Vulkan Lab 5: Texture Mapping (Vulkan 1.3)

1 INTRODUCTION

In previous labs, we've given our objects colour and simulated lighting, but they still look flat and artificial. **Texture Mapping** is the technique that provides the single biggest jump in visual realism. It's the process of "wrapping" a 2D image—called a texture—onto the surface of a 3D model.

This allows us to add intricate details like wood grain, brick patterns, text, or any complex surface property without increasing the geometric complexity of the model itself. This lab will guide you through the complete Vulkan workflow for loading, managing, and sampling a texture, which is one of the most fundamental skills in modern 3D graphics.

2 LEARNING OBJECTIVES

Upon successful completion of this lab, you will be able to:

- **Understand Texture Coordinates (UVs):** Explain how 2D (u, v) coordinates are used to map a 3D model's vertices to a 2D image.
- **Modify Vertex Data:** Extend your Vertex struct and vertex input description to include texture coordinates.
- **Load Image Files:** Use the stb_image.h library to load image data from a file into CPU memory.
- **Manage Vulkan Image Resources:** Create, allocate, and manage core Vulkan objects: VkImage (GPU memory), VkImageView (the "lens"), and VkSampler (the "reader").
- **Transfer Image Data:** Implement the crucial process of using a staging buffer to copy pixel data from the CPU to optimal-tiled GPU memory.
- **Execute Image Layout Transitions:** Correctly use Vulkan barriers to transition a VkImage between layouts (e.g., UNDEFINED -> TRANSFER_DST -> SHADER_READ_ONLY).
- **Bind Textures:** Update your descriptor set layout, pool, and sets to bind a texture and sampler to your shaders.
- **Modify GLSL Shaders:** Use GLSL's sampler2D type and texture() function to sample the texture and apply it to your model's color.

3 CORE CONCEPTS

- **Texture Coordinates (UVs):** Every vertex in a 3D mesh can store a 2D coordinate, known as a (u, v) or texture coordinate. This coordinate maps that specific vertex to a point on a 2D texture image. The u axis represents the horizontal direction (0.0 to 1.0, left to right), and the v axis represents the vertical direction (0.0 to 1.0, top to bottom). The GPU interpolates these coordinates across the surface of a triangle, so every pixel on the surface knows which pixel (or "texel") to read from the texture.

- **Image Loading (stb_image.h):** Vulkan itself does not know how to load a .png or .jpg file. We must use a separate library for this. stb_image.h is a popular, lightweight, header-only library. You simply add it to your project, and it provides functions like stbi_load() to read an image file into a raw buffer of pixel data on the CPU.

- **VkImage vs. VkBuffer:** A VkBuffer is a general-purpose, 1D block of memory. A VkImage is a specialized memory object optimized for 1D, 2D, or 3D image data. It can store data in an optimal tiling format, which arranges pixels in a way that is extremely fast for the GPU to read from (known as "texture locality").

- **Staging and Layout Transitions:** We cannot directly write pixel data from the CPU into an OPTIMAL tiled VkImage. The workflow is:

  1. **CPU:** Load image pixels using stb_image.h.
  2. **Staging VkBuffer:** Create a VkBuffer with HOST_VISIBLE memory (CPU-accessible) and TRANSFER_SRC *usage. memcpy* the pixel data into this buffer.
  3. **GPU VkImage:** Create the final VkImage with DEVICE_LOCAL (fast GPU-only) memory, OPTIMAL tiling, and TRANSFER_DST + SAMPLED usage.
  4. Commands: Record a command buffer to:
     a. Transition the VkImage layout from UNDEFINED to TRANSFER_DST_OPTIMAL (preparing it to be written to).
     b. Copy data from the Staging Buffer to the VkImage (vkCmdCopyBufferToImage).
     c. Transition the VkImage layout from TRANSFER_DST_OPTIMAL to SHADER_READ_ONLY_OPTIMAL (preparing it to be read by the shader).
  5. **Submit:** Submit this command buffer once during initialization.

