Vulkan Lab 1 - Simple Shapes

Week 2 - Lab A

EXERCISE 1: DRAW TWO TRIANGLES WITHOUT USING VERTEX INDICES

Goal: Render two distinct triangles instead of one quad using vkCmdDraw().

Solution:

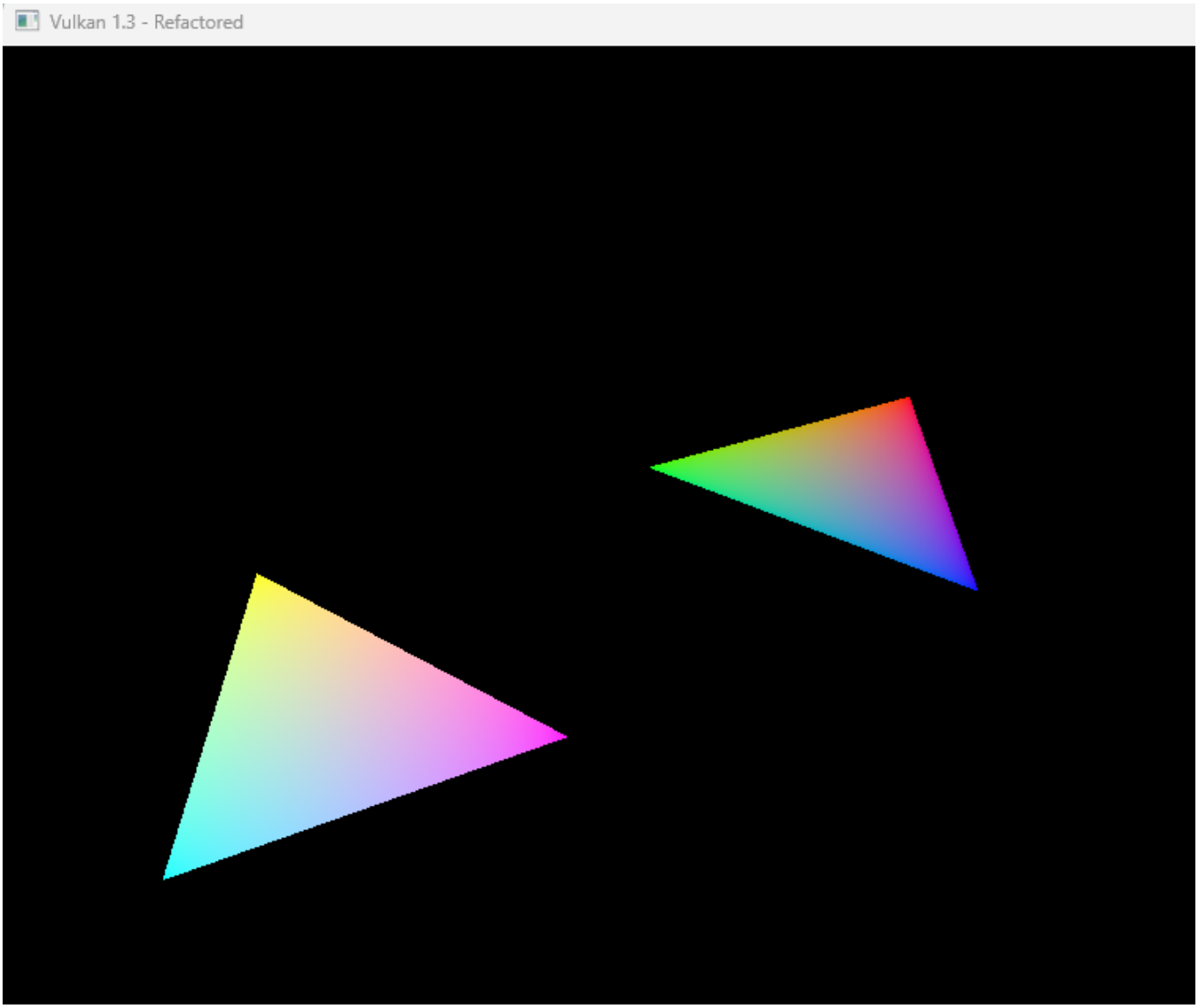
```
void loadModel() {
    // Unique vertex data for two triangles.
    vertices = {
        // First triangle (shifted left by -0.6 on X)
        {{-0.5f - 0.6f, -0.5f, 0.0f}, {1.0f, 0.0f, 0.0f}},
        {{0.5f - 0.6f, -0.5f, 0.0f},  {0.0f, 1.0f, 0.0f}},
        {{0.0f - 0.6f,  0.5f, 0.0f},  {0.0f, 0.0f, 1.0f}},

        // Second triangle (shifted right by +0.6 on X)
        {{0.5f + 0.6f, -0.5f, 0.0f},  {1.0f, 1.0f, 0.0f}},
        {{0.9f + 0.6f,  0.5f, 0.0f},  {0.0f, 1.0f, 1.0f}},
        {{0.0f + 0.6f,  0.5f, 0.0f},  {1.0f, 0.0f, 1.0f}},
    };


    // indices cleared as the to triangle will have s
    indices.clear();
}
```

```
void HelloTriangleApplication::initVulkan() {
    createInstance();
    setupDebugMessenger();
    createSurface();
    pickPhysicalDevice();
    createLogicalDevice();
    createSwapChain();
    createImageViews();
    createDescriptorSetLayout();
    createGraphicsPipeline();
    createCommandPool();

    loadModel();

    createVertexBuffer();
    //createIndexBuffer();  I don't need index buffer because the triangles are
not sharing vertices.
    createUniformBuffers();
    createDescriptorPool();
```

```
    createDescriptorSets();
    createCommandBuffers();
