Vulkan Lab 5: Texture Mapping

EXERCISE 1: PREPARING THE APPLICATION FOR TEXTURES

Solution:

The external image-loading header file stb_image.h was downloaded from the official GitHub repository and placed into a Dependencies folder within the project for better organization. The macro STB_IMAGE_IMPLEMENTATION was defined before including <stb_image.h> to enable loading of PNG and JPG files during runtime. A wall texture image was placed into an Assets directory next to the application executable so that the file can be accessed with a relative path when rendering. The Vertex structure was extended to include a glm::vec2 texCoord attribute, providing texture coordinates for each vertex. The Vulkan vertex input description was also updated to include an attribute for this new UV data using the correct memory formatting and offset. Finally, the cube vertex data was modified so that all 36 vertices now include normalized UV coordinates, ensuring proper texture mapping across all six faces of the cube. These completed changes prepare the application to apply textures correctly in later exercises.

- Include STB Image Implementation:

```
#define STB_IMAGE_IMPLEMENTATION
#include <stb_image.h>
```

- Vertex Structure Update:

```
struct Vertex {
    glm::vec3 pos;
    glm::vec3 color;
    glm::vec3 normal;        // <- - Normal Vector
    glm::vec2 texCoord;     // <- - Texture Coordinate

    static VkVertexInputBindingDescription getBindingDescription() {
        VkVertexInputBindingDescription bindingDescription{};
        bindingDescription.binding = 0;
        bindingDescription.stride = sizeof(Vertex);
        bindingDescription.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
        return bindingDescription;
    }

    static std::array<VkVertexInputAttributeDescription, 3>
getAttributeDescriptions() {
        std::array<VkVertexInputAttributeDescription, 3> attributeDescriptions{};
        attributeDescriptions[0] = { 0, 0, VK_FORMAT_R32G32B32_SFLOAT,
offsetof(Vertex, pos) };
        attributeDescriptions[1] = { 1, 0, VK_FORMAT_R32G32B32_SFLOAT,
offsetof(Vertex, color) };
        attributeDescriptions[1] = { 1, 0, VK_FORMAT_R32G32B32_SFLOAT,
offsetof(Vertex, normal) };
```

```
        attributeDescriptions[2] = { 2, 0, VK_FORMAT_R32G32_SFLOAT,
offsetof(Vertex, texCoord) };    // <- - Texture Coordinate
        return attributeDescriptions;
    }
};
```

Reflection:

This exercise introduced the foundational requirements for supporting textures within a Vulkan application. I learned that the graphics pipeline must be informed not only of vertex position and colour data, but also of UV coordinates that define how a 2D image is mapped onto 3D geometry. Updating the Vertex structure and vertex attribute descriptions demonstrated the importance of maintaining consistent memory layouts between CPU-side data and shader inputs. Adding stb_image.h reinforced the need for image-loading utilities because Vulkan itself does not provide built-in support for reading texture files. Generating full UV coordinates for each cube vertex highlighted how texture mapping relies on precise indexing to avoid distortion. Overall, this exercise improved my understanding of how both data structure design and pipeline configuration contribute to correct texture usage in Vulkan.

EXERCISE 2: LOADING AND CREATING VULKAN IMAGE RESOURCES

Solution:

In this exercise, I implemented the complete Vulkan texture-loading workflow from reading an image file on the CPU to creating a fully functional GPU texture object ready for sampling in the shader. The process involved understanding how data flows between host and device memory, how Vulkanhandles image layouts and synchronization, and how samplers define how textures are accessed.

I began by declaring four main Vulkan objects (VkImage, VkDeviceMemory, VkImageView, and VkSampler) to represent the texture resource, its memory, view, and sampling configuration. I used stb_image to load an image file (wall.jpg) and obtain pixel data in RGBA format, then created a staging buffer that temporarily stored this data in host-visible memory. The pixel data was mapped into this buffer using vkMapMemory() and later un-mapped before freeing the CPU copy with stbi_image_free().

Next, I created a GPU-resident VkImage using device-local memory. This step ensured that the texture resides in high-performance memory optimized for sampling. Before the texture could be used, it was transitioned through layout states: first from UNDEFINED to TRANSFER_DST_OPTIMAL for data transfer, and then from TRANSFER_DST_OPTIMAL to SHADER_READ_ONLY_OPTIMAL for shader access. These transitions were handled using pipeline barriers recorded and executed within one-time command buffers. The data transfer itself was performed using vkCmdCopyBufferToImage().

After transferring, I created an image view (VkImageView) to define how the image would be accessed in the shader, and a sampler (VkSampler) that controls filtering and addressing behavior. The sampler used linear filtering, repeat wrapping, and anisotropic filtering based on device limits. Finally, I cleaned up the staging buffer since the data had already been copied to GPU memory.

- Creating the Image:

```
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, height, 1 };
    imageInfo.mipLevels = 1;
    imageInfo.arrayLayers = 1;
    imageInfo.format = format;
    imageInfo.tiling = tiling;
    imageInfo.usage = usage;
    imageInfo.samples = VK_SAMPLE_COUNT_1_BIT;
    imageInfo.sharingMode = VK_SHARING_MODE_EXCLUSIVE;

    vkCreateImage(device, &imageInfo, nullptr, &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);

    vkAllocateMemory(device, &allocInfo, nullptr, &imageMemory);
    vkBindImageMemory(device, image, imageMemory, 0);
}
```

- Transitioning Image Layouts:

```
void HelloTriangleApplication::transitionImageLayout(
    VkImage image, VkImageLayout oldLayout, VkImageLayout newLayout,
VkImageAspectFlags aspectMask) {

    VkCommandBuffer cmd = 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.levelCount = 1;
    barrier.subresourceRange.layerCount = 1;
```

```
    if (oldLayout == VK_IMAGE_LAYOUT_UNDEFINED &&
        newLayout == VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL) {
        barrier.srcAccessMask = 0;
        barrier.dstAccessMask = VK_ACCESS_TRANSFER_WRITE_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;
    }

    vkCmdPipelineBarrier(cmd,
        VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
        VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT,
        0, 0, nullptr, 0, nullptr, 1, &barrier);

    endSingleTimeCommands(cmd);
}
```

- Copying Buffer to Image:

```
void HelloTriangleApplication::copyBufferToImage(
    VkBuffer buffer, VkImage image, uint32_t width, uint32_t height) {

    VkCommandBuffer cmd = beginSingleTimeCommands();

    VkBufferImageCopy region{};
    region.imageSubresource.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
    region.imageSubresource.layerCount = 1;
    region.imageExtent = { width, height, 1 };

    vkCmdCopyBufferToImage(cmd, buffer, image,
        VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL, 1, &region);

    endSingleTimeCommands(cmd);
}
```

- Creating Image View:

```
VkImageView HelloTriangleApplication::createImageView(
    VkImage image, VkFormat format, VkImageAspectFlags aspectMask) {

    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 = aspectMask;
    viewInfo.subresourceRange.levelCount = 1;
```

```
    viewInfo.subresourceRange.layerCount = 1;

    VkImageView imageView;
    vkCreateImageView(device, &viewInfo, nullptr, &imageView);
    return imageView;
}
```

- Creating Texture Sampler:

```
void HelloTriangleApplication::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;

    vkCreateSampler(device, &samplerInfo, nullptr, &textureSampler);
}
```

Reflection:

This exercise deepened my understanding of how Vulkan manages textures at a low level. I learned that loading an image involves several explicit steps such as creating a staging buffer, transferring data to a GPU-local image, performing layout transitions, and setting up an image view and sampler.

The process highlighted Vulkan's emphasis on explicit control. Each stage, from CPU data upload to GPU sampling, had to be manually defined, including synchronization through layout transitions (UNDEFINED ? TRANSFER_DST ? SHADER_READ_ONLY). Implementing these barriers taught me how crucial proper pipeline ordering is for correctness and performance.

Creating the image view and sampler clarified how Vulkan separates texture storage from how it is accessed. The view defines which parts of the image are visible, while the sampler defines how textures are filtered and wrapped.

