Vulkan Lab 8: Post-Processing Effects

EXERCISE 1: RENDER-TO-TEXTURE ON A CUBE

Solution: To complete this exercise, I used the lecture material and the lab sheet to implement the render to texture technique in Vulkan. The process begins by creating an offscreen image that the first pass can write into, along with an image view and a sampler that allow the second pass to read from it. I then created a post process descriptor set layout so that the offscreen texture could be bound in the second pass. The command buffer was recorded with two stages. The first stage renders the cube normally into the offscreen image by transitioning it into a color attachment state, beginning a dynamic rendering block, binding the main graphics pipeline, and drawing the indexed geometry. After the first stage finishes, the image is transitioned into a shader read only state so that the fragment shader in the next stage can safely sample from it. The second stage begins a new rendering pass to the swapchain image, binds the post processing pipeline, and draws using the descriptor set that contains the offscreen sampler. The shader used in this pass simply maps the offscreen texture onto the cube, allowing the cube to display a live snapshot of itself that was created in the first stage.

- Create offscreen resources

```
void createOffscreenResources() {
    VkFormat fmt = swapChainImageFormat;

    createImage(
        swapChainExtent.width,
        swapChainExtent.height,
        fmt,
        VK_IMAGE_TILING_OPTIMAL,
        VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT |
        VK_IMAGE_USAGE_SAMPLED_BIT,
        VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
        offscreenImage,
        offscreenImageMemory
    );

    offscreenImageView = createImageView(
        offscreenImage,
        fmt,
        VK_IMAGE_ASPECT_COLOR_BIT
    );

    VkSamplerCreateInfo info{};
    info.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
    info.magFilter = VK_FILTER_LINEAR;
    info.minFilter = VK_FILTER_LINEAR;
    info.addressModeU = VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE;
    info.addressModeV = VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE;
    info.addressModeW = VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE;
```

```
    vkCreateSampler(device, &info, nullptr, &offscreenSampler);
}
```

- Create post-processing descriptor set layout

```
void createPostDescriptorSetLayout() {
    VkDescriptorSetLayoutBinding ubo{};
    ubo.binding = 0;
    ubo.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    ubo.descriptorCount = 1;
    ubo.stageFlags = VK_SHADER_STAGE_VERTEX_BIT |
VK_SHADER_STAGE_FRAGMENT_BIT;

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

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

    VkDescriptorSetLayoutCreateInfo ci{};
    ci.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
    ci.bindingCount = 2;
    ci.pBindings = bindings.data();

    vkCreateDescriptorSetLayout(device, &ci, nullptr,
&postDescriptorSetLayout);
}
```

