Vulkan Lab 7: Cube Mapping and Particle Systems

EXERCISE 1: IMPLEMENTING AND CONTROLLING THE DEPTH BUFFER

Solution: I had already implement the depth buffer in a previous lab. So to complete this exercise, I review my implementation to make sure it aligned with the snippets you have provided. I created a lambda function to toggle the depth buffer settings so I coould see the difference when rendering. The reason a lambda function is used is to encapsulate the depth buffer control logic in a concise manner, allowing for easy toggling of depth testing features without cluttering the main rendering code. This approach enhances code readability and maintainability.

- Depth Buffer Control Flag:

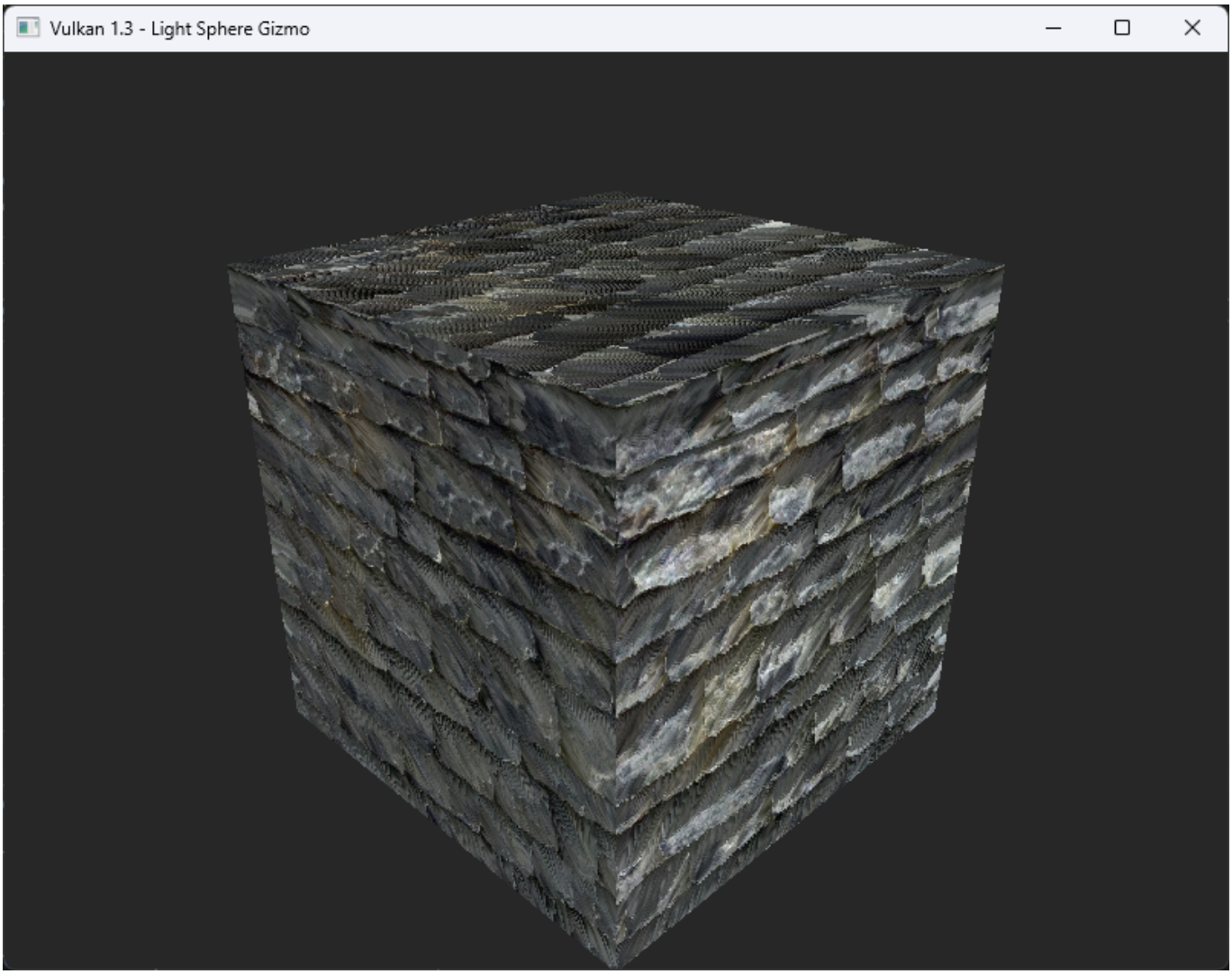
```
static constexpr bool USE_DEPTH = true;
```

- Default Depth Stencil State Creation Info:

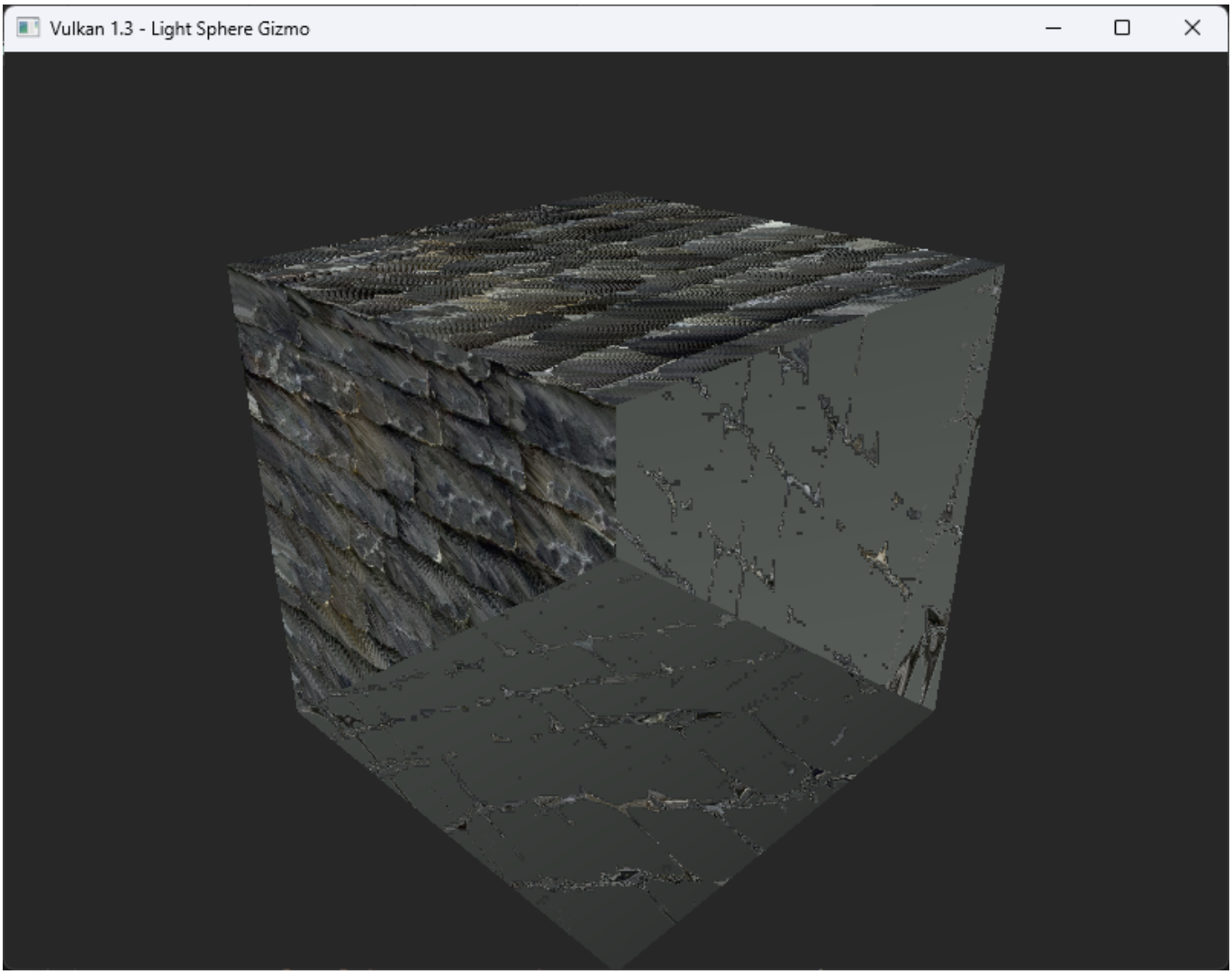
```
VkPipelineDepthStencilStateCreateInfo depth{};
depth.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
depth.depthTestEnable  = USE_DEPTH ? VK_TRUE : VK_FALSE;
depth.depthWriteEnable = USE_DEPTH ? VK_TRUE : VK_FALSE;
depth.depthCompareOp   = VK_COMPARE_OP_LESS;
depth.depthBoundsTestEnable = VK_FALSE;
depth.stencilTestEnable     = VK_FALSE;
```

Output:

- Depth Buffer Enabled



- Depth Buffer Disabled



Reflection: This exercise reinforced my understanding of depth buffering in Vulkan. By implementing a toggle for depth testing, I could visually observe the impact of depth buffering on rendering. It also put me on the path of making my rendering engine more flexible and configurable, which is crucial for developing complex graphics applications.

