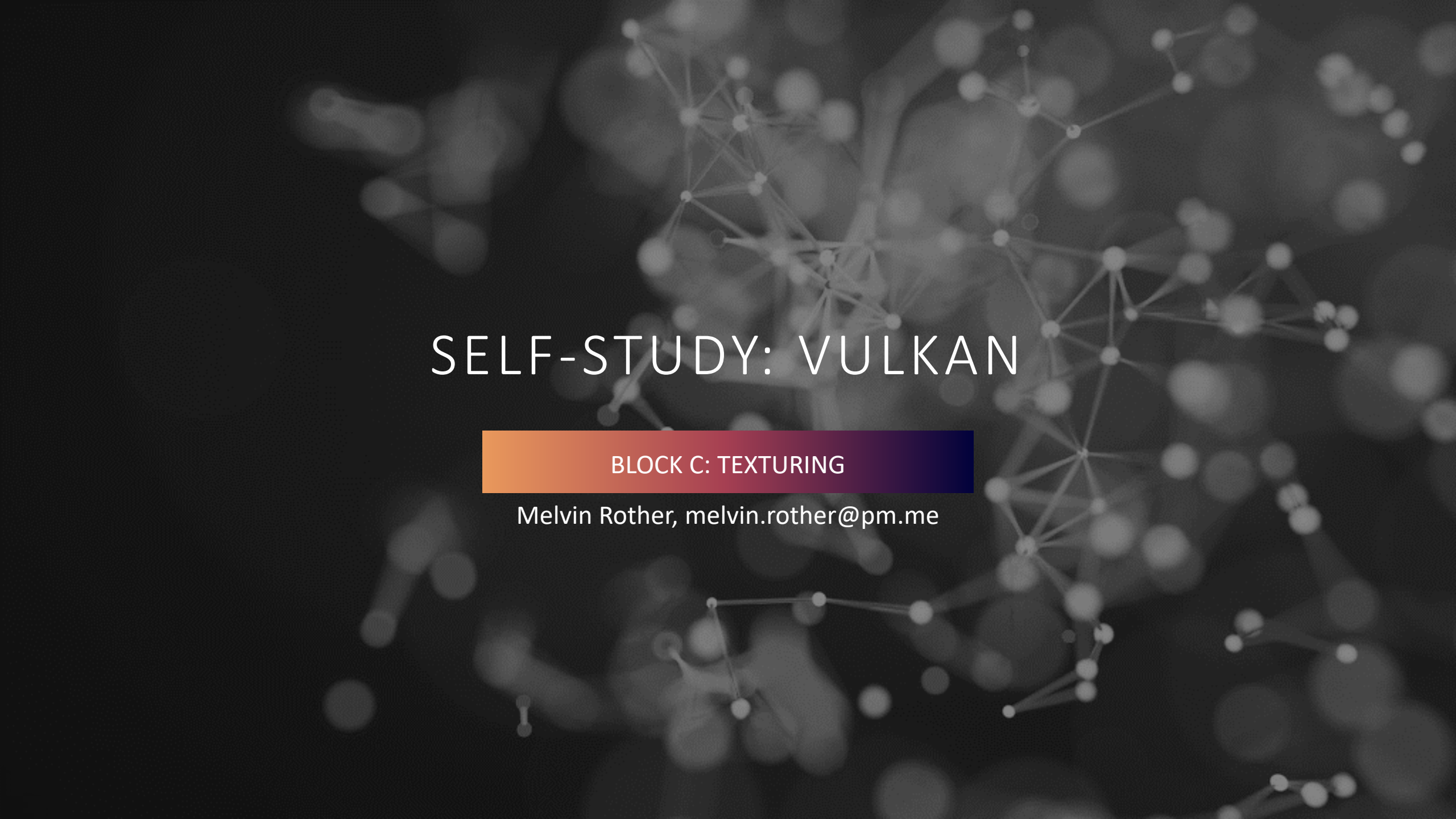


SELF-STUDY: VULKAN

BLOCK C: TEXTURING

Melvin Rother, melvin.rother@pm.me

CONTENT

Topics

0. Preparation: Generic, staging and index buffer
1. Uniform Buffers
2. Texture Mapping

PREPARATION

Before starting texturing, I will need to learn a few things. First, I need to abstract my Vertex buffer class into a generic buffer class to allow me to send any kind of data to the GPU instead of only vertices.

