptorPool(descriptorPool);
	mainDevice.logicalDevice.destroyDescriptorSetLayout(descriptorSetLayout);
	for (size_t i = 0; i < swapchainImages.size(); ++i)
	{
		mainDevice.logicalDevice.destroyBuffer(vpUniformBuffer[i]);
		mainDevice.logicalDevice.freeMemory(vpUniformBufferMemory[i]);
		mainDevice.logicalDevice.destroyBuffer(modelUniformBufferDynamic[i]);
		mainDevice.logicalDevice.freeMemory(modelUniformBufferMemoryDynamic[i]);
	}
	...
```

Setup the new data

We can now set some test data and call the `allocateDynamicBufferTransferSpace` function.

`VulkanRenderer.cpp`

```
int VulkanRenderer::init(GLFWwindow* windowP)
{
        window = windowP;
        try
        {
                // Objects
                ...
                VulkanMesh firstMesh =
                        VulkanMesh(mainDevice.physicalDevice, mainDevice.logicalDevice,
                        graphicsQueue, graphicsCommandPool, &meshVertices1, &meshIndices);
                VulkanMesh secondMesh =
                        VulkanMesh(mainDevice.physicalDevice, mainDevice.logicalDevice,
                        graphicsQueue, graphicsCommandPool, &meshVertices2, &meshIndices);
                meshes.push_back(firstMesh);
                meshes.push_back(secondMesh);

                glm::mat4 meshModelMatrix = meshes[0].getModel().model;
                meshModelMatrix = glm::rotate(meshModelMatrix, glm::radians(45.0f),
                        glm::vec3(0.0f, 0.0f, 1.0f));
                meshes[1].setModel(meshModelMatrix);

                // Data
                allocateDynamicBufferTransferSpace();        // NEW
                createUniformBuffers();
                createDescriptorPool();
                createDescriptorSets();

                // Commands
                createGraphicsCommandBuffers();
                recordCommands();
                createSynchronisation();
        }
        ...
}
```

Now, one of the squares should have rotated alone. This is just test code, and rotation still happen in a wierd way. Let's do proper 3D geometry.

Proper transformations

First, change the vertices we have:

```
                  vector<Vertex> meshVertices1{
                          {{-0.4f,  0.4f, 0.0f}, {1.0f, 0.0f, 0.0f}},        // 0
                          {{-0.4f, -0.4f, 0.0f}, {0.0f, 1.0f, 0.0f}},        // 1
                          {{ 0.4f, -0.4f, 0.0f}, {0.0f, 0.0f, 1.0f}},        // 2
                          {{ 0.4f,  0.4f, 0.0f}, {1.0f, 1.0f, 0.0f}},        // 3
                  };

                  vector<Vertex> meshVertices2{
                          {{-0.2f,  0.6f, 0.0f}, {1.0f, 0.0f, 0.0f}},        // 0
                          {{-0.2f, -0.6f, 0.0f}, {0.0f, 1.0f, 0.0f}},        // 1
                          {{ 0.2f, -0.6f, 0.0f}, {0.0f, 0.0f, 1.0f}},        // 2
                          {{ 0.2f,  0.6f, 0.0f}, {1.0f, 1.0f, 0.0f}},        // 3
                  };
```

Change the `updateModel` function:

```
void VulkanRenderer::updateModel(int modelId, glm::mat4 modelP)
{
        if (modelId >= meshes.size()) return;

        meshes[modelId].setModel(modelP);
}
```

Change the view matric and remove the test code from `init`:

```
...
// CHANGE THIS
                ...
                viewProjection.view = glm::lookAt(glm::vec3(0.0f, 1.0f, 2.0f),
                        glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
                ...

/*  AND        REMOVE THIS
                glm::mat4 meshModelMatrix = meshes[0].getModel().model;
                meshModelMatrix =
                        glm::rotate(meshModelMatrix, glm::radians(45.0f), glm::vec3(0.0f, 0.0f, 1.
                meshes[1].setModel(meshModelMatrix);
        END REMOVE
*/
```

Now update `main` to use update model and place properly out two squares:

```
...
while (!glfwWindowShouldClose(window))
{
        glfwPollEvents();

        float now = glfwGetTime();
        deltaTime = now - lastTime;
        lastTime = now;

        angle += 10.0 * deltaTime;
        if (angle > 360.0f) { angle -= 360.0f; }

        glm::mat4 firstModel(1.0f);
        glm::mat4 secondModel(1.0f);

        firstModel = glm::translate(firstModel, glm::vec3(-2.0f, 0.0f, -5.0f));
        firstModel = glm::rotate(firstModel, glm::radians(angle), glm::vec3(0.0f, 0.0f, 1.0f));

        secondModel = glm::translate(secondModel, glm::vec3(2.0f, 0.0f, -5.0f));
        secondModel =
                glm::rotate(secondModel, glm::radians(-angle * 100), glm::vec3(0.0f, 0.0f, 1.0f));

        vulkanRenderer.updateModel(0, firstModel);
        vulkanRenderer.updateModel(1, secondModel);

