# Basic rendering techniques

## Prerequisites

This lesson will need the use of your hardware graphics card. On laptop computers, this can be a problem, because for power reason, your OS can try using the CPU integrated graphics card. In order to force the use of the hardware GPU, paste this code at the beginning of your main file.

`main.cpp`

```
#define DWORD unsigned int
#if defined(WIN32) || defined(_WIN32)
        extern "C" { __declspec(dllexport) DWORD NvOptimusEnablement = 0x00000001; }
        extern "C" {
                __declspec(dllexport) DWORD AmdPowerXpressRequestHighPerformance = 0x00000001; }
#else
        extern "C" { int NvOptimusEnablement = 1; }
        extern "C" { int AmdPowerXpressRequestHighPerformance = 1; }
#endif
```

# Textures

## Concepts

Textures are split in two parts, accessed through descriptor sets:

- The **Image** data itself
- The **Sampler** : an object that accesses the image using pre-defined methods, such as picking a point between texels or beyond the edge of the image.

You can have separate descriptor sets, but there is a **Combined Image Sampler** to get both at the same time.

To load image data, we will use stb_image, then copy data to a buffer, with the usual process:

- Create a Buffer and Memory
- Use `memcpy` to map image data to memory
- Use Staging Buffers because the texture data won't be changing and can be stored on GPU without host access.

We will create our image with the layout `vk::ImageLayout::eUndefined`. Then when transfering the Image from Staging Buffer to a destination, the image will need to be in layout `vk::ImageLayout::eTransferDstOptimal`. After this transfer, in order to be used, it will switch layout to `vk::ImageLayout::eShaderReadOnlyOptimal`. Usually we transitions automatically attachments in the render pass through subpass dependencies. But images are not attachments, so we will have to transition layouts manually. To do this we will use **Pipeline Barriers**.

Pipeline barriers are similar to semaphores because it is used to synchronise CPU actions, with the difference we can set thems wherever we want. They are barriers in the pipeline to organise events and ensure no overlaps. Pipeline barriers will separate queues of command between those which are before the barrier and those which are after. They are useful in our cases because they can force layout transitions when crossing the barrier.

We will use a specific barrier called **Image Memory Barrier**. They restricts order of access to image resource and allow image layout transitions. They will state two points in time:

- A time a transition can only occur after;
- A time a transition must occur before.

Once the data in a buffer, we must define the sampler. Options are similar as in OpenGL : filtering, addressing, anisotropy. Anisotropy will require a feature to be enabled on the Device.

The descriptor sets creation will be similar to the previous ones. This time however the descriptor set layout will differ and the stage it will connect will be the fragment shader's. The descriptor pool size will be of Compined Image Sampler type. The descriptor write will use `pImageInfo` as opposed to `pBufferInfo`.

To use the descriptor set, we will do same way we did before. We will just ensure that vertex data is updated to hold texel values. And of course we will update the shaders.

We will setup the code to use multiple textures at the same time.

# Loading a texture file

## File loading

We will use `stb_image` to load files. Include the `stb_image.h` file to laod images : go to https://github.com/nothings/stb/blob/master/stb_image.h , get to the raw file, download and include it in your project. Also create a `textures` folder into your project folder.

`VulkanRenderer.h`

```
...
#include "stb_image.h"
...
```

`Main.cpp`

```
#define STB_IMAGE_IMPLEMENTATION
```

Use stb_image to create a public load image function:

```
stbi_uc* VulkanRenderer::loadTextureFile(
        const string& filename, int* width, int* height,
        vk::DeviceSize* imageSize)
{
        // Number of channel image uses
        int channels;

        // Load pixel data for image
        string path = "textures/" + filename;
        stbi_uc* image = stbi_load(path.c_str(), width, height, &channels, STBI_rgb_alpha);

        if (!image)
        {
                throw std::runtime_error("Failed to load texture file: " + path);
        }

        // RGBA has 4 channels
        *imageSize = *width * *height * 4;
        return image;
}
```

## Loading image texture on staging buffer

We first need to create vectors that will hold multiple textures and their memory.

`VulkanRenderer.h`

```
        vector<VkImage> textureImages;
        vector<vk::ImageView> textureImageViews;
        vector<VkDeviceMemory> textureImageMemory;
```

A function will ensure the texture load and put the data in the staging buffer.

`VulkanRenderer.cpp`

```
int VulkanRenderer::createTextureImage(const string& filename)
{
        // Load image file
        int width, height;
        vk::DeviceSize imageSize;
        stbi_uc* imageData = loadTextureFile(filename, &width, &height, &imageSize);

        // Create staging buffer to hold loaded data, ready to copy to device
        vk::Buffer imageStagingBuffer;
        vk::DeviceMemory imageStagingBufferMemory;
        createBuffer(mainDevice.physicalDevice, mainDevice.logicalDevice,
                imageSize, vk::BufferUsageFlagBits::eTransferSrc,
                vk::MemoryPropertyFlagBits::eHostVisible
                        | vk::MemoryPropertyFlagBits::eHostCoherent,
                &imageStagingBuffer, &imageStagingBufferMemory);

        // Copy image data to the staging buffer
        void* data;
        mainDevice.logicalDevice.mapMemory(imageStagingBufferMemory, {}, imageSize, {}, &data);
        memcpy(data, imageData, static_cast<size_t>(imageSize));
        mainDevice.logicalDevice.unmapMemory(imageStagingBufferMemory);

        // Free original image data
        stbi_image_free(imageData);
    ...
}
```

This is only the beginning of this function. We now have to copy the staging buffer to an image. We already have a function to copy a buffer to an other buffer. We will modify this function so that it can copy a buffer to an image.

## Copy buffer to the image

We start with a little refactoring:

`VulkanUtilities.h`

```
static vk::CommandBuffer beginCommandBuffer(vk::Device device, vk::CommandPool commandPool)
{
	// Command buffer to hold transfer commands
	vk::CommandBuffer commandBuffer;

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

	// Allocate command buffer from pool
	commandBuffer = 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
	commandBuffer.begin(beginInfo);
	return commandBuffer;
}

static void endAndSubmitCommandBuffer(vk::Device device, vk::CommandPool commandPool,
	vk::Queue queue, vk::CommandBuffer commandBuffer)
{
	// End record commands
	commandBuffer.end();

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

	// Submit transfer commands to transfer queue and wait until it finishes
	queue.submit(1, &submitInfo, nullptr);
	queue.waitIdle();

	// Free temporary command buffer
	device.freeCommandBuffers(commandPool, 1, &commandBuffer);
}

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 = beginCommandBuffer(device, transferCommandPool);

        // 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);

        // Submit and free
        endAndSubmitCommandBuffer(device, transferCommandPool,
                transferQueue, transferCommandBuffer);
}
```

Now we can create a function to copy from the buffer to the image:

```
static void copyImageBuffer(vk::Device device, vk::Queue transferQueue,
        vk::CommandPool transferCommandPool, vk::Buffer srcBuffer, vk::Image dstImage,
        uint32_t width, uint32_t height)
{
        // Create buffer
        vk::CommandBuffer transferCommandBuffer =
                beginCommandBuffer(device, transferCommandPool);

        vk::BufferImageCopy imageRegion{};
        // All data of image is tightly packed
        // -- Offset into data
        imageRegion.bufferOffset = 0;
        // -- Row length of data to calculate data spacing
        imageRegion.bufferRowLength = 0;
        // -- Image height of data to calculate data spacing
        imageRegion.bufferImageHeight = 0;

        // Which aspect to copy (here: colors)
        imageRegion.imageSubresource.aspectMask = vk::ImageAspectFlagBits::eColor;
        // Mipmap level to copy
        imageRegion.imageSubresource.mipLevel = 0;
        // Starting array layer if array
        imageRegion.imageSubresource.baseArrayLayer = 0;
        // Number of layers to copy starting at baseArray
        imageRegion.imageSubresource.layerCount = 1;
        // Offset into image (as opposed to raw data into bufferOffset)
        imageRegion.imageOffset = vk::Offset3D { 0, 0, 0 };
        // Size of region to copy (xyz values)
        imageRegion.imageExtent = vk::Extent3D { width, height, 1 };

        // Copy buffer to image
        transferCommandBuffer.copyBufferToImage(srcBuffer,
                dstImage, vk::ImageLayout::eTransferDstOptimal, 1, &imageRegion);

        endAndSubmitCommandBuffer(device, transferCommandPool,
                transferQueue, transferCommandBuffer);
}
```

We can now finish our `createTextureImage` function:

`VulkanRenderer.cpp`

```
int VulkanRenderer::createTextureImage(const string& filename)
{
	// Load image file
	int width, height;
	vk::DeviceSize imageSize;
	stbi_uc* imageData = loadTextureFile(filename, &width, &height, &imageSize);

	// Create staging buffer to hold loaded data, ready to copy to device
	vk::Buffer imageStagingBuffer;
	vk::DeviceMemory imageStagingBufferMemory;
	createBuffer(mainDevice.physicalDevice, mainDevice.logicalDevice,
		imageSize, vk::BufferUsageFlagBits::eTransferSrc,
		vk::MemoryPropertyFlagBits::eHostVisible
			| vk::MemoryPropertyFlagBits::eHostCoherent,
		&imageStagingBuffer, &imageStagingBufferMemory);

	// Copy image data to the staging buffer
	void* data;
	mainDevice.logicalDevice.mapMemory(imageStagingBufferMemory, {}, imageSize, {}, &data);
	memcpy(data, imageData, static_cast<size_t>(imageSize));
	mainDevice.logicalDevice.unmapMemory(imageStagingBufferMemory);

	// Free original image data
	stbi_image_free(imageData);

	// Create image to hold final texture
	vk::Image texImage;
	vk::DeviceMemory texImageMemory;
	texImage = createImage(width, height,
		vk::Format::eR8G8B8A8Unorm, vk::ImageTiling::eOptimal,
		vk::ImageUsageFlagBits::eTransferDst | vk::ImageUsageFlagBits::eSampled,
		vk::MemoryPropertyFlagBits::eDeviceLocal, &texImageMemory);

	// -- COPY DATA TO IMAGE --
	// Copy image data
	copyImageBuffer(mainDevice.logicalDevice, graphicsQueue,
		graphicsCommandPool, imageStagingBuffer, texImage, width, height);

	// Add texture data to vector for reference
	textureImages.push_back(texImage);
	textureImageMemory.push_back(texImageMemory);

	// Destroy stagin buffers
	mainDevice.logicalDevice.destroyBuffer(imageStagingBuffer, nullptr);
	mainDevice.logicalDevice.freeMemory(imageStagingBufferMemory, nullptr);
```

```
        // Return index of new texture image
        return textureImages.size() - 1;
}
```

Wew

## Clean up and use of the texture

Let's not forget to destroy the texture images and memory:

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

        for (auto i = 0; i < textureImages.size(); ++i)
        {
                mainDevice.logicalDevice.destroyImage(textureImages[i], nullptr);
                mainDevice.logicalDevice.freeMemory(textureImageMemory[i], nullptr);
        }
  ...
}
```

And now test a texture load:

```
void VulkanRenderer::init()
{
    ...
    // Pipeline
    ...
    createGraphicsCommandPool();

    // Texture
    int catTexture = createTextureImage("cat.jpg");

    // Objects
    ...
}
```

The code launches but there is a validation error:

```
Submitted command buffer expects VkNonDispatchableHandle [...] to be in layout
        VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL -- instead, current layout
        is VK_IMAGE_LAYOUT_UNDEFINED.
```

We will have to take the image and transition it to the vk::ImageLayout::eTransferDstOptimal format.

## Transitionning the layout

We will use a pipeline barrier to make this transition happen. First we will create a function to handle transition with a barrier:

`VulkanUtilities.h`

```
static void transitionImageLayout(vk::Device device, vk::Queue queue, vk::CommandPool commandPool,
        vk::Image image, vk::ImageLayout oldLayout, vk::ImageLayout newLayout)
{
        vk::CommandBuffer commandBuffer = beginCommandBuffer(device, commandPool);

        vk::ImageMemoryBarrier imageMemoryBarrier{};
        imageMemoryBarrier.oldLayout = oldLayout;
        imageMemoryBarrier.newLayout = newLayout;
        // Queue family to transition from
        imageMemoryBarrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
        // Queue family to transition to
        imageMemoryBarrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
        // Image being accessed and modified as part fo barrier
        imageMemoryBarrier.image = image;
        imageMemoryBarrier.subresourceRange.aspectMask = vk::ImageAspectFlagBits::eColor;
        // First mip level to start alterations on
        imageMemoryBarrier.subresourceRange.baseMipLevel = 0;
        // Number of mip levels to alter starting from baseMipLevel
        imageMemoryBarrier.subresourceRange.levelCount = 1;
        // First layer to starts alterations on
        imageMemoryBarrier.subresourceRange.baseArrayLayer = 0;
        // Number of layers to alter starting from baseArrayLayer
        imageMemoryBarrier.subresourceRange.layerCount = 1;

        vk::PipelineStageFlags srcStage;
        vk::PipelineStageFlags dstStage;

        // If transitioning from new image to image ready to receive data
        if (oldLayout == vk::ImageLayout::eUndefined &&
                newLayout == vk::ImageLayout::eTransferDstOptimal)
        {
                // Memory access stage transition must happen after this stage
                imageMemoryBarrier.srcAccessMask = vk::AccessFlagBits::eNone;
                // Memory access stage transition must happen before this stage
                imageMemoryBarrier.dstAccessMask = vk::AccessFlagBits::eTransferWrite;

                // Transfer from old layout to new layout has to occur after any
                // point of the top of the pipeline and before it attemps to to a
                // transfer write at the transfer stage of the pipeline
                srcStage = vk::PipelineStageFlagBits::eTopOfPipe;
                dstStage = vk::PipelineStageFlagBits::eTransfer;
        }

        commandBuffer.pipelineBarrier(
                // Pipeline stages (match to src and dst AccessMasks)
                srcStage, dstStage,
```

```
                // Dependency flags
                {},
                // Memory barrier count and data
                0, nullptr,
                // Buffer memory barrier count and data
                0, nullptr,
                // Image memory barrier count and data
                1, &imageMemoryBarrier
        );

        endAndSubmitCommandBuffer(device, commandPool, queue, commandBuffer);
}
```

We can now use this function juste before copying the texture data:

`VulkanRenderer.cpp`

```
int VulkanRenderer::createTextureImage(string filename)
{
    ...
        // -- COPY DATA TO IMAGE --
        // Transition image to be DST for copy operations
        transitionImageLayout(mainDevice.logicalDevice, graphicsQueue, graphicsCommandPool,
                texImage, vk::ImageLayout::eUndefined, vk::ImageLayout::eTransferDstOptimal);

        // Copy image data
        copyImageBuffer(mainDevice.logicalDevice, graphicsQueue,
                graphicsCommandPool, imageStagingBuffer, texImage, width, height);
    ...
}
```

Now it should work without an error.

Nevertheless, our image format is now optimal for a destination of a transfer. But we want it to be in a format optimal for a shader to use it.

# Prepare shader use

Update `transitionImageLayout`:

`VulkanUtilities.h`

```
...
    // If transitioning from new image to image ready to receive data
    if (oldLayout == vk::ImageLayout::eUndefined &&
            newLayout == vk::ImageLayout::eTransferDstOptimal)
    {
            // Memory access stage transition must happen after this stage
            imageMemoryBarrier.srcAccessMask = vk::AccessFlagBits::eNone;
            // Memory access stage transition must happen before this stage
            imageMemoryBarrier.dstAccessMask = vk::AccessFlagBits::eTransferWrite;

            // Transfer from old layout to new layout has to occur after any
            // point of the top of the pipeline and before it attemps to to a
            // transfer write at the transfer stage of the pipeline
            srcStage = vk::PipelineStageFlagBits::eTopOfPipe;
            dstStage = vk::PipelineStageFlagBits::eTransfer;
    }
    else if (oldLayout == vk::ImageLayout::eTransferDstOptimal &&
            newLayout == vk::ImageLayout::eShaderReadOnlyOptimal)
    {
            // Transfer is finished
            imageMemoryBarrier.srcAccessMask = vk::AccessFlagBits::eTransferWrite;
            // Before the shader reads
            imageMemoryBarrier.dstAccessMask = vk::AccessFlagBits::eShaderRead;

            srcStage = vk::PipelineStageFlagBits::eTransfer;
            dstStage = vk::PipelineStageFlagBits::eFragmentShader;
    }
...
```

We put the transition to action. Update VulkanRenderer::createTextureImage:

```
    ...
        // -- COPY DATA TO IMAGE --
        // Transition image to be DST for copy operations
        transitionImageLayout(mainDevice.logicalDevice, graphicsQueue, graphicsCommandPool,
                texImage, vk::ImageLayout::eUndefined, vk::ImageLayout::eTransferDstOptimal);

        // Copy image data
        copyImageBuffer(mainDevice.logicalDevice, graphicsQueue,
                graphicsCommandPool, imageStagingBuffer, texImage, width, height);

        // -- READY FOR SHADER USE --
        transitionImageLayout(mainDevice.logicalDevice, graphicsQueue, graphicsCommandPool,
                texImage, vk::ImageLayout::eTransferDstOptimal,
                vk::ImageLayout::eShaderReadOnlyOptimal);

        // Add texture data to vector for reference
    ...
```

## Create textures

Now we will use the images to create ImageViews, in order to be able to display them. Declare a new `createTexture` function.