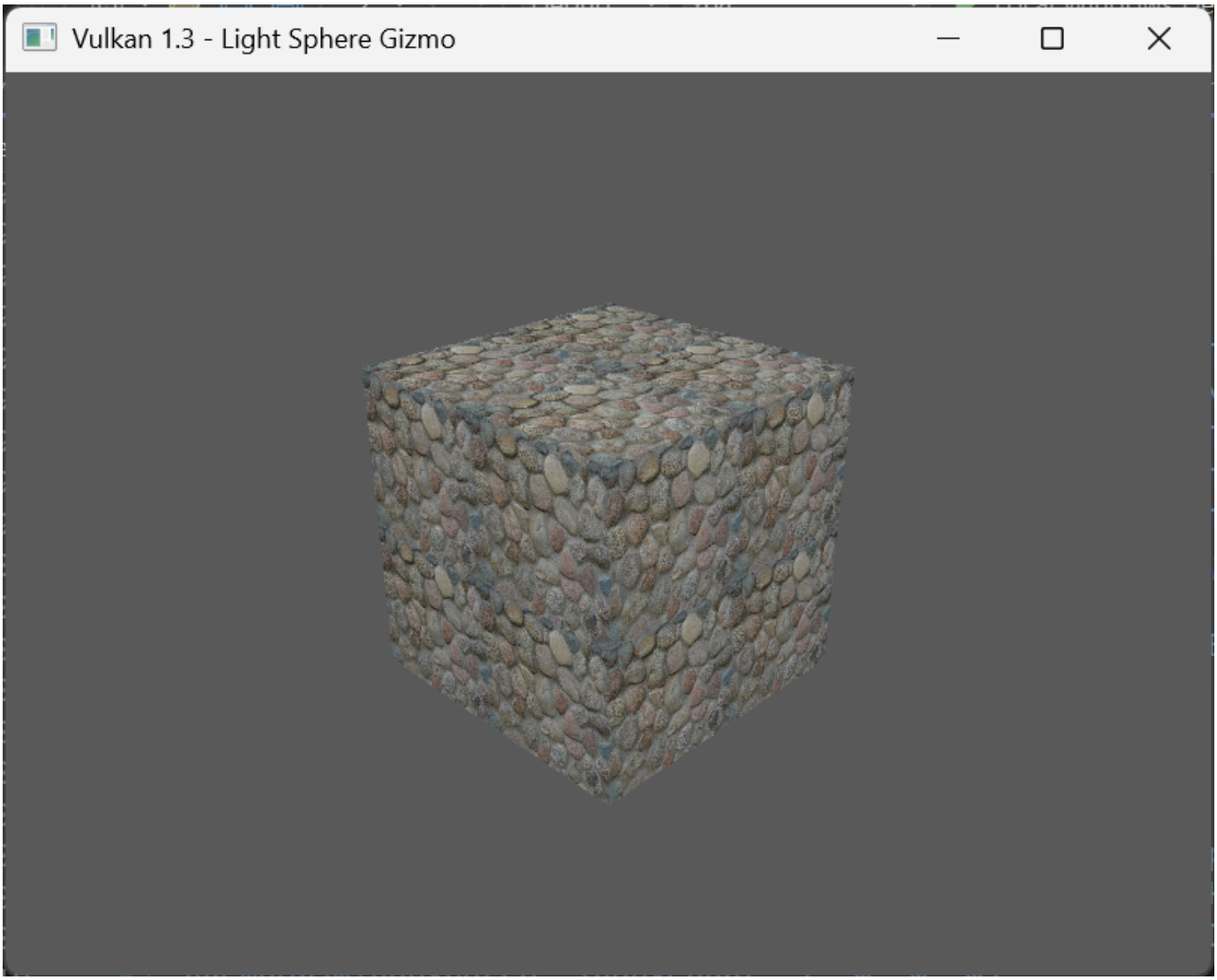
- Record command buffer with two passes

```
// Pass 1: Offscreen → COLOR_ATTACHMENT
VkImageMemoryBarrier2 offToColor{};
offToColor.oldLayout = VK_IMAGE_LAYOUT_UNDEFINED;
offToColor.newLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
offToColor.srcStageMask  = VK_PIPELINE_STAGE_2_TOP_OF_PIPE_BIT;
offToColor.dstStageMask  =
VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT;
offToColor.srcAccessMask = 0;
offToColor.dstAccessMask = VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT;
offToColor.image = offscreenImage;
offToColor.subresourceRange = { VK_IMAGE_ASPECT_COLOR_BIT, 0, 1, 0, 1 };
vkCmdPipelineBarrier2(cb, &depOffToColor);

// --- PASS 1 BEGIN ---
vkCmdBeginRendering(cb, &sceneRenderingInfo);
vkCmdBindPipeline(cb, VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);
vkCmdBindDescriptorSets(cb, VK_PIPELINE_BIND_POINT_GRAPHICS,
```