EXERCISE 2: IMPLEMENTING THE SKYBOX

Solution: In this exercise, I refactored the rendering pipeline from the previous lab to support a single cubemap-based skybox instead of multiple textured objects. I removed the two extra 2D samplers and updated the descriptor set layout to include only a uniform buffer (binding 0) and a cubemap sampler (binding 1). The graphics pipeline was simplified to use static depth testing with VK_COMPARE_OP_LESS_OR_EQUAL and depth writes disabled, ensuring the skybox always renders correctly behind all geometry.

I converted the texture-loading routine for the third sampler into a cubemap loader, which reads six images representing the skybox faces and creates a VK_IMAGE_VIEW_TYPE_CUBE image view. The fragment shader was replaced with a single samplerCube lookup, while the vertex shader removes camera translation to keep the cube centered around the viewer.

To make the scene interactive, I implemented a keyboard-controlled camera. The handleInput() function processes W/A/S/D for horizontal movement, Q/E for vertical movement, and arrow keys for yaw and pitch

rotation. These inputs update the camera position and orientation every frame, and the new view matrix is written to the uniform buffer so that the skybox view reacts smoothly to user movement.

- Skybox Depth Stencil State:

```
VkPipelineDepthStencilStateCreateInfo depth{};
    depth.sType =
VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
    depth.depthTestEnable = USE_DEPTH ? VK_TRUE : VK_FALSE;
    depth.depthWriteEnable = VK_FALSE;
    depth.depthCompareOp = VK_COMPARE_OP_LESS_OR_EQUAL;
    depth.depthBoundsTestEnable = VK_FALSE;
    depth.stencilTestEnable = VK_FALSE;
```

- Cull Front Faces for Skybox:

```
rs.cullMode  = VK_CULL_MODE_FRONT_BIT;
```

Output:

Reflection: During this exercise, I encountered an issue where the window opened and closed immediately after launch. After reviewing my code, I discovered that I had mistakenly left in the texture-loading and rendering logic from the previous lab while introducing the cubemap skybox. This created conflicts in resource allocation because the old descriptors and samplers no longer matched the new descriptor set layout. Once I removed all references to the previous textures and ensured that only the cubemap and uniform buffer were active, the application initialized correctly, and the skybox displayed as intended.

This experience taught me how critical careful resource management is in Vulkan. Even a small mismatch between descriptors, pipelines, or shaders can prevent a program from running. It also helped me appreciate how important it is to keep each rendering stage consistent and well organized.

By focusing only on the cubemap, I also learned more about how the skybox works in relation to the view matrix. The camera's translation is removed so the skybox remains centered around the viewer, which creates the illusion of an infinite environment. Once I implemented keyboard movement, I could move freely within the scene, and this demonstrated how changes in camera position and orientation directly influence what is drawn each frame.

Overall, this exercise strengthened my understanding of scene setup, pipeline configuration, and resource cleanup. It showed how attention to detail is essential when modifying an existing rendering system and how correct management of resources ensures that only the intended elements appear in the final sce

---

EXERCISE 3:

Solution: I kept one `createGraphicsPipeline` function but created two different pipeline state objects for the skybox and reflective meshes. One was resonsible for the skybox rendering with front-face culling and depth writes disabled, while the other handled the reflective objects. The shaders were modified to include a uniform flag indicating whether the current draw call was for the skybox or a reflective mesh. I computed the world space normal and view direction and then calculated the reflection vector using GLSL's built-in reflect function. As the camera moves around, the reflection vector updates accordingly, allowing the reflective objects to accurately mirror the skybox environment.

- Pipeline States:

```
// skybox:
VkPipelineRasterizationStateCreateInfo rsSky = rsCommon;
rsSky.cullMode = VK_CULL_MODE_FRONT_BIT;
VkPipelineDepthStencilStateCreateInfo dzSky{
VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO };
dzSky.depthTestEnable  = VK_TRUE;
dzSky.depthWriteEnable = VK_FALSE;
dzSky.depthCompareOp   = VK_COMPARE_OP_LESS_OR_EQUAL;

// reflective mesh:
VkPipelineRasterizationStateCreateInfo rsRefl = rsCommon;
rsRefl.cullMode = VK_CULL_MODE_BACK_BIT;
VkPipelineDepthStencilStateCreateInfo dzRefl{
VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO };
dzRefl.depthTestEnable  = VK_TRUE;
dzRefl.depthWriteEnable = VK_TRUE;
dzRefl.depthCompareOp   = VK_COMPARE_OP_LESS_OR_EQUAL;
```