Then I will need to implement staging buffers to be able to copy pixels to the GPU.

Finally, I will need to learn about and implement descriptor layouts and buffers to buffer the MVP matrix to the GPU without including it in the vertex data.

Afterwards, I will have everything set up to upload a texture to the GPU, sample and apply it.

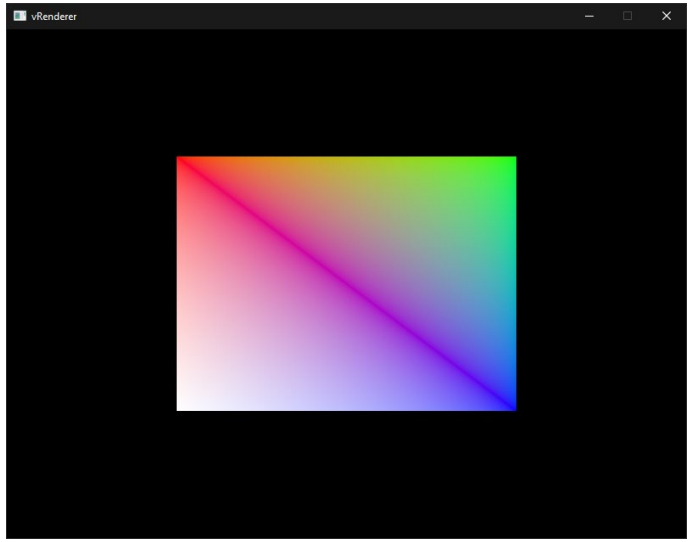
0.1: GENERIC AND STAGING BUFFERS

- Abstracting the vertex buffer is relatively straightforward:
  1. Pass the BufferSize, BufferUsageFlags and theMemoryPropertyFlags into the CreateBuffer function instead of hardcoding them
  2. Change the data contained in the buffer from a vector of vertices to a void pointer pointing to where the data is in memory
  3. Then I reworked the VertexBuffer as a child class to the Buffer class that automatically passes the right flags to the base class' CreateBuffer function

- Using the new generic buffer class, I can then implement staging buffers. That is necessary because while the memory type of the vertex buffer allows for it to be accessed from the CPU, it is not the one that is the most efficient for the GPU to read. However, the one that is the most optimal (which is the one having the VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT) is usually not accessible from the CPU.
- A staging buffer can remedy that situation. By creating two vertex buffers, one CPU-accessible staging buffer and the actual vertex buffer in Device Local Memory. Then the vertex data is uploaded to the CPU-accessible staging buffer and copied into the GPU-accessible final vertex buffer from which it is then read
- Implementing this for the Vertex Buffer is straightforward:
  1. Modify the vertex buffer creation to create two buffers (the same way all buffers are created)
  2. The staging buffer needs the VK_BUFFER_USAGE_TRANSFER_SRC_BIT to enable the use of the Transfer command, allowing to copy data from it to the vertex buffer
  3. The vertex buffer needs the VK_BUFFER_USAGE_TRANSFER_DST_BIT to enable the use of the Transfer command, allowing to copy data to it from the staging buffer, the VK_BUFFER_USAGE_VERTEX_BUFFER_BIT to designate it as a Vertex Buffer and the VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT flag to mark it as located in device local memory
- Data can then be copied from the staging buffer to the vertex buffer
  1. Create a temporary command buffer (using vkAllocateCommandBuffers)
  2. Record the command buffer

0.2: INDEX BUFFER

- To avoid copying a lot of redundant data and to make my life easier, I need to implement an index buffer (a buffer containing pointers to vertices in the vertex array to enable me to reuse them for several triangles)
- Creating an Index Buffer is like creating a vertex buffer. There are only two differences:
  - The data passed into it is not a vector of vertices, but rather a vector of uint16_t
  - The usage type of the index buffer is now VK_BUFFER_USAGE_INDEX_BUFFER_BIT
- Using the index buffer is then trivial:
  - Bind the Index Buffer using vkCmdBindIndexBuffer and set the index type to VK_INDEX_TYPE_UINT16
  - Then, instead of using VkCmdDraw use VkCmdDrawIndexed
- Now memory is saved by reusing vertices depending on the indices provided



- To optimize even further it is recommended to store multiple buffers of the same type into a single VkBuffer and make use of offsets many commands allow you to set. This allows for better memory alignment and is thus more cache-friendly