        vulkanRenderer.draw();
}
...
```

Now the shapes should rotate independantly ! This project is set up in VulkanApp_Data4 folder.

Further improves

Our code is running, but it is slower it should be. Inside `draw`, we call `updateUniformBuffers`. Every frame, we map the model data memory : this is not efficient. Memory mapping is very slow.

As we have said:

- Uniform buffers are good for a single piece of unchanging data that all of our objects use
- Dynamic uniform buffers are good for multiple *unchanging* data, unique for each object. Here, we are using them for changing data.

The solution is to use **push constants** for multiple changing data.

Push constantes

The use of push constants follow a different logic than the buffers we used until now. We will this time directly push values. We will have to re-record our command every single draw. Still, this is less expensive than re-allocate memory every frame.

Reset command pool each frame

Let's start by allowing to reset the command buffer in the command pool configuration:

`VulkanRenderer.cpp`

```
void VulkanRenderer::createGraphicsCommandPool()
{
        QueueFamilyIndices queueFamilyIndices = getQueueFamilies(mainDevice.physicalDevice);

        vk::CommandPoolCreateInfo poolInfo{};
        // Queue family type that buffers from this command pool will use
        poolInfo.queueFamilyIndex = queueFamilyIndices.graphicsFamily;
        poolInfo.flags = vk::CommandPoolCreateFlagBits::eResetCommandBuffer;
        ...
```

Now the commabd buffer will implicitly reset everytime we call `vk::CommandBuffer::begin`.

Recording the commands each frame

Remove `recordCommands` from `init` and move it to `draw`, just before `updateUniformBuffers`. Indeed, we want to record the commands each frame.

```
...
        // 1. Get next available image to draw and set a semaphore to signal
        // when we're finished with the image.
        uint32_t imageToBeDrawnIndex;
        vkAcquireNextImageKHR(mainDevice.logicalDevice, swapchain,
                std::numeric_limits<uint32_t>::max(), imageAvailable[currentFrame],
                VK_NULL_HANDLE, &imageToBeDrawnIndex);

        recordCommands();
        updateUniformBuffers(imageToBeDrawnIndex);

        // 2. Submit command buffer to queue for execution, make sure it waits
        // for the image to be signaled as available before drawing, and
        // signals when it has finished rendering.
...
```

The point is we don't want to update uniform buffers each frame. We only want to push constants. Add a uint32_t currentImage parameter to recordCommands. We do not want to loop through the command buffer anymore. We will replace `i` with the `currentImage` in this function.

```
void VulkanRenderer::recordCommands(uint32_t currentImage)
{
        // How to begin each command buffer
        vk::CommandBufferBeginInfo commandBufferBeginInfo{};
        // Buffer can be resubmited when it has already been submited
        //commandBufferBeginInfo.flags = vk::CommandBufferUsageFlagBits::eSimultaneousUse;

        // Information about how to being a render pass (only for graphical apps)
        vk::RenderPassBeginInfo renderPassBeginInfo{};
        // Render pass to begin
        renderPassBeginInfo.renderPass = renderPass;
        // Start point of render pass in pixel
        renderPassBeginInfo.renderArea.offset = vk::Offset2D { 0, 0 };
        // Size of region to run render pass on
        renderPassBeginInfo.renderArea.extent = swapchainExtent;

        vk::ClearValue clearValues{};
        std::array<float, 4> colors { 0.6f, 0.65f, 0.4f, 1.0f };
        clearValues.color = vk::ClearColorValue{ colors };

        renderPassBeginInfo.pClearValues = &clearValues;
        renderPassBeginInfo.clearValueCount = 1;

        // Because 1-to-1 relationship
        renderPassBeginInfo.framebuffer = swapchainFramebuffers[currentImage];

        // Start recording commands to command buffer
        commandBuffers[currentImage].begin(commandBufferBeginInfo);

        // Begin render pass
        // All draw commands inline (no secondary command buffers)
        commandBuffers[currentImage].beginRenderPass(
                renderPassBeginInfo, vk::SubpassContents::eInline);

        // Bind pipeline to be used in render pass,
        // you could switch pipelines for different subpasses
        commandBuffers[currentImage].bindPipeline(
                vk::PipelineBindPoint::eGraphics, graphicsPipeline);
        // Draw all meshes
        for (size_t j = 0; j < meshes.size(); ++j)
        {
                // Bind vertex buffer
                vk::Buffer vertexBuffers[] = { meshes[j].getVertexBuffer() };
                vk::DeviceSize offsets[] = { 0 };
                commandBuffers[currentImage].bindVertexBuffers(0, 1, vertexBuffers, offsets);

                // Bind index buffer
                commandBuffers[currentImage].bindIndexBuffer(
                        meshes[j].getIndexBuffer(), 0, vk::IndexType::eUint32);

                // Dynamic offet amount
                uint32_t dynamicOffset = static_cast<uint32_t>(modelUniformAlignement) * j;

                // Bind descriptor sets
                commandBuffers[currentImage].bindDescriptorSets(vk::PipelineBindPoint::eGraphics,
                        pipelineLayout, 0, 1, &descriptorSets[currentImage], 1, &dynamicOffset);