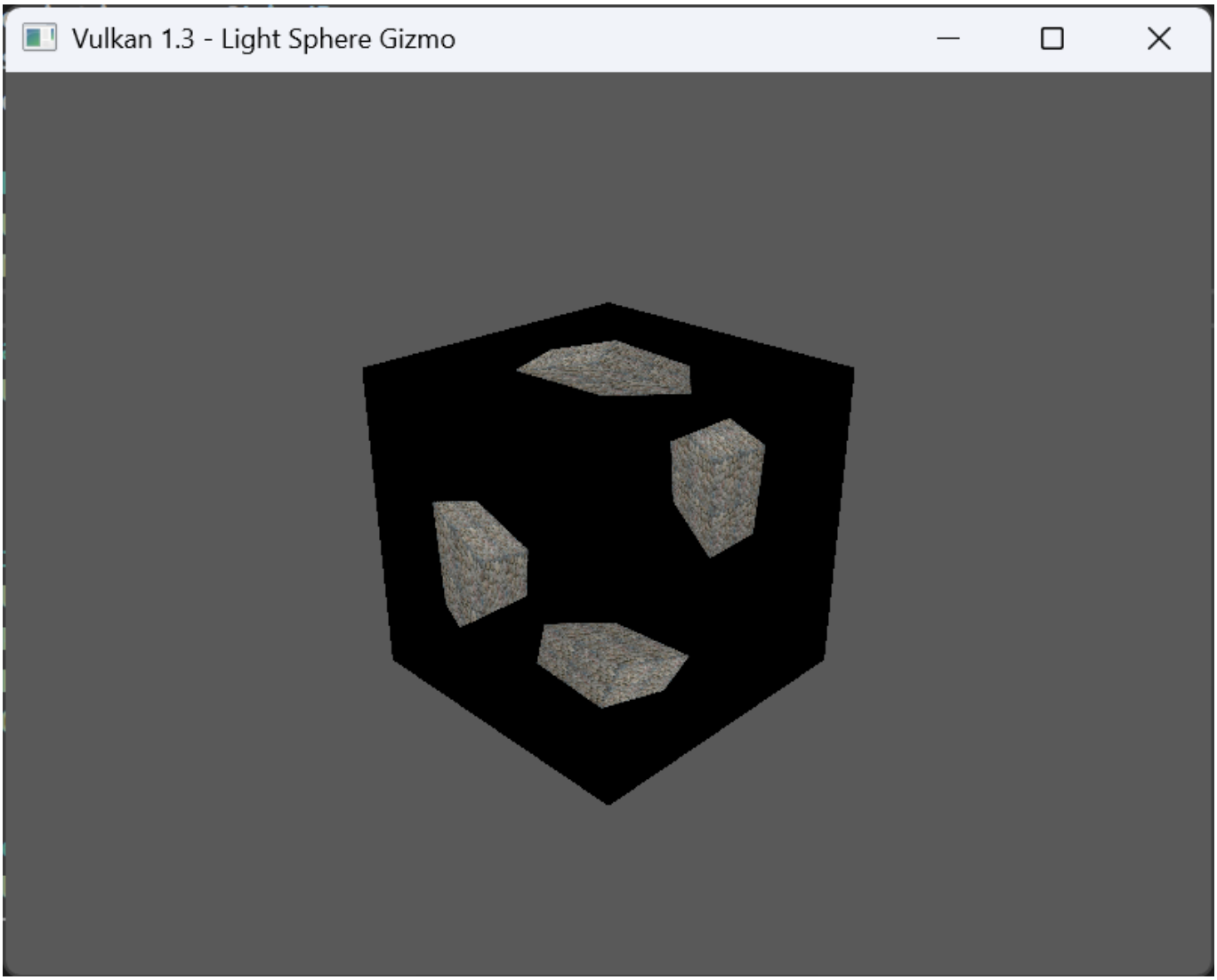
```
                        pipelineLayout, 0, 1, &descriptorSets[i], 0,
nullptr);
vkCmdDrawIndexed(cb, indexCount, 1, 0, 0, 0);
vkCmdEndRendering(cb);

// Pass 1 → Pass 2: COLOR_ATTACHMENT → SHADER_READ_ONLY
VkImageMemoryBarrier2 offToSample{};
offToSample.oldLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
offToSample.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
offToSample.srcStageMask  =
VK_PIPELINE_STAGE_2_COLOR_ATTACHMENT_OUTPUT_BIT;
offToSample.dstStageMask  = VK_PIPELINE_STAGE_2_FRAGMENT_SHADER_BIT;
offToSample.srcAccessMask = VK_ACCESS_2_COLOR_ATTACHMENT_WRITE_BIT;
offToSample.dstAccessMask = VK_ACCESS_2_SHADER_READ_BIT;
offToSample.image = offscreenImage;
offToSample.subresourceRange = { VK_IMAGE_ASPECT_COLOR_BIT, 0, 1, 0, 1 };
vkCmdPipelineBarrier2(cb, &depOffToSample);

// --- PASS 2 BEGIN ---
vkCmdBeginRendering(cb, &finalRenderingInfo);
vkCmdBindPipeline(cb, VK_PIPELINE_BIND_POINT_GRAPHICS, postPipeline);
vkCmdBindDescriptorSets(cb, VK_PIPELINE_BIND_POINT_GRAPHICS,
                        postPipelineLayout, 0, 1, &postDescriptorSets[i],
0, nullptr);
vkCmdDrawIndexed(cb, indexCount, 1, 0, 0, 0);
vkCmdEndRendering(cb);
```

Output:

- Pass 1: Render scene to a texture



- Pass 2: Map the texture onto a cube



Reflection: This exercise developed my understanding of how Vulkan manages multi stage rendering and how textures can be generated entirely within the graphics pipeline. Implementing the offscreen pass helped me become more confident with image layout transitions, memory barriers, and the use of dynamic rendering. Binding the same scene data in two different stages also highlighted how flexible descriptor sets and pipelines are when constructing rendering workflows. Seeing the cube textured with an image of itself confirmed that the first stage and second stage were communicating correctly, and it helped reinforce how render to texture forms the basis of many advanced effects such as mirrors, shadows, post processing, and reflections.