Overall, I learned how textures flow from disk to GPU memory and how Vulkan requires precise management of memory and synchronization. This exercise built a solid foundation for understanding image-based resources and will be essential for later topics such as mipmapping, depth buffers, and material systems.

EXERCISE 4: BINDING AND SHADER UPDATES

Solution:

With the texture image, view, and sampler successfully created, I updated the descriptor set layout to include a second binding for the combined image sampler. This ensured that the fragment shader could access the texture along with the uniform buffer data. The descriptor pool was expanded to support both uniform buffer and sampler descriptors, and each descriptor set was written with two bindings, one for the uniform buffer (binding 0) and one for the texture sampler (binding 1). The shaders were then updated to include texture coordinates and a `sampler2D` uniform bound to binding 1. The fragment shader sampled the texture using the interpolated UV coordinates and output the resulting texture color directly to the framebuffer. The vertex structure and pipeline configuration were modified to pass texture coordinates from the vertex shader to the fragment shader. During initialization, the texture setup functions `createTextureImage()`, `createTextureImageView()`, and `createTextureSampler()` were called before creating vertex and index buffers so that the texture resources were ready when command buffers were recorded. In the cleanup routine, I ensured that all texture-related Vulkan objects, including the sampler, image view, image, and its device memory, were properly destroyed to prevent resource leaks. This completed the full Vulkan texture-loading workflow, allowing the cube to render with the applied texture. Through this exercise, I gained a clear understanding of how Vulkan handles image creation, memory transitions, descriptor bindings, and sampler configuration to enable textured rendering.

- Descriptor Set Layout with Texture Sampler Binding:

```
VkDescriptorSetLayoutBinding ubo{};
ubo.binding = 0;
ubo.descriptorType  = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
ubo.descriptorCount = 1;
ubo.stageFlags      = VK_SHADER_STAGE_VERTEX_BIT;

VkDescriptorSetLayoutBinding sampler{};
sampler.binding = 1;
sampler.descriptorType  = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
sampler.descriptorCount = 1;
sampler.stageFlags      = VK_SHADER_STAGE_FRAGMENT_BIT;

std::array<VkDescriptorSetLayoutBinding, 2> bindings{ ubo, sampler };

VkDescriptorSetLayoutCreateInfo layoutInfo{
VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO };
layoutInfo.bindingCount = static_cast<uint32_t>(bindings.size());
layoutInfo.pBindings    = bindings.data();

vkCreateDescriptorSetLayout(device, &layoutInfo, nullptr, &descriptorSetLayout);
```

- Descriptor Pool Creation with Sampler Support:

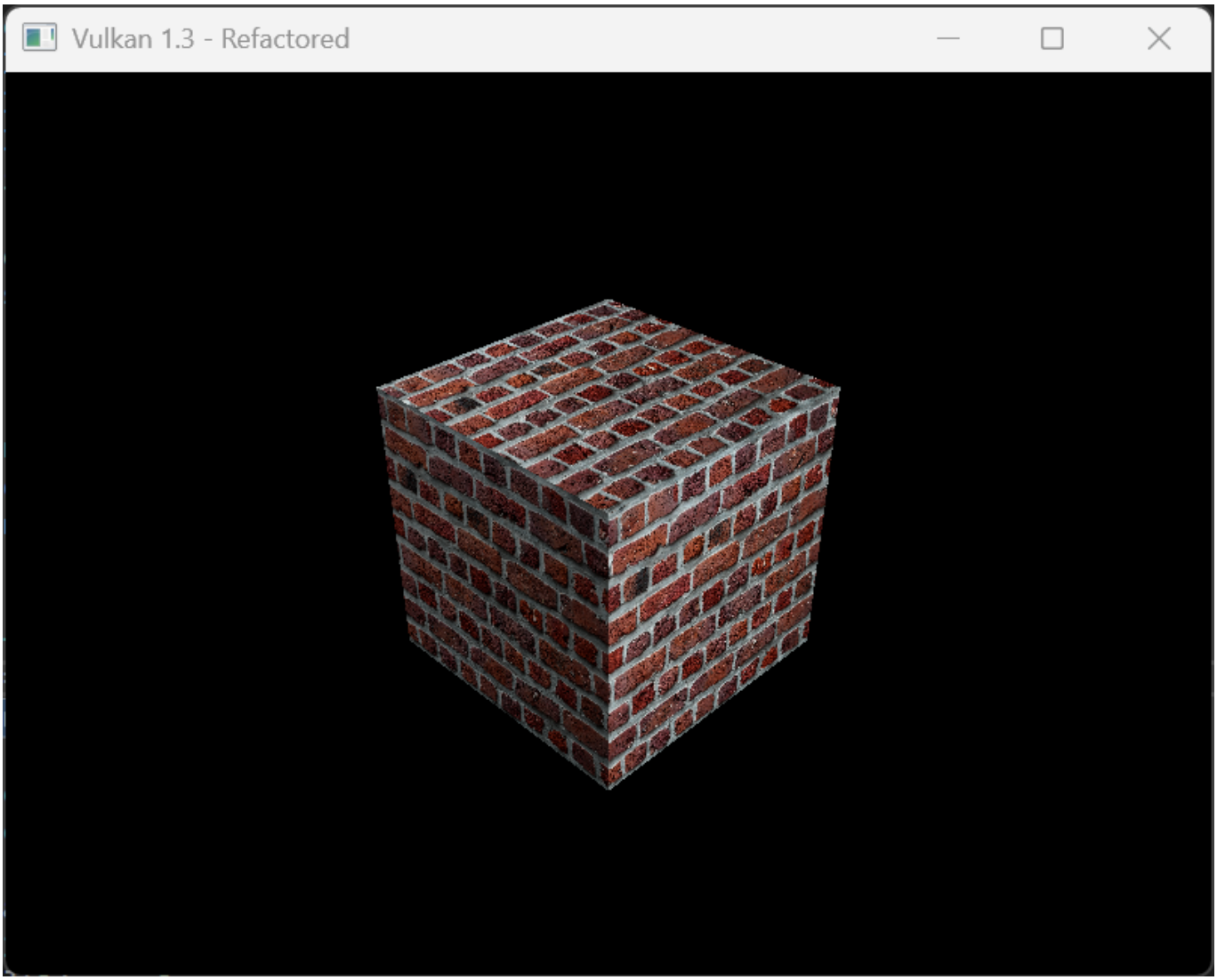
```
std::array<VkDescriptorPoolSize, 2> poolSizes{};
poolSizes[0] = { VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER,         MAX_FRAMES_IN_FLIGHT
};
```

```
poolSizes[1] = { VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, MAX_FRAMES_IN_FLIGHT
}; // new

VkDescriptorPoolCreateInfo poolInfo{ VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO
};
poolInfo.poolSizeCount = static_cast<uint32_t>(poolSizes.size());
poolInfo.pPoolSizes    = poolSizes.data();
poolInfo.maxSets       = MAX_FRAMES_IN_FLIGHT;

vkCreateDescriptorPool(device, &poolInfo, nullptr, &descriptorPool);
```

- Writing Descriptor Sets with Texture Sampler:

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

VkWriteDescriptorSet samplerWrite{ VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET };
samplerWrite.dstSet          = descriptorSets[i];
samplerWrite.dstBinding      = 1; // matches layout(binding=1)
samplerWrite.descriptorType  = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
samplerWrite.descriptorCount = 1;
samplerWrite.pImageInfo      = &imageInfo;

vkUpdateDescriptorSets(device, 1, &samplerWrite, 0, nullptr);
```

Output:



Reflection:

This exercise helped me understand how to actually get textures working in Vulkan instead of just loading them. I learned that creating the image isn't enough; you have to connect it to the shaders through descriptor layouts, descriptor sets, and samplers. Setting up the combined image sampler in the descriptor layout made it clear how Vulkan links texture data to the fragment shader. I also realized how hands-on Vulkan is, since you have to manage every part of the process yourself, from adding the sampler binding to updating descriptor sets and binding them during draw calls. Writing the vertex and fragment shaders tied it all together, because I could see how the texture coordinates flowed through the pipeline to sample the image. Overall, it was satisfying to finally see how everything fits together and how the CPU-side texture setup, GPU memory, and shaders all work in sync to display a textured cube.

EXERCISE 5: A WOODEN CUBE

Solution:

For this exercise, I integrated both per-fragment lighting and a moving light sphere (gizmo) to visualize the light source. The original vertex colour input from the cube geometry was replaced with a texture sampled from the converted wood.jpg image (originally wood.dds). This was achieved by loading the texture through stb_image.h, creating a Vulkan image and sampler, and binding it via the descriptor set at binding = 1. The fragment shader performs per-fragment Phong lighting, combining ambient, diffuse, and specular terms, and

uses the texture colour (texture(texSampler, vUV).rgb) instead of vertex colour to achieve the wood material effect. A push constant controls the unlit white light sphere, drawn at the current light position, while the cube uses the uniform buffer's light data for dynamic illumination.

- Loading Texture and Binding to Descriptor Set:

```
// Load wood.jpg texture using stb_image
int w, h, ch;
stbi_uc* pixels = stbi_load("wood.jpg", &w, &h, &ch, STBI_rgb_alpha);
VkDeviceSize imageSize = (VkDeviceSize)w * h * 4;

// Create a staging buffer and upload texture data
createBuffer(imageSize, VK_BUFFER_USAGE_TRANSFER_SRC_BIT,
    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT,
    stagingBuffer, stagingMemory);
void* data;
vkMapMemory(device, stagingMemory, 0, imageSize, 0, &data);
memcpy(data, pixels, (size_t)imageSize);
vkUnmapMemory(device, stagingMemory);
stbi_image_free(pixels);

// Bind texture to descriptor set at binding = 1
VkDescriptorImageInfo imageInfo{};
imageInfo.imageLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
imageInfo.imageView = textureImageView;
imageInfo.sampler = textureSampler;
write[1].descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER;
write[1].pImageInfo = &imageInfo;
```

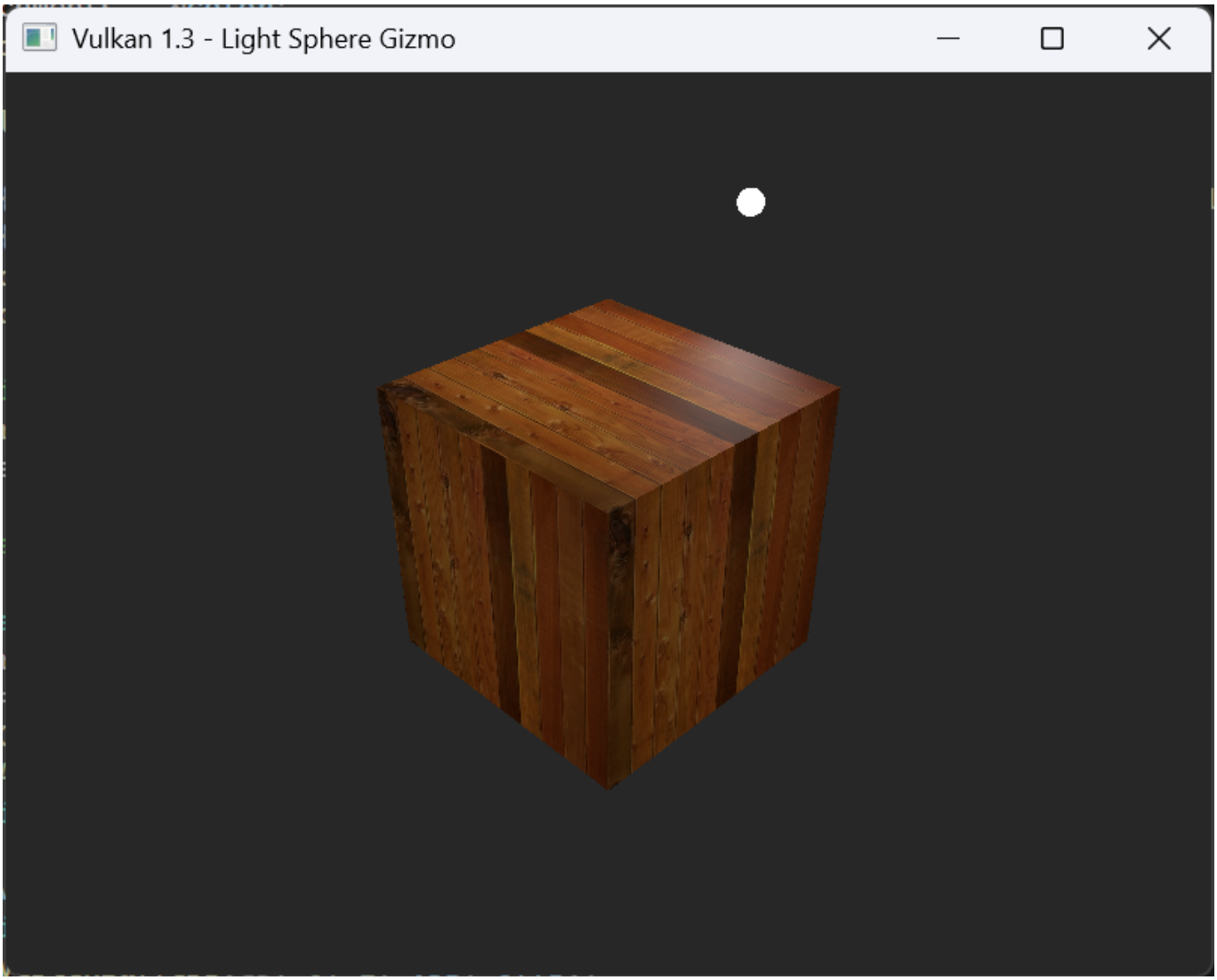
- Fragment Shader with Texture Sampling and Phong Lighting:

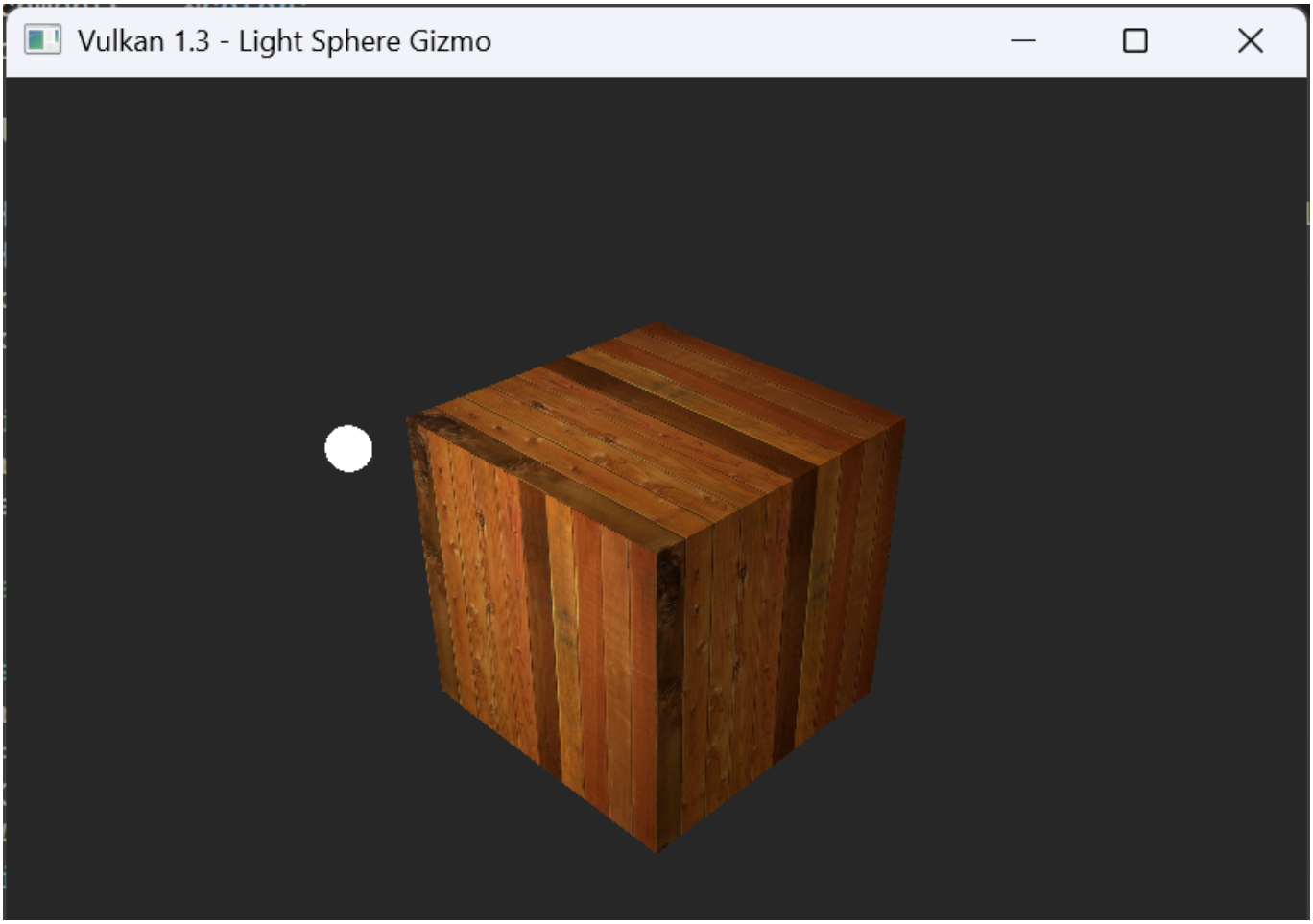
```
vec3 N = normalize(vWorldNormal);
vec3 L = normalize(ubo.lightPos - vWorldPos);
vec3 V = normalize(ubo.eyePos - vWorldPos);
vec3 R = reflect(-L, N);