                // Execute pipeline
                commandBuffers[currentImage].drawIndexed(
                        static_cast<uint32_t>(meshes[j].getIndexCount()), 1, 0, 0, 0);
        }

        // End render pass
        commandBuffers[currentImage].endRenderPass();

        // Stop recordind to command buffer
        commandBuffers[currentImage].end();
```

Do not forget to add the parameter to `recordCommands`:

```
        recordCommands(imageToBeDrawnIndex);
```

Everything is now set up, we can push the constants.

Update the shader and unroll the thread

`shader.vert`

```
#version 450

// From vertex input stage
layout(location = 0) in vec3 pos;
layout(location = 1) in vec3 col;

// Uniform Buffer Object
layout(binding = 0) uniform ViewProjection {
    mat4 projection;
    mat4 view;
} viewProjection;

// Push constant
layout(push_constant) uniform PushModel {
    mat4 model;
} pushModel;

// To fragment shader
layout(location = 0) out vec3 fragColor;

void main() {
    gl_Position = viewProjection.projection * viewProjection.view *
                  pushModel.model * vec4(pos, 1.0);
    fragColor = col;
}
```

You can also comment out the call to the allocation of the dynamic uniform buffer in `init`:

```
                // Data
                //allocateDynamicBufferTransferSpace();
```

And comment out the dynamic unifor buffer part in the `updateUniformBuffers` function:

```
void VulkanRenderer::updateUniformBuffers(uint32_t imageIndex)
{
	// Copy view projection data
	void* data;
	mainDevice.logicalDevice.mapMemory(vpUniformBufferMemory[imageIndex], {},
		sizeof(ViewProjection), {}, &data);
	memcpy(data, &viewProjection, sizeof(ViewProjection));
	mainDevice.logicalDevice.unmapMemory(vpUniformBufferMemory[imageIndex]);

	/*
	// Copy model data
	for (size_t i = 0; i < meshes.size(); ++i)
	{
		// Get a pointer to model in modelTransferSpace
		Model* model = (Model*)((uint64_t)modelTransferSpace + i * modelUniformAlignement)
		// Copy data at this adress
		*model = meshes[i].getModel();
	}
	mainDevice.logicalDevice.mapMemory(modelUniformBufferMemoryDynamic[imageIndex], {},
		modelUniformAlignement * meshes.size(), {}, &data);
	memcpy(data, &modelTransferSpace, modelUniformAlignement * meshes.size());
	mainDevice.logicalDevice.unmapMemory(modelUniformBufferMemoryDynamic[imageIndex]);
	*/
}
```

Remove the reference to the dynamic ubniform buffer in `clean`:

```
...
	for (size_t i = 0; i < swapchainImages.size(); ++i)
	{
		mainDevice.logicalDevice.destroyBuffer(vpUniformBuffer[i]);
		mainDevice.logicalDevice.freeMemory(vpUniformBufferMemory[i]);
		//mainDevice.logicalDevice.destroyBuffer(modelUniformBufferDynamic[i]);
		//mainDevice.logicalDevice.freeMemory(modelUniformBufferMemoryDynamic[i]);
	}
...
```

Same with `createDescriptorSetLayout` , `createDescriptorPool` , `createDescriptorSets` and `createUniformBuffers`:

```
void VulkanRenderer::createDescriptorSetLayout()
{
        // ViewProjection binding information
        vk::DescriptorSetLayoutBinding vpLayoutBinding;
        // Binding number in shader
        vpLayoutBinding.binding = 0;
        // Type of descriptor (uniform, dynamic uniform, samples...)
        vpLayoutBinding.descriptorType = vk::DescriptorType::eUniformBuffer;
        // Number of descriptors for binding
        vpLayoutBinding.descriptorCount = 1;
        // Shader stage to bind to (here: vertex shader)
        vpLayoutBinding.stageFlags = vk::ShaderStageFlagBits::eVertex;
        // For textures : can make sample data un changeable
        vpLayoutBinding.pImmutableSamplers = nullptr;

        /*
        // Model
        vk::DescriptorSetLayoutBinding modelLayoutBinding;
        modelLayoutBinding.binding = 1;
        modelLayoutBinding.descriptorType = vk::DescriptorType::eUniformBufferDynamic;
        modelLayoutBinding.descriptorCount = 1;
        modelLayoutBinding.stageFlags = vk::ShaderStageFlagBits::eVertex;
        modelLayoutBinding.pImmutableSamplers = nullptr;

        vector<vk::DescriptorSetLayoutBinding> layoutBindings {
                vpLayoutBinding, modelLayoutBinding
        };
        */

        vector<vk::DescriptorSetLayoutBinding> layoutBindings{
                vpLayoutBinding
        };

        // Descriptor set layout with given binding
        vk::DescriptorSetLayoutCreateInfo layoutCreateInfo{};
        layoutCreateInfo.bindingCount = static_cast<uint32_t>(layoutBindings.size());
        layoutCreateInfo.pBindings = layoutBindings.data();