1. UNIFORM BUFFER

- Data can be passed to the Vertex Shader as part of the vertex attributes already, but doing so is not always a good use of memory
- Global variables (like the MVP matrix) could be included in the vertex attributes, but since only one is needed, that would be a huge amount of wasted memory
- This can be remedied by creating a uniform buffer that contains the global data and then accessing it through resource descriptors. To do so I have to
    1. Create a descriptor layout which describes the type of resource that is going to be accessed in the shader
    2. Allocate a descriptor set which describes which buffer or image will be bound to the descriptors
    3. Bind the descriptor set
    4. Create the uniform buffer object

1.1 UNIFORM BUFFER DESCRIPTOR SET LAYOUT

Creating a Descriptor Set layout is similar to everything else created in Vulkan:

1. Create a VkDescriptorSetlayoutBinding struct and fill its members
    1. binding: the “slot” from which the data will be accessible in the shader
    2. descriptorType: what type of descriptor, in this case, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER
    3. stageFlags: in which shader stages can the descriptor be referenced
    4. pImmutableSamplers: used for image samplers, used in texturing later
2. Create a VkDescriptorSetLayoutCreateInfo struct and fill its members
    1. Binding count: the amount of VkDescriptorSetLayoutBindings that should be bound
    2. pBindings: a pointer to the previously created DescriptorSetLayoutBindings
3. Call vkCreateDescriptorSetLayout to create the Layout
4. Bind it in the VkPipelineLayoutCreateInfo by setting the setLayoutCount and pSetLayouts members
5. When closing the program, make sure to call vkDestroyDescriptorSetLayout to properly clean them up

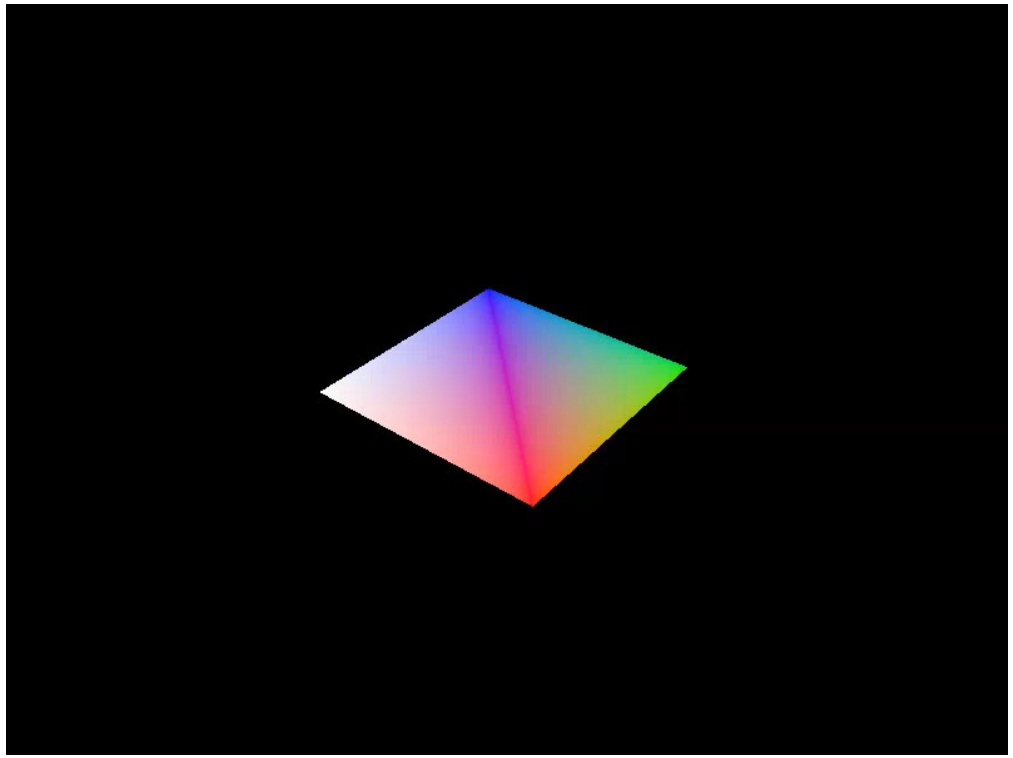
1.2 CREATING THE UNIFORM BUFFER

- Creating the Uniform Buffer itself can be done similarly to making the generic buffer (in fact, my own implementation is simply a child class of the generic buffer).
- The only difference is that it maps its memory only once and never unmaps it. That way, a persistent pointer can be stored that allows constant direct access to the memory that is mapped to the GPU memory on the CPU.
- Since this data is accessed & modified every frame, this is more performant than the standard map memory -> copy data into it -> unmap memory cycle that other buffers use as mapping memory is not a free operation
- It also requires no staging buffer
- It is then easy to update the Uniform data by copying the data to the persistently mapped memory using memcpy