- **VkImageView and VkSampler:**
  - **VkImageView:** An image object (VkImage) holds the raw data. A VkImageView is a "view" or "lens" that describes *how to interpret* that data (e.g., as a 2D texture, its format, etc.).
  - **VkSampler:** A VkSampler is a separate object that tells the shader *how to read* from the image. It controls:

    - **Filtering:** VK_FILTER_LINEAR (blurry, smooth) vs. VK_FILTER_NEAREST (pixelated, sharp).
    - **Addressing:** What to do outside the [0, 1] UV range? VK_SAMPLER_ADDRESS_MODE_REPEAT, CLAMP_TO_EDGE, etc.
- **Combined Image Sampler:** Vulkan shaders often use a sampler2D type, which combines an image view and a sampler. We bind these using a special descriptor, VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER.

4 LAB EXERCISES

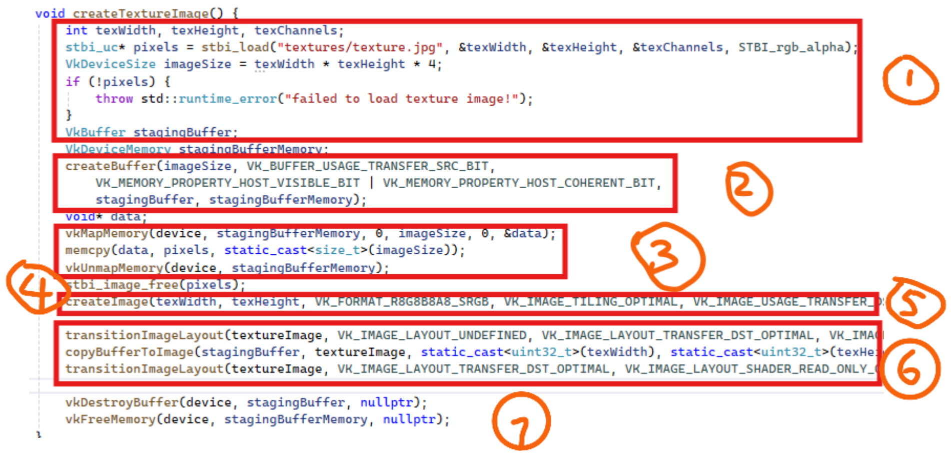
EXERCISE 1: PREPARING THE APPLICATION FOR TEXTURES

1. **Get stb_image.h:** Download the stb_image.h file from the official GitHub repository (github.com/nothings/stb) and add it to your project's include directory.
2. **Implement stb_image.h:** In *one* of your .cpp files, before including it, you must define STB_IMAGE_IMPLEMENTATION:
```
#define STB_IMAGE_IMPLEMENTATION
#include <stb_image.h>
```
3. **Get a Texture:** Find a simple texture file (e.g., container.jpg, wall.png, or a simple brick pattern) and place it in a location your application can read (like the build directory).
4. **Update Vertex Struct:** Add glm::vec2 texCoord; to your C++ Vertex struct.
```
struct Vertex {
  glm::vec3 pos;
  glm::vec3 color;
  glm::vec3 normal;
  glm::vec2 texCoord; // New
};
```
5. **Update Vertex Input Description:** Add a new VkVertexInputAttributeDescription for the texture coordinates. Update the location and offset values for all attributes to be correct.
6. **Update Cube Data:** This is a critical step. You must provide UV coordinates for all 36 vertices of your cube. Each face should map to the full [0, 1] range.
    - glm::vec2(0.0f, 1.0f): Bottom-left
    - glm::vec2(1.0f, 1.0f): Bottom-right

- glm::vec2(1.0f, 0.0f): Top-right
- glm::vec2(0.0f, 0.0f): Top-left

For example, one face (two triangles) would look like this:// Front face
{{-0.5f, -0.5f, 0.5f}, {1.0f, 1.0f, 1.0f}, {0.0f, 0.0f, 1.0f}, {0.0f, 1.0f}}, // Bottom-left
{{ 0.5f, -0.5f, 0.5f}, {1.0f, 1.0f, 1.0f}, {0.0f, 0.0f, 1.0f}, {1.0f, 1.0f}}, // Bottom-right
... ...
You must do this for all 6 faces.