vec3 tex = texture(texSampler, vUV).rgb;

float diff = max(dot(N, L), 0.0);
float spec = pow(max(dot(R, V), 0.0), 32.0);

vec3 ambient  = 0.15 * tex;
vec3 diffuse  = 0.85 * diff * tex;
vec3 specular = 0.15 * spec * vec3(1.0);

outColor = vec4(ambient + diffuse + specular, 1.0);
```

- Dynamic Light Position Update:

```
// Update light position each frame
float R = 1.0f, omega = 1.5f;
ubo.lightPos = glm::vec3(R * cos(omega * t), 0.5f, R * sin(omega * t));
```

Output:


Vulkan 1.3 - Light Sphere Gizmo


Vulkan 1.3 - Light Sphere Gizmo

Reflection:

Through this task, I deepened my understanding of how Vulkan's per-fragment lighting pipeline integrates texture sampling, uniform buffers, and push constants. I learned how texture mapping replaces per-vertex colours in the fragment stage and how descriptor sets allow textures to be bound and sampled efficiently. Implementing the moving light sphere reinforced how push constants can drive unlit objects without additional pipelines. I also noticed the risk of inconsistencies when computing the light position separately for the UBO and for drawing the sphere. Next time, I will compute the light position once per frame and cache it, reusing the same value for both the uniform buffer update and the sphere transform to avoid divergence and improve clarity and performance.

Exercise 6. TEXTURE WRAPPING MODE

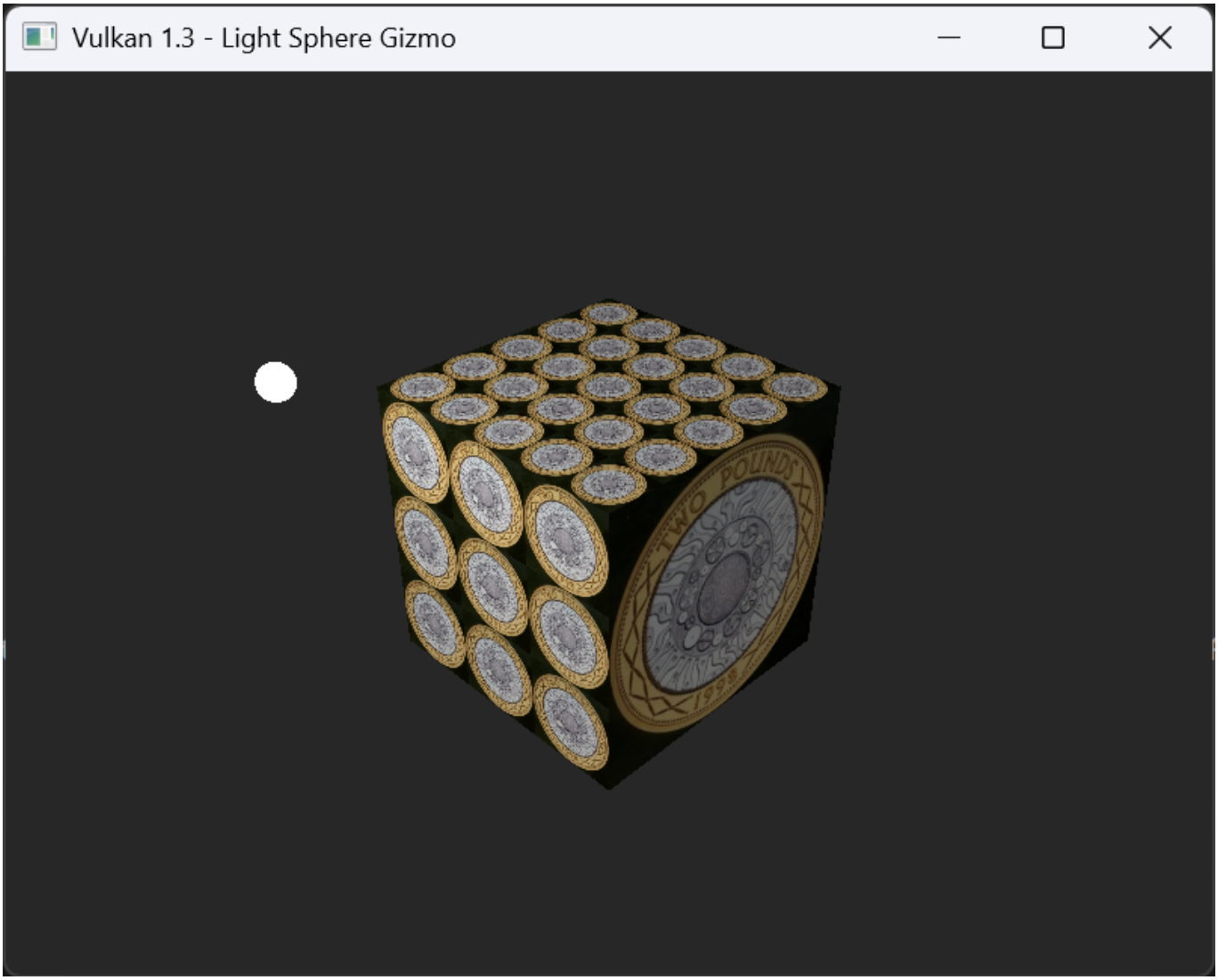
Solution:

I created a texture-mapped cube using the Coin.jpg texture, with each face displaying a different number of coin repetitions. This was achieved by modifying the per-vertex UV coordinates rather than altering the texture or shader. I assigned UV ranges that extended beyond the standard [0,1] range, causing the texture to wrap and repeat according to the sampler's VK_SAMPLER_ADDRESS_MODE_REPEAT setting. The sampler was configured to repeat in both the U and V directions, ensuring seamless tiling across each face. The result was a cube where each side showed progressively more tiled coin patterns—from one on the front face to six on the bottom—demonstrating effective use of texture wrapping and coordinate scaling.