        // Create descriptor set layout
        descriptorSetLayout = mainDevice.logicalDevice.createDescriptorSetLayout(layoutCreateInfo)
}

...

void VulkanRenderer::createDescriptorPool()
{
        // One descriptor in the pool for each image

        // View projection pool
        vk::DescriptorPoolSize vpPoolSize{};
        vpPoolSize.descriptorCount = static_cast<uint32_t>(vpUniformBuffer.size());

        /*
        // Model pool
        vk::DescriptorPoolSize modelPoolSize{};
        modelPoolSize.descriptorCount = static_cast<uint32_t>(modelUniformBufferDynamic.size());

        vector<vk::DescriptorPoolSize> poolSizes{ vpPoolSize, modelPoolSize };
        */

        vector<vk::DescriptorPoolSize> poolSizes{ vpPoolSize };

        // One descriptor set that contains one descriptor
        vk::DescriptorPoolCreateInfo poolCreateInfo{};
        poolCreateInfo.maxSets = static_cast<uint32_t>(swapchainImages.size());
        poolCreateInfo.poolSizeCount = static_cast<uint32_t>(poolSizes.size());
        poolCreateInfo.pPoolSizes = poolSizes.data();

        // Create pool
        descriptorPool = mainDevice.logicalDevice.createDescriptorPool(poolCreateInfo);
}

void VulkanRenderer::createDescriptorSets()
{
        // One descriptor set for every image/buffer
        descriptorSets.resize(swapchainImages.size());

        // We want the same layout for the right number of descriptor sets
        vector<vk::DescriptorSetLayout> setLayouts(swapchainImages.size(), descriptorSetLayout);

        // Allocation from the pool
        vk::DescriptorSetAllocateInfo setAllocInfo{};
        setAllocInfo.descriptorPool = descriptorPool;
        setAllocInfo.descriptorSetCount = static_cast<uint32_t>(swapchainImages.size());
        setAllocInfo.pSetLayouts = setLayouts.data();

        // Allocate multiple descriptor sets
        vk::Result result = mainDevice.logicalDevice.allocateDescriptorSets(
                &setAllocInfo, descriptorSets.data());
        if (result != vk::Result::eSuccess)
        {
                throw std::runtime_error("Failed to allocate descriptor sets.");
        }

        // We have a connection between descriptor set layouts and descriptor sets,
        // but we don't know how link descriptor sets and the uniform buffers.

        // Update all of descriptor set buffer bindings
        for (size_t i = 0; i < swapchainImages.size(); ++i)
        {
                // -- VIEW PROJECTION DESCRIPTOR --
                // Description of the buffer and data offset
                vk::DescriptorBufferInfo vpBufferInfo{};
                // Buffer to get data from
                vpBufferInfo.buffer = vpUniformBuffer[i];
                // We bind the whole data
                vpBufferInfo.offset = 0;
                // Size of data
                vpBufferInfo.range = sizeof(ViewProjection);

                // Data about connection between binding and buffer
                vk::WriteDescriptorSet vpSetWrite{};
                // Descriptor sets to update
                vpSetWrite.dstSet = descriptorSets[i];
                // Binding to update (matches with shader binding)
                vpSetWrite.dstBinding = 0;
                // Index in array to update
                vpSetWrite.dstArrayElement = 0;
                vpSetWrite.descriptorType = vk::DescriptorType::eUniformBuffer;
                // Amount of descriptor sets to update
                vpSetWrite.descriptorCount = 1;
                // Information about buffer data to bind
                vpSetWrite.pBufferInfo = &vpBufferInfo;

                /*
                // -- MODEL DESCRIPTOR --
                // Description of the buffer and data offset
                vk::DescriptorBufferInfo modelBufferInfo{};
                modelBufferInfo.buffer = modelUniformBufferDynamic[i];
                modelBufferInfo.offset = 0;
                modelBufferInfo.range = modelUniformAlignement;

                // Data about connection between binding and buffer
                vk::WriteDescriptorSet modelSetWrite{};
                modelSetWrite.dstSet = descriptorSets[i];
                modelSetWrite.dstBinding = 1;
                modelSetWrite.dstArrayElement = 0;
                modelSetWrite.descriptorType = vk::DescriptorType::eUniformBufferDynamic;
                modelSetWrite.descriptorCount = 1;
                modelSetWrite.pBufferInfo = &modelBufferInfo;

                // Descriptor set writes vector
                vector<vk::WriteDescriptorSet> setWrites{ vpSetWrite, modelSetWrite };
                */

                vector<vk::WriteDescriptorSet> setWrites{ vpSetWrite };

                // Update descriptor set with new buffer/binding info
                mainDevice.logicalDevice.updateDescriptorSets(
                        static_cast<uint32_t>(setWrites.size()), setWrites.data(), 0, nullptr);
        }
}

...

void VulkanRenderer::createUniformBuffers()
```

```
{
        // Buffer size will be size of all 3 variables
        vk::DeviceSize vpBufferSize = sizeof(ViewProjection);

        /*
        // Model buffer size
        vk::DeviceSize modelBufferSize = modelUniformAlignement * MAX_OBJECTS;
        */