EXERCISE 2: LOADING AND CREATING VULKAN IMAGE RESOURCES



```
void createTextureImage() {
    int texWidth, texHeight, texChannels;
    stbi_uc* pixels = stbi_load("textures/texture.jpg", &texWidth, &texHeight, &texChannels, STBI_rgb_alpha);
    VkDeviceSize imageSize = texWidth * texHeight * 4;
    if (!pixels) {
        throw std::runtime_error("failed to load texture image!");
    }
    VkBuffer stagingBuffer;
    VkDeviceMemory stagingBufferMemory;
    createBuffer(imageSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
        VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
        stagingBuffer, stagingBufferMemory);
    void* data;
    vkMapMemory(device, stagingBufferMemory, 0, imageSize, 0, &data);
    memcpy(data, pixels, static_cast<size_t>(imageSize));
    vkUnmapMemory(device, stagingBufferMemory);
    stbi_image_free(pixels);
    createImage(texWidth, texHeight, VK_FORMAT_R8G8B8A8_SRGB, VK_IMAGE_TILING_OPTIMAL, VK_IMAGE_USAGE_TRANSFER_

    transitionImageLayout(textureImage, VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, VK_IMA
    copyBufferToImage(stagingBuffer, textureImage, static_cast<uint32_t>(texWidth), static_cast<uint32_t>(texHe
    transitionImageLayout(textureImage, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, VK_IMAGE_LAYOUT_SHADER_READ_ONLY_

    vkDestroyBuffer(device, stagingBuffer, nullptr);
    vkFreeMemory(device, stagingBufferMemory, nullptr);
}
```

1. **Declare relevant Vulkan Objects to be created:**

```
VkImage textureImage;
VkDeviceMemory textureImageMemory;
VkImageView textureImageView;
VkSampler textureSampler;
```

2. **Load Image with stb_image:**

1

```
int texWidth, texHeight, texChannels;
stbi_uc* pixels = stbi_load("textures/texture.jpg", &texWidth, &texHeight, &texChannels, STBI_rgb_alpha);
VkDeviceSize imageSize = texWidth * texHeight * 4;
if (!pixels) {
    throw std::runtime_error("failed to load texture image!");
}
```

3. **Create Staging Buffer:** Create a VkBuffer (stagingBuffer) and VkDeviceMemory (stagingBufferMemory).
   - size: imageSize
   - usage: VK_BUFFER_USAGE_**TRANSFER_SRC**_BIT
   - properties: VK_MEMORY_PROPERTY_**HOST_VISIBLE**_BIT | VK_MEMORY_PROPERTY_**HOST_COHERENT**_BIT

2

```
createBuffer(VkDeviceSize size, VkBufferUsageFlags usage, VkMemoryPropertyFlags properties,
             VkBuffer& buffer, VkDeviceMemory& bufferMemory) {
```

```
    VkBufferCreateInfo bufferInfo{};
    bufferInfo.sType = VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
    bufferInfo.size = size;
    bufferInfo.usage = usage;
    bufferInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

    if (vkCreateBuffer(device, &bufferInfo, nullptr, &buffer) != VK_SUCCESS) {
        throw std::runtime_error("failed to create buffer!");
    }

    VkMemoryRequirements memRequirements;
    vkGetBufferMemoryRequirements(device, buffer, &memRequirements);

    VkMemoryAllocateInfo allocInfo{};
    allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
    allocInfo.allocationSize = memRequirements.size;
    allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, properties);

    if (vkAllocateMemory(device, &allocInfo, nullptr, &bufferMemory) != VK_SUCCESS) {
        throw std::runtime_error("failed to allocate buffer memory!");
    }

    vkBindBufferMemory(device, buffer, bufferMemory, 0);
}
```

4. **Copy Data to Staging Buffer:** Map the buffer, memcpy the pixels data into it, and unmap.

