Vulkan Lab 7: Cube Mapping and Particle Systems (Vulkan 1.3)

1 Introduction

This lab introduces two powerful and essential techniques in modern 3D graphics: **Environment Mapping** using a cubemap and **Particle Systems**.

First, we will use **Cube Mapping** to implement a "skybox," creating the illusion that our scene exists within a vast, 360-degree environment. We will then leverage this same environment map to create realistic, dynamic reflections and refractions on our objects, making them appear metallic or glass-like.

Second, we will build a sprite-based **Particle System** to render dynamic special effects like fire or smoke. This will involve "billboarding" (making 2D sprites always face the camera) and **Alpha Blending** to handle transparency.

Both of these techniques require precise control over the graphics pipeline, specifically when and how objects interact with the depth buffer and how their colours are blended.

2 Learning Objectives

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

- **Master Depth Buffer Control:** Understand and control depthTestEnable and depthWriteEnable in the Vulkan pipeline.
- **Implement a Skybox:** Load 6 images corresponding to the 6 sides of a cube to build a cube texture object and render it as a seamless 3D background that appears infinitely distant.
- **Create Cube Map Textures:** Understand the Vulkan process for creating an VkImage and VkImageView with VK_IMAGE_VIEW_TYPE_CUBE.
- **Implement Reflections and Refractions:** Use GLSL's reflect and refract functions with a samplerCube to simulate mirrored and glass-like surfaces.
- **Understand Blending:** Configure a pipeline for additive or alpha blending to render transparent objects.
- **Implement Billboarding:** Use a vertex shader to make 2D quads (sprites) always face the camera.
- **Build a Particle System:** Manage the lifecycle of many small particles on the CPU and render them efficiently as a group of blended, billboarded sprites.

3 CORE CONCEPTS

- **Depth Buffer Control:** The depth buffer is a texture that stores the "depth" (distance from camera) of the closest pixel rendered so far.
  - depthTestEnable: If VK_TRUE, Vulkan checks the depth buffer before drawing a new pixel. If the new pixel is *behind* the existing one, it is discarded.
  - depthWriteEnable: If VK_TRUE, Vulkan *updates* the depth buffer with the new pixel's depth if it passes the test.
  - **Skyboxes:** Are drawn with depthTestEnable = VK_FALSE and depthWriteEnable = VK_FALSE, so they are always in the background and never block other objects.
  - **Transparent Particles:** Are drawn with test *enabled* (so they don't draw in front of solid walls) but write *disabled* (so a particle doesn't block other particles behind it).
- **Cube Maps:** A special type of texture consisting of 6 square 2D textures, one for each face of a cube (+X, -X, +Y, -Y, +Z, -Z). In Vulkan, this is a single VkImage with arrayLayers = 6 and the VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT flag. It is sampled in GLSL using a samplerCube and a 3D direction vector, not a 2D UV coordinate.
- **Blending:** Blending combines the colour of a fragment being rendered (the "source") with the colour already in the framebuffer (the "destination").
  - **Alpha Blending:** FinalColor = SrcAlpha * SrcColor + (1 - SrcAlpha) * DstColor. Used for standard transparency.
  - **Additive Blending:** FinalColor = SrcColor + DstColor. Used for effects like fire and magic, where light is being added to the scene.
- **Billboarding:** The process of orienting a 2D quad in 3D space so that its front face is always parallel to the camera's view plane. This is achieved in the vertex shader by applying the camera's rotation (but not its translation) to the particle's vertices.

4 LAB EXERCISES

EXERCISE 1: IMPLEMENTING AND CONTROLLING THE DEPTH BUFFER

This exercise is split into two parts. First, you will implement the depth buffer itself. Second, you will use it to observe how depth testing and writing are controlled.

1. **Goal:** To create a depth buffer and then practically observe the effect of depthTestEnable and depthWriteEnable.

2. **Implementation:**

Before you can control depth, you must create the depth buffer's resources. This involves creating a VkImage and VkImageView that will be used as the depth attachment.