```
auto faceUV = [&](float S, glm::vec2 uv) { return uv * S; };

std::vector<Vertex> cubeVertices = {
    // Front (+Z), 1×1 coins
    {{-0.5f,-0.5f,  0.5f}, {1,0,0}, {0,0, 1}, faceUV(1.0f,{0,1})},
    {{ 0.5f,-0.5f,  0.5f}, {1,0,0}, {0,0, 1}, faceUV(1.0f,{1,1})},
    {{ 0.5f, 0.5f,  0.5f}, {1,0,0}, {0,0, 1}, faceUV(1.0f,{1,0})},
    {{ 0.5f, 0.5f,  0.5f}, {1,0,0}, {0,0, 1}, faceUV(1.0f,{1,0})},
    {{-0.5f, 0.5f,  0.5f}, {1,0,0}, {0,0, 1}, faceUV(1.0f,{0,0})},
    {{-0.5f,-0.5f,  0.5f}, {1,0,0}, {0,0, 1}, faceUV(1.0f,{0,1})},

    // Back (−Z), 2×2 coins
    {{ 0.5f,-0.5f, -0.5f}, {0,1,0}, {0,0,-1}, faceUV(2.0f,{0,1})},
    {{-0.5f,-0.5f, -0.5f}, {0,1,0}, {0,0,-1}, faceUV(2.0f,{1,1})},
    {{-0.5f, 0.5f, -0.5f}, {0,1,0}, {0,0,-1}, faceUV(2.0f,{1,0})},
    {{-0.5f, 0.5f, -0.5f}, {0,1,0}, {0,0,-1}, faceUV(2.0f,{1,0})},
    {{ 0.5f, 0.5f, -0.5f}, {0,1,0}, {0,0,-1}, faceUV(2.0f,{0,0})},
    {{ 0.5f,-0.5f, -0.5f}, {0,1,0}, {0,0,-1}, faceUV(2.0f,{0,1})},
```

```
info.addressModeU = VK_SAMPLER_ADDRESS_MODE_REPEAT;
info.addressModeV = VK_SAMPLER_ADDRESS_MODE_REPEAT;
info.addressModeW = VK_SAMPLER_ADDRESS_MODE_REPEAT;
```

Output:



Reflection:

Through this exercise, I learned how texture coordinates directly influence how textures are mapped and repeated on 3D surfaces. Adjusting the UVs at the vertex level provided an intuitive way to control the scale and repetition of the texture without needing additional shaders or multiple draw calls. It also reinforced my understanding of how the REPEAT wrapping mode interacts with UV coordinates beyond [0,1]. In future implementations, I plan to explore dynamic approaches such as using push constants or per-face uniform values to adjust texture scaling programmatically rather than embedding the UV scaling into the vertex data. This would make the solution more flexible and reusable across different models.

Exercise 7. TEXTURE FILTERING TECHNIQUES.

Solution:

In this exercise I scaled the cube geometry along the viewing direction to create the visual effect of a long flat road extending into the distance. This transformation gave the scene depth and helped to show how different texture filtering techniques behave under strong perspective distortion. I then implemented and compared four filtering modes which were nearest neighbour, bilinear, bicubic which was approximated using trilinear filtering in Vulkan, and anisotropic filtering. Nearest neighbour sampling produced blocky textures, while bilinear filtering softened the transitions between texels but introduced a small amount of blurring. Bicubic filtering was approximated by enabling both linear texture sampling and linear interpolation between mip levels, since Vulkan does not include a true bicubic option. Anisotropic filtering provided the highest visual quality, keeping the distant sections of the road clear and reducing aliasing at oblique viewing angles.

```
VkSamplerCreateInfo info{};
```

- Nearest Neighbor Filtering:

```
info.magFilter       = VK_FILTER_NEAREST;
info.minFilter       = VK_FILTER_NEAREST;
info.mipmapMode      = VK_SAMPLER_MIPMAP_MODE_NEAREST;
info.anisotropyEnable = VK_FALSE;
```

- Bilinear Filtering:

```
info.magFilter       = VK_FILTER_LINEAR;
info.minFilter       = VK_FILTER_LINEAR;
info.mipmapMode      = VK_SAMPLER_MIPMAP_MODE_NEAREST;
info.anisotropyEnable = VK_FALSE;
```

- Bicubic Filtering:

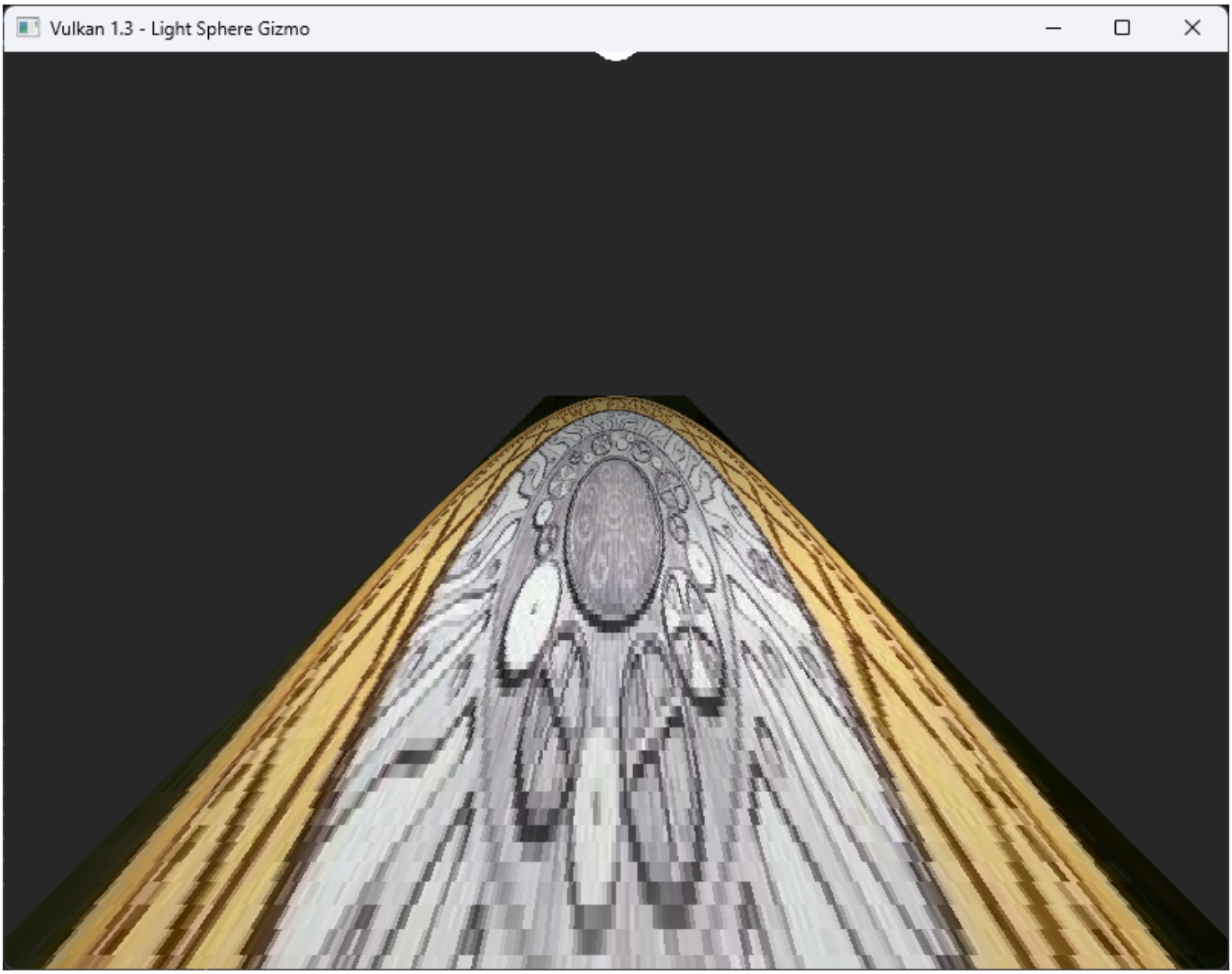
```
info.magFilter       = VK_FILTER_LINEAR;
info.minFilter       = VK_FILTER_LINEAR;
info.mipmapMode      = VK_SAMPLER_MIPMAP_MODE_LINEAR;   // trilinear
info.anisotropyEnable = VK_FALSE;
```

- Anisotropic Filtering:

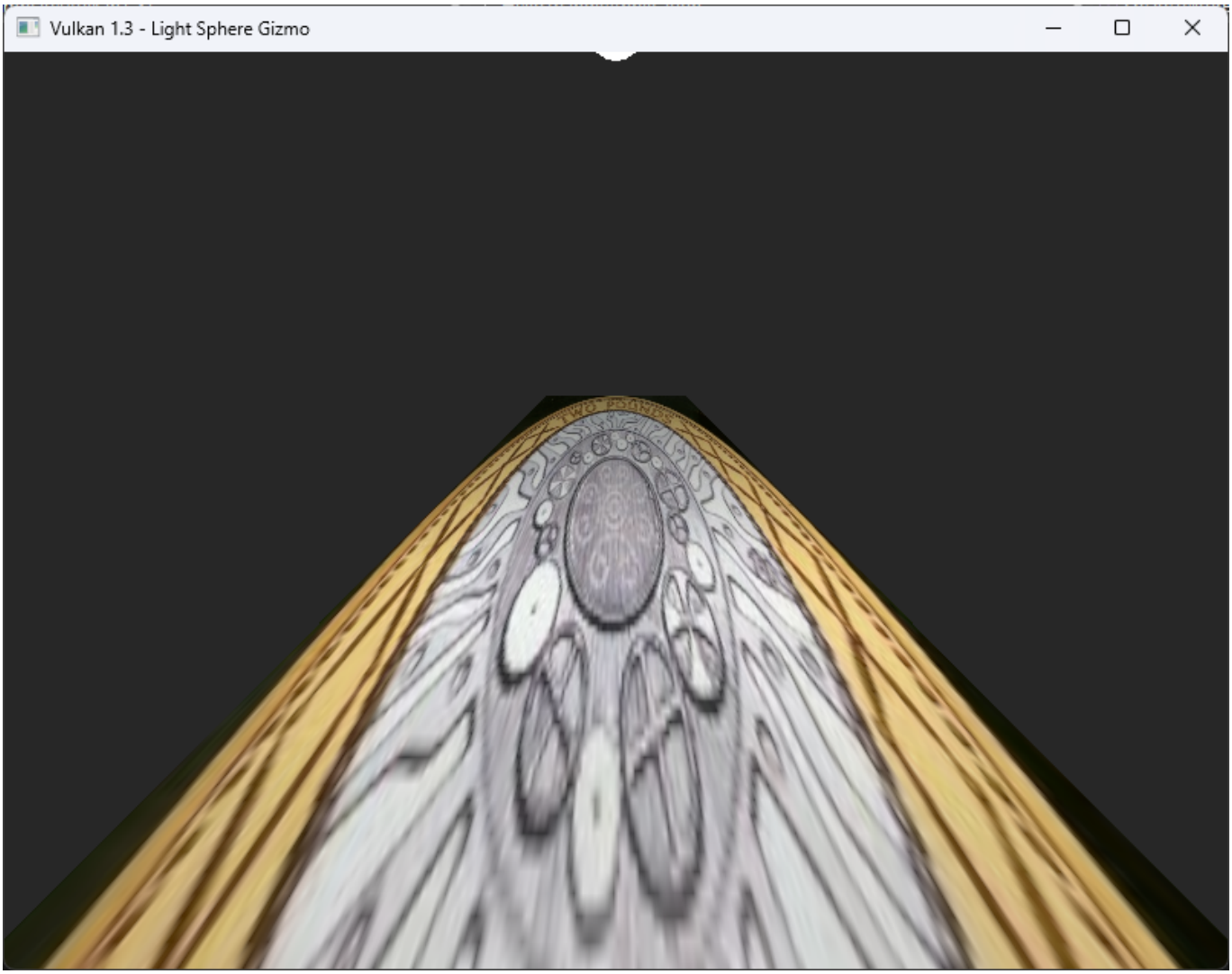
```
info.magFilter = VK_FILTER_LINEAR;
info.minFilter = VK_FILTER_LINEAR;
info.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
info.anisotropyEnable = VK_TRUE;
info.maxAnisotropy = props.limits.maxSamplerAnisotropy;
```

Output:

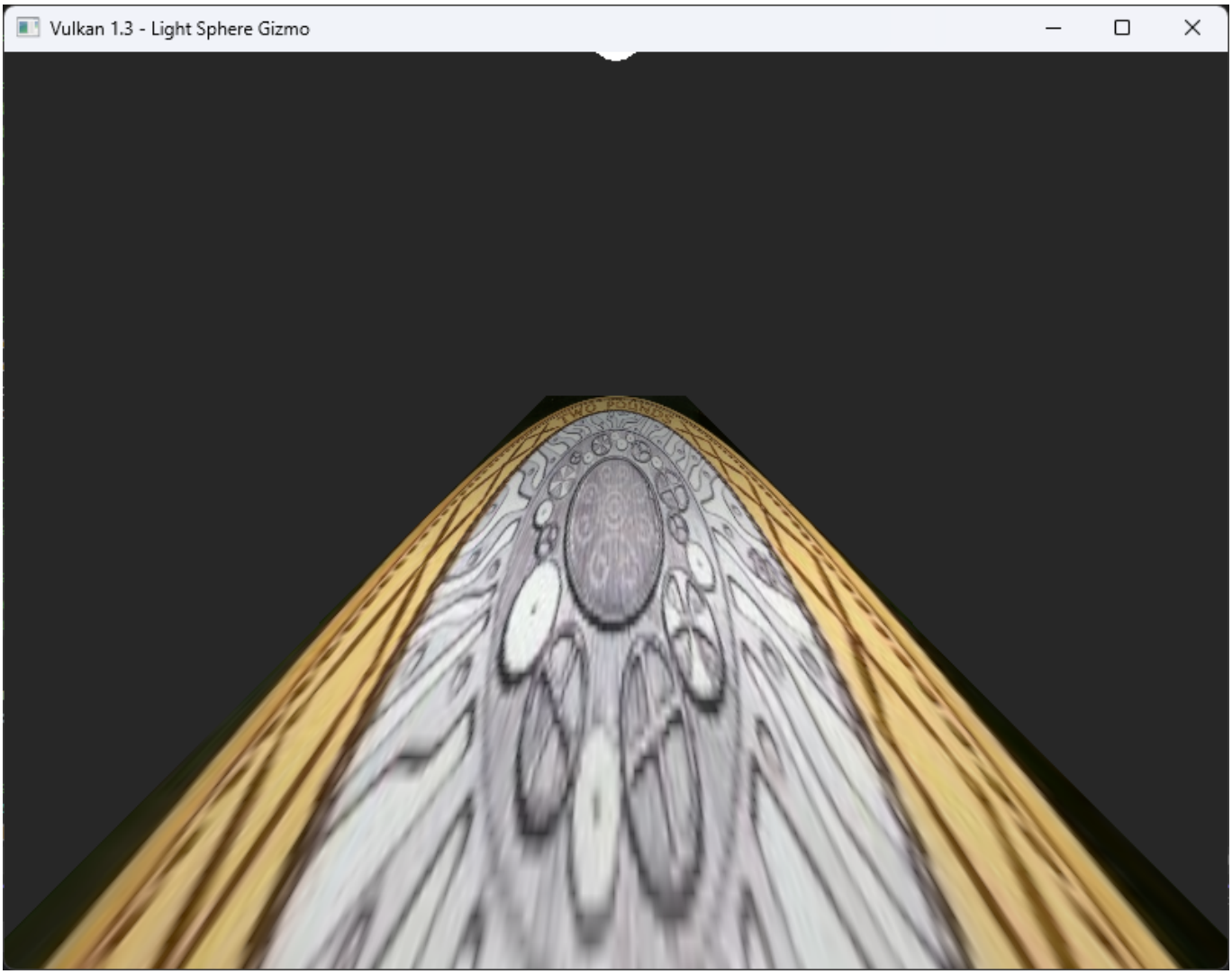
- Nearest Neighbor Filtering:



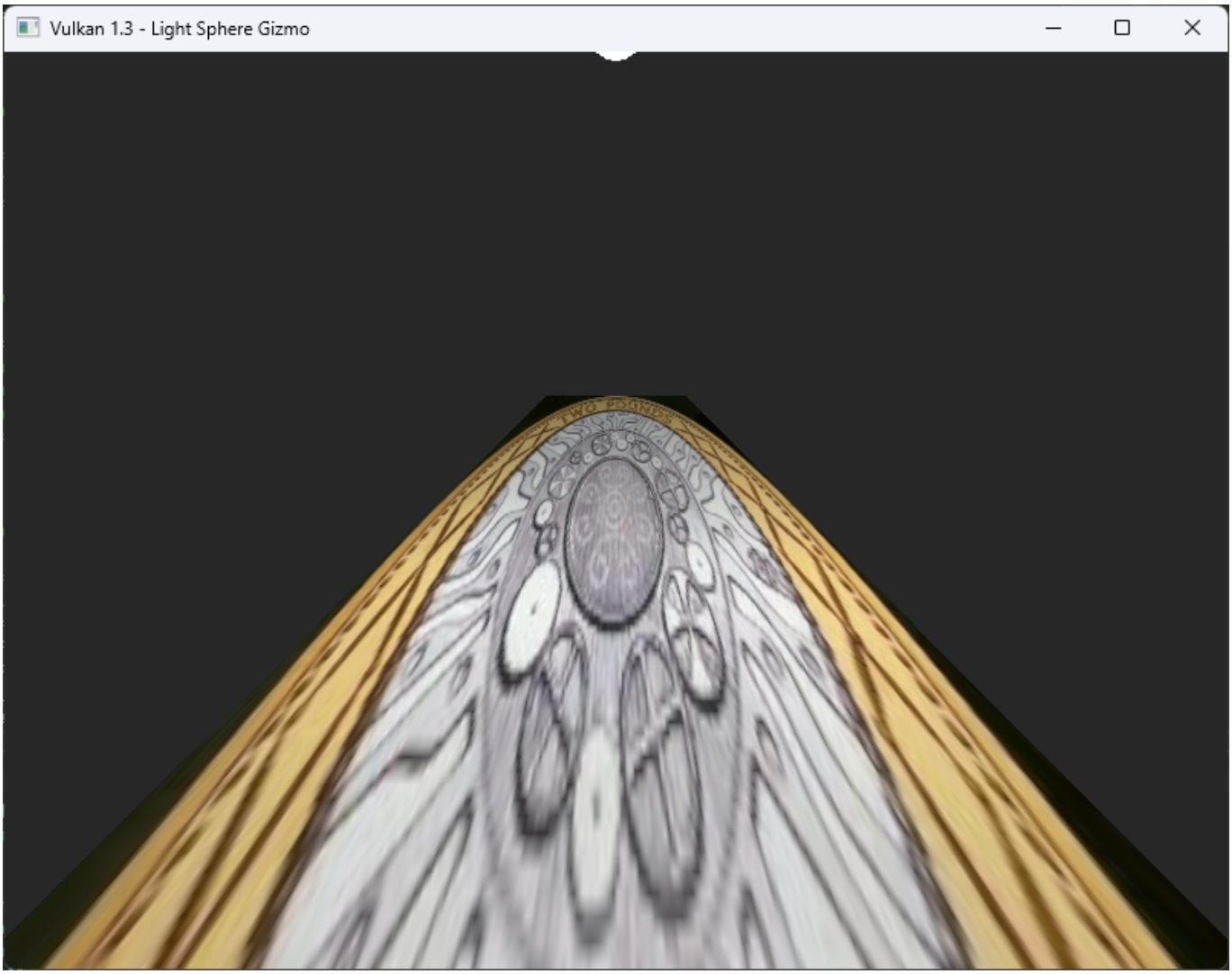
- Bilinear Filtering:



- Bicubic Filtering:



- Anisotropic Filtering:



Reflection:

Through this exercise I learned how texture filtering choices directly affect perceived image quality in real time rendering. I observed that nearest neighbour filtering is fast but causes visible aliasing, while bilinear and trilinear filtering produce smoother results by averaging samples. The bicubic approximation showed how Vulkan trilinear filtering can be used in place of true bicubic filtering when it is not supported. Anisotropic filtering was the most effective technique for keeping detail sharp on surfaces viewed at shallow angles such as the stretched road. Overall I gained a deeper understanding of how texture sampling methods reduce blurring and aliasing and how careful filtering selection can significantly improve visual realism without changing the geometry.

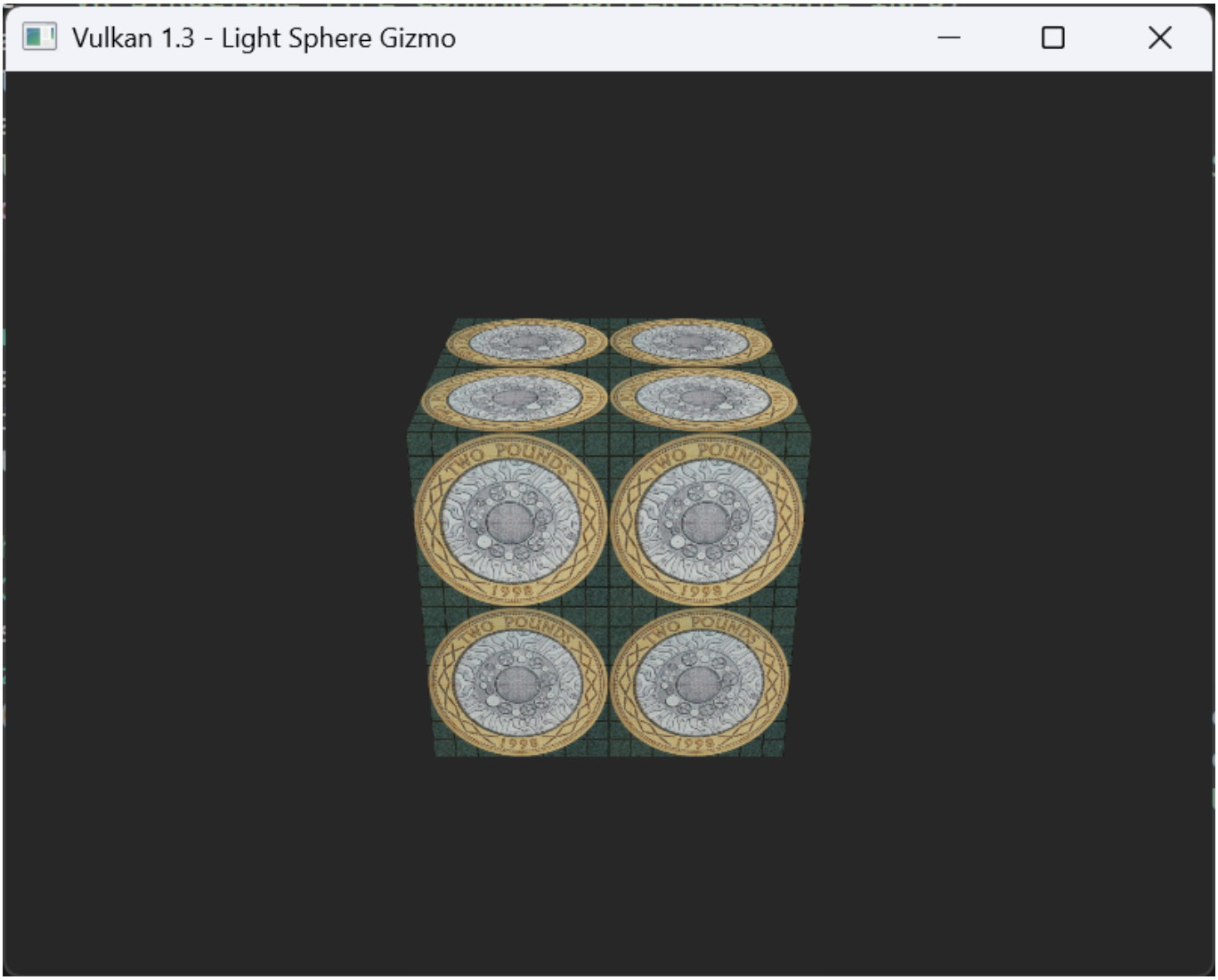
EXERCISE 8: MULTIPLE TEXTURING

Solution:

To complete this exercise I extended the existing texture system by creating one additional texture rather than redesigning the whole pipeline. The original texture setup was kept for the coin image, and a new texture was added for the tiled surface. This required defining one more texture image, view, and sampler in the C++ code, and updating the descriptor set layout to include the extra binding. The fragment shader was then modified to read from both textures and combine them, using the coin texture as an overlay on top of the tile base. Minor adjustments were also made to the shader to simplify the lighting so that the cube appears