EXERCISE 2: TEXTURE SMOOTHING (BOX BLUR)

Solution: To complete this exercise, I reused the render to texture setup from the previous task and extended it with a post processing blur stage. The first pass renders the scene into an offscreen image that is created with color attachment and sampled usage. After rendering the cube into this image, the command buffer performs a layout transition so that the image can be read by the fragment shader. In the second pass, the code begins a new rendering operation that targets the swapchain image and binds a special post processing pipeline. This pipeline uses a full screen triangle and a blur shader that repeatedly samples nearby texels from the offscreen texture. By binding the post descriptor set, which contains the uniform buffer and the offscreen sampler, the second pass applies the box blur to produce a smoothed version of the scene. The output is the blurred image displayed on the screen.

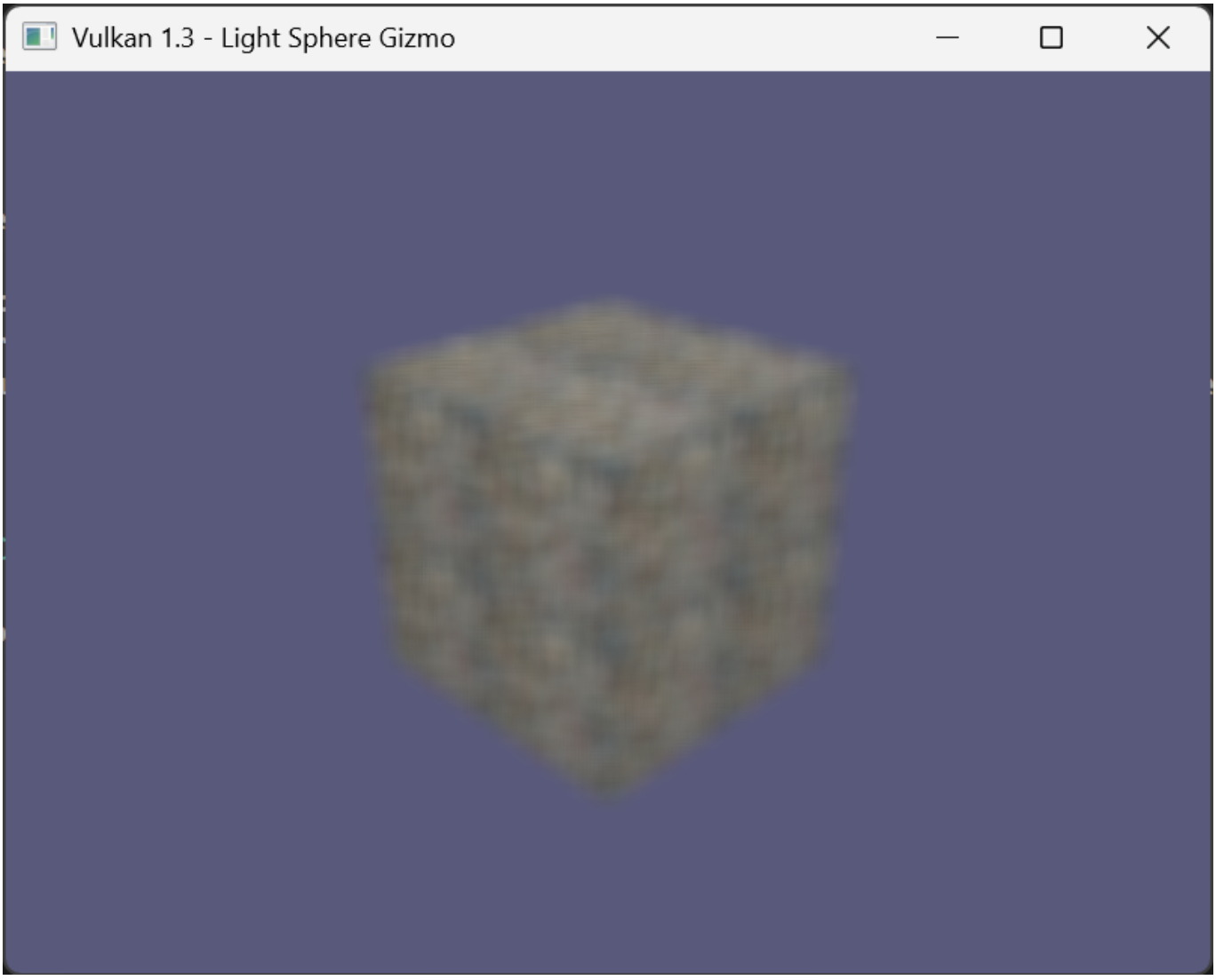
- Command buffer recording for two passes

```
// PASS 2: Fullscreen Blur (sample from offscreen image)
VkRenderingAttachmentInfo colorAtt2{};
colorAtt2.sType = VK_STRUCTURE_TYPE_RENDERING_ATTACHMENT_INFO;
colorAtt2.imageView = swapChainImageViews[imageIndex];