C++ (Class Members): Add new Vulkan objects to your application class to manage the depth resources.

```
VkImage depthImage;
VkDeviceMemory depthImageMemory;
VkImageView depthImageView;
VkFormat depthFormat;
```

- **C++ (Implementation):**

  1. **Find Depth Format:** Create a helper function to find a suitable depth format supported by the physical device.

```
VkFormat HelloTriangleApplication::findSupportedFormat(const std::vector<VkFormat>& candidates,
VkImageTiling tiling, VkFormatFeatureFlags features) {
    for (VkFormat format : candidates) {
        VkFormatProperties props;
        vkGetPhysicalDeviceFormatProperties(physicalDevice, format, &props);
        if (tiling == VK_IMAGE_TILING_LINEAR && (props.linearTilingFeatures & features) ==
features) {
            return format;
        }
        else if (tiling == VK_IMAGE_TILING_OPTIMAL && (props.optimalTilingFeatures & features) ==
features) {
            return format;
        }
    }
    throw std::runtime_error("failed to find supported format!");
}


VkFormat HelloTriangleApplication::findDepthFormat() {

    return findSupportedFormat(
        { VK_FORMAT_D32_SFLOAT, VK_FORMAT_D32_SFLOAT_S8_UINT, VK_FORMAT_D24_UNORM_S8_UINT },
        VK_IMAGE_TILING_OPTIMAL,  VK_FORMAT_FEATURE_DEPTH_STENCIL_ATTACHMENT_BIT );
   }
```

  2. **Create Depth Resources:** Create a new function createDepthResources() that will be called during initialization (and swapchain recreation).

```
void HelloTriangleApplication::createDepthResources() {
    depthFormat = findDepthFormat();

    // Re-use your existing createImage helper function
    createImage(swapChainExtent.width, swapChainExtent.height, depthFormat,
        VK_IMAGE_TILING_OPTIMAL,
        VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT,
        VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT,
        depthImage, depthImageMemory);

    // Re-use your existing createImageView helper function
    depthImageView = createImageView(depthImage, depthFormat, VK_IMAGE_ASPECT_DEPTH_BIT);

}
```

3. **Update Pipeline:** In createGraphicsPipeline(), you must now configure the pipeline to *use* depth.

   - Describe VkPipelineDepthStencilStateCreateInfo:

```
VkPipelineDepthStencilStateCreateInfo depthStencil{};
depthStencil.sType = VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
depthStencil.depthTestEnable = VK_TRUE;
depthStencil.depthWriteEnable = VK_TRUE; // Enable writing
depthStencil.depthCompareOp = VK_COMPARE_OP_LESS; // Standard for perspective
depthStencil.depthBoundsTestEnable = VK_FALSE;
depthStencil.stencilTestEnable = VK_FALSE;
```

   - Update VkPipelineRenderingCreateInfo by adding information on depthAttachmentFormat:

```
renderingCreateInfo.depthAttachmentFormat = depthFormat;
```

   - Update `VkGraphicsPipelineCreateInfo`:

```
pipelineInfo.pNext = &renderingCreateInfo;
pipelineInfo.pDepthStencilState = &depthStencil;
```

4. **Update Rendering:** In recordCommandBuffer(), you must now attach the depth buffer to the rendering process.

   - Define the depth attachment info:

```
VkRenderingAttachmentInfo depthAttachment{};
depthAttachment.sType = VK_STRUCTURE_TYPE_RENDERING_ATTACHMENT_INFO;
depthAttachment.imageView = depthImageView;
depthAttachment.imageLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;
depthAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR; // Clear depth at start of frame
depthAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
depthAttachment.clearValue.depthStencil = { 1.0f, 0 };
```

   - In VkRenderingInfo, link this depth attachment:

```
VkRenderingInfo renderingInfo{};
renderingInfo.sType = VK_STRUCTURE_TYPE_RENDERING_INFO;
renderingInfo.renderArea = { {0, 0}, swapChainExtent };
renderingInfo.layerCount = 1;
renderingInfo.colorAttachmentCount = 1;
renderingInfo.pColorAttachments = &colorAttachment;
renderingInfo.pDepthAttachment = &depthAttachment;
```

```
//… … …
vkCmdBeginRendering(commandBuffer, &renderingInfo);
```

5. **Recreation:** Remember to call vkDestroyImageView, vkDestroyImage, and vkFreeMemory for the depth resources in cleanupSwapChain() and call createDepthResources() in recreateSwapChain().