    createSyncObjects();
}
```

```
void HelloTriangleApplication::cleanup() {
    cleanupSwapChain();

    vkDestroyPipeline(device, graphicsPipeline, nullptr);
    vkDestroyPipelineLayout(device, pipelineLayout, nullptr);
    vkDestroyDescriptorSetLayout(device, descriptorSetLayout, nullptr);

    /*
    This whole section is commented out because it deals with handling the memory
for vertex and index buffers
    which I am no longer using.

    vkDestroyBuffer(device, indexBuffer, nullptr);
    vkFreeMemory(device, indexBufferMemory, nullptr);

    vkDestroyBuffer(device, vertexBuffer, nullptr);
    vkFreeMemory(device, vertexBufferMemory, nullptr);
    */
```

```
 VkBuffer vertexBuffers[] = { vertexBuffer };
    VkDeviceSize offsets[] = { 0 };
    vkCmdBindVertexBuffers(commandBuffer, 0, 1, vertexBuffers, offsets);
    /*
    * This is the function call to bind the index buffer, which I am no longer
using.
    vkCmdBindIndexBuffer(commandBuffer, indexBuffer, 0, VK_INDEX_TYPE_UINT16);
    */
    vkCmdBindDescriptorSets(commandBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS,
pipelineLayout, 0, 1, &descriptorSets[currentFrame], 0, nullptr);
    /*
    * I'm no longer using indexed drawing.
    vkCmdDrawIndexed(commandBuffer, static_cast<uint16_t>(indices.size()), 1, 0,
0, 0);
    */
    vkCmdDraw(commandBuffer, 6, 1, 0, 0); // 6 for the unique vertex count
    vkCmdEndRendering(commandBuffer);
```