colorAtt2.imageLayout = VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
colorAtt2.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
colorAtt2.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
colorAtt2.clearValue = { {{0.0f, 0.0f, 0.0f, 1.0f}} };

VkRenderingInfo render2{};
render2.sType = VK_STRUCTURE_TYPE_RENDERING_INFO;
render2.colorAttachmentCount = 1;
render2.pColorAttachments = &colorAtt2;
render2.renderArea = { {0, 0}, swapChainExtent };
render2.layerCount = 1;

vkCmdBeginRendering(cb, &render2);

// Set fullscreen viewport + scissor
VkViewport vp2{};
vp2.x = 0.0f;
vp2.y = 0.0f;
vp2.width  = (float)swapChainExtent.width;
vp2.height = (float)swapChainExtent.height;
vp2.minDepth = 0.0f;
vp2.maxDepth = 1.0f;
vkCmdSetViewport(cb, 0, 1, &vp2);

VkRect2D sc2{};
sc2.offset = {0, 0};
sc2.extent = swapChainExtent;
vkCmdSetScissor(cb, 0, 1, &sc2);

// Bind blur pipeline + descriptor set
vkCmdBindPipeline(cb, VK_PIPELINE_BIND_POINT_GRAPHICS, postPipeline);
vkCmdBindDescriptorSets(
    cb, VK_PIPELINE_BIND_POINT_GRAPHICS,
    postPipelineLayout,
    0, 1,
    &postDescriptorSets[currentFrame],
    0, nullptr
);

// Fullscreen triangle (no VBO needed)
vkCmdDraw(cb, 3, 1, 0, 0);

vkCmdEndRendering(cb);
```