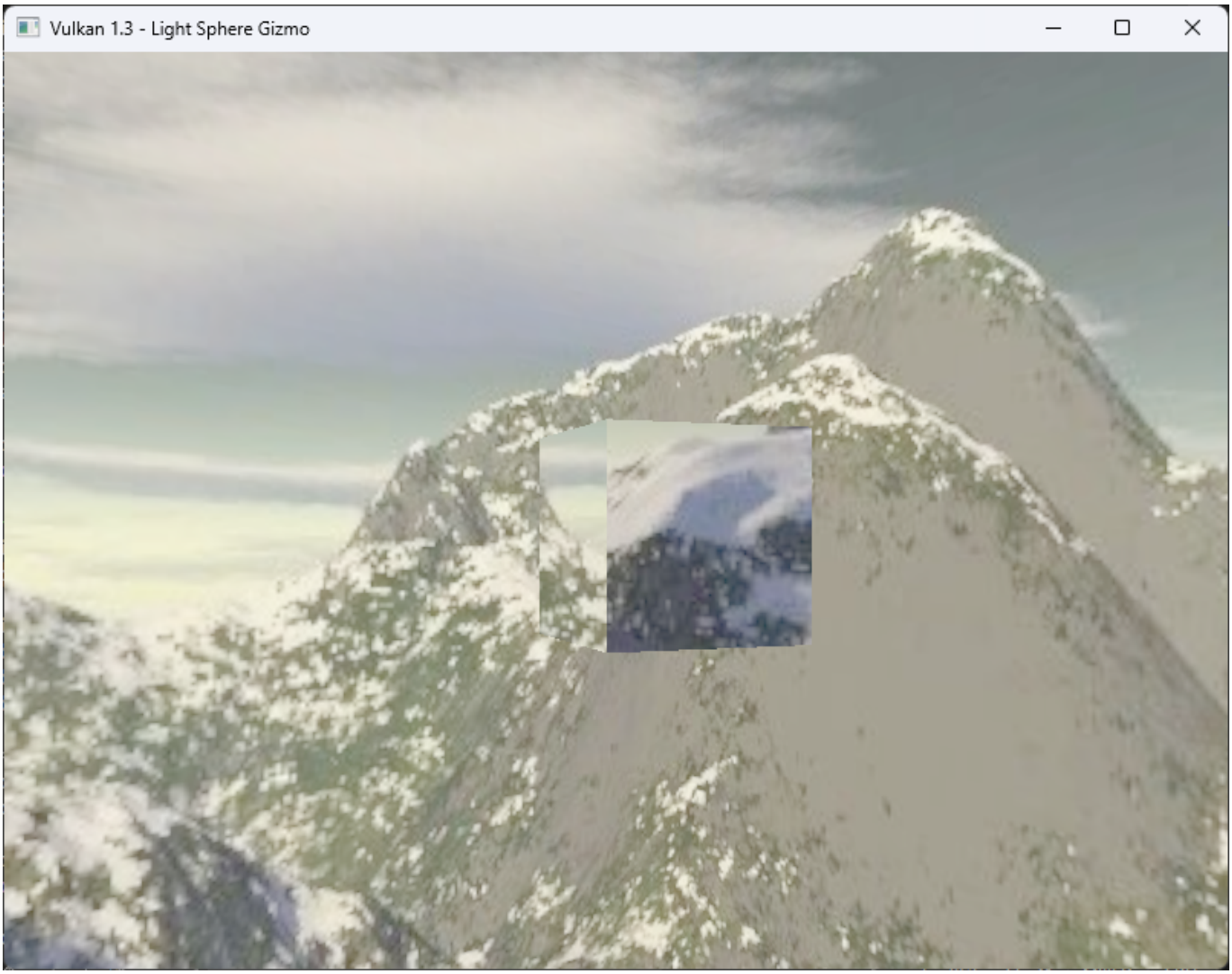
- Fragment Shader:

```
void main() {
    if (pc.unlit != 0u) {
        vec3 dir = normalize(vDir);
        vec3 col = texture(uSky, dir).rgb;
        outColor = vec4(pow(col, vec3(1.0 / 2.2)), 1.0);
        return;
    }

    vec3 N = normalize(vWorldNormal);
    vec3 V = normalize(ubo.eyePos.xyz - vWorldPos);
    vec3 I = -V;

    vec3 R = reflect(I, N);

    vec3 col = texture(uSky, R).rgb;

    outColor = vec4(pow(col, vec3(1.0 / 2.2)), 1.0);
}
```