1.3 DESCRIPTOR POOLS & SETS

- The descriptor layout describes the type of descriptors that can be bound
- The descriptor pool allows to allocate descriptor sets from them that can then be bound
- Creating a descriptor Pool is simple
    1. Create and Fill the members of a VkDescriptorPoolSize struct
    2. Then create a VkDescriptorPoolCreateInfo struct
    3. Then call VkCreateDescriptorPool and pass it the device, the VkDescriptorPoolCreateInfo and a pointer to the VkDescriptorPool that should hold the handle to the newly created Pool
    4. Destroy the Pool after the program has finished using VkDestroyDescriptorPool
- From there, allocating descriptor sets is trivial:
    1. Create a VkDescriptorSetAllocateInfo struct and pass it the descriptor pool, the descriptor set count and a pointer to the descriptor layout raw data
    2. Then call vkAllocateDescriptorSets
- Now the only thing still left to do before using the descriptors is to configure them
    1. Loop over the descriptors
    2. Create a VkDescriptorBufferInfo and fill its members
    3. Then update the descriptor configuration using a VkWriteDescriptorSet struct -> set the descriptor type to VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER and set the pBufferInfo member to point to the previously created VkDescriptorBufferInfo struct
    4. Then call VkUpdateDescriptorSets
- To use the created descriptors, simply call vkCmdBindDescriptorSets before vkCmdDrawIndexed

1.3 DESCRIPTOR POOLS & SETS



2. TEXTURE MAPPING

- Adding texturing to an application requires these four steps
    1. Creating a Vulkan Image object
    2. Load an image file and fill the image object with the loaded pixels
    3. Create an Image Sampler
    4. Add a combined image sampler descriptor to sample colors from the created texture

2.1 CREATING A VULKAN IMAGE OBJECT

- Creating a Vulkan Image Object and filling it with data is similar in principle to vertex buffer creation
- It also makes use of a staging resource which is then copied into a final image that can be used for rendering
  - You can either use a staging image or a VkBuffer object to stage the data before copying it into the final image. Using a staging buffer is faster on some hardware than using a staging image, however
- Creating an image involves similar steps as creating buffers
  1. Query the memory requirements
  2. Allocate device memory
  3. Bind device memory
- An additional step for image creation is using the right layout -> which layout is the most optimal for the operations you plan to execute on it?
  - VK_IMAGE_LAYOUT_PRESENT_SRC_KHR: best for presentation
  - VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL: optimal as an attachment for writing colours from the fragment shader
  - VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL: optimal as a source for transferring data, for example when calling vkCmdCopyImageToBuffer
  - VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL: optimal to transfer data to, for example when calling vkCmdCopyImageToBuffer
  - VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL: optimal for sampling from a shader
- To make sure that the image sources are synchronized (e.g. make sure images are written before they are read etc), the use of resource barriers is also required

2.1 CREATING A VULKAN IMAGE OBJECT

- Creating a Vulkan Image Object and filling it with data is similar in principle to vertex buffer creation
- It also makes use of a staging resource which is then copied into a final image that can be used for rendering
  - You can either use a staging image or a VkBuffer object to stage the data before copying it into the final image. Using a staging buffer is faster on some hardware than using a staging image, however
- Creating an image involves similar steps as creating buffers
  1. Query the memory requirements
  2. Allocate device memory
  3. Bind device memory
- An additional step for image creation is using the right layout -> which layout is the most optimal for the operations you plan to execute on it?
  - VK_IMAGE_LAYOUT_PRESENT_SRC_KHR: best for presentation
  - VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL: optimal as an attachment for writing colours from the fragment shader
  - VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL: optimal as a source for transferring data, for example when calling vkCmdCopyImageToBuffer
  - VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL: optimal to transfer data to, for example when calling vkCmdCopyImageToBuffer
  - VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL: optimal for sampling from a shader
- To make sure that the image sources are synchronized (e.g. make sure images are written before they are read etc), the use of resource barriers is also required (see next slide)