```
int VulkanRenderer::createTexture(const string& filename)
{
        int textureImageLocation = createTextureImage(filename);

        vk::ImageView imageView = createImageView(textureImages[textureImageLocation],
                vk::Format::eR8G8B8A8Unorm, vk::ImageAspectFlagBits::eColor);
        textureImageViews.push_back(imageView);

        // TODO : Create Descriptor sets

        return 0;
}
```

In `init`, replace the function:

```
void VulkanRenderer::init()
{
    ...
    // Pipeline
    ...
    createGraphicsCommandPool();

    // Texture
                int catTexture = createTexture("cat.jpg");

    // Objects
    ...
}
```

Because we create an image view we have to clean it:

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

        for (auto i = 0; i < textureImages.size(); ++i)
        {
                mainDevice.logicalDevice.destroyImageView(textureImageViews[i], nullptr);
                ...
        }
  ...
```

# Create a texture sampler

`VulkanRenderer.h`

```
        vk::Sampler textureSampler;
        void createTextureSampler();
```

Use it in `init`:

`VulkanRenderer.cpp`

```
...
    // Commands
    createGraphicsCommandBuffers();
    createTextureSampler();
    createSynchronisation();
...
```

```
void VulkanRenderer::createTextureSampler()
{
        vk::SamplerCreateInfo samplerCreateInfo{};
        // How to render when image is magnified on screen
        samplerCreateInfo.magFilter = vk::Filter::eLinear;
        // How to render when image is minified on screen
        samplerCreateInfo.minFilter = vk::Filter::eLinear;
        // Texture wrap in the U direction
        samplerCreateInfo.addressModeU = vk::SamplerAddressMode::eRepeat;
        // Texture wrap in the V direction
        samplerCreateInfo.addressModeV = vk::SamplerAddressMode::eRepeat;
        // Texture wrap in the W direction
        samplerCreateInfo.addressModeW = vk::SamplerAddressMode::eRepeat;
        // When no repeat, texture become black beyond border
        samplerCreateInfo.borderColor = vk::BorderColor::eIntOpaqueBlack;
        // Coordinates ARE normalized. When true, coords are between 0 and image size
        samplerCreateInfo.unnormalizedCoordinates = false;
        // Fade between two mipmaps is linear
        samplerCreateInfo.mipmapMode = vk::SamplerMipmapMode::eLinear;
        // Add a bias to the mimmap level
        samplerCreateInfo.mipLodBias = 0.0f;
        samplerCreateInfo.minLod = 0.0f;
        samplerCreateInfo.maxLod = 0.0f;
        // Overcome blur when a texture is stretched because of perspective with angle
        samplerCreateInfo.anisotropyEnable = true;
        // Anisotropy number of samples
        samplerCreateInfo.maxAnisotropy = 16;

        textureSampler = mainDevice.logicalDevice.createSampler(samplerCreateInfo);
}
```

Don't forget to clean:

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

        mainDevice.logicalDevice.destroySampler(textureSampler);
  ...
```

If we run the code now, there is an error: we have not enable anisotropic sampling. We have to enable this feature in `createLogicalDevice`:

```
void VulkanRenderer::createLogicalDevice()
{
        ...
        // Features
        vk::PhysicalDeviceFeatures deviceFeatures {};
        deviceFeatures.samplerAnisotropy = true;
        deviceCreateInfo.pEnabledFeatures = &deviceFeatures;
    ...
```

We have to check the graphics card supports this feature in `checkDeviceSuitable`:

```
bool VulkanRenderer::checkDeviceSuitable(vk::PhysicalDevice device)
{
        ...
        return indices.isValid() && extensionSupported && swapchainValid
                && deviceFeatures.samplerAnisotropy;
}
```

Now we will setup the descriptor sets

# Sampler descriptor sets

## Pool

We need a descriptor pool for the sampler:

`VulkanRenderer.h`

```
        vk::DescriptorPool samplerDescriptorPool;
```

We start by creating descriptor pools at the end of `createDescriptorPool`:

```
void VulkanRenderer::createDescriptorPool()
{
	...
	// -- SAMPLER DESCRIPTOR POOL --
	// Texture sampler pool
	vk::DescriptorPoolSize samplerPoolSize{};
	// We assume one texture byobject.
	samplerPoolSize.descriptorCount = MAX_OBJECTS;

	vk::DescriptorPoolCreateInfo samplerPoolCreateInfo{};
	// The maximum for this is actually very high
	samplerPoolCreateInfo.maxSets = MAX_OBJECTS;
	samplerPoolCreateInfo.poolSizeCount = 1;
	samplerPoolCreateInfo.pPoolSizes = &samplerPoolSize;

	samplerDescriptorPool =
		mainDevice.logicalDevice.createDescriptorPool(samplerPoolCreateInfo);
}
```

## Layout

We currently have two descriptor sets inputs. We now create a descriptor set layout for the sampler.

`VulkanRenderer.h`

```
	vk::DescriptorSetLayout samplerDescriptorSetLayout;
```

```
void VulkanRenderer::createDescriptorSetLayout()
{
	// -- UNIFORM VALUES DESCRIPTOR SETS LAYOUT --
	...
	// -- SAMPLER DESCRIPTOR SETS LAYOUT --
	vk::DescriptorSetLayoutBinding samplerLayoutBinding;
	// Binding 0 for descriptor set 1
	samplerLayoutBinding.binding = 0;
	samplerLayoutBinding.descriptorType = vk::DescriptorType::eCombinedImageSampler;
	samplerLayoutBinding.descriptorCount = 1;
	samplerLayoutBinding.stageFlags = vk::ShaderStageFlagBits::eFragment;
	samplerLayoutBinding.pImmutableSamplers = nullptr;

	vector<vk::DescriptorSetLayoutBinding> samplerLayoutBindings{ samplerLayoutBinding };

	vk::DescriptorSetLayoutCreateInfo textureLayoutCreateInfo{};
	textureLayoutCreateInfo.bindingCount =
		static_cast<uint32_t>(samplerLayoutBindings.size());
	textureLayoutCreateInfo.pBindings = samplerLayoutBindings.data();

	samplerDescriptorSetLayout =
		mainDevice.logicalDevice.createDescriptorSetLayout(textureLayoutCreateInfo);
}
```

In `createGraphicsPipeline`, go to the pipeline lyout part and update it to add the new descriptor set:

```
void VulkanRenderer::createGraphicsPipeline()
{
    ...
        // -- PIPELINE LAYOUT --
        array<vk::DescriptorSetLayout, 2> descriptorSetLayouts{
                descriptorSetLayout, samplerDescriptorSetLayout
        };

        vk::PipelineLayoutCreateInfo pipelineLayoutCreateInfo{};
        pipelineLayoutCreateInfo.setLayoutCount =
                static_cast<uint32_t>(descriptorSetLayouts.size());
        pipelineLayoutCreateInfo.pSetLayouts = descriptorSetLayouts.data();
        pipelineLayoutCreateInfo.pushConstantRangeCount = 1;
        pipelineLayoutCreateInfo.pPushConstantRanges = &pushConstantRange;

        // Create pipeline layout
        pipelineLayout = mainDevice.logicalDevice.createPipelineLayout(pipelineLayoutCreateInfo);
    ...
}
```

## Descriptor sets

`VulkanRenderer.h`

```
        vector<vk::DescriptorSet> samplerDescriptorSets;
        int createTextureDescriptor(vk::ImageView textureImageView);
```

Previously we had one descriptor set per swapchain images, because the values would change for each image. But here, we will have one for each texture, that won't change.

```
int VulkanRenderer::createTextureDescriptor(VkImageView textureImageView)
{
	vk::DescriptorSet descriptorSet;

	vk::DescriptorSetAllocateInfo setAllocInfo{};
	setAllocInfo.descriptorPool = samplerDescriptorPool;
	setAllocInfo.descriptorSetCount = 1;
	setAllocInfo.pSetLayouts = &samplerDescriptorSetLayout;

	vk::Result result =
		mainDevice.logicalDevice.allocateDescriptorSets(&setAllocInfo, &descriptorSet);
	if (result != vk::Result::eSuccess)
	{
		throw std::runtime_error("Failed to allocate texture descriptor set.");
	}

	// Texture image info
	vk::DescriptorImageInfo imageInfo{};
	// Image layout when in use
	imageInfo.imageLayout = vk::ImageLayout::eShaderReadOnlyOptimal;
	// Image view to bind to set
	imageInfo.imageView = textureImageView;
	// Sampler to use for set
	imageInfo.sampler = textureSampler;

	// Write info
	vk::WriteDescriptorSet descriptorWrite{};
	descriptorWrite.dstSet = descriptorSet;
	descriptorWrite.dstBinding = 0;
	descriptorWrite.dstArrayElement = 0;
	descriptorWrite.descriptorType = vk::DescriptorType::eCombinedImageSampler;
	descriptorWrite.descriptorCount = 1;
	descriptorWrite.pImageInfo = &imageInfo;

	// Update new descriptor set
	mainDevice.logicalDevice.updateDescriptorSets(1, &descriptorWrite, 0, nullptr);

	// Add descriptor set to list
	samplerDescriptorSets.push_back(descriptorSet);

	return samplerDescriptorSets.size() - 1;
}
```

We can now use this function in `createTexture`:

```
int VulkanRenderer::createTexture(const string& filename)
{
        int textureImageLocation = createTextureImage(filename);

        vk::ImageView imageView = createImageView(textureImages[textureImageLocation],
                vk::Format::eR8G8B8A8Unorm, vk::ImageAspectFlagBits::eColor);
        textureImageViews.push_back(imageView);

        int descriptorLoc = createTextureDescriptor(imageView);

        // Return location of set with texture
        return descriptorLoc;
}
```

### Clean and init

Don't forget to clean:

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

        mainDevice.logicalDevice.destroyDescriptorPool(samplerDescriptorPool, nullptr);
        mainDevice.logicalDevice.destroyDescriptorSetLayout(samplerDescriptorSetLayout, nullptr);
    ...
}
```

Also move all the code of objects and texture creation at the end of `init` to avoid errors.