- Vertex Shader:

```
void main() {
    mat4 M = (pc.useOverride != 0u) ? pc.modelOverride : ubo.model;

    vec4 world = M * vec4(inPos, 1.0);
    vWorldPos = world.xyz;
    vWorldNormal = normalize(mat3(M) * inNormal);

    mat4 V = ubo.view;
    if (pc.unlit != 0u) {
        mat4 Vsky = V;
        Vsky[3] = vec4(0.0, 0.0, 0.0, 1.0);
        vDir = mat3(inverse(Vsky)) * world.xyz;
        vec4 clip = ubo.proj * Vsky * vec4(world.xyz, 1.0);
        gl_Position = vec4(clip.xy, clip.w, clip.w);
    } else {
        vDir = mat3(inverse(V)) * world.xyz;
        gl_Position = ubo.proj * V * vec4(world.xyz, 1.0);
    }
}
```

Output:



Reflection: Working through this exercise is reinforcing that I need to make my code more modular. As this would make it easier to add features like multiple pipelines without having to rewrite large sections of code. I was able to complete this exercise without having to completely refactor, but for future projects I plan on investing time into building a more flexible rendering architecture from the start. This will help me avoid issues where different objects compete for the same pipeline settings.

EXERCISE 4:

Solution: To create the refractive effect, I modified the fragment shader to compute the refraction vector using GLSL's built-in refract function. The cube shader now calculates the view direction and normal in world space, then applies Snell's law to determine how light bends as it passes through the refractive material. I set the index of refraction (eta) to 1.33, simulating water. A blend factor based on the Fresnel equations was also implemented to mix the refracted color with a slight tint, enhancing realism. Finally, I enabled alpha blending in the pipeline to allow for transparency effects.

- Fragment Shader:

```
void main() {
    if (pc.unlit != 0u) {
        vec3 dir = normalize(vDir);
        vec3 col = texture(uSky, dir).rgb;
        outColor = vec4(pow(col, vec3(1.0 / 2.2)), 1.0);
        return;
    }

    vec3 N = normalize(vWorldNormal);
    vec3 V = normalize(ubo.eyePos.xyz - vWorldPos);

    vec3 I = -V;
    float eta = 1.0 / 1.33;
    vec3 T = refract(I, N, eta);

    vec3 col = texture(uSky, T).rgb;

    float F0 = 0.04;
    float F = F0 + (1.0 - F0) * pow(clamp(1.0 - dot(N, V), 0.0, 1.0),
5.0);
    col = mix(col * 0.8, col, F);

    col = pow(col, vec3(1.0 / 2.2));
    float alpha = 0.4;

    outColor = vec4(col, alpha);
}
```