2.1.2 LAYOUT TRANSITIONS

- Before data can be copied from the Buffer to the Image, the image has to be in the right layout first
- To transition an image to a specific layout, a few steps are necessary

1. Create an image memory barrier. A pipeline barrier like this is used to synchronize access to resources
    1. Create a VkImageMemoryBarrier Struct
    2. Fill its members
        - oldLayout: the old layout of the image. Use VK_IMAGE_LAYOUT_UNDEFINED if it does not matter what happens to the existing content of the image
        - newLayout: self-explanatory
        - src/dstQueueFamilyIndex: if the barrier is used to transfer queue family ownership, these are the source and destination queues. Use VK_QUEUE_FAMILY_IGNORED if you do not want to do this (this is not the default)
        - Image/subresourceRange: the image & which specific part of the image that is affected

    3. Call vkCmdPipelineBarrier to submit the barrier to the Command Buffer
        1. Parameter: command buffer
        2. Parameter: which pipeline stage the operations occur before the barrier
        3. Parameter: pipeline stage in which operations will wait on the barrier
        4. Parameter: either 0 or VK_DEPENDENCY_BY_REGION (can already read the regions of a resource that have already been finished, even before the whole resource has finished being written to
        - The last three reference arrays of pipeline barriers of the three available types of memory

2.1 CREATING A VULKAN IMAGE OBJECT

Now we can create our image:

1. Create a VkImageCreateInfo struct and fill its members, then call Call vkCreateImage
   - imageType: what kind of coordinate system will Vulkan use to address the texels. In case of a texture, this will be VK_IMAGE_TYPE_2D
   - imageFormat: the format of the image, should be the same for the texels as the pixels in the staging buffer to make sure the copy operation succeeds.
   - Tiling: how the texels are laid out. Can either be VK_IMAGE_TILING_LINEAR (row-major order) or VK_IMAGE_TILING_OPTIMAL (implementation defined order for optimal access)
   - initialLayout: whether or not to keep the texels after the very first transition. Can be either VK_IMAGE_LAYOUT_UNDEFINED (discard the texels, not usable by the GPU) or VK_IMAGE_LAYOUT_PREINITIALIZED (keep texels, not usable by the GPU)
   - usage: usage flags (same semantics as during buffer creation). In this case should be VK_IMAGE_USAGE_TRANSFER_DST_BIT to make sure it can receive the data from the staging buffer and VK_IMAGE_USAGE_SAMPLED_BIT
   - sharingMode: whether or not the image will be shared between queue families
   - samples: multisampling. In this case VK_SAMPLE_COUNT_1_BIT
2. Allocate memory for the image by querying the memory requirements, creating a VkMemoryAllocateInfo struct and passing it to vkAllocateMemory
   - To query the memory requirements, create a VkmemoryRequirements struct and pass it to vkGetImageMemoryRequirements to initialize it
   - For the VkMemoryAllocateinfo, set allocationSize to memoryRequirements.size and set the memoryTypeIndex using by querying it using memoryRequirement.memoryTypeBits
3. Afterwards, call vkBindMemory and pass it the newly created image and the image memory
4. Then copy over the data from the staging buffer to the image. First transition the image layout (as described on the previous slide) to VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL. Then copy the data by creating a VkBufferImageCopy struct, filling its members and calling vkCopyBufferToImage (do not forget to record a command buffer while doing so). Afterwards, transition the image again to VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL

2.1 CREATING A VULKAN IMAGE OBJECT

The last thing left to do before creating a sampler is to create an image view for the newly created image. As can be seen from the creation of the swap chain images and the frame buffer, images are not directly accessed. Rather, they are accessed through image views, and the images used for texturing are no different.