- Fragment shader for box blur effect

```
#version 450

layout(location = 0) out vec2 uv;

vec2 POS[3] = vec2[](
    vec2(-1, -1),
    vec2( 3, -1),
    vec2(-1,  3)
);

vec2 UV[3] = vec2[](
    vec2(0, 0),
    vec2(2, 0),
    vec2(0, 2)
);

void main() {
    gl_Position = vec4(POS[gl_VertexIndex], 0, 1);
    uv = UV[gl_VertexIndex];
}
```

Output:



Reflection: This exercise helped me understand how multi pass rendering works in Vulkan and how post processing effects are created by applying shaders to an image produced in a previous pass. Implementing the blur required setting up the offscreen resources correctly, managing the layout transitions between color attachment and shader sampled states, and drawing a full screen triangle that uses texture coordinates to read from the offscreen image. The fragment shader performs repeated sampling to average neighbouring pixels, and seeing the effect confirmed how texture based filters operate in real time graphics. This task strengthened my understanding of render pipelines, descriptor sets, and how post processing effects are layered on top of normal scene rendering.

EXERCISE 3: SIMPLE GLOW EFFECT

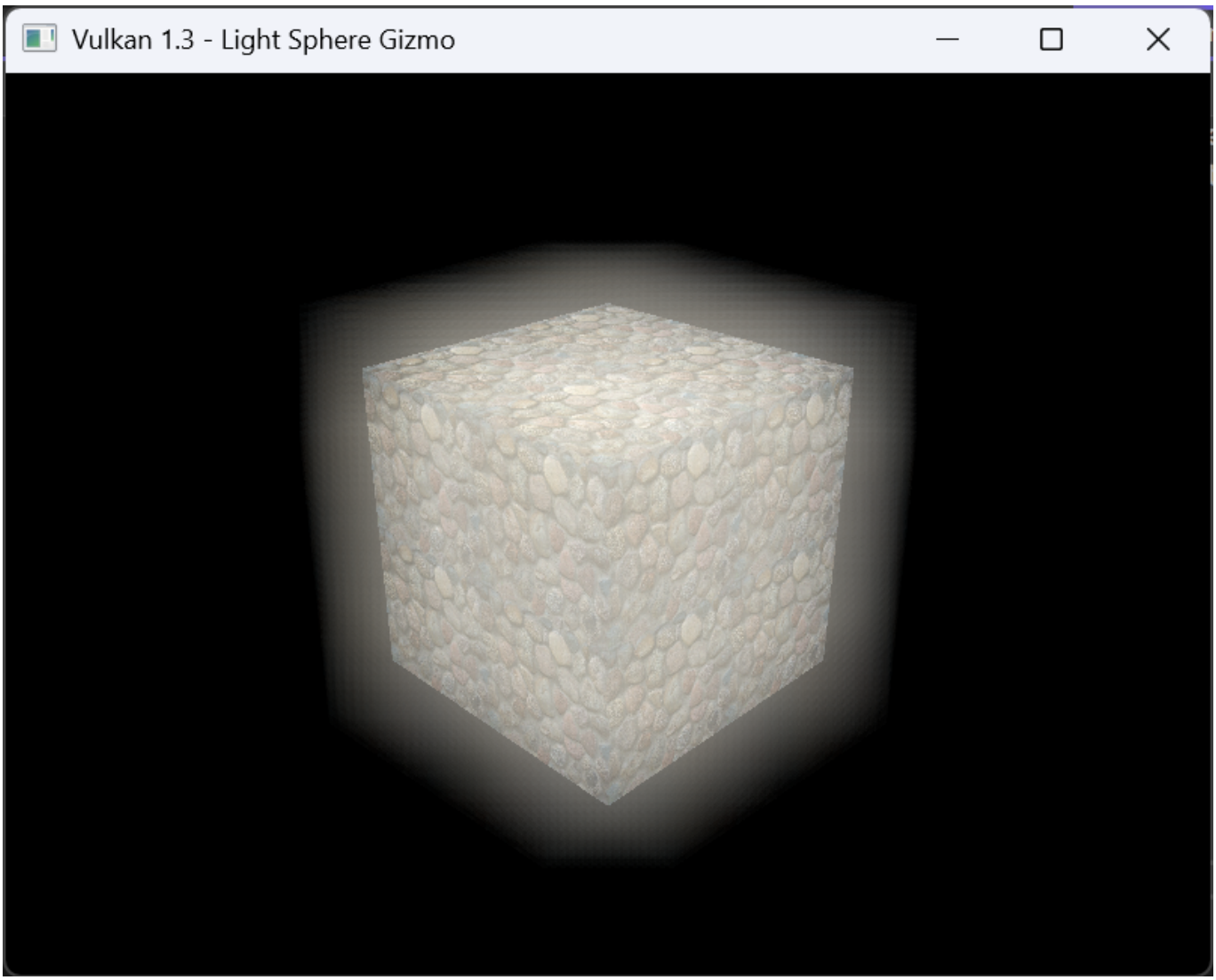
Solution: For this exerise, the first thing I did was to create a fragment shader that implements a glow effect by combining a blurred version of the scene.The shader uses the same logic that was use in the previous full screen blur effect to sample nearby texels and compute a blurred color. It then adds this blurred color to the original sharp color sampled from the texture, scaled by a glow strength factor. This creates a glowing effect around bright areas of the scene. I then integrated this shader into the existing render to texture framework from the previous exercises. The command buffer records two passes: the first pass renders the scene into an offscreen image, and the second pass applies the glow shader to produce the final output on the swapchain image. By binding the appropriate descriptor sets and pipelines, the glow effect is applied in real time as the scene is rendered.

- Fragment shader for glow effect

```
#version 450

layout(location = 0) in vec2 uv;
layout(location = 0) out vec4 outColor;

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

void main() {
    int   boxSize   = 7;
    float stepSize  = 6.0;

    vec2 texel = stepSize / vec2(textureSize(sceneTexture, 0));

    vec3 sum   = vec3(0.0);
    int  total = 0;

    for (int x = -boxSize; x <= boxSize; x++) {
        for (int y = -boxSize; y <= boxSize; y++) {
            vec2 offset = vec2(x, y) * texel;
            sum += texture(sceneTexture, uv + offset).rgb;
            total++;
        }
    }

    vec3 blurred = sum / float(total);
    vec3 sharp = texture(sceneTexture, uv).rgb;
```

```
    float glowStrength = 2.5;
    vec3 result = sharp + blurred * glowStrength;

    outColor = vec4(result, 1.0);
}
```