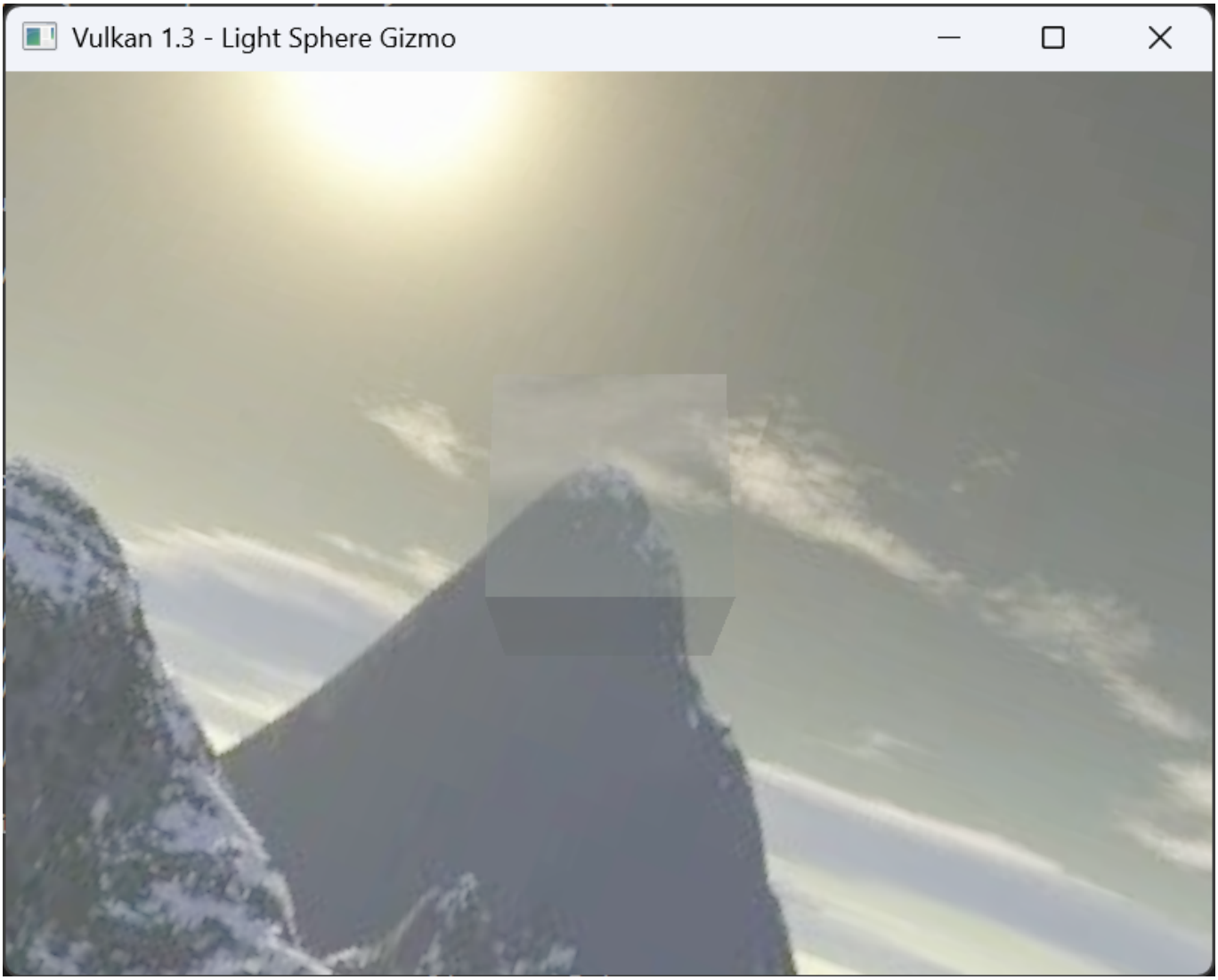
- Blend State:

```
VkPipelineColorBlendAttachmentState cba{};
cba.colorWriteMask =
    VK_COLOR_COMPONENT_R_BIT |
    VK_COLOR_COMPONENT_G_BIT |
    VK_COLOR_COMPONENT_B_BIT |
    VK_COLOR_COMPONENT_A_BIT;

cba.blendEnable         = VK_TRUE;
cba.srcColorBlendFactor = VK_SRC_ALPHA;
cba.dstColorBlendFactor = VK_ONE_MINUS_SRC_ALPHA;
cba.colorBlendOp        = VK_BLEND_OP_ADD;
cba.srcAlphaBlendFactor = VK_ONE;
cba.dstAlphaBlendFactor = VK_ONE_MINUS_SRC_ALPHA;
cba.alphaBlendOp        = VK_BLEND_OP_ADD;
```

- Refractive Pipeline Creation:

```
VkPipelineRasterizationStateCreateInfo rsRefract = rsCommon;
rsRefract.cullMode = VK_CULL_MODE_BACK_BIT;

VkPipelineDepthStencilStateCreateInfo dzRefract{
VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO };
dzRefract.depthTestEnable = USE_DEPTH ? VK_TRUE : VK_FALSE;
dzRefract.depthWriteEnable = VK_TRUE;
dzRefract.depthCompareOp = VK_COMPARE_OP_LESS_OR_EQUAL;

gps[1] = gps[0];
gps[1].pRasterizationState = &rsRefract;
gps[1].pDepthStencilState = &dzRefract;
gps[1].pColorBlendState = &cbRefract;

if (vkCreateGraphicsPipelines(device, VK_NULL_HANDLE, 1, &gps[1], nullptr,
&reflectPipeline) != VK_SUCCESS) {
    throw std::runtime_error("Failed to create refractive pipeline!");
}
```

Output:



Reflection: This exercise was particularly challenging as it required a solid understanding of both the mathematical principles behind refraction and the practical implementation in GLSL. Implementing Snell's law and the Fresnel equations in the shader code was a great learning experience, as it deepened my understanding of how light interacts with different materials. Enabling alpha blending also introduced me to the complexities of rendering transparent objects in Vulkan, which is not as straightforward as opaque rendering.

EXERCISE 5:

Solution: I implemented the fire effect by following the structure demonstrated in the lecture, using a set of quads that represent individual particles arranged on a small circular emitter positioned at the top of the cube. Each particle is assigned a unique identifier that drives a looping lifetime value through fract(), which allows the particles to rise smoothly before restarting at the base. The vertex shader uses this identifier to position the particles over time, adding height, outward drift, and a slight swirl, while also orienting each quad to face the camera using the inverse view matrix. The fragment shader shapes each quad into a soft circular glow, blends warm fire colours through the particle’s height, and adds a flickering effect using noise. A dedicated particle pipeline was configured with depth testing enabled but without depth writing, and with additive blending to create a bright, layered fire effect. This particle pass was then integrated into the main rendering sequence so it animates in harmony with the rest of the scene.