        // One uniform buffer for each image / each command buffer
        vpUniformBuffer.resize(swapchainImages.size());
        vpUniformBufferMemory.resize(swapchainImages.size());
        /*
        modelUniformBufferDynamic.resize(swapchainImages.size());
        modelUniformBufferMemoryDynamic.resize(swapchainImages.size());
        */

        // Create uniform buffers
        for (size_t i = 0; i < swapchainImages.size(); ++i)
        {
                createBuffer(mainDevice.physicalDevice, mainDevice.logicalDevice, vpBufferSize,
                        vk::BufferUsageFlagBits::eUniformBuffer,
                        vk::MemoryPropertyFlagBits::eHostVisible
                                | vk::MemoryPropertyFlagBits::eHostCoherent,
                        &vpUniformBuffer[i], &vpUniformBufferMemory[i]);

                /*
                createBuffer(mainDevice.physicalDevice, mainDevice.logicalDevice, modelBufferSize,
                        vk::BufferUsageFlagBits::eUniformBuffer,
                        vk::MemoryPropertyFlagBits::eHostVisible
                                | vk::MemoryPropertyFlagBits::eHostCoherent,
                        &modelUniformBufferDynamic[i], &modelUniformBufferMemoryDynamic[i]);
                */
        }
}
```

Push constants

Declare the push constant and associated function:

`VulkanRenderer.h`

```
	VkPushConstantRange pushConstantRange;
	void createPushConstantRange();
```

`VulkanRenderer.cpp`

```
void VulkanRenderer::createPushConstantRange()
{
	// Shader stage push constant will go to
	pushConstantRange.stageFlags = vk::ShaderStageFlagBits::eVertex;
	pushConstantRange.offset = 0;
	pushConstantRange.size = sizeof(Model);
}
```

Call this function in `init`:

```
		createDescriptorSetLayout();
		createPushConstantRange();
		createGraphicsPipeline();
```

Let's now update the pipeline layout.

```
	// -- PIPELINE LAYOUT --
	vk::PipelineLayoutCreateInfo pipelineLayoutCreateInfo{};
	pipelineLayoutCreateInfo.setLayoutCount = 1;
	pipelineLayoutCreateInfo.pSetLayouts = &descriptorSetLayout;
	pipelineLayoutCreateInfo.pushConstantRangeCount = 1;
	pipelineLayoutCreateInfo.pPushConstantRanges = &pushConstantRange;
```

...and go to the `recordCommands` function. We will update the draw part to add push constants, and remove the Dynamic uniform buffer's descriptor sets.

```
...
        // Draw all meshes
        for (size_t j = 0; j < meshes.size(); ++j)
        {
                ...

                /*
                // Dynamic offet amount
                uint32_t dynamicOffset = static_cast<uint32_t>(modelUniformAlignement) * j;

                // Bind descriptor sets
                commandBuffers[currentImage].bindDescriptorSets(vk::PipelineBindPoint::eGraphics,
                        pipelineLayout, 0, 1, &descriptorSets[currentImage], 1, &dynamicOffset);
                */

                // Push constants to given shader stage
                Model model = meshes[j].getModel();
                commandBuffers[currentImage].pushConstants(pipelineLayout,
                        vk::ShaderStageFlagBits::eVertex, 0, sizeof(Model), &model);

                // Bind descriptor sets
                commandBuffers[currentImage].bindDescriptorSets(vk::PipelineBindPoint::eGraphics,
                        pipelineLayout, 0, 1, &descriptorSets[currentImage], 0, nullptr);

                // Execute pipeline
                commandBuffers[currentImage].drawIndexed(
                        static_cast<uint32_t>(meshes[j].getIndexCount()), 1, 0, 0, 0);
        }
...
```

Now the data sent each frame is way lighter. This project is setup in the VulkanApp_Data5 folder.

Nevertheless, we still have a problem. If we try to translate the square in front of the fast turning rectangle, it does not work. The second quad is drawn in front of the first, just because it is drawn second. We can solve this problem by implementing a depth buffer.

Depth buffer

Concepts

The **Depth buffer** will keeps track of an object distance from the camera. It has a second Image that holds the depth of each pixel, in place of color data.

We will need a new Image. This time the swapchain won't be used to create it, we will generate it manually. To create an image, we use a `vk::ImageCreateInfo`. However, this object is only a struct, a header describing the image. We will have to designate `DeviceMemory` for the image, then bound this memory to the `vk::Image`, then generate an `ImageView` from this last element. Nevertheless, we won't need to map memory: the depth values are calculated by the pipeline itself.

Much like the swapchain images, the depth image will be an attachment, transitioned at each stage of the render pass (for now we only had the color attachment). The new attachment will starts as undefined, transition to `vk::ImageLayout::eAttachmentOptimal`, that is used for both depth and stencil. Transitions will happen at the same time as swapchain images. Subpass does not refer to the depth image in the color attachment, but instead throught the `pDepthStencilAttachment`.

The depth buffer's image view is then attached to the framebuffer. If the render pass has color attachment 0 and depth attachment 1, then the framebuffer will also need to respect this order. Otherwise the depth will output to the color and the color to the depth.

We have to not forget tho clear the depth attachement at the start of each renderpass. The position of the clears should match the order of the render pass and framebuffer attachment.

Data and functions

At he beginning of the main, tell glm to make depth going from 0 to 1, instead it goind from -1 to 1

`Main.cpp`