## Connect mesh and texture

### Texture id

We create a texture id for the mesh:

`VulkanMesh.h`

```
public:
        VulkanMesh(vk::PhysicalDevice physicalDeviceP, vk::Device deviceP,
                vk::Queue transferQueue, vk::CommandPool transferCommandPool,
                vector<Vertex>* vertices, vector<uint32_t>* indices, int texIdP);
        ...
        int getTexId() const { return texId; }
private:
        ...
        int texId;
```

We use this new function in `init`. We will create two texture with the same file instead of one, but this is for example sake, so that our code supports multiple textures.

`VulkanRenderer.cpp`

```
                VulkanMesh firstMesh = VulkanMesh(mainDevice.physicalDevice,
                        mainDevice.logicalDevice, graphicsQueue, graphicsCommandPool,
                        &meshVertices1, &meshIndices, createTexture("cat.jpg"));

                VulkanMesh secondMesh = VulkanMesh(mainDevice.physicalDevice,
                        mainDevice.logicalDevice, graphicsQueue, graphicsCommandPool,
                        &meshVertices2, &meshIndices, createTexture("cat.jpg"));
```

## Record commands

We can update the bind descriptor sets command:

```
void VulkanRenderer::recordCommands(uint32_t currentImage)
{
	...
	// Draw all meshes
	for (size_t j = 0; j < meshes.size(); ++j)
	{
		...
		// Bind descriptor sets
		array<vk::DescriptorSet, 2> descriptorSetsGroup {
			descriptorSets[currentImage], samplerDescriptorSets[meshes[j].getTexId()]
		};
		commandBuffers[currentImage].bindDescriptorSets(vk::PipelineBindPoint::eGraphics,
			pipelineLayout, 0, static_cast<uint32_t>(descriptorSetsGroup.size()),
			descriptorSetsGroup.data(), 0, nullptr);
			...
		}
		...
}
```

## Texture coordinates and shader update

Start with updating the Vertex struct:

`VulkanUtilities.h`

```
struct Vertex
{
	glm::vec3 pos;
	glm::vec3 col;
	glm::vec2 tex;
};
```

Nom in `init` add the required data:

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

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

We now have to update the attribute descriptions in `createGraphicsPipeline`:

```
    // Create pipeline

    ...
    // Different attributes
    array<VkVertexInputAttributeDescription, 3> attributeDescriptions;

    ...

    // Texture attributes
    attributeDescriptions[2].binding = 0;
    attributeDescriptions[2].location = 2;
    attributeDescriptions[2].format = VK_FORMAT_R32G32_SFLOAT;
    attributeDescriptions[2].offset = offsetof(Vertex, tex);
```

And finally modify the shaders:

`shader.vert`

```
#version 450

// From vertex input stage
layout(location = 0) in vec3 pos;
layout(location = 1) in vec3 col;
layout(location = 2) in vec2 tex;

// Uniform Buffer Object
layout(set = 0, 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;
layout(location = 1) out vec2 fragTex;

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

`shader.frag`

```
#version 450

// Input colors from vertex shader
layout(location = 0) in vec3 fragColor;
layout(location = 1) in vec2 fragTex;

layout(set = 1, binding = 0) uniform sampler2D textureSampler;

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

void main() {
    outColor = texture(textureSampler, fragTex);
}
```

Also change model matrices in the `main` to easily check your models:
`main.cpp`

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

        secondModel = glm::translate(secondModel, glm::vec3(0.0f, -1.0f, -1.0f));
        secondModel = glm::rotate(secondModel, glm::radians(-angle * 2),
                glm::vec3(0.0f, 0.0f, 1.0f));
...
```

And now the texture is displayed ! This project is available in the VulkanApp_Basic01_Textures folder.

# 3D models

## Preparation

### Open asset importer library

To load 3D meshes, we will use Assimp. You can get this 3d model loading library on by folloying instructions at https://github.com/assimp/assimp/blob/master/Build.md . On Windows, the easiest way is to use the vcpkg system.

Once the library in built and installed, include the files and link the lib. If you installed assimp with vcpkg, you will find the `include` and `lib` folders here : C:[path-to]\vcpkg\packages\assimp_x86-windows . Do not forget to name the library you are using in Linker/Input.

## New class type for model representations

Create a `VulkanMeshModel` class.

`VulkanMeshModel.h`

```
#pragma once
#include <glm/glm.hpp>
#include <assimp/scene.h>

#include <vector>
using std::vector;

#include "VulkanMesh.h"

class VulkanMeshModel
{
public:
	VulkanMeshModel();
	VulkanMeshModel(vector<VulkanMesh> meshesP);
	~VulkanMeshModel();

	size_t getMeshCount() const { return meshes.size(); };
	VulkanMesh* getMesh(size_t index);

	glm::mat4 getModel() const { return model; };
	void setModel(glm::mat4 modelP) { model = modelP; }
	void destroyMeshModel();


private:
	vector<VulkanMesh> meshes;
	glm::mat4 model;
};
```

A scene in assimp is a hierarchy of mesh models. We will need to get the texture filenames for each model mesh. Here is the implementation:

`VulkanMeshModel.cpp`

```
#include "VulkanMeshModel.h"

VulkanMeshModel::VulkanMeshModel()
{
}

VulkanMeshModel::VulkanMeshModel(vector<VulkanMesh> meshesP)
	: meshes(meshesP), model(glm::mat4(1.0f))
{
}

VulkanMeshModel::~VulkanMeshModel()
{
}

VulkanMesh* VulkanMeshModel::getMesh(size_t index)
{
	if (index >= meshes.size())
	{
		throw std::runtime_error("Attempted to access a mesh with not attributed index");
	}
	return &meshes[index];
}

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

# Loading a model

Include assimp and some new variables in the renderer:

`VulkanRenderer.h`

```
#include <assimp/Importer.hpp>
#include <assimp/scene.h>
#include <assimp/postprocess.h>
#include "VulkanMeshModel.h"
...
vector<VulkanMeshModel> meshModels;
void createMeshModel(const string& filename);
```

## Loading texture

We will first load the textures of a model. We will ensure to have a one-to-one relatationship between the loaded textures and the vulkan's texture ids.

`VulkanMeshModel.h`

```
static vector<string> loadMaterials(const aiScene* scene);
```

`VulkanMeshModel.cpp`

```
vector<string> VulkanMeshModel::loadMaterials(const aiScene* scene)
{
	// Create one-to-one size list of texture
	vector<string> textures(scene->mNumMaterials);
	// Go through each material and copy its texture file name if it exists
	for (size_t i = 0; i < textures.size(); ++i)
	{
		// Get material
		aiMaterial* material = scene->mMaterials[i];
		// Initialize texture name with empty string
		textures[i] = "";
		// Check for a diffuse texture
		if (material->GetTextureCount(aiTextureType_DIFFUSE))
		{
			// Get the path of the texture file
			aiString path;
			if (material->GetTexture(aiTextureType_DIFFUSE, 0, &path) == AI_SUCCESS)
			{
				// Cut off any absolute directory information already present
				int index = string(path.data).rfind("\\");
				string filename = string(path.data).substr(index + 1);
				textures[i] = filename;
			}
		}
	}
	return textures;
}
```

We can now start implementing `createMeshModel`:

`VulkaRenderer.cpp`

```
void VulkanRenderer::createMeshModel(const string& filename)
{
	// Import model scene
	Assimp::Importer importer;
	// We want the model to be in triangles, to flip vertically texels uvs,
	// and optimize the use of vertices
	const aiScene* scene = importer.ReadFile(filename, aiProcess_Triangulate |
		aiProcess_FlipUVs | aiProcess_JoinIdenticalVertices);
	if (!scene)
	{
		throw std::runtime_error("Failed to load mesh model: " + filename);
	}

	// Load materials with one to one relationship with texture ids
	vector<string> textureNames = VulkanMeshModel::loadMaterials(scene);

	// Conversion to material list ID to descriptor array ids (we don't keep empty files)
	vector<int> matToTex(textureNames.size());
	// Loop over texture names and create textures for them
	for (size_t i = 0; i < textureNames.size(); ++i)
	{
		if (textureNames[i].empty())
		{
			// Texture 0 will be reserved for a default texture
			matToTex[i] = 0;
		}
		else
		{
			// Return the texture's id
			matToTex[i] = createTexture(textureNames[i]);
		}
	}
	...
}
```

The function is not finished. We will now load the meshs of the model.

## Loading meshes

Meshes are represented under the shape of a node tree. We will have to go through this tree to retrieve the hierarchy of meshes, represented as nodes.

`VulkanMeshModel.h`

```
        static VulkanMesh loadMesh(vk::PhysicalDevice physicalDeviceP,
                vk::Device deviceP, vk::Queue transferQueue,
                vk::CommandPool transferCommandPool,
                aiMesh* mesh, const aiScene* scene, vector<int> matToTex);

        static vector<VulkanMesh> loadNode(vk::PhysicalDevice physicalDeviceP,
                vk::Device deviceP, vk::Queue transferQueue,
                vk::CommandPool transferCommandPool,
                aiNode* node, const aiScene* scene, vector<int> matToTex);
```

We will first load the mesh by getting all vertices and indices, then use this function to load the node, ie the mesh plus its children in node hierarchy.

`VulkanMeshModel.cpp`