Output:



Reflection: This exercise deepened my understanding of how post processing effects can be layered on top of standard rendering techniques in Vulkan. By implementing a glow effect that combines a blurred version of the scene with the original sharp image, I learned how to manipulate texture data in shaders to achieve visually appealing results. The process of sampling nearby texels and blending them with the original color highlighted the flexibility of fragment shaders in creating custom effects. Integrating this shader into the existing render to texture framework reinforced my knowledge of multi pass rendering, descriptor sets, and pipeline management in Vulkan.

EXERCISE 4: OBJECT ON FIRE ANIMATION

Solution: To extend the basic glow effect into a fire effect, I began by modifying the behaviour of the post-processing blur so that it no longer spread evenly in all directions. The original glow simply averaged

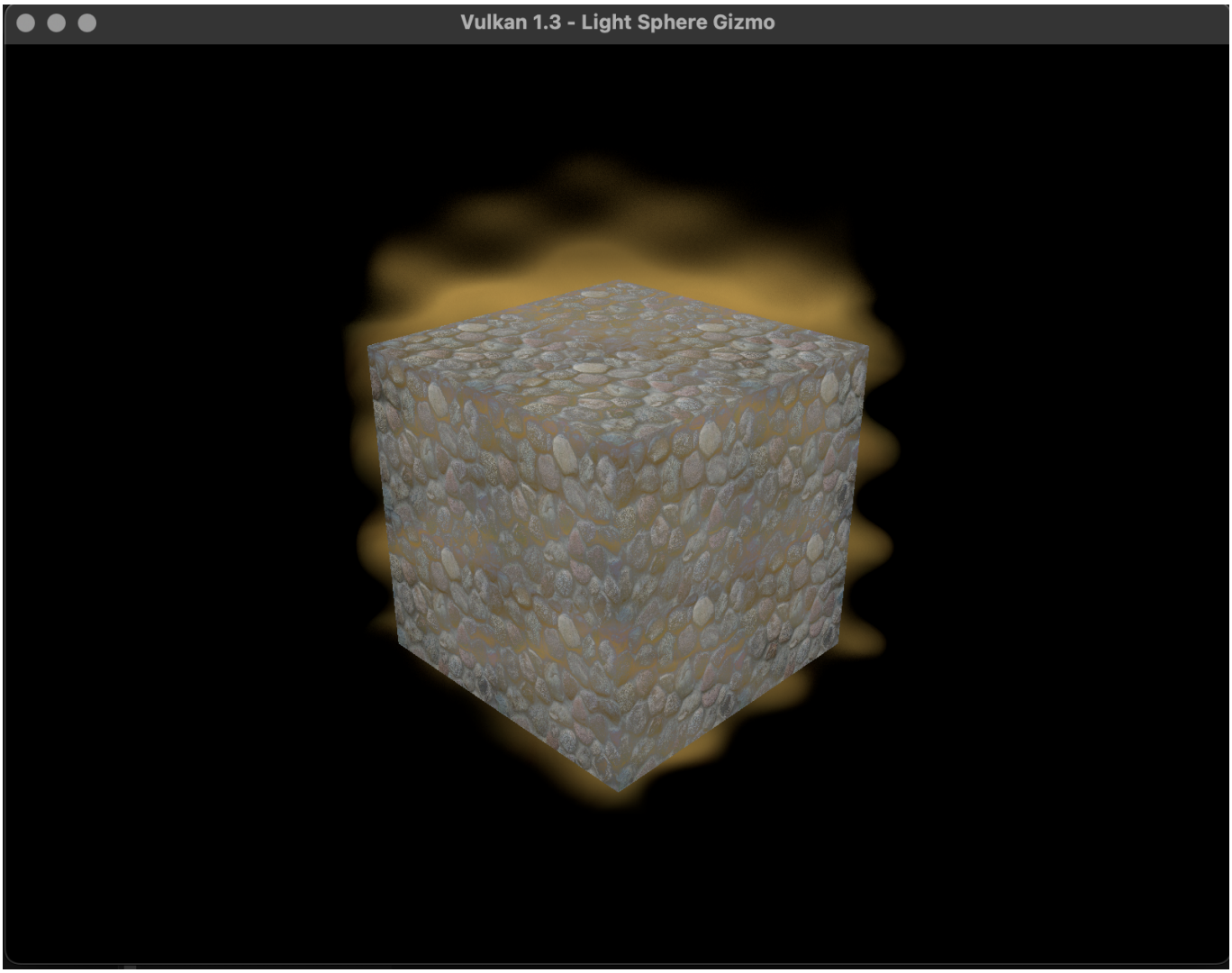
neighbouring texels with a uniform box filter, producing a soft halo. To push this towards a fire-like appearance, I added animated distortion to the sampling coordinates using a combination of sine-based heat wobble and a small hash noise function. This caused the blurred samples to shift unpredictably over time, giving the impression of turbulent heat. I then biased the blur vertically so that samples were pulled upwards, creating a rising motion characteristic of flames. The blur kernel was reshaped to favour vertical stretching and reduced horizontally, which stopped the effect from forming a thick band around the cube and instead made it taper more naturally above it. Additional masking was introduced to fade the effect as it moved away from the cube's centre and upper region, preventing the fire from looking uniform. Finally, a warm colour tint and a time-varying flicker were applied to the blurred component, giving the result the bright orange/yellow appearance of burning. These combined adjustments transformed the original glow into a dynamic, directional, and visually irregular fire effect around the cube.