```
#define GLM_FORCE_DEPTH_ZERO_TO_ONE
```

Create the data we need.

`VulkanRenderer.h`

```
        // Depth
        vk::Image depthBufferImage;
        vk::DeviceMemory depthBufferImageMemory;
        vk::ImageView depthBufferImageView;
        void createDepthBufferImage();
```

Use it in the `init` function:

```
...
                createGraphicsPipeline();
                createDepthBufferImage();
                createFramebuffers();
...
```

Create an image

We will create a function specifically to create images:

`VulkanRenderer.h`

```
VkImage createImage(uint32_t width, uint32_t height,
        VkFormat format, VkImageTiling tiling,
        VkImageUsageFlags useFlags, VkMemoryPropertyFlags propFlags,
        VkDeviceMemory *imageMemory);
```

`VulkanRenderer.cpp`

```
VkImage VulkanRenderer::createImage(uint32_t width, uint32_t height,
        VkFormat format, VkImageTiling tiling, VkImageUsageFlags useFlags,
        VkMemoryPropertyFlags propFlags, VkDeviceMemory* imageMemory)
{
        vk::ImageCreateInfo imageCreateInfo{};
        imageCreateInfo.imageType = vk::ImageType::e2D;
        imageCreateInfo.extent.width = width;
        imageCreateInfo.extent.height = height;
        // Depth is 1, no 3D aspect
        imageCreateInfo.extent.depth = 1;
        imageCreateInfo.mipLevels = 1;
        // Number of levels in image array
        imageCreateInfo.arrayLayers = 1;
        imageCreateInfo.format = format;
        // How image data should be "tiled" (arranged for optimal reading)
        imageCreateInfo.tiling = tiling;
        // Initial layout in the render pass
        imageCreateInfo.initialLayout = vk::ImageLayout::eUndefined;
        // Bit flags defining what image will be used for
        imageCreateInfo.usage = useFlags;
        // Number of samples for multi sampling
        imageCreateInfo.samples = vk::SampleCountFlagBits::e1;
        // Whether image can be shared between queues (no)
        imageCreateInfo.sharingMode = vk::SharingMode::eExclusive;

        // Create the header of the image
        vk::Image image = mainDevice.logicalDevice.createImage(imageCreateInfo);

        // Now we need to setup and allocate memory for the image
        vk::MemoryRequirements memoryRequierements =
                mainDevice.logicalDevice.getImageMemoryRequirements(image);

        vk::MemoryAllocateInfo memoryAllocInfo{};
        memoryAllocInfo.allocationSize = memoryRequierements.size;
        memoryAllocInfo.memoryTypeIndex = findMemoryTypeIndex(mainDevice.physicalDevice,
                memoryRequierements.memoryTypeBits, propFlags);

        auto result = mainDevice.logicalDevice.allocateMemory(
                &memoryAllocInfo, nullptr, imageMemory);
        if (result != vk::Result::eSuccess)
        {
                throw std::runtime_error("Failed to allocate memory for an image.");
        }

        // Connect memory to image
        mainDevice.logicalDevice.bindImageMemory(image, *imageMemory, 0);

        return image;
}
```

We also need a function to get the supported format for our image:

```
vk::Format VulkanRenderer::chooseSupportedFormat(
        const vector<vk::Format>& formats, vk::ImageTiling tiling,
        vk::FormatFeatureFlags featureFlags)
{
        // Loop through the options and find a compatible format
        for (vk::Format format : formats)
        {
                // Get properties for a given format on this device
                vk::FormatProperties properties =
                        mainDevice.physicalDevice.getFormatProperties(format);

                // If the tiling is linear and all feature flags match
                if (tiling == vk::ImageTiling::eLinear
                        && (properties.linearTilingFeatures & featureFlags) == featureFlags)
                {
                        return format;
                }
                // If the tiling is optimal and all feature flags match
                else if (tiling == vk::ImageTiling::eOptimal &&
                        (properties.optimalTilingFeatures & featureFlags) == featureFlags)
                {
                        return format;
                }
        }
        throw std::runtime_error("Failed to find a matching format.");
}
```

We can now use those functions to create the buffer image:

```
void VulkanRenderer::createDepthBufferImage()
{
        std::vector<vk::Format> formats {
                // Look for a format with 32bits death buffer and stencil buffer
                vk::Format::eD32SfloatS8Uint,
                // if not found, without stencil
                vk::Format::eD32Sfloat,
                // if not 24bits depth and stencil buffer
                vk::Format::eD24UnormS8Uint
        };

        vk::Format depthFormat = chooseSupportedFormat(formats, vk::ImageTiling::eOptimal,
                // Format supports depth and stencil attachment
                vk::FormatFeatureFlagBits::eDepthStencilAttachment);