1. Create a VkImageViewCreateInfo struct and initialise its members:
    - image: the just created image
    - viewType: the same as the image type, in this case VK_IMAGE_VIEW_TYPE_2D
    - format: same as image, in this case VK_FORMAT_S8R8G8B8A8_SRGB
    - aspectMask: VK_IMAGE_ASPECT_COLOR_BIT

2. Call vkCreateImageView and pass the VkImageViewCreateInfo to it

2.3 IMAGE SAMPLER

There are three things to be done to use an image sampler: create it, bind it and use it in the shader. Samplers can apply filtering and transformations, so they are more often used to access textures than reading texels directly

- Creating the sampler usually happens after the creation of the texture image and its image view
- To create a sampler, create a VkSamplerCreateInfo struct, initialize its members and then pass it to vkCreateSampler
  - magFilter: how to interpolate texels that are magnified. In this case VK_FILTER_LINEAR, but there are many more filters that can be applied
  - minFilter: how to interpolate texels that are minified. In this case VK_FILTER_LINEAR
  - addressModeU/V/W: the addressing mode per axis. Can be
    - VK_SAMPLER_ADDRESS_MODE_REPEAT: repeats the texture after going beyond its dimensions. Used here
    - VK_SAMPLER _ADDRESS_MODE_MIRRORED_REPEAT: same as REPEAT, but with inverted coordinates to mirror the image beyond its bounds
    - VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE: uses the colour of the edge closest to the coordinates beyond the image bounds
    - VK_SAMPLER _ADDRESS_MODE_MIRROR _CLAMP_TO_EDGE: like CLAMP_TO_EDGE, but mirrored
    - VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER: when sampling beyond the bounds of the image, return solid colour
  - anisotropyEnable: whether or not anisotropic filtering should be used. VK_TRUE, unless the performance is a concern. Update the query for physical device features to also include samplerAnisotropy as it is an optional device feature
  - maxAnisotropy: depends on the VkPhysicalDeviceProperties. It contains a member limits that in itself has a member called maxSamplerAnisotropy that can be used here
  - borderColor: which colour is returned when sampling beyond the image's bounds using CLAMP_TO_BORDER
  - unnormalizedCoordinates: which coordinate system should be used to address the texels. If VK_FALSE, coordinates are in the range 0 – 1, if VK_TRUE coordinates are in the range 0 – TexWidth/Height
  - compareEnable: VK_FALSE (mostly used in percentage-closer filtering)
  - compareOp: VK_COMPARE_OP_ALWAYS
  - mipmapMode: how to sample the mipmaps, in this case VK_SAMPLER_MODE_LINEAR
  - mipLodBias, minLod, maxLod: level of detail settings, in this case 0.0f
- Do not forget to call VkDestroySampler before destroying the image views

2.3 IMAGE SAMPLER

The last thing left to do is to make it possible for shaders to access an image resource through a sampler. This is done through descriptors, or more accurately through a combined image sampler descriptor. The process is similar to the descriptors for the uniform buffers.

1. Modify the descriptor layout, descriptor pool and descriptor set to include a combined image sampler
2. Modify the fragment shader to read colours from the texture using the sampler

- Modify the descriptor layout by adding a VkDescriptorSetLayoutBinding for a combined image sampler. This works the same as for the uniform buffer, with a few changes:
  - binding: 1, as it is the second binding
  - descriptorType: VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER
  - stageFlags: VK_SHADER_STAGE_FRAGMENT_BIT (this needs to be accessible in the fragment shader, unlike the uniform buffer that needs to be accessible in the vertex shader)
- Create a larger descriptor pool to make room for the additional sampler
  - Add another VkPoolSize of Type VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER to the DescriptorPoolCreateInfo
  - Add a VkDescriptorPoolSize for the new descriptor
- Bind the image and sampler to the descriptor set
  - Add a VkDescriptorImageInfo struct. Make sure its imageLayout is VK_IMAGE_LAYOUT_SHADER_READ_OPTIMAL
  - Add a VkDescriptorBufferInfo struct for the descriptor set to take the ImageInfo into account. That is the same as with the Uniform buffer, except that the the descriptorType is VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER

2.3 IMAGE SAMPLER

Modifying the shader is then trivial. Although this is not Vulkan code, I will still note it here for completeness' sake. Make sure that the vertices include texture coordinates and that the VertexInputBindingDescription takes them into account.

- Make the vertex shader pass through the texture coordinates
  - Add an incoming layout that accepts the texture coordinates (layout(location = 2) in vec2 inTexCoord)
  - Then add an output that returns the texture coordinates the same way: layout(location = 1) out vec2 outTexCoord

- Modify the fragment shader to accept and use the texture coordinates:
  - Add an incoming layout that accepts the texture coordinates from the vertex shader the same way as done above
  - Bind the sampler: layout(binding = 1) uniform sampler2D textureSampler;
  - When calculating the color, call texture(textureSampler, outTexCoord);

FINAL RESULT