Vulkan Lab 8: Post-Processing Effects (Vulkan 1.3)

1 INTRODUCTION

Welcome to the next major step in advanced rendering. So far, we have rendered all of our scenes directly to the **swapchain** (the images presented to the screen). This lab introduces **Post-Processing**, a powerful technique where we first render our entire 3D scene to an intermediate, off-screen texture.

Once the scene is rendered to a texture, we can apply 2D image-processing effects before it appears on the screen. This is achieved by drawing a full-screen quad textured with the rendered scene. In the quad's fragment shader, we can implement effects such as blur, colour tinting, edge detection, or the glow (bloom) effect featured in this lab. This "render-to-texture" workflow is the foundation for almost all modern visual effects.

2 LEARNING OBJECTIVES

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

- **Implement "Render-to-Texture" (RTT):** Create and configure a VkImage that can be used as both a colour attachment (to be rendered to) and a sampled texture (to be read from).
- **Manage Multiple Render Passes:** Structure your command buffer to perform two sequential render passes: one to an off-screen texture and one to the swapchain.
- **Use Image Barriers:** Correctly use VkImageMemoryBarrier to transition your off-screen texture between COLOR_ATTACHMENT_OPTIMAL and SHADER_READ_ONLY_OPTIMAL layouts.
- **Re-use Rendered Textures:** Bind the output of the first render pass as an input texture to the second render pass.
- **Implement a Blur Shader:** Write a simple 2D convolution (blur) filter in a fragment shader.
- **Create a Glow (Bloom) Effect:** Combine the RTT and blur techniques to create a simple but effective glow effect by additively blending a blurred version of the scene back onto itself.

3 CORE CONCEPTS

- **Render-to-Texture (RTT):** The core idea is to change our render target. Instead of pointing our VkRenderingInfo's color attachment to a swapchain VkImageView, we point it to a *new* VkImageView that we create ourselves. This new "off-screen" image must be created with two key

usage flags: VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT (so we can render to it) and VK_IMAGE_USAGE_SAMPLED_BIT (so we can read from it in a shader).

- **Multi-Pass Rendering:** We will structure our command buffer to perform two distinct passes:
    - **Pass 1 (Scene Pass):** Renders the 3D scene to our off-screen texture.
    - **Pass 2 (Post-Process Pass):** This pass renders the scene to the screen using the texture created in the first pass. This pass typically just draws a single, screen-sized quad. The vertex shader for this quad is minimal; its only job is to draw the quad and provide texture coordinates. The fragment shader does all the work, sampling from the texture we just rendered our scene into. In this pass, the fragment shader will sample from the off-screen texture (which now holds an image of the scene) and apply it to a scene object, for example, the cube's surface.
- **Image Layout Transitions (Barriers):** This is the most critical concept in this lab. A VkImage cannot be used as a render target and a sampled texture at the same time. We must tell Vulkan to change its "layout" (its internal memory format) between our render passes.
    1. **Scene Pass:** Before rendering the 3D scene, transition the off-screen image from VK_IMAGE_LAYOUT_UNDEFINED to VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL.
    2. **Barrier:** After the scene pass, we must insert a **pipeline barrier**. This barrier transitions the image layout from VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL to VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL. This barrier ensures that the first pass (writing) is completely finished before the second pass (reading) begins.
    3. **Main Pass:** The image is now safe to be used as a sampler2D in the second pass.
- **Convolution Filter (Blur):** A blur is a "convolution," which means each pixel's final colour is a weighted average of itself and its neighbours. A simple **box blur** involves sampling the 9 pixels in a 3x3 grid (the centre pixel and its 8 neighbours) and averaging their colours.

4 LAB EXERCISES

EXERCISE 1: RENDER-TO-TEXTURE ON A CUBE

1. **Goal:** Render your 3D cube (with vertex colors) to an off-screen texture. Then, in a second pass, render the *same 3D cube* to the screen, using the off-screen texture as its new surface texture.
2. **Implementation:**
    - **C++ (Create Off-screen Resources):**
        - Create a new VkImage (offscreenImage), VkDeviceMemory (offscreenImageMemory), and VkImageView (offscreenImageView).

- When creating offscreenImage, use your swapchain's format and extent, and set usage flags: VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_SAMPLED_BIT.
- Create a new VkSampler (offscreenSampler).