3. Expected Outcome:

Now you have a functioning depth buffer, you can experiment with its settings further. Draw a solid cube and observe how the depth setting may affect the visual result.

EXERCISE 2: IMPLEMENTING THE SKYBOX

1. **Goal:** Render a 6-sided cube map as a non-interactive background.
2. **Implementation:**
   - **C++ (Assets):** Load 6 separate images for the skybox faces (front, back, top, bottom, left, right).
   - **C++ (Vulkan):**
     - In createImage () used in Lab 6, set imageType = VK_IMAGE_TYPE_2D, arrayLayers = 6, and flags = VK_IMAGE_CREATE_CUBE_COMPATIBLE_BIT.
     - Create an array of VkImage from the 6 images. The order of the six images in the array should be as follows:

Layer Index	Cube Face	Direction
0	+X	Right
1	–X	Left
2	+Y	Top
3	–Y	Bottom
4	+Z	Front
5	–Z	Back

     - Create a VkImageView. Set viewType = VK_IMAGE_VIEW_TYPE_CUBE and layerCount = 6.
     - Create a VkSampler.
   - **C++ (Pipeline):**
     - Create a new skyboxPipeline.

```
VkPipeline skyboxPipeline = VK_NULL_HANDLE;
```

- Write a createSkyboxPipeline() like creating the original createGraphicsPipeline() method. In the method, describe the VkPipelineDepthStencilStateCreateInfo in the following way by specifying depthTestEnable = VK_TRUE, depthWriteEnable = VK_FALSE.

  We want the skybox to be "at" the far plane, so we use VK_COMPARE_OP_LESS_OR_EQUAL.

```
VkPipelineDepthStencilStateCreateInfo depthStencil{};
 depthStencil.sType =
    VK_STRUCTURE_TYPE_PIPELINE_DEPTH_STENCIL_STATE_CREATE_INFO;
 depthStencil.depthTestEnable = VK_TRUE;
depthStencil.depthWriteEnable = VK_FALSE;
depthStencil.depthCompareOp = VK_COMPARE_OP_LESS_OR_EQUAL;
```

- Configure rasterization state VkPipelineRasterizationStateCreateInfo:

  cullMode = VK_CULL_MODE_FRONT_BIT.

  Image we are *inside* the cube, so we must cull the front faces to see the back faces.

○ **Shaders (GLSL):**

- skybox.vert: The goal is to make the cube seem infinitely far and always centered on the camera (please read the lecture note for the detailed description of the principle).

```
 layout(binding = 0) uniform UniformBufferObject {
     mat4 model;
     mat4 view;
     mat4 proj;
     vec3 eyePos;
 } ubo;


 layout(location = 0) in vec3 inPosition;
 … …

layout(location = 1) out vec3 viewDir;


 void main() {
      viewDir = inPosition;
      vec3 wPos=inPosition+ubo.eyePos;
      gl_Position = ubo.proj * ubo.view * vec4(inPosition, 1.0);
 }
```