3

```
void* data;
vkMapMemory(device, stagingBufferMemory, 0, imageSize, 0, &data);
memcpy(data, pixels, static_cast<size_t>(imageSize));
vkUnmapMemory(device, stagingBufferMemory);
```

4

5. **Free CPU Pixels:** Call stbi_image_free(pixels); now that the data is on the staging buffer.

6. **Create VkImage:** Create your final VkImage (textureImage).
    - imageType: VK_IMAGE_TYPE_2D
    - extent: { (uint32_t)texWidth, (uint32_t)texHeight, 1 }
    - mipLevels: 1
    - arrayLayers: 1
    - format: VK_FORMAT_R8G8B8A8_SRGB (Use SRGB for colour textures)
    - tiling: VK_IMAGE_TILING_OPTIMAL
    - initialLayout: VK_IMAGE_LAYOUT_UNDEFINED
    - usage: VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT
    - sharingMode: VK_SHARING_MODE_EXCLUSIVE
    - samples: VK_SAMPLE_COUNT_1_BIT

5

```
void HelloTriangleApplication::createImage(uint32_t width, uint32_t height, VkFormat format, VkImageTiling tiling, VkImageUsageFlags
usage, VkMemoryPropertyFlags properties, VkImage& image, VkDeviceMemory& imageMemory) {
    VkImageCreateInfo imageInfo{};
    imageInfo.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
    imageInfo.imageType = VK_IMAGE_TYPE_2D;
    imageInfo.extent.width = width;
    imageInfo.extent.height = height;
    imageInfo.extent.depth = 1;
    imageInfo.mipLevels = 1;
    imageInfo.arrayLayers = 1;
    imageInfo.format = format;
    imageInfo.tiling = tiling;
    imageInfo.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
    imageInfo.usage = usage;
```

```
        imageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
        imageInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;
        if (vkCreateImage(device, &imageInfo, nullptr, &image) != VK_SUCCESS) {
            throw std::runtime_error("failed to create image!");
        }
        VkMemoryRequirements memRequirements;
        vkGetImageMemoryRequirements(device, image, &memRequirements);
        VkMemoryAllocateInfo allocInfo{};
        allocInfo.sType = VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
        allocInfo.allocationSize = memRequirements.size;
        allocInfo.memoryTypeIndex = findMemoryType(memRequirements.memoryTypeBits, properties);
        if (vkAllocateMemory(device, &allocInfo, nullptr, &imageMemory) != VK_SUCCESS) {
            throw std::runtime_error("failed to allocate image memory!");
        }
        vkBindImageMemory(device, image, imageMemory, 0);
    }
```

7. **Allocate VkImage Memory:** Allocate VkDeviceMemory (textureImageMemory) for this VkImage with VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT and bind it.

```
createImage( texWidth, texHeight, VK_FORMAT_R8G8B8A8_SRGB,
             VK_IMAGE_TILING_OPTIMAL, VK_IMAGE_USAGE_TRANSFER_DST_BIT | VK_IMAGE_USAGE_SAMPLED_BIT,
             VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT, textureImage, textureImageMemory
            );
```