- **C++ (Create Post-Process Pipeline):**
  - Your existing graphicsPipeline (e.g., from you have a achieved in Lab 5 on texture mapping) will be used for **Pass 1** (rendering the cube object to the texture).
  - Create a new VkDescriptorSetLayout. It needs **two** bindings:
    - binding = 0: VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER (for the UBO)
    - binding = 1: VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER (for the offscreenImage)
  - Create a new VkPipeline: VkPipelineLayout, and VkPipeline. This pipeline will be almost identical to the existing graphicsPipeline, but it will use the new VkPipelineLayout and new shaders.
- **C++ (Create Descriptors):**
  - Create a VkDescriptorPool and VkDescriptorSet (texturedCubeDescriptorSet) for the new pipeline.
  - Update this descriptor set to bind both the uniformBuffers[i] (at binding 0) and the offscreenSampler/offscreenImageView (at binding 1).
- **GLSL (Shaders for Pass 1 - No Change):**
  - vert.spv (from your graphicsPipeline).
  - frag.spv (from your graphicsPipeline).
- **GLSL (Shaders for Pass 2 - Modified):**
  - textured_cube.vert: This will be your *main* vertex shader, vert.spv, modified to pass texture coordinates.

```
#version 450
layout(binding = 0) uniform UniformBufferObject {
  mat4 model;
  mat4 view;
  mat4 proj;
} ubo;

layout(location = 0) in vec3 inPosition;
... ... ...
```

```
layout(location = 1) out vec2 fragTexCoord; // New

void main() {
  ... ... ...
  fragTexCoord = inTexCoord; // Pass texcoord
}
```

- texture_map.frag: This is the new fragment shader for your textured cube using the texture created in the first pass.

```
#version 450
... ...
layout(location = 1) in vec2 fragTexCoord;

layout(binding = 1) uniform sampler2D sceneTexture; // From Pass 1

layout(location = 0) out vec4 outColor;

void main() {
    outColor = texture(sceneTexture, fragTexCoord);
}
```

- **C++ (Update recordCommandBuffer):**

1) **Pass 1: Render Scene to Texture**
   - Insert barrier to transition offscreenImage from UNDEFINED to COLOR_ATTACHMENT.
   - Create VkRenderingAttachmentInfo (colorAttachmentInfo) pointing to offscreenImageView.
   - Create VkRenderingInfo (sceneRenderingInfo) using colorAttachmentInfo and your depthAttachmentInfo.
   - vkCmdBeginRendering(commandBuffer, &sceneRenderingInfo);
   - Bind your *original 3D scene pipeline* and descriptor sets.
   - Bind vertex and index buffers.
   - vkCmdDrawIndexed(...);
   - vkCmdEndRendering(commandBuffer);
2) **Pass 1 -> 2 Barrier**

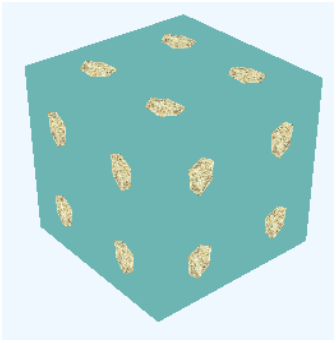
- Insert barrier to transition offscreenImage from COLOR_ATTACHMENT to SHADER_READ_ONLY. This is mandatory.

3) **Pass 2: Render Textured Cube to Screen**
   - Insert barrier to transition *swapchain image* from UNDEFINED to COLOR_ATTACHMENT.
   - Create VkRenderingAttachmentInfo (swapchainAttachmentInfo) pointing to the *current swapchain image view*.
   - Create VkRenderingInfo (finalRenderingInfo) using swapchainAttachmentInfo and your depthAttachmentInfo (clear the depth buffer again).
   - vkCmdBeginRendering(commandBuffer, &finalRenderingInfo);
   - Bind your new texturedCubePipeline and texturedCubeDescriptorSet.
   - Bind vertex and index buffers *again*.
   - vkCmdDrawIndexed(...);
   - vkCmdEndRendering(commandBuffer);
4) **Final Barrier:** Transition *swapchain image* to PRESENT_SRC_KHR.

- **C++ (Cleanup):** Remember to destroy the new texturedCubePipeline, texturedCubeLayout, and related descriptor resources.