        // Create image and image view
        depthBufferImage = createImage(swapchainExtent.width,
                swapchainExtent.height, depthFormat, vk::ImageTiling::eOptimal,
                vk::ImageUsageFlagBits::eDepthStencilAttachment,
                vk::MemoryPropertyFlagBits::eDeviceLocal, &depthBufferImageMemory);

        depthBufferImageView = createImageView(
                depthBufferImage, depthFormat, vk::ImageAspectFlagBits::eDepth);
}
```

We can now update the `clean` function:

```
void VulkanRenderer::clean()
{
        mainDevice.logicalDevice.waitIdle();

        mainDevice.logicalDevice.destroyImageView(depthBufferImageView);
        mainDevice.logicalDevice.destroyImage(depthBufferImage);
        mainDevice.logicalDevice.freeMemory(depthBufferImageMemory);
        ...
}
```

Use the depth buffer in the render pass

We will tell the renderpass how it is going to handle the depth image and how it is going to get the data from the pipeline and read it. We will add a depth attachment in addition to the color attachment, and a reference to it.

```
void VulkanRenderer::createRenderPass()
{
        vk::RenderPassCreateInfo renderPassCreateInfo{};

        // Attachement description : describe color buffer output, depth buffer output...
        // e.g. (location = 0) in the fragment shader is the first attachment
        vk::AttachmentDescription colorAttachment{};
        // Format to use for attachment
        colorAttachment.format = swapchainImageFormat;
        // Number of samples t write for multisampling
        colorAttachment.samples = vk::SampleCountFlagBits::e1;
        // What to do with attachement before renderer. Here, clear when we start the render pass.
        colorAttachment.loadOp = vk::AttachmentLoadOp::eClear;
        // What to do with attachement after renderer. Here, store the render pass.
        colorAttachment.storeOp = vk::AttachmentStoreOp::eStore;
        // What to do with stencil before renderer. Here, don't care, we don't use stencil.
        colorAttachment.stencilLoadOp = vk::AttachmentLoadOp::eDontCare;
        // What to do with stencil after renderer. Here, don't care, we don't use stencil.
        colorAttachment.stencilStoreOp = vk::AttachmentStoreOp::eDontCare;

        // Framebuffer images will be stored as an image, but image can have different layouts
        // to give optimal use for certain operations
        // Image data layout before render pass starts
        colorAttachment.initialLayout = vk::ImageLayout::eUndefined;
        // Image data layout after render pass
        colorAttachment.finalLayout = vk::ImageLayout::ePresentSrcKHR;

        // Depth attachment of renderpass
        vk::AttachmentDescription depthAttachment{};

        std::vector<vk::Format> formats {
                vk::Format::eD32SfloatS8Uint,
                vk::Format::eD32Sfloat,
                vk::Format::eD24UnormS8Uint
        };
        depthAttachment.format = chooseSupportedFormat(
                formats, vk::ImageTiling::eOptimal,
                vk::FormatFeatureFlagBits::eDepthStencilAttachment);
        depthAttachment.samples = vk::SampleCountFlagBits::e1;
        // Clear when we start the render pass.
        depthAttachment.loadOp = vk::AttachmentLoadOp::eClear;
        // We do not do anything after depth buffer image is calculated
        depthAttachment.storeOp = vk::AttachmentStoreOp::eDontCare;
        depthAttachment.stencilLoadOp = vk::AttachmentLoadOp::eDontCare;
        depthAttachment.stencilStoreOp = vk::AttachmentStoreOp::eDontCare;
        depthAttachment.initialLayout = vk::ImageLayout::eUndefined;
        depthAttachment.finalLayout = vk::ImageLayout::eDepthStencilAttachmentOptimal;

        array<vk::AttachmentDescription, 2> renderPassAttachments{
                colorAttachment, depthAttachment
        };
        renderPassCreateInfo.attachmentCount =
                static_cast<uint32_t>(renderPassAttachments.size());
        renderPassCreateInfo.pAttachments = renderPassAttachments.data();

        // -- REFERENCES --
        // Attachment reference uses an attachment index that refers to index
        // in the attachement list passed to renderPassCreateInfo
        vk::AttachmentReference colorAttachmentReference{};
        colorAttachmentReference.attachment = 0;
        // Layout of the subpass (between initial and final layout)
        colorAttachmentReference.layout = vk::ImageLayout::eColorAttachmentOptimal;

        vk::AttachmentReference depthAttachmentReference{};
        depthAttachmentReference.attachment = 1;
        depthAttachmentReference.layout = vk::ImageLayout::eDepthStencilAttachmentOptimal;

        // -- SUBPASSES --
        // Subpass description, will reference attachements
        vk::SubpassDescription subpass{};
        // Pipeline type the subpass will be bound to.
        // Could be compute pipeline, or nvidia raytracing...
        subpass.pipelineBindPoint = vk::PipelineBindPoint::eGraphics;
        subpass.colorAttachmentCount = 1;
        subpass.pColorAttachments = &colorAttachmentReference;
        subpass.pDepthStencilAttachment = &depthAttachmentReference;

        renderPassCreateInfo.subpassCount = 1;
        renderPassCreateInfo.pSubpasses = &subpass;