8. **Create VkImageView:** Create a VkImageView (textureImageView) for the textureImage.
   - image: textureImage
   - viewType: VK_IMAGE_VIEW_TYPE_2D
   - format: VK_FORMAT_R8G8B8A8_SRGB
   - subresourceRange.aspectMask: VK_IMAGE_ASPECT_COLOR_BIT

```
void createTextureImageView() {
    textureImageView = createImageView(textureImage, VK_FORMAT_R8G8B8A8_SRGB, VK_IMAGE_ASPECT_COLOR_BIT);
}

VkImageView HelloTriangleApplication::createImageView(VkImage image, VkFormat format, VkImageAspectFlags aspectFlags)
{
    VkImageViewCreateInfo viewInfo{};
    viewInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
    viewInfo.image = image;
    viewInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
    viewInfo.format = format;
    viewInfo.subresourceRange.aspectMask = aspectFlags;
    viewInfo.subresourceRange.baseMipLevel = 0;
    viewInfo.subresourceRange.levelCount = 1;
    viewInfo.subresourceRange.baseArrayLayer = 0;
    viewInfo.subresourceRange.layerCount = 1;
    VkImageView imageView;
    if (vkCreateImageView(device, &viewInfo, nullptr, &imageView) != VK_SUCCESS) {
        throw std::runtime_error("failed to create image view!");
    }
    return imageView;
}
```

9. **Execute Transfer Commands:** You need to record and submit a one-time command buffer to perform the copy.

   transitionImageLayout(textureImage, ..., VK_IMAGE_LAYOUT_UNDEFINED, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL);

6

```
void transitionImageLayout(VkImage image, VkImageLayout oldLayout, VkImageLayout newLayout, VkImageAspectFlags aspectMask) {
        VkCommandBuffer commandBuffer = beginSingleTimeCommands();

        VkImageMemoryBarrier barrier{};
        barrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
        barrier.oldLayout = oldLayout;
        barrier.newLayout = newLayout;
        barrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
        barrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
        barrier.image = image;
        barrier.subresourceRange.aspectMask = aspectMask;
        barrier.subresourceRange.baseMipLevel = 0;
        barrier.subresourceRange.levelCount = 1;
        barrier.subresourceRange.baseArrayLayer = 0;
        barrier.subresourceRange.layerCount = 1;

        VkPipelineStageFlags sourceStage;
        VkPipelineStageFlags destinationStage;

        if (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED && newLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL) {
            barrier.srcAccessMask = 0;
            barrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
            sourceStage = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
            destinationStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
        }
        else if (oldLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL && newLayout == VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL) {
            barrier.srcAccessMask = VK_ACCESS_TRANSFER_WRITE_BIT;
            barrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
            sourceStage = VK_PIPELINE_STAGE_TRANSFER_BIT;
            destinationStage = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
        }
        else if (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED && newLayout == VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL) {
            barrier.srcAccessMask = 0;
            barrier.dstAccessMask = VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_READ_BIT | VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
            sourceStage = VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT;
            destinationStage = VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT;
        }
        else {
            throw std::invalid_argument("unsupported layout transition!");
        }

        vkCmdPipelineBarrier(commandBuffer, sourceStage, destinationStage, 0, 0, nullptr, 0, nullptr, 1, &barrier);

        endSingleTimeCommands(commandBuffer);
    }
```

- copyBufferToImage(stagingBuffer, textureImage, static_cast<uint32_t>(texWidth), static_cast<uint32_t>(texHeight));

```
void HelloTriangleApplication::copyBufferToImage(VkBuffer buffer, VkImage image, uint32_t width, uint32_t height) {
    VkCommandBuffer commandBuffer = beginSingleTimeCommands();
    VkBufferImageCopy region{};
    region.bufferOffset = 0;
    region.bufferRowLength = 0;
    region.bufferImageHeight = 0;
    region.imageSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    region.imageSubresource.mipLevel = 0;
    region.imageSubresource.baseArrayLayer = 0;
    region.imageSubresource.layerCount = 1;
    region.imageOffset = { 0, 0, 0 };
    region.imageExtent = { width, height, 1 };
    vkCmdCopyBufferToImage(commandBuffer, buffer, image, VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, &region);
    endSingleTimeCommands(commandBuffer);
}

VkCommandBuffer HelloTriangleApplication::beginSingleTimeCommands() {
    VkCommandBufferAllocateInfo allocInfo{};
    allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
    allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
    allocInfo.commandPool = commandPool;
    allocInfo.commandBufferCount = 1;
    VkCommandBuffer commandBuffer;
    vkAllocateCommandBuffers(device, &allocInfo, &commandBuffer);
    VkCommandBufferBeginInfo beginInfo{};
    beginInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
    beginInfo.flags = VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;
    vkBeginCommandBuffer(commandBuffer, &beginInfo);
    return commandBuffer;
}



void HelloTriangleApplication::endSingleTimeCommands(VkCommandBuffer commandBuffer) {
    vkEndCommandBuffer(commandBuffer);
    VkSubmitInfo submitInfo{};
    submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
    submitInfo.commandBufferCount = 1;
    submitInfo.pCommandBuffers = &commandBuffer;
    vkQueueSubmit(graphicsQueue, 1, &submitInfo, VK_NULL_HANDLE);
    vkQueueWaitIdle(graphicsQueue);
    vkFreeCommandBuffers(device, commandPool, 1, &commandBuffer);
}
```