3. **Expected Outcome:** You will see a textured cube. The texture on the cube will be a "snapshot" of the vertex-colored cube rendered in Pass 1. If the cube is rotating, the texture will appear to "swim" or "slide" across the surface in a disorienting but correct way. If you stop the rotation, the texture on the cube will be a static image of itself.

Screen image



Screen image used a texture

EXERCISE 2: TEXTURE SMOOTHING (BOX BLUR)

1. **Goal:** Apply a simple image smoothing algorithm to blur the texture generated in Pass 1 *before* it is rendered to the screen-aligned quad.

2. **Implementation:**

   - **New Shaders:**

     - fullscreen.vert: Write a new vertex shader to draw a screen-aligned quad.
     - blur.frag: Write a fragment shader to smooth the texture generated in the first pass.

```
#version 450
layout(location = 0) in vec2 fragTexCoord;
layout(binding = 0) uniform sampler2D sceneTexture; // Input is offscreenImage
layout(location = 0) out vec4 outColor;

void main() {
   float stepSize = 3.0;  //stepSize = 1.0, 2.0, ...
   vec2 texelSize = stepSize / textureSize(sceneTexture, 0);
   vec4 result = vec4(0.0);
   int boxSize = ... ; // boxSize = 1, 2, ...
   for (int x = - boxSize; x <= boxSize; x++) {
      for (int y = - boxSize; y <= boxSize; y++) {
         result += texture(sceneTexture, fragTexCoord + vec2(x, y) * texelSize);
      }
   }
   outColor = result /( boxSize * boxSize +1);
}
```

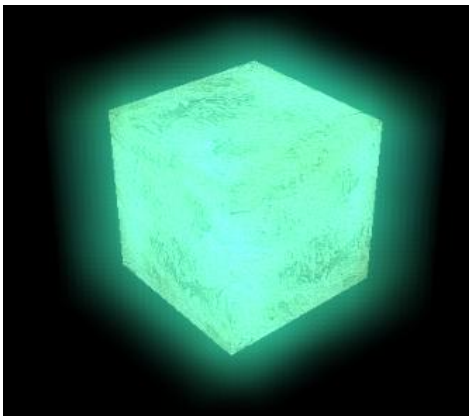
3. **Expected Outcome:** The cube is rendered to the screen. Its texture is now a *blurred* version of its vertex-colored self.

EXERCISE 3: SIMPLE GLOW EFFECT

1. **Goal:** Create a "glow" by additively blending the blurred scene back onto the original, sharp scene.
2. **Implementation:**

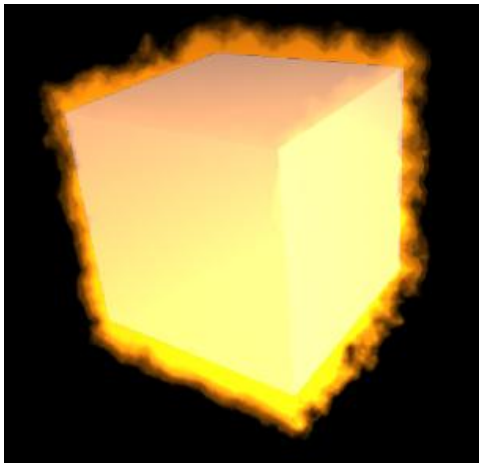
   - Revise your Exercise 2 fragment shader used in the second pass so that it blends the original texture generated from the first pass with the smoothed version of the texture.

3. **Expected Outcome:** A "dreamy" glow effect. The sharp cube will be visible, additively blended with a full-screen blurred version of itself.



EXERCISE 4: OBJECT ON FIRE ANIMATION

1. **Goal:** Modify the glow effect to create a "fire" aura.
2. **Implementation:**
    - Use the same setup as Exercise 2
        - Capture the target scene object that you want to appear as burning.
        - Apply a blur effect to the captured texture.
        - Introduce a timer to animate the blurred texture, creating a dynamic fire-like behaviour.
        - Blend the processed (blurred) texture with the original texture, and render the result onto a screen-aligned quad.
3. **Expected Outcome:** The cube will have a bright, fiery orange/yellow glow, making it look like it's superheated or on fire.



5 FURTHER EXPLORATION (OPTIONAL)

- **True Bloom (Thresholding):** The current glow effect applies to the entire scene, making everything glow. In contrast, true bloom targets only the brightest areas. Enhance the effect you created in Exercise 3 to make it more visually realistic.