- Particle System Vertex Generation:

```
static void buildParticles() {
    particleVertices.clear();

    const int   particleCount = 200;
    const float emitterRadius = 0.25f;
    const float cubeTopY = 0.5f;

    const glm::vec2 corners[4] = {
        {-1.0f, -1.0f},
        { 1.0f, -1.0f},
        { 1.0f,  1.0f},
        {-1.0f,  1.0f}
    };

    for (int i = 0; i < particleCount; ++i) {
        float id = float(i) / float(particleCount);

        float seed = id;
        float angle = seed * glm::two_pi<float>();
        float r = emitterRadius * std::sqrt(seed);

        float x = r * std::cos(angle);
        float z = r * std::sin(angle);


        glm::vec3 basePos(x, cubeTopY, id);

        for (int c = 0; c < 4; ++c) {
            ParticleVertex v{};
            v.particlePos = basePos;
            v.corner = corners[c];
            particleVertices.push_back(v);
        }
    }
}
```

- Particle Vertex Shader:

```
#version 450

layout(location = 0) in vec3 inParticlePos;
layout(location = 1) in vec2 inCorner;

layout(location = 0) out vec2 texCoord;
layout(location = 1) out float t;

layout(std140, set = 0, binding = 0) uniform UBO {
    mat4 model;
```

```
    mat4 view;
    mat4 proj;
    vec4 lightPos;
    vec4 eyePos;
    float time;
    vec3 _pad;
} ubo;

#define particleSpeed         0.48
#define particleSpread        0.40
#define particleSize          0.30
#define particleSystemHeight  2.0

void main() {
    float id = inParticlePos.z;
    t = fract(id + particleSpeed * ubo.time);

    vec3 base = vec3(inParticlePos.x, 0.5, inParticlePos.y);

    float angle = 50.0 * id;
    float radius = particleSpread * (1.0 - t);

    vec3 center;
    center.x = base.x + radius * cos(angle);
    center.z = base.z + radius * sin(angle);

    center.y = base.y + t * particleSystemHeight;

    mat4 viewInv = inverse(ubo.view);
    vec3 right = viewInv[0].xyz;
    vec3 up    = viewInv[1].xyz;

    float sizeH = particleSize;
    float sizeV = particleSize * 1.8;

    vec3 worldPos =
        center +
        right * (inCorner.x * sizeH) +
        up    * (inCorner.y * sizeV);

    gl_Position = ubo.proj * ubo.view * vec4(worldPos, 1.0);

    texCoord = inCorner * 0.5 + 0.5;
}
```