Reflection: After designing two distinct triangles with custom vertex data, I ensured they were clearly visible and did not overlap on the screen. This initial step helped me understand how to define and manage vertex positions manually.

Next, I systematically removed all references to the index buffer throughout the codebase. This included eliminating the createIndexBuffer() function call from initVulkan() and cleaning up any related logic that previously relied on indexed drawing.

With the index buffer removed, I updated the drawFrame() function to use vkCmdDraw() instead of vkCmdDrawIndexed(). Since indexed drawing was no longer applicable, I specified the vertex count directly in the draw call, aligning it with the number of vertices I had defined. This exercise was a valuable learning experience. It not only taught me how to create and manage custom vertex data but also gave me deeper insight into the structure of Vulkan applications. By removing index buffer usage, I was able to explore various parts of the Vulkan pipeline and understand how different components interact when rendering geometry.

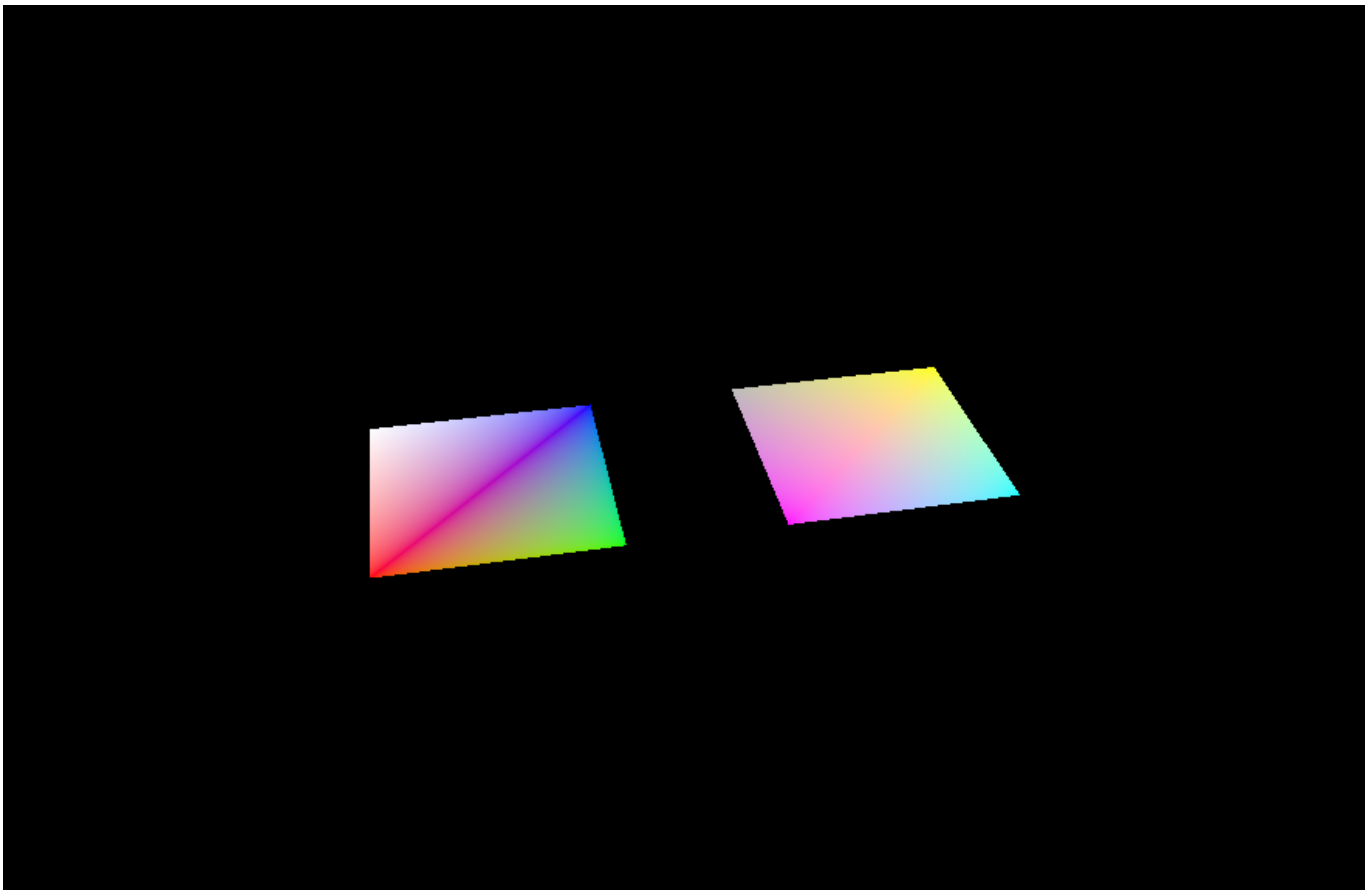
EXERCISE 2: DRAW TWO SQUARES USING AN INDEX BUFFER

Goal: Draw two squares (each composed of two triangles) using an index buffer to reuse vertices

Solution:

```
const std::vector<Vertex> twoSquare_vertices = {
    {{-0.8f,-0.5f,0},{1,0,0}}, {{-0.2f,-0.5f,0},{0,1,0}},
    {{-0.2f, 0.1f,0},{0,0,1}}, {{-0.8f, 0.1f,0},{1,1,1}},
    {{ 0.2f,-0.5f,0},{1,0,1}}, {{ 0.8f,-0.5f,0},{0,1,1}},
    {{ 0.8f, 0.1f,0},{1,1,0}}, {{ 0.2f, 0.1f,0},{0.5f,0.5f,0.5f}},
};

const std::vector<uint16_t> twoSquares_indices = {
    0,1,2, 2,3,0,
    4,5,6, 6,7,4
};
```

```
void loadModel() {
    vertices = twoSquare_vertices;
    indices = twoSquares_indices;
}
```



Reflection: After I had been exploring vulkan with the first exercise, this one was very straightforward, I just had to uncomment the index buffer I had removed to get the code using the index again.