```
VulkanMesh VulkanMeshModel::loadMesh(vk::PhysicalDevice physicalDeviceP,
	vk::Device deviceP, vk::Queue transferQueue,
	vk::CommandPool transferCommandPool, aiMesh* mesh,
	const aiScene* scene, vector<int> matToTex)
{
	vector<Vertex> vertices(mesh->mNumVertices);
	vector<uint32_t> indices;

	// Copy all vertices
	for (size_t i = 0; i < mesh->mNumVertices; ++i)
	{
		// Position
		vertices[i].pos = {
			mesh->mVertices[i].x, mesh->mVertices[i].y, mesh->mVertices[i].z
		};
		// Tex coords if they exists
		if (mesh->mTextureCoords[0])
		{
			vertices[i].tex = {
				mesh->mTextureCoords[0][i].x, mesh->mTextureCoords[0][i].y
			};
		}
		else
		{
			vertices[i].tex = { 0.0f, 0.0f };

		}
		// Vertex color (white)
		vertices[i].col = { 1.0f, 1.0f, 1.0f };
	}

	// Copy all indices, stored by face (triangle)
	for (size_t i = 0; i < mesh->mNumFaces; ++i)
	{
		aiFace face = mesh->mFaces[i];
		for (size_t j = 0; j < face.mNumIndices; ++j)
		{
			indices.push_back(face.mIndices[j]);
		}
	}

	// Create new mesh
	VulkanMesh newMesh = VulkanMesh(physicalDeviceP, deviceP, transferQueue,
		transferCommandPool, &vertices, &indices, matToTex[mesh->mMaterialIndex]);
	return newMesh;
}
```

```
vector<VulkanMesh> VulkanMeshModel::loadNode(vk::PhysicalDevice physicalDeviceP,
        vk::Device deviceP, vk::Queue transferQueue,
        vk::CommandPool transferCommandPool, aiNode* node,
        const aiScene* scene, vector<int> matToTex)
{
        vector<VulkanMesh> meshes;
        // Go through each mesh at this node and create it, then add it to our meshList
        for (size_t i = 0; i < node->mNumMeshes; ++i)
        {
                // Load mesh
                meshes.push_back(loadMesh(physicalDeviceP, deviceP, transferQueue,
                        transferCommandPool, scene->mMeshes[node->mMeshes[i]], scene, matToTex));
                // Explanation of scene->mMeshes[node->mMeshes[i]]:
                // The scene actually hold the data for the meshes, and the nodes store ids of
                // meshes, that relate to the scene meshes.
        }

        // Go through each node attached to this node and load it,
        // then append their meshes to this node's meshes
        for (size_t i = 0; i < node->mNumChildren; ++i)
        {
                vector<VulkanMesh> newMeshes = loadNode(physicalDeviceP, deviceP, transferQueue,
                        transferCommandPool, node->mChildren[i], scene, matToTex);
                meshes.insert(end(meshes), begin(newMeshes), end(newMeshes));
        }

        return meshes;
}
```

We can now get back and complete `createMeshModel`:

`VulkanRenderer.cpp`

```
void VulkanRenderer::createMeshModel(string filename)
{
	// Import model scene
	Assimp::Importer importer;
	// We want the model to be in triangles, to flip vertically texels uvs,
	// and optimize the use of vertices
	const aiScene* scene = importer.ReadFile(filename, aiProcess_Triangulate |
		aiProcess_FlipUVs | aiProcess_JoinIdenticalVertices);
	if (!scene)
	{
		throw std::runtime_error("Failed to load mesh model: " + filename);
	}

	// Load materials with one to one relationship with texture ids
	vector<string> textureNames = VulkanMeshModel::loadMaterials(scene);

	// Conversion to material list ID to descriptor array ids (we don't keep empty files)
	vector<int> matToTex(textureNames.size());
	// Loop over texture names and create textures for them
	for (size_t i = 0; i < textureNames.size(); ++i)
	{
		if (textureNames[i].empty())
		{
			// Texture 0 will be reserved for a default texture
			matToTex[i] = 0;
		}
		else
		{
			// Return the texture's id
			matToTex[i] = createTexture(textureNames[i]);
		}
	}

	// Load in all our meshes
	vector<VulkanMesh> modelMeshes =
		VulkanMeshModel::loadNode(mainDevice.physicalDevice, mainDevice.logicalDevice,
		graphicsQueue, graphicsCommandPool, scene->mRootNode, scene, matToTex);

	auto meshModel = VulkanMeshModel(modelMeshes);
	meshModels.push_back(meshModel);
}
```

## Cleanup

`VulkanRenderer.cpp`

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

	for (auto& model : meshModels)
	{
		model.destroyMeshModel();
	}
	...
```

# Display the 3d model

## Get and load a model

Download a 3d .obj model on free3d.com. You must keep the `.obj` file, the `.mtl` (material) file and the textures folder. Put the obj and material files into a `models` folder inside your VulkanApp project, and the textures into the `textures` folder.

At the end of `init`, you can add:

```
createMeshModel("models/Futuristic combat jet.obj");    // Name of your object
```

You will get an error:

```
ERROR: Failed to allocate texture descriptor set.
```

The error will come from the fact we don't have enough descriptor sets for our new object. Push the max number of objects to 20:

```
const int MAX_OBJECTS = 20;
```

(In function of your model, this number may need to be higher.)

You scene should now load.

## Draw our models

Let's modify the commands in `recordCommands`. The goal is now to push one constant for a mesh model, giving the model matrix, then go through all children meshes to draw them.

```
void VulkanRenderer::recordCommands(uint32_t currentImage)
{
	...
	// Draw all meshes
	for (size_t j = 0; j < meshModels.size(); ++j)
	{
		// Push constants to given shader stage
		VulkanMeshModel model = meshModels[j];
		glm::mat4 modelMatrix = model.getModel();
		commandBuffers[currentImage].pushConstants(pipelineLayout,
			vk::ShaderStageFlagBits::eVertex, 0, sizeof(Model), &modelMatrix);

		// We have one model matrix for each object, then several children meshes

		for (size_t k = 0; k < model.getMeshCount(); ++k)
		{
			// Bind vertex buffer
			vk::Buffer vertexBuffers[] = { model.getMesh(k)->getVertexBuffer() };
			vk::DeviceSize offsets[] = { 0 };
			commandBuffers[currentImage].bindVertexBuffers(0, 1,
				vertexBuffers, offsets);

			// Bind index buffer
			commandBuffers[currentImage].bindIndexBuffer(
				model.getMesh(k)->getIndexBuffer(), 0, vk::IndexType::eUint32);

			// Bind descriptor sets
			array<vk::DescriptorSet, 2> descriptorSetsGroup{
					descriptorSets[currentImage],
					samplerDescriptorSets[model.getMesh(k)->getTexId()] };
			commandBuffers[currentImage].bindDescriptorSets(
				vk::PipelineBindPoint::eGraphics, pipelineLayout,
				0, static_cast<uint32_t>(descriptorSetsGroup.size()),
				descriptorSetsGroup.data(), 0, nullptr);

			// Execute pipeline
			commandBuffers[currentImage].drawIndexed(
				static_cast<uint32_t>(model.getMesh(k)->getIndexCount()),
				1, 0, 0, 0);
		}
	}
	...
```

Your code should now draw. Nevertheless, the view is not optimal. Change it in `init` (view matrix) so you can see your model. For me it will be:

```
viewProjection.view = glm::lookAt(glm::vec3(10.0f, 10.0f, 20.0f),
        glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(0.0f, 1.0f, 0.0f));
```

You can remove the quad's code. In order to use a default texture when the model texture cannot load, you can create a texture in `init` before loading the model:

```
                // Default texture
                createTexture("cat.jpg");

                // Load model
                createMeshModel("models/Futuristic combat jet.obj");
```

We also need to change the `updateModel` function:

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

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

## Loading in main

We can also set `createMeshModel` to public and use it in 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;

	// Load model
	vulkanRenderer.createMeshModel("models/Futuristic combat jet.obj");

	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 rotationModelMatrix(1.0f);
		rotationModelMatrix = glm::translate(rotationModelMatrix,
			glm::vec3(-0.0f, 0.0f, -1.0f));
		rotationModelMatrix = glm::rotate(rotationModelMatrix, glm::radians(angle),
			glm::vec3(0.0f, 1.0f, 0.0f));
		vulkanRenderer.updateModel(0, rotationModelMatrix);

		vulkanRenderer.draw();
	}

	clean();
	return 0;
}
```

The only problem is that the 0 in `vulkanRenderer.updateModel(0, rotationModelMatrix);` is hard coded. We need an id from the list of models. Let's modify `createMeshModel` to return such an id.