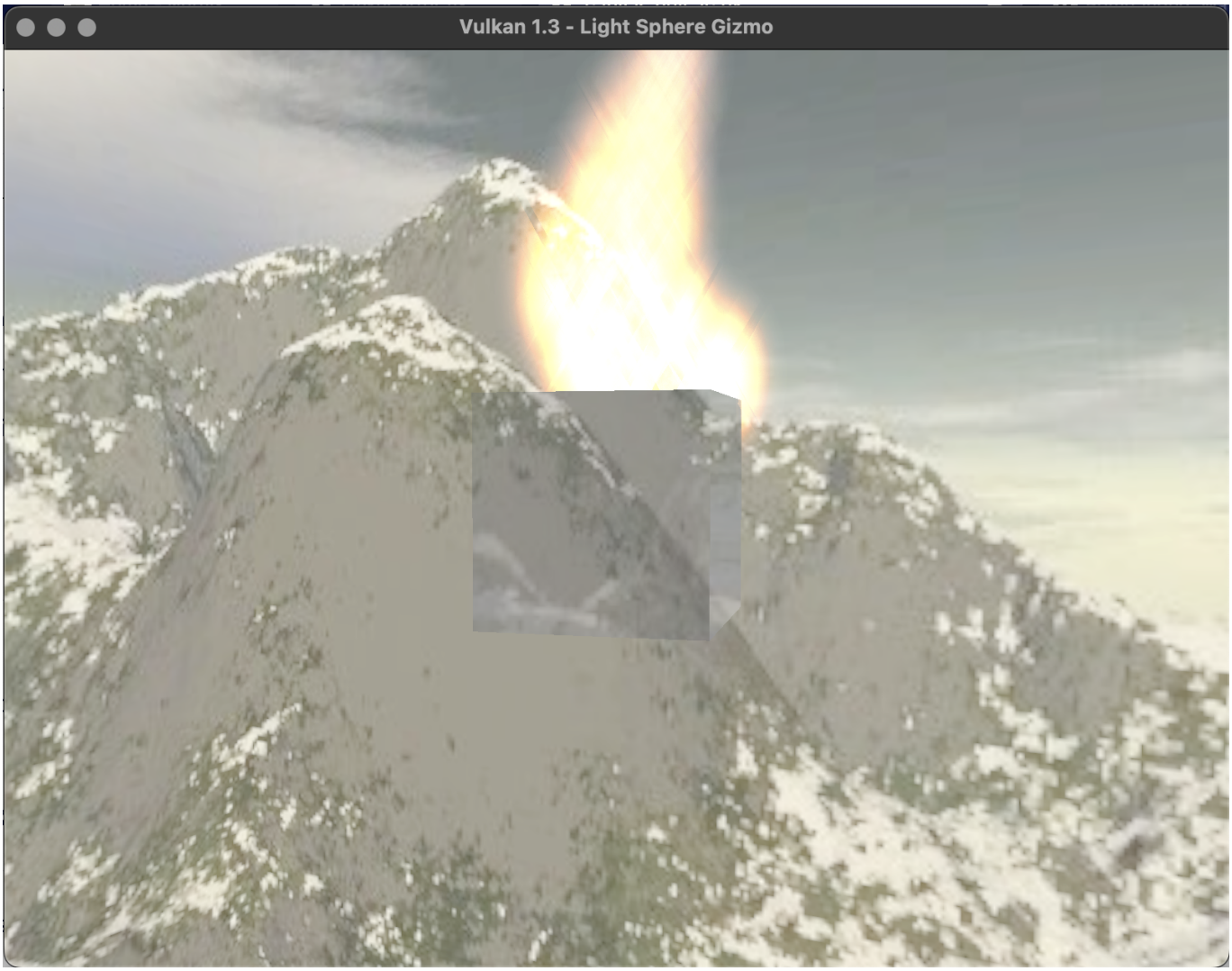
- Particle Fragment Shader:

```
#version 450

layout(location = 0) in vec2 texCoord;
layout(location = 1) in float t;
```

```
layout(location = 0) out vec4 outColor;

layout(std140, set = 0, binding = 0) uniform UBO {
    mat4 model;
    mat4 view;
    mat4 proj;
    vec4 lightPos;
    vec4 eyePos;
    float time;
    vec3 _pad;
} ubo;

float hash21(vec2 p) {
    p = fract(p * vec2(123.34, 345.45));
    p += dot(p, p + 34.345);
    return fract(p.x * p.y);
}

void main() {
    vec2 uv = texCoord * 2.0 - 1.0;
    float r = length(uv);

    float radial = smoothstep(1.0, 0.0, r);

    float h = clamp(t, 0.0, 1.0);
    float body = 1.0 - h;
    body = max(body, 0.15);

    float alpha = radial * body;

    float noise = hash21(texCoord * 12.3 + ubo.time * 3.7);
    float flicker = 0.8 + 0.2 * noise;
    alpha *= flicker;

    if (alpha < 0.02)
        discard;

    float heat = (1.0 - h) * radial;

    vec3 colBottom = vec3(1.0, 0.55, 0.10);
    vec3 colMid    = vec3(1.0, 0.85, 0.40);
    vec3 colTop    = vec3(0.25, 0.20, 0.20);

    vec3 color;
    if (heat > 0.6) {
        float k = (heat - 0.6) / 0.4;
        color = mix(colBottom, colMid, k);
    } else if (heat > 0.3) {
        float k = (heat - 0.3) / 0.3;
        color = mix(vec3(0.6, 0.15, 0.02), colBottom, k);
    } else {
        float k = heat / 0.3;
        color = mix(colTop, vec3(0.5, 0.2, 0.05), k);
    }
```

```
    color *= (0.6 + 0.4 * alpha);

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

Output:



Reflection: This exercise showed how a convincing fire effect can be created by combining simple geometry with shader-driven motion and colour control rather than relying on complex simulations. Using a looping lifetime value clarified how a continuous particle stream can be maintained without managing creation or removal, making the animation smooth and predictable. Implementing billboarding helped me understand how to keep particles visually oriented toward the viewer regardless of camera direction, and working with depth testing and additive blending revealed the importance of ordering and transparency for layered visual effects. Developing the colour transitions, fading behaviour, and flicker directly in the fragment shader highlighted how much visual richness can be achieved through procedural shading alone, resulting in a lively and coherent fire animation.