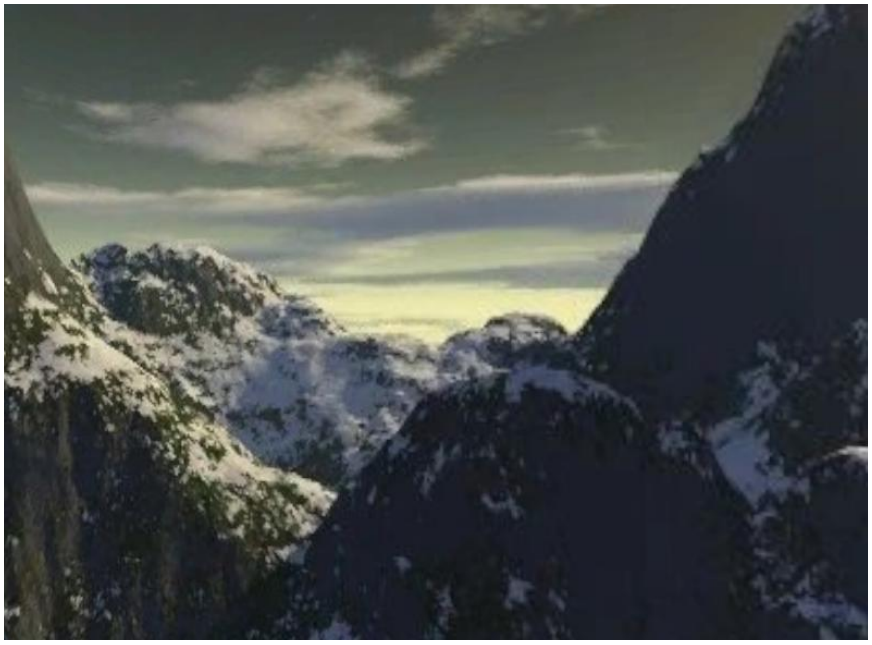
- skybox.frag:

```
layout(location = 0) in vec3 viewDir;
layout(binding = 1) uniform samplerCube skySampler;
layout(location = 0) out vec4 outColor;

void main() {
    // Sample the cubemap using the vertex's position as a 3D direction
    outColor = texture(skySampler, fragTexCoord);
}
```

- **C++ (Rendering):** In your render loop, draw the skybox **first**, before any solid objects. This allows its depth (1.0) to fill the depth buffer, so any objects in front of it will be drawn correctly.

3. **Expected Outcome:** Your scene is now surrounded by a 360-degree sky. When you rotate the camera, the skybox rotates with you, but it appears to be infinitely far away. The following image is the view at the eye position at (2, 0, 5).



Exercise 3: Implementing Reflections

1. **Goal:** Make one of your solid objects (e.g., a sphere or cube) perfectly reflective, like a mirror.
2. **Implementation:**
   - **C++:** Ensure your main object's descriptor set also includes the skySampler from Exercise 2 (at a different binding).
   - **Shaders (GLSL):** Modify your *main solid object's* fragment shader.
     - Add layout(binding = 1) uniform samplerCube skySampler;.
     - Instead of calculating lighting, calculate the reflection vector.

```
// ... in variables ...
layout(location = 1) in vec3 inWorldPos;
layout(location = 2) in vec3 inWorldNormal;

// ... UBO ...
vec3 viewPos;

void main() {
    vec3 N = normalize(inWorldNormal);
    vec3 V = normalize(ubo.viewPos - inWorldPos); // View vector

    // Calculate the reflection vector
    vec3 R = reflect(-V, N); // reflect() expects incident vector

    // Sample the skybox with the reflection vector
    vec3 reflectionColor = texture(skySampler, R).rgb;

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

- **C++ (Rendering):** Draw this object normally with your pipelineSolid.

3. **Expected Outcome:** The object will act as a perfect mirror, reflecting the skybox. As you move the camera, the reflection will change realistically.

EXERCISE 4: IMPLEMENTING REFRACTION

1. **Goal:** Simulate a transparent, refractive object (like glass or water).
2. **Implementation:**
    - **Shaders (GLSL):** Modify the fragment shader from Exercise 3.
        - Instead of reflect, use refract.

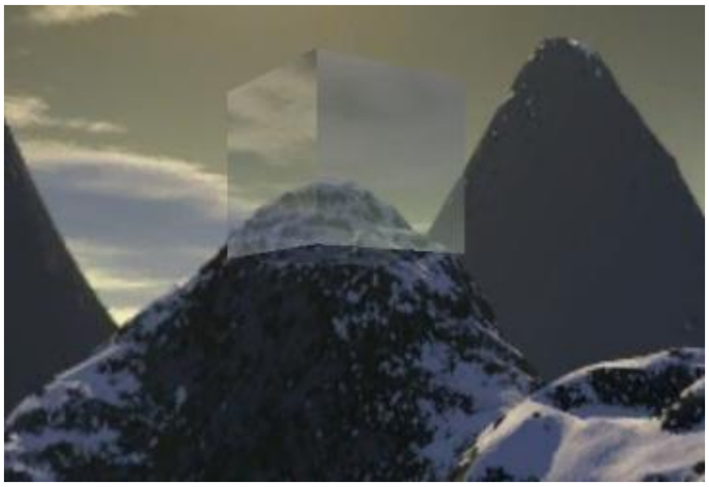
```
// ... same as Exercise 3 ...
void main() {
    vec3 N = normalize(inWorldNormal);
    vec3 V = normalize(ubo.viewPos - inWorldPos);

    float IOR = 1.00 / 1.33; // Index of Refraction (air to water)