```
int VulkanRenderer::createMeshModel(string filename)
{
	...
	return meshModels.size() - 1;
}
```

Now we can use it in main:

`Main.cpp`

```
	// Load model
	int modelId = vulkanRenderer.createMeshModel("models/Futuristic combat jet.obj");

	while (!glfwWindowShouldClose(window))
	{
		...
		vulkanRenderer.updateModel(modelId, rotationModelMatrix);
		...
	}
	...
```

## Conclusion

We now have seen the basics of a vulkan renderer, loading a 3D model. The complete code is available in the VulkanApp_Basic02_3DMesh folder.

This code is not at all ready for production. It was meant more as a demonstration of the various usual vulkan concepts. If you want to implement a real-time engine, use this knowledge with https://vkguide.dev, especially if you are interested in engine programming.

# Mipmaps

Mipmaps are a Level Of Details system for textures. You generate a collection of smaller textures that will be used when the texture display is too far for all the details to be relevent.

## Creating mipmap levels

We first need to choose the number of mipmap levels we need for our texture. There is a simple algorithm to do so:

`VulkanRenderer.cpp`

```
int VulkanRenderer::createTextureImage(const string& filename, uint32_t& mipLevels)
{
        // Load image file
        int width, height;
        vk::DeviceSize imageSize;
        stbi_uc* imageData = loadTextureFile(filename, &width, &height, &imageSize);
        mipLevels = static_cast<uint32_t>(std::floor(std::log2(std::max(width, height)))) + 1;
        ...
```

This calculates the number of levels in the mip chain. The `max` function selects the largest dimension. The `log2` function calculates how many times that dimension can be divided by 2. The `floor` function handles cases where the largest dimension is not a power of 2. 1 is added so that the original image has a mip level.

To use this value, we need to change the `createImage`, `createImageView`, and `transitionImageLayout` functions to allow us to specify the number of mip levels. Add a mipLevels parameter to the functions:

`VulkanUtilities.h`

```
static void transitionImageLayout(vk::Device device, vk::Queue queue, vk::CommandPool commandPool,
        vk::Image image, vk::ImageLayout oldLayout, vk::ImageLayout newLayout, uint32_t mipLevels)
{
        ...
        imageMemoryBarrier.subresourceRange.levelCount = mipLevels;
        ...
}
```

`VulkanRenderer.cpp`

```
int VulkanRenderer::createTexture(const string& filename)
{
	uint32_t mipLevels{ 0 };
	int textureImageLocation = createTextureImage(filename, mipLevels);

	vk::ImageView imageView = createImageView(textureImages[textureImageLocation],
		vk::Format::eR8G8B8A8Unorm, vk::ImageAspectFlagBits::eColor, mipLevels);
	textureImageViews.push_back(imageView);

	int descriptorLoc = createTextureDescriptor(imageView);

	// Return location of set with texture
	return descriptorLoc;
}

vk::Image VulkanRenderer::createImage(uint32_t width, uint32_t height, uint32_t mipLevels,
	vk::Format format, vk::ImageTiling tiling,
	vk::ImageUsageFlags useFlags, vk::MemoryPropertyFlags propFlags,
	vk::DeviceMemory* imageMemory)
{
	...
	imageCreateInfo.mipLevels = mipLevels;
	...
}

vk::ImageView VulkanRenderer::createImageView(vk::Image image, vk::Format format,
	vk::ImageAspectFlagBits aspectFlags, uint32_t mipLevels)
{
	...
	// Number of mipmap level to view
	viewCreateInfo.subresourceRange.levelCount = mipLevels;
}

int VulkanRenderer::createTextureImage(const string& filename, uint32_t& mipLevels)
{
	...
	texImage =
		createImage(width, height, mipLevels, vk::Format::eR8G8B8A8Unorm,
		vk::ImageTiling::eOptimal,
		vk::ImageUsageFlagBits::eTransferDst | vk::ImageUsageFlagBits::eSampled,
		vk::MemoryPropertyFlagBits::eDeviceLocal, &texImageMemory);

	// -- COPY DATA TO IMAGE --
	// Transition image to be DST for copy operations
	transitionImageLayout(mainDevice.logicalDevice, graphicsQueue, graphicsCommandPool,
		texImage, vk::ImageLayout::eUndefined, vk::ImageLayout::eTransferDstOptimal,
```

```
                mipLevels);

        // Copy image data
        copyImageBuffer(mainDevice.logicalDevice, graphicsQueue,
                graphicsCommandPool, imageStagingBuffer, texImage, width, height);

        // -- READY FOR SHADER USE --
        transitionImageLayout(mainDevice.logicalDevice, graphicsQueue, graphicsCommandPool,
                texImage, vk::ImageLayout::eTransferDstOptimal,
                vk::ImageLayout::eShaderReadOnlyOptimal, mipLevels);
        ...
}

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

void VulkanRenderer::createSwapchain()
{
        ...
        for (VkImage image : images) // We are using handles, not values
        {
                ...
                // Create image view
                swapchainImage.imageView = createImageView(image, swapchainImageFormat,
                        vk::ImageAspectFlagBits::eColor, 1);

                swapchainImages.push_back(swapchainImage);
        }
}

void VulkanRenderer::createDepthBufferImage()
{
        ...
        depthBufferImageView = createImageView(depthBufferImage, depthFormat,
                vk::ImageAspectFlagBits::eDepth, 1);
}
```

# Generating mipmaps

Our texture image now has multiple mip levels, but the staging buffer can only be used to fill mip level 0. The other levels are still undefined. To fill these levels we need to generate the data from the single level that we have. We will use the `vk::CommandBuffer::blitImage` command. This command performs copying, scaling, and filtering operations. We will call this multiple times to blit data to each level of our texture image.

`blitImage` is considered a transfer operation, so we must inform Vulkan that we intend to use the texture image as both the source and destination of a transfer. Add `vk::ImageUsageFlagBits::eTransferSrc` to the texture image's usage flags in createTextureImage:

`VulkanRenderer.cpp`

```
int VulkanRenderer::createTextureImage(const string& filename, uint32_t& mipLevels)
{
	...
	// Create image to hold final texture
	vk::Image texImage;
	vk::DeviceMemory texImageMemory;
	texImage = createImage(width, height, mipLevels, vk::Format::eR8G8B8A8Unorm,
		vk::ImageTiling::eOptimal,vk::ImageUsageFlagBits::eTransferDst
		vk::ImageUsageFlagBits::eSampled | vk::ImageUsageFlagBits::eTransferSrc,
		vk::MemoryPropertyFlagBits::eDeviceLocal, &texImageMemory);
	...
	/*
	// -- READY FOR SHADER USE --
	transitionImageLayout(mainDevice.logicalDevice, graphicsQueue, graphicsCommandPool,
		texImage, vk::ImageLayout::eTransferDstOptimal,
		vk::ImageLayout::eShaderReadOnlyOptimal, mipLevels);
	*/
	...
```

Each level will be transitioned to `vk::ImageLayout::eShaderReadOnlyOptimal` after the blit command reading from it is finished.

We're now going to write the function that generates the mipmaps. We're going to make several transitions, so we'll use `vk::CommandBuffer::pipelineBarrier`. Step to step explanations are given in the code comments :

`VulkanUtilities.h`

```
static void generateMipmaps(vk::Device device, vk::Queue queue, vk::CommandPool commandPool,
        vk::Image image, int32_t texWidth, int32_t texHeight, uint32_t mipLevels)
{
        vk::CommandBuffer commandBuffer = beginCommandBuffer(device, commandPool);

        // The fields set below will remain the same for all barriers.
        // On the contrary, subresourceRange.miplevel, oldLayout, newLayout, srcAccessMask,
        // and dstAccessMask will be changed for each transition.
        vk::ImageMemoryBarrier barrier{};
        barrier.image = image;
        barrier.srcQueueFamilyIndex = vk::QueueFamilyIgnored;
        barrier.dstQueueFamilyIndex = vk::QueueFamilyIgnored;
        barrier.subresourceRange.aspectMask = vk::ImageAspectFlagBits::eColor;
        barrier.subresourceRange.baseArrayLayer = 0;
        barrier.subresourceRange.layerCount = 1;
        barrier.subresourceRange.levelCount = 1;

        int32_t mipWidth = texWidth;
        int32_t mipHeight = texHeight;

        // This loop will record each of the blitImage commands.
        // Note that the loop variable starts at 1, not 0.
        for (uint32_t i = 1; i < mipLevels; i++)
        {
                // First, we transition level i - 1 to vk::ImageLayout::eTransferSrcOptimal.
                // This transition will wait for level i - 1 to be filled, either from the
                //  previous blit command, or from vk::CommandBuffer::copyBufferToImage.
                barrier.subresourceRange.baseMipLevel = i - 1;
                barrier.oldLayout = vk::ImageLayout::eTransferDstOptimal;
                barrier.newLayout = vk::ImageLayout::eTransferSrcOptimal;
                barrier.srcAccessMask = vk::AccessFlagBits::eTransferWrite;
                barrier.dstAccessMask = vk::AccessFlagBits::eTransferRead;

                // The current blit command will wait on this transition.
                commandBuffer.pipelineBarrier(
                        vk::PipelineStageFlagBits::eTransfer,
                        vk::PipelineStageFlagBits::eTransfer, {},
                        0, nullptr, 0, nullptr, 1, &barrier);

                // Next, we specify the regions that will be used in the blit operation.
                // The source mip level is i - 1 and the destination mip level is i.
                // The two elements of the srcOffsets array determine the 3D region
                // that data will be blitted from. dstOffsets determines the region
                // that data will be blitted to.
                vk::ImageBlit blit{};
                blit.srcOffsets[0] = vk::Offset3D{ 0, 0, 0 };
```