- transitionImageLayout(textureImage, ..., VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL);
- (These transition and copy functions will record barrier and copy commands into a command buffer, which you must then submit and wait on).

10. **Clean Up Staging Buffer:** After the transfer is complete, destroy stagingBuffer and free stagingBufferMemory.

7

```
vkDestroyBuffer(device, stagingBuffer, nullptr);
vkFreeMemory(device, stagingBufferMemory, nullptr);
```

11. **Create VkSampler:** Create a VkSampler (textureSampler) to tell the shader how to read the image.

    ○ magFilter: VK_FILTER_LINEAR

    ○ minFilter: VK_FILTER_LINEAR

    ○ addressModeU/V/W: VK_SAMPLER_ADDRESS_MODE_REPEAT

    ○ anisotropyEnable: VK_TRUE (or VK_FALSE if not supported)

    ○ maxAnisotropy: 16.0f (or 1.0f if disabled)

    ○ borderColor: VK_BORDER_COLOR_INT_OPAQUE_BLACK

    ○ unnormalizedCoordinates: VK_FALSE

```
void createTextureSampler() {
    VkPhysicalDeviceProperties properties{};
    vkGetPhysicalDeviceProperties(physicalDevice, &properties);
    VkSamplerCreateInfo samplerInfo{};
    samplerInfo.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
    samplerInfo.magFilter = VK_FILTER_LINEAR;
    samplerInfo.minFilter = VK_FILTER_LINEAR;
    samplerInfo.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    samplerInfo.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    samplerInfo.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
    samplerInfo.anisotropyEnable = VK_TRUE;
    samplerInfo.maxAnisotropy = properties.limits.maxSamplerAnisotropy;
    samplerInfo.borderColor = VK_BORDER_COLOR_INT_OPAQUE_BLACK;
    samplerInfo.unnormalizedCoordinates = VK_FALSE;
    samplerInfo.compareEnable = VK_FALSE;
    samplerInfo.compareOp = VK_COMPARE_OP_ALWAYS;
    samplerInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
    if (vkCreateSampler(device, &samplerInfo, nullptr, &textureSampler) != VK_SUCCESS) {
        throw std::runtime_error("failed to create texture sampler!");
    }
}
```

EXERCISE 4: BINDING AND SHADER UPDATES

1. **Update Descriptor Set Layout:** Add a new binding to your VkDescriptorSetLayoutCreateInfo.
   VkDescriptorSetLayoutBinding samplerLayoutBinding{};
   samplerLayoutBinding.binding = 1; // 0 is already used by the UBO
   samplerLayoutBinding.descriptorType =
   VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
   samplerLayoutBinding.descriptorCount = 1;
   samplerLayoutBinding.stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;

   Add this to your array of bindings when creating the layout.