    // Calculate the refraction vector
    vec3 R = refract(-V, N, IOR);

    vec3 refractionColor = texture(skySampler, R).rgb;
    outColor = vec4(refractionColor, 1.0);
}
```

3. **Expected Outcome:** The object now appears transparent, and the skybox seen *through* it is distorted, as if looking through a lens or a block of glass.



EXERCISE 5: SPRITE-BASED PARTICLE SYSTEM

1. **Goal:** Create a simple fire or smoke effect using billboards and blending.
2. **Implementation:**
   - **C++ (Data):** Create a sequence of quads along the z-axis with identical size of [-1, 1]x[-1, 1] for xy-dimensions and an increased z-coordinate values varying from, say, 0 to 1. The value of z-coordinate can be used to identify the position of each particle in the vertex shader.
   - **C++ (Time):** Pass time to vertex shader as a uniform variable.
   - **C++ (Pipeline):** Create a particlePipeline and configure depth testing and colour blending.
     - VkPipelineDepthStencilStateCreateInfo: depthTestEnable = VK_TRUE, depthWriteEnable = VK_FALSE.

        - VkPipelineColorBlendAttachmentState: blendEnable = VK_TRUE, srcColorBlendFactor = VK_BLEND_FACTOR_SRC_ALPHA, dstColorBlendFactor = VK_BLEND_FACTOR_ONE (for additive fire) or VK_BLEND_FACTOR_ONE_MINUS_SRC_ALPHA (for smokey transparency).
        - VkPipelineRasterizationStateCreateInfo: cullMode = VK_CULL_MODE_NONE_BIT.
    - **Shaders (GLSL):**
        - particle.vert: Implement billboarding. The vertex inPosition will be the particle's center, and inTexCoord can store the quad's corner offset (e.g., (-1, -1), (1, 1)).

```
layout(location = 0) in vec3 inParticlePos;
… … …

layout(location = 0) out vec2 texCoord;
layout(location = 1) out t;
… … … …
// ... out ...

# define particleSpeed 0.48
#define  particleSpread 20.48
#define  particleShape 0.37
#define  particleSize 6.0
#define particleSystemHeight 60.0

void main() {
  // slice the time and loop particles
  float t = fract(inParticlePos.z + particleSpeed * time);
  //reposition the quads based on their z-coordinate:
  vec3 pos;
  // Spread particles in a semi-random fashion
  pos.x = particleSpread * t * cos(50.0 * inParticlePos.z);
```

```
        pos.z = particleSpread * t * sin(120.0 * inParticlePos.z);

      // Find the inverse of the view matrix
      mat4 viewInv = inverse(view);
      //align quad orientation with view image plane (billboarding)
      vec3 BBPos =(pos.x * viewInv[0] + pos.y * viewInv[1]).xyz;
      //resize the quad
     pos += particleSize * BBPos;
    gl_Position = ubo.proj * ubo.view * vec4(worldPos, 1.0);
    texCoord = inParticlePos.xy;
     … … …
}
```

- particle.frag: Define the pixel colour at each location based on texCoord and fade the colour with time:

```
… … …
layout(location = 0) in vec2 texCoord;
layout(location = 1) in t;
layout(location = 0) out vec4 outColor;
… … …

void main() {
   outColor = … … … ;
}
```

3. **Expected Outcome:** You will see a cloud of 2D sprites (e.g., 50-100) that look 3D, move according to your physics, and blend correctly with each other and the solid scene.

5 Further Exploration (Optional)

- **Fresnel Effect:** Combine Exercises 3 and 4. Use the dot product of the view vector and normal to blend *between* the refraction colour and the reflection colour. This is called the Fresnel effect, and it realistically simulates that objects are more reflective at grazing angles.