```
blit.srcOffsets[1] = vk::Offset3D{ mipWidth, mipHeight, 1 };
blit.srcSubresource.aspectMask = vk::ImageAspectFlagBits::eColor;
blit.srcSubresource.mipLevel = i - 1;
blit.srcSubresource.baseArrayLayer = 0;
blit.srcSubresource.layerCount = 1;
// The X and Y dimensions of the dstOffsets[1] are divided by two since each mip
// level is half the size of the previous level.The Z dimension of
// srcOffsets[1] and dstOffsets[1] must be 1, since a 2D image has
// a depth of 1.
blit.dstOffsets[0] = vk::Offset3D{ 0, 0, 0 };
blit.dstOffsets[1] = vk::Offset3D{ mipWidth > 1 ? mipWidth / 2 : 1,
        mipHeight > 1 ? mipHeight / 2 : 1, 1 };
blit.dstSubresource.aspectMask = vk::ImageAspectFlagBits::eColor;
blit.dstSubresource.mipLevel = i;
blit.dstSubresource.baseArrayLayer = 0;
blit.dstSubresource.layerCount = 1;


// Now, we record the blit command. Note that textureImage is used for both
// the srcImage and dstImage parameter. This is because we're blitting between
// different levels of the same image. The source mip level was just transitioned
// to vk::ImageLayout::eTransferSrcOptimal and the destination level
// is still in vk::ImageLayout::eTransferDstOptimal from createTextureImage.
// The last parameter is the same filtering options here that we had when making
// the vk::Sampler. We use the vk::Filter::eLinear to enable interpolation.
commandBuffer.blitImage(
        image, vk::ImageLayout::eTransferSrcOptimal,
        image, vk::ImageLayout::eTransferDstOptimal,
        1, &blit,
        vk::Filter::eLinear);

// This barrier transitions mip level i - 1 to
// vk::ImageLayout::eShaderReadOnlyOptimal. This transition waits on the
// current blit command to finish. All sampling operations will wait
// on this transition to finish.
barrier.oldLayout = vk::ImageLayout::eTransferSrcOptimal;
barrier.newLayout = vk::ImageLayout::eShaderReadOnlyOptimal;
barrier.srcAccessMask = vk::AccessFlagBits::eTransferRead;
barrier.dstAccessMask = vk::AccessFlagBits::eShaderRead;

commandBuffer.pipelineBarrier(
        vk::PipelineStageFlagBits::eTransfer,
        vk::PipelineStageFlagBits::eFragmentShader, {},
        0, nullptr,
        0, nullptr,
        1, &barrier);
```

```
            // At the end of the loop, we divide the current mip dimensions by two. We
            // check each dimension before the division to ensure that dimension never
            // becomes 0. This handles cases where the image is not square, since one
            // of the mip dimensions would reach 1 before the other dimension. When
            // this happens, that dimension should remain 1 for all remaining
            // levels.
            if (mipWidth > 1) mipWidth /= 2;
            if (mipHeight > 1) mipHeight /= 2;
        }

        // Before we end the command buffer, we insert one more pipeline barrier. This barrier
        // transitions the last mip level from vk::ImageLayout::eTransferDstOptimal to
        // vk::ImageLayout::eShaderReadOnlyOptimal. This wasn't handled by the loop, since the
        // last mip level is never blitted from.
        barrier.subresourceRange.baseMipLevel = mipLevels - 1;
        barrier.oldLayout = vk::ImageLayout::eTransferDstOptimal;
        barrier.newLayout = vk::ImageLayout::eShaderReadOnlyOptimal;
        barrier.srcAccessMask = vk::AccessFlagBits::eTransferWrite;
        barrier.dstAccessMask = vk::AccessFlagBits::eShaderRead;

        commandBuffer.pipelineBarrier(
                vk::PipelineStageFlagBits::eTransfer,
                vk::PipelineStageFlagBits::eFragmentShader, {},
                0, nullptr,
                0, nullptr,
                1, &barrier);

        endAndSubmitCommandBuffer(device, commandPool, queue, commandBuffer);
}
```

We can now call this function in `createTextureImage`:

`VulkanRenderer.cpp`

```
int VulkanRenderer::createTextureImage(const string& filename, uint32_t& mipLevels)
{
        ...
        // -- READY FOR SHADER USE --
        /*
        transitionImageLayout(mainDevice.logicalDevice, graphicsQueue, graphicsCommandPool,
                texImage, vk::ImageLayout::eTransferDstOptimal,
                vk::ImageLayout::eShaderReadOnlyOptimal, mipLevels);
        */

        // -- GENERATE MIPMAPS AND READY FOR SHADER USE --
        generateMipmaps(mainDevice.logicalDevice, graphicsQueue, graphicsCommandPool,
                texImage, width, height, mipLevels);
        ...
}
```

## Linear filtering support

It is very convenient to use a built-in function like `vk::CommandBuffer::blitImage` to generate all the mip levels, but unfortunately it is not guaranteed to be supported on all platforms. It requires the texture image format we use to support linear filtering, which can be checked with the `vk::PhysiclDevice::getFormatProperties` function. We will add a check to the `generateMipmaps` function for this. First add two additional parameters that pqss the physical device and specifies the image format:

`VulkanUtilities.h`

```
static void generateMipmaps(vk::Device device, vk::PhysicalDevice physicalDevice,
  vk::Queue queue, vk::CommandPool commandPool, vk::Image image,
        vk::Format imageFormat, int32_t texWidth, int32_t texHeight,
        uint32_t mipLevels)
```

`VulkanRenderer.cpp`

```
...
generateMipmaps(mainDevice.logicalDevice, mainDevice.physicalDevice, graphicsQueue,
         graphicsCommandPool, texImage, vk::Format::eR8G8B8A8Srgb, width, height, mipLevels);
...
```

Then check the image format is supported:

`VulkanUtilities.h`

```
static void generateMipmaps(vk::Device device, vk::PhysicalDevice physicalDevice,
        vk::Queue queue, vk::CommandPool commandPool, vk::Image image, vk::Format imageFormat,
        int32_t texWidth, int32_t texHeight, uint32_t mipLevels)
{
        // Check if image format supports linear blitting. We create a texture image with
        // the optimal tiling format, so we need to check optimalTilingFeatures.
        vk::FormatProperties formatProperties = physicalDevice.getFormatProperties(imageFormat);
        if (!(formatProperties.optimalTilingFeatures
                & vk::FormatFeatureFlagBits::eSampledImageFilterLinear))
        {
                throw std::runtime_error(
                        "texture image format does not support linear blitting!"
                );
        }

        vk::CommandBuffer commandBuffer = beginCommandBuffer(device, commandPool);
        ...
}
```

There are two alternatives in the case the image format is not supported. You could implement a function that searches common texture image formats for one that does support linear blitting, or you could implement the mipmap generation in software with a library like `stb_image_resize`. Each mip level can then be loaded into the image in the same way that you loaded the original image.

It should be noted that it is uncommon in practice to generate the mipmap levels at runtime anyway. Usually they are pregenerated and stored in the texture file alongside the base level to improve loading speed.

## Sampler

Some theory about the sampler.

While the `vk::Image` holds the mipmap data, `vk::Sampler` controls how that data is read while rendering. Vulkan allows us to specify `minLod`, `maxLod`, `mipLodBias`, and `mipmapMode`. When a texture is sampled, the sampler selects a mip level according to the following pseudocode (not to be included in your project):

```
// Smaller when the object is close, may be negative
lod = getLodLevelFromScreenSize();
lod = clamp(lod + mipLodBias, minLod, maxLod);

// Clamped to the number of mip levels in the texture
level = clamp(floor(lod), 0, texture.mipLevels - 1);

if (mipmapMode == vk::SamplerMipmapMode::eNearest) {
    color = sample(level);
} else {
    color = blend(sample(level), sample(level + 1));
}
```

If `samplerInfo.mipmapMode` is `vk::SamplerMipmapMode::eNearest`, lod selects the mip level to sample from. If the mipmap mode is `vk::SamplerMipmapMode::eLinear`, lod is used to select two mip levels to be sampled. Those levels are sampled and the results are linearly blended.

The sample operation is also affected by lod:

```
if (lod <= 0) {
    color = readTexture(uv, magFilter);
} else {
    color = readTexture(uv, minFilter);
}
```

If the object is close to the camera, `magFilter` is used as the filter. If the object is further from the camera, `minFilter` is used. Normally, lod is non-negative, and is only 0 when close the camera. `mipLodBias` lets us force Vulkan to use lower lod and level than it would normally use.

In our case, the different textures from our model's material have 10 or 12 mip levels. We will choose the minimum for the sampler.

`VulkanRenderer.cpp`