        // Subpass dependencies: transitions between subpasses + from the last subpass
        // to what happens after. Need to determine when layout transitions
        // occur using subpass dependencies.
        // Will define implicitly layout transitions.
        array<vk::SubpassDependency, 2> subpassDependencies;
        // -- From layout undefined to color attachment optimal
        // ---- Transition must happens after
        // External means from outside the subpasses
        subpassDependencies[0].srcSubpass = VK_SUBPASS_EXTERNAL;
        // Which stage of the pipeline has to happen before
        subpassDependencies[0].srcStageMask = vk::PipelineStageFlagBits::eBottomOfPipe;
        subpassDependencies[0].srcAccessMask = vk::AccessFlagBits::eMemoryRead;
        // ---- But must happens before
        // Conversion should happen before the first subpass starts
        subpassDependencies[0].dstSubpass = 0;
        subpassDependencies[0].dstStageMask = vk::PipelineStageFlagBits::eColorAttachmentOutput;
        // ...and before the color attachment attempts to read or write
        subpassDependencies[0].dstAccessMask =
                vk::AccessFlagBits::eMemoryRead | vk::AccessFlagBits::eMemoryWrite;
        subpassDependencies[0].dependencyFlags = {}; // No dependency flag

        // -- From layout color attachment optimal to image layout present
        // ---- Transition must happens after
        subpassDependencies[1].srcSubpass = 0;
        subpassDependencies[1].srcStageMask =
                vk::PipelineStageFlagBits::eColorAttachmentOutput;
        subpassDependencies[1].srcAccessMask =
                vk::AccessFlagBits::eMemoryRead | vk::AccessFlagBits::eMemoryWrite;
        // ---- But must happens before
        subpassDependencies[1].dstSubpass = VK_SUBPASS_EXTERNAL;
        subpassDependencies[1].dstStageMask = vk::PipelineStageFlagBits::eBottomOfPipe;
        subpassDependencies[1].dstAccessMask = vk::AccessFlagBits::eMemoryRead;
        subpassDependencies[1].dependencyFlags = vk::DependencyFlags();

        renderPassCreateInfo.dependencyCount =
                static_cast<uint32_t>(subpassDependencies.size());
        renderPassCreateInfo.pDependencies = subpassDependencies.data();
```

```
	renderPass = mainDevice.logicalDevice.createRenderPass(renderPassCreateInfo);
}
```

We also need to update the attachments in the `createFramebuffers` function:

```
void VulkanRenderer::createFramebuffers()
{
	// Create one framebuffer for each swapchain image
	swapchainFramebuffers.resize(swapchainImages.size());
	for (size_t i = 0; i < swapchainFramebuffers.size(); ++i)
	{
		// Setup attachments
		array<vk::ImageView, 2> attachments{
			swapchainImages[i].imageView, depthBufferImageView
		};
		...
```

We still need the pipeline to output the depth data.

```
void VulkanRenderer::createGraphicsPipeline()
{
        ...
        // -- DEPTH STENCIL TESTING --
        vk::PipelineDepthStencilStateCreateInfo depthStencilCreateInfo{};
        // Enable checking depth
        depthStencilCreateInfo.depthTestEnable = true;
        // Enable writing (replace old values) to depth buffer
        depthStencilCreateInfo.depthWriteEnable = true;
        depthStencilCreateInfo.depthCompareOp = vk::CompareOp::eLess;
        // Does the depth value exist between two bounds?
        depthStencilCreateInfo.depthBoundsTestEnable = false;
        // Enable stencil test
        depthStencilCreateInfo.stencilTestEnable = false;



        // -- PASSES --
        // Passes are composed of a sequence of subpasses that can pass data from one to another

        // -- GRAPHICS PIPELINE CREATION --
        ...
        graphicsPipelineCreateInfo.pDepthStencilState = &depthStencilCreateInfo;
```

Finally, we update the `recordCommands` function. We just have to modify the clear values, to clear the depth buffer as well as the color buffer.

```
void VulkanRenderer::recordCommands(uint32_t currentImage)
{
        ...
        array<vk::ClearValue, 2> clearValues{};
        std::array<float, 4> colors { 0.6f, 0.65f, 0.4f, 1.0f };
        clearValues[0].color = vk::ClearColorValue{colors};
        clearValues[1].depthStencil.depth = 1.0f;

        renderPassBeginInfo.pClearValues = clearValues.data();
        renderPassBeginInfo.clearValueCount = static_cast<uint32_t>(clearValues.size());
```

Change the second model translation coordinates to superpose the two models and enjoy your brand new depth buffer !

`main.cpp`

```
...
                secondModel = glm::translate(secondModel, glm::vec3(-2.0f, 0.0f, -4.0f));
...
```

And now, finally, we have a proper way to set the transform of our 3D quads ! The final code is available in the VulkanApp_Data6 folder.

Next lessons will be about setting up basics 3D graphics techniques, such as textures, use of models and multiple subpasses.