evenly lit, allowing the two textures to blend clearly. This addition of a second texture and the shader update achieved the multiple-texturing outcome required for the exercise.

Output:



Reflection:

Through this exercise I learned how to extend a Vulkan texture pipeline to support multiple textures and how descriptor bindings directly connect data on the CPU side to shader inputs on the GPU. Adding a second texture showed the importance of managing image views, samplers, and descriptor updates in parallel to ensure both textures are accessible within the same fragment shader. Modifying the shader to blend two textures deepened my understanding of how texture sampling and layering work together to produce composite materials. It also reinforced how even small lighting adjustments can affect how clearly textures are perceived. Overall, this task helped strengthen my confidence in handling descriptor layouts, texture management, and shader coordination to achieve a richer and more controlled rendering result.

EXERCISE 9: AN OPEN BOX

Solution:

To achieve the cube with rock on the outside and wood on the inside while leaving the top open, I configured the Vulkan pipeline to use two textures and handle both surfaces within a single mesh. I loaded two images,

rock.jpg and wood.jpg, created their image views and samplers, and bound them to descriptor set bindings 1 and 2 alongside the uniform buffer at binding 0. In the pipeline settings I disabled face culling so that both the front and back faces of the cube could be rendered. The cube geometry was defined without its top face to allow viewing inside. In the vertex shader I output the world position, normal, and UV coordinates, scaling the UVs slightly to extend the rock texture across the outer faces. In the fragment shader I sampled both textures and selected which to display based on surface orientation. Front facing surfaces sampled from the rock texture while back facing surfaces sampled from the wood texture. I then applied simple diffuse lighting using the light and camera positions from the uniform buffer to produce realistic shading. The result was an open cube where the exterior appears rocky, the interior wooden, and the lighting consistent across both materials.

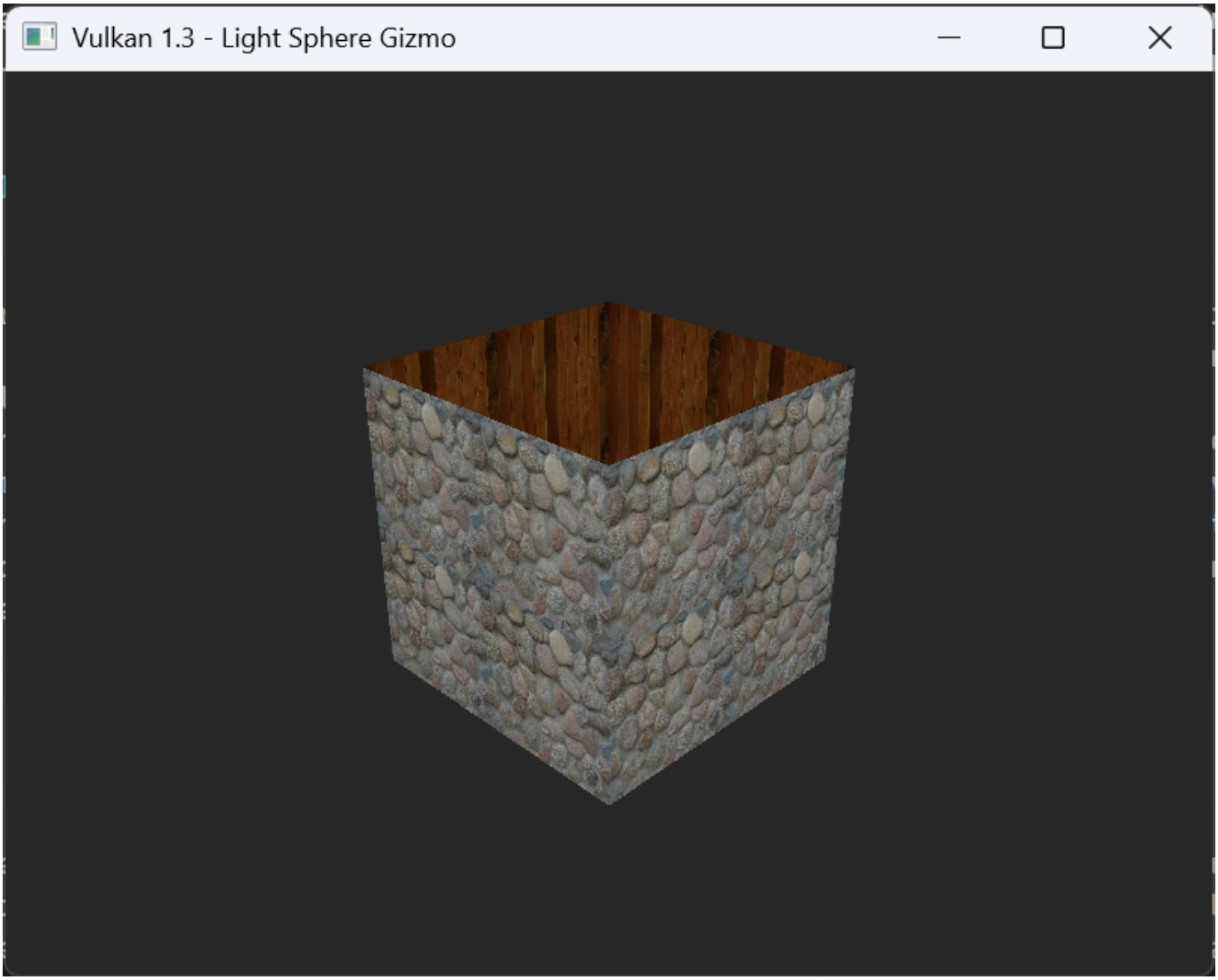
- Descriptor Set Layout with Two Texture Bindings:

```
void HelloTriangleApplication::createDescriptorSetLayout() {
    VkDescriptorSetLayoutBinding ubo{0, VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, 1,
        VK_SHADER_STAGE_VERTEX_BIT | VK_SHADER_STAGE_FRAGMENT_BIT};

    VkDescriptorSetLayoutBinding tex1{1,
VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, 1,
        VK_SHADER_STAGE_FRAGMENT_BIT};

    VkDescriptorSetLayoutBinding tex2{2,
VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, 1,
        VK_SHADER_STAGE_FRAGMENT_BIT};

    std::array<VkDescriptorSetLayoutBinding, 3> bindings{ubo, tex1, tex2};

    VkDescriptorSetLayoutCreateInfo
info{VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO};
    info.bindingCount = static_cast<uint32_t>(bindings.size());
    info.pBindings = bindings.data();
    vkCreateDescriptorSetLayout(device, &info, nullptr, &descriptorSetLayout);
}
```

- Fragment Shader Sampling Both Textures Based on Face Orientation:

```
vec3 color1 = texture(texSampler1, vUV).rgb;  // rock
vec3 color2 = texture(texSampler2, vUV).rgb;  // wood

vec3 normalViewSpace = normalize(mat3(transpose(inverse(ubo.model))) *
vWorldNormal);
bool isRearFace = isRearFaceByNormal(normalViewSpace);

vec3 finalColor = isRearFace ? color2 : color1;
```

Output:



Reflection: In completing this exercise I learned how to combine multiple textures and lighting within a single Vulkan pipeline to create realistic materials on a 3D object. I gained a clearer understanding of how fragment shaders can use surface orientation to dynamically select between different textures, allowing the same geometry to display different materials on its inside and outside faces. I also learned the importance of enabling depth buffering so that fragments are rendered in the correct order and 3D scenes appear spatially accurate. Implementing the depth image, view, and pipeline configuration helped me understand how Vulkan manages per-fragment depth testing. Overall, this exercise strengthened my grasp of texture sampling, normal transformations, and the role of depth in achieving visually correct 3D rendering.