2. **Update Descriptor Pool:** Make sure your VkDescriptorPool is created with a VkDescriptorPoolSize for VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER.

```
void createDescriptorPool() {
    std::array<VkDescriptorPoolSize, 2> poolSizes{};
    poolSizes[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    poolSizes[0].descriptorCount = static_cast<uint32_t>(MAX_FRAMES_IN_FLIGHT);
    poolSizes[1].type = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
    poolSizes[1].descriptorCount = static_cast<uint32_t>(MAX_FRAMES_IN_FLIGHT);
    VkDescriptorPoolCreateInfo poolInfo{};
    poolInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
    poolInfo.poolSizeCount = static_cast<uint32_t>(poolSizes.size());
    poolInfo.pPoolSizes = poolSizes.data();
    poolInfo.maxSets = static_cast<uint32_t>(MAX_FRAMES_IN_FLIGHT);
    if (vkCreateDescriptorPool(device, &poolInfo, nullptr, &descriptorPool) != VK_SUCCESS) {
        throw std::runtime_error("failed to create descriptor pool!");
    }
}
```

3. **Update Descriptor Set:** When you create your VkDescriptorSet, you must now write two descriptors to it.

   - One VkDescriptorBufferInfo for the UBO at binding = 0.
   - One VkDescriptorImageInfo for the texture at binding = 1.

```
VkDescriptorImageInfo imageInfo{};
imageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
imageInfo.imageView = textureImageView;
imageInfo.sampler = textureSampler;

//2 descriptors: uniform buffer + texture sampler

std::array<VkWriteDescriptorSet, 2> descriptorWrites{};

 //(1) Uniform Buffer
descriptorWrites[0].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[0].dstSet = descriptorSets[i];
descriptorWrites[0].dstBinding = 0;
descriptorWrites[0].dstArrayElement = 0;
descriptorWrites[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
descriptorWrites[0].descriptorCount = 1;
descriptorWrites[0].pBufferInfo = &bufferInfo;

  //(2) Texture Sampler
descriptorWrites[1].sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
descriptorWrites[1].dstSet = descriptorSets[i];
descriptorWrites[1].dstBinding = 1;
descriptorWrites[1].dstArrayElement = 0;
descriptorWrites[1].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
descriptorWrites[1].descriptorCount = 1;
descriptorWrites[1].pImageInfo = &imageInfo;

vkUpdateDescriptorSets(device, static_cast<uint32_t>(descriptorWrites.size()),
                       descriptorWrites.data(), 0, nullptr);
```

4. Update Descriptor set layout:

```
void HelloTriangleApplication::createDescriptorSetLayout() {
    VkDescriptorSetLayoutBinding uboLayoutBinding{};
    uboLayoutBinding.binding = 0;
    uboLayoutBinding.descriptorCount = 1;
    uboLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    uboLayoutBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;

    // This is the new piece:
    VkDescriptorSetLayoutBinding samplerLayoutBinding{};
    samplerLayoutBinding.binding = 1;
    samplerLayoutBinding.descriptorCount = 1;
    samplerLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
    samplerLayoutBinding.stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
```

```
        std::array<VkDescriptorSetLayoutBinding, 2> bindings = { uboLayoutBinding, samplerLayoutBinding };
        VkDescriptorSetLayoutCreateInfo layoutInfo{};
        layoutInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
        layoutInfo.bindingCount = static_cast<uint32_t>(bindings.size());
        layoutInfo.pBindings = bindings.data();

        if (vkCreateDescriptorSetLayout(device, &layoutInfo, nullptr, &descriptorSetLayout) != VK_SUCCESS) {
            throw std::runtime_error("Failed to create descriptor set layout!");
        }
    }
```

5. Add necessary cleanup for your new texture resources in the cleanup function to prevent memory leaks.

6. Update initVulkan()

```
… …
 createTextureImage();
createTextureImageView();
createTextureSampler();
```

7. **Vertex Shader (shader.vert):**
   - Add the new input: layout(location = 3) in vec2 inTexCoord; (Adjust location as needed)
   - Add a new output: layout(location = 3) out vec2 fragTexCoord;
   - In main(), pass it through: fragTexCoord = inTexCoord;

8. **Fragment Shader (shader.frag):**
   - Add the new uniform sampler: layout(binding = 1) uniform sampler2D texSampler;
   - Add the new input: layout(location = 3) in vec2 fragTexCoord;
   - In main(), sample the texture and combine it with your lighting.