```
void VulkanRenderer::createTextureSampler()
{
...
        samplerCreateInfo.maxLod = 10.0f;
...
}
```

Now your model is textured using mipmaps. In addition to optimal texture resource usage, it avoids the Moiré effect (https://i.stack.imgur.com/pPOY1.png) on your textures. You can see the use of lower

resolution texture by setting `samplerCreateInfo.minLod` to `5.0f` in the `createTextureSampler` function.

The code for this project is available in the `VulkanApp_Basics03_Mipmaps` folder.

# Multisampling

We can see, on our 3D scene, that the model edges pixels are scaling on background. This effect is called aliasing. We can remove it by using multisampling anti-aliasing (MSAA). Such a method sample multiple points inside the targeted pixel and average the colors of each point to obtain the final pixel color. This will result in a smoother transition between different models or between models and background.

## Available sample count

First, we need to know how many samples our GPU can support. We update `VulkanRenderer::getPhysicalDevice` to do so.

`VulkanRenderer.h`

```
        vk::SampleCountFlagBits msaaSamples{ vk::SampleCountFlagBits::e1 };
```

``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;

vk::SampleCountFlags counts = deviceProperties.limits.framebufferColorSampleCounts
        & deviceProperties.limits.framebufferDepthSampleCounts;
if (counts & vk::SampleCountFlagBits::e64) msaaSamples = vk::SampleCountFlagBits::e64;
else if (counts & vk::SampleCountFlagBits::e32) msaaSamples = vk::SampleCountFlagBits::e32;
else if (counts & vk::SampleCountFlagBits::e16) msaaSamples = vk::SampleCountFlagBits::e16;
else if (counts & vk::SampleCountFlagBits::e8) msaaSamples = vk::SampleCountFlagBits::e8;
else if (counts & vk::SampleCountFlagBits::e4) msaaSamples = vk::SampleCountFlagBits::e4;
else if (counts & vk::SampleCountFlagBits::e2) msaaSamples = vk::SampleCountFlagBits::e2;
else msaaSamples = vk::SampleCountFlagBits::e1;
}
```

## Setting up a render target

In MSAA, each pixel is sampled in an offscreen buffer which is then rendered to the screen. This new buffer is slightly different from regular images we've been rendering to - they have to be able to store more than one sample per pixel. Once a multisampled buffer is created, it has to be resolved to the default framebuffer (which stores only a single sample per pixel). This is why we have to create an additional render target and modify our current drawing process. We only need one render target since only one drawing operation is active at a time, just like with the depth buffer. Add the following class members:

`VulkanRenderer.h`

```
        vk::Image colorImage;
        vk::DeviceMemory colorImageMemory;
        vk::ImageView colorImageView;
```

This new image will have to store the desired number of samples per pixel. Modify the `createImage` function by adding a `numSamples` parameter:

`VulkanRenderer.cpp`

```
vk::Image VulkanRenderer::createImage(uint32_t width, uint32_t height, uint32_t mipLevels,
        vk::SampleCountFlagBits numSamples, vk::Format format, vk::ImageTiling tiling,
        vk::ImageUsageFlags useFlags, vk::MemoryPropertyFlags propFlags,
        vk::DeviceMemory* imageMemory)
{
        ...
        // Number of samples for multi sampling
        imageCreateInfo.samples = numSamples;
        ...
}
```

Update the call to this function to add the missing argument. Set it on one sample.

We will now create a multisampled color buffer. We're also only one mip level, since this is enforced by the Vulkan specification in case of images with more than one sample per pixel. Also, this color buffer doesn't need mipmaps since it's not going to be used as a texture:

`VulkanRenderer.cpp`

```
void VulkanRenderer::createColorBufferImage()
{
        vk::Format colorFormat = swapchainImageFormat;

        colorImage = createImage(swapchainExtent.width, swapchainExtent.height, 1,
         msaaSamples, colorFormat, vk::ImageTiling::eOptimal,
         vk::ImageUsageFlagBits::eTransientAttachment | vk::ImageUsageFlagBits::eColorAttachment,
         vk::MemoryPropertyFlagBits::eDeviceLocal, &colorImageMemory);
        colorImageView = createImageView(colorImage, colorFormat, vk::ImageAspectFlagBits::eColor,
}
```

Call this function in `init`:

```
...
                createGraphicsPipeline();
                createColorBufferImage();               // HERE
                createDepthBufferImage();
...
```

Now that we have a multisampled color buffer in place it's time to take care of depth. Modify `createDepthBufferImage` and update the number of samples used by the depth buffer:

```
void VulkanRenderer::createDepthBufferImage()
{
...
        // Create image and image view
        depthBufferImage = createImage(swapchainExtent.width, swapchainExtent.height, 1,
                msaaSamples, depthFormat, vk::ImageTiling::eOptimal,
                vk::ImageUsageFlagBits::eDepthStencilAttachment,
                vk::MemoryPropertyFlagBits::eDeviceLocal,
                &depthBufferImageMemory);
...
}
```

Do not forget to clean:

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

        mainDevice.logicalDevice.destroyImageView(colorImageView);
        mainDevice.logicalDevice.destroyImage(colorImage);
        mainDevice.logicalDevice.freeMemory(colorImageMemory);
        ...
}
```

We made it past the initial MSAA setup, now we need to start using this new resource in our graphics pipeline, framebuffer, render pass and see the results!

## Adding new attachments

Let's take care of the render pass first. Modify `createRenderPass` and update color and depth attachment creation info:

```
void VulkanRenderer::createRenderPass()
{
        ...
        // Number of samples to write for multisampling
        colorAttachment.samples = msaaSamples;
        ...
        // Image data layout after render pass
        colorAttachment.finalLayout = vk::ImageLayout::eColorAttachmentOptimal;
        ...
        // Depth attachment of renderpass
        ...
        depthAttachment.samples = msaaSamples;
```

You'll notice that we have changed the finalLayout to `vk::ImageLayout::eColorAttachmentOptimal`. That's because multisampled images cannot be presented directly. We first need to resolve them to a regular image. This requirement does not apply to the depth buffer, since it won't be presented at any point. Therefore we will have to add only one new attachment for color which is a so-called **resolve attachment**:

```
void VulkanRenderer::createRenderPass()
{
...
        // Color resolve attachment
        vk::AttachmentDescription colorAttachmentResolve{};
        colorAttachmentResolve.format = swapchainImageFormat;
        colorAttachmentResolve.samples = vk::SampleCountFlagBits::e1;
        colorAttachmentResolve.loadOp = vk::AttachmentLoadOp::eDontCare;
        colorAttachmentResolve.storeOp = vk::AttachmentStoreOp::eStore;
        colorAttachmentResolve.stencilLoadOp = vk::AttachmentLoadOp::eDontCare;
        colorAttachmentResolve.stencilStoreOp = vk::AttachmentStoreOp::eDontCare;
        colorAttachmentResolve.initialLayout = vk::ImageLayout::eUndefined;
        colorAttachmentResolve.finalLayout = vk::ImageLayout::ePresentSrcKHR;

        array<vk::AttachmentDescription, 3> renderPassAttachments{
                        colorAttachment, depthAttachment, colorAttachmentResolve };
        renderPassCreateInfo.attachmentCount =
                static_cast<uint32_t>(renderPassAttachments.size());
        renderPassCreateInfo.pAttachments = renderPassAttachments.data();

        // -- REFERENCES --
        ...
        vk::AttachmentReference colorAttachmentResolveReference{};
        colorAttachmentResolveReference.attachment = 2;
        colorAttachmentResolveReference.layout = vk::ImageLayout::eColorAttachmentOptimal;

        // -- SUBPASSES --
        ...
        subpass.pResolveAttachments = &colorAttachmentResolveReference;
}
```

With the render pass in place, modify `createFramebuffers` and add the new image view to the list:

```
        // Setup attachments
        array<vk::ImageView, 3> attachments{
                colorImageView, depthBufferImageView, swapchainImages[i].imageView
        };
...
```

Finally, tell the newly created pipeline to use more than one sample by modifying `createGraphicsPipeline`:

```
...
        // -- MULTISAMPLING --
        ...
        // Number of samples to use per fragment
        multisamplingCreateInfo.rasterizationSamples = msaaSamples;
...
```

If you run the code now, you'll see the edges of the model are now smooth.

## Quality improvements

There are certain limitations of our current MSAA implementation which may impact the quality of the output image in more detailed scenes. For example, we're currently not solving potential problems caused by shader aliasing, i.e. MSAA only smoothens out the edges of geometry but not the interior filling. This may lead to a situation when you get a smooth polygon rendered on screen but the applied texture will still look aliased if it contains high contrasting colors. One way to approach this problem is to enable Sample Shading which will improve the image quality even further, though at an additional performance cost:

```
void createLogicalDevice() {
        ...
        // Features
        vk::PhysicalDeviceFeatures deviceFeatures{};
        deviceFeatures.samplerAnisotropy = true;
        deviceFeatures.sampleRateShading = true;
        deviceCreateInfo.pEnabledFeatures = &deviceFeatures;
        ...
}

void createGraphicsPipeline() {
        ...
        // -- MULTISAMPLING --
        ...
        // Enable sample shading in the pipeline
        multisamplingCreateInfo.sampleShadingEnable = true;
        // Min fraction for sample shading; closer to one is smoother
        multisamplingCreateInfo.minSampleShading = 0.2f;
        ...
}
```

# Conclusion

This ends the Vulkan introduction lesson. You can find the final project in the VulkanApp_Basic04_Multisampling folder.

In the next lessons, we'll learn more modern rendering techniques.