- Fragment shader for fire effect

```
#version 450

layout(location = 0) in vec2 uv;
layout(location = 0) out vec4 outColor;

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

layout(push_constant) uniform FireParams {
    float time;
    float intensity;
    float pad0;
    float pad1;
} fire;

float hash(vec2 p) {
    return fract(sin(dot(p, vec2(127.1, 311.7))) * 43758.5453123);
}

void main() {
    vec2 texSize = vec2(textureSize(sceneTex, 0));

    vec2 texel = 3.5 / texSize;
    vec4 sharp = texture(sceneTex, uv);

    float noise = sin(uv.y * 60.0 + fire.time * 10.0) *
                  cos(uv.x * 40.0 - fire.time * 7.0);
    vec2 wobble = texel * 8.0 * noise;

    int   radius    = 10;
    vec3  blurSum   = vec3(0.0);
    float weightSum = 0.0;


    float v = uv.y;

    float centerX = 0.5;
    float distFromCenter = abs(uv.x - centerX);
```

```
    for (int x = -radius; x <= radius; x++) {
        for (int y = -radius; y <= radius; y++) {
            vec2 o = vec2(x, y);

            float dist = length(vec2(o.x * 0.35, o.y));
            float w = max(0.0, 1.0 - dist / float(radius + 1));
            float nTap = hash(o * 3.17 + uv * 25.0 + fire.time);

            float baseUp = float(radius + 1 - y) / float(radius + 1);
            baseUp = max(baseUp, 0.0);

            float uneven = 0.6 + 0.8 * nTap;


            float heightMask = 1.0 - smoothstep(0.45, 0.85, v);
            float sideMask = 1.0 - smoothstep(0.18, 0.25, distFromCenter);

            float upward = baseUp * uneven * heightMask * sideMask;
            upward = pow(upward, 1.3);

            float stretch = 0.05;

            vec2 sampleUV =
                uv
                + wobble
                + o * texel
                + vec2(0.0, upward * stretch);

            vec3 sampleCol = texture(sceneTex, sampleUV).rgb;

            float weightJitter = 0.7 + 0.6 * (nTap - 0.5);
            blurSum   += sampleCol * w * weightJitter;
            weightSum += w * weightJitter;
        }
    }

    vec3 blurred = blurSum / max(weightSum, 0.0001);

    vec3 aura = max(blurred - sharp.rgb, 0.0);

    vec3 fireTint = vec3(1.0, 0.6, 0.1);
    float flicker = 0.7 + 0.3 * sin(fire.time * 8.0 + uv.y * 50.0);

    vec3 color = sharp.rgb + aura * fire.intensity * fireTint * flicker;

    outColor = vec4(color, 1.0);
}
```

Output:



Reflection: Turning the glow effect into a fire effect helped me understand how post processing can shape the behaviour of a visual effect. I expected stronger blurring to be enough, but I discovered that fire needs motion, direction, and variation to look convincing. Introducing noise, upward stretching, and uneven sampling showed me how small adjustments to texture coordinates can dramatically change the result. I also realised that the first pass must contain enough brightness for the second pass to build on, because fire cannot form from a flat or dim input. Overall this task showed me that effective visual effects often come from combining simple techniques such as blurring, distortion, colour tinting, and animation to create something that feels natural and dynamic.