     ```
     …. …
     vec4 texColor = texture(texSampler, fragTexCoord);

     // … (Your lighting calculations: ambient, diffuse, specular) …

     // Modulate the material's colour (from the texture) with the light
     vec3 result = (ambient + diffuse) * texColor.rgb + specular;
     outColor = vec4(result, 1.0);
     ```

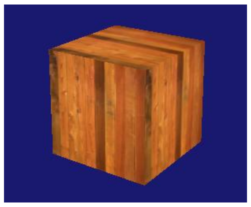
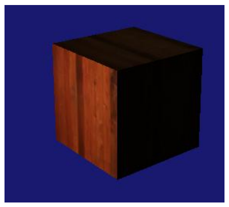
9. **Run:** Compile and run. Your cube should now be textured!

EXERCISE 5: A WOODEN CUBE

Convert the provided wood.dds texture and convert it to jpg format using an online texture converter, for example, https://convert.guru/converter. the existing texture file texture.jpg with the newly converted wood.jpg.

In your implementation of per-fragment lighting from Lab 4, substitute the vertex colour input with the wood.jpg texture. This change will produce the intended visual effect as illustrated below.

:

Exercise 6. TEXTURE WRAPPING MODE

Create a texture-mapped cube using the coin texture "Coin.jpg", such that different faces of the cube have different number of coin patterns.

Exercise 7. TEXTURE FILTERING TECHNIQUES.

Scale the cube geometry along the view direction to create the visual impression of a long, straight road extending into the distance. This transformation aligns the object with the camera's perspective, enhancing depth perception and mimicking the appearance of linear structures such as highways or corridors.

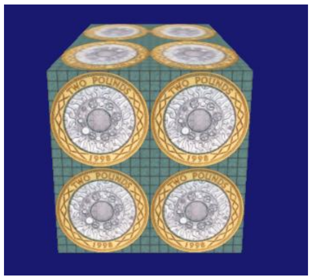
Next, apply various texture filtering techniques to address common issues in texture mapping—specifically, **minification** and **magnification**. Techniques such as nearest-neighbour, bilinear, bi-cubic, and anisotropic filtering can be used to mitigate aliasing and blurring artifacts.

As you implement and compare these filtering methods, observe how each affects the visual fidelity of the rendered image.

This exercise highlights the importance of texture sampling strategies in real-time rendering and their impact on perceived image quality, especially under non-uniform geometric transformations.

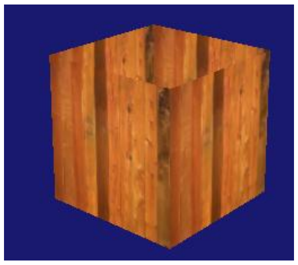
EXERCISE 8. MULTIPLE TEXTURING

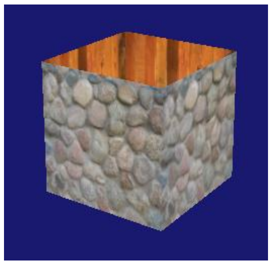
Convert the provided tile.dds texture to jpg format and use it together with the coin texture to create the following effect.



EXERCISE 9. AN OPEN BOX

Create the following open box effects using the wood.jpg and rock.jpg textures.





5 FURTHER EXPLORATION

- **Address Modes:** Change addressModeU and addressModeV to VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE. Scale your UV coordinates in the vertex shader (e.g., fragTexCoord = inTexCoord * 2.0;) and observe how the edges of the texture are smeared instead of repeating.
- **Procedural Textures:** Generate procedural textures directly in code by defining a function that computes the colour of each fragment based on mathematical expressions or patterns. You can find inspiration from the examples on Shadertoy.com, which demonstrate how to simulate textures without relying on external image files.