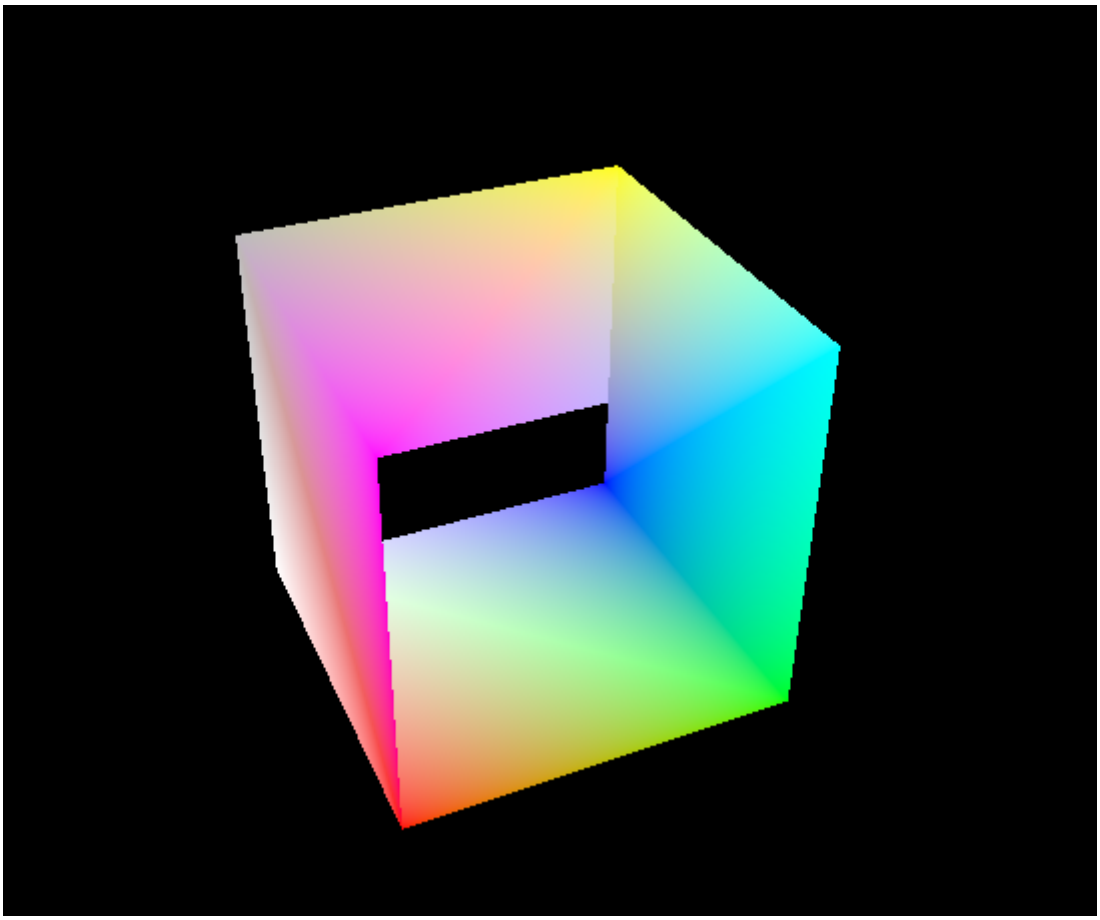
EXERCISE 3: DRAW THE FOUR WALLS OF A CUBE

Goal: Extend the previous exercise to draw the four side faces of a cube.

Solution:

```
const std::vector<Vertex> cube_vertices = {
    // back (-Z)
    {{-0.5f, -0.5f, -0.5f}, {1,0,0}}, // 0
    {{ 0.5f, -0.5f, -0.5f}, {0,1,0}}, // 1
    {{ 0.5f,  0.5f, -0.5f}, {0,0,1}}, // 2
    {{-0.5f,  0.5f, -0.5f}, {1,1,1}}, // 3

    // front (+Z)
    {{-0.5f, -0.5f,  0.5f}, {1,0,1}}, // 4
    {{ 0.5f, -0.5f,  0.5f}, {0,1,1}}, // 5
    {{ 0.5f,  0.5f,  0.5f}, {1,1,0}}, // 6
    {{-0.5f,  0.5f,  0.5f}, {0.5,0.5,0.5}}, // 7
};

// In this format because it helps me visualize the triangles
const std::vector<uint16_t> cube_indices = {
    // front (+Z)
    4,5,6,  6,7,4,
    // right (+X)
    5,1,2,  2,6,5,
    // back (-Z)
    1,0,3,  3,2,1,
    // left (-X)
    0,4,7,  7,3,0,
};
```

```
    rasterizer.cullMode = VK_CULL_MODE_NONE;
```



Reflection: I was able to draw the four walls of a cube by defining the appropriate vertices and indices. I attempted to asjust the view matrix to get a better perspective of the cube, but I was unable to get it looking it down into the way I wanted. I think I need to understand better how the view matrix works. Even though I was able to set the cube, I still feel like i could have a better understanding of visualising 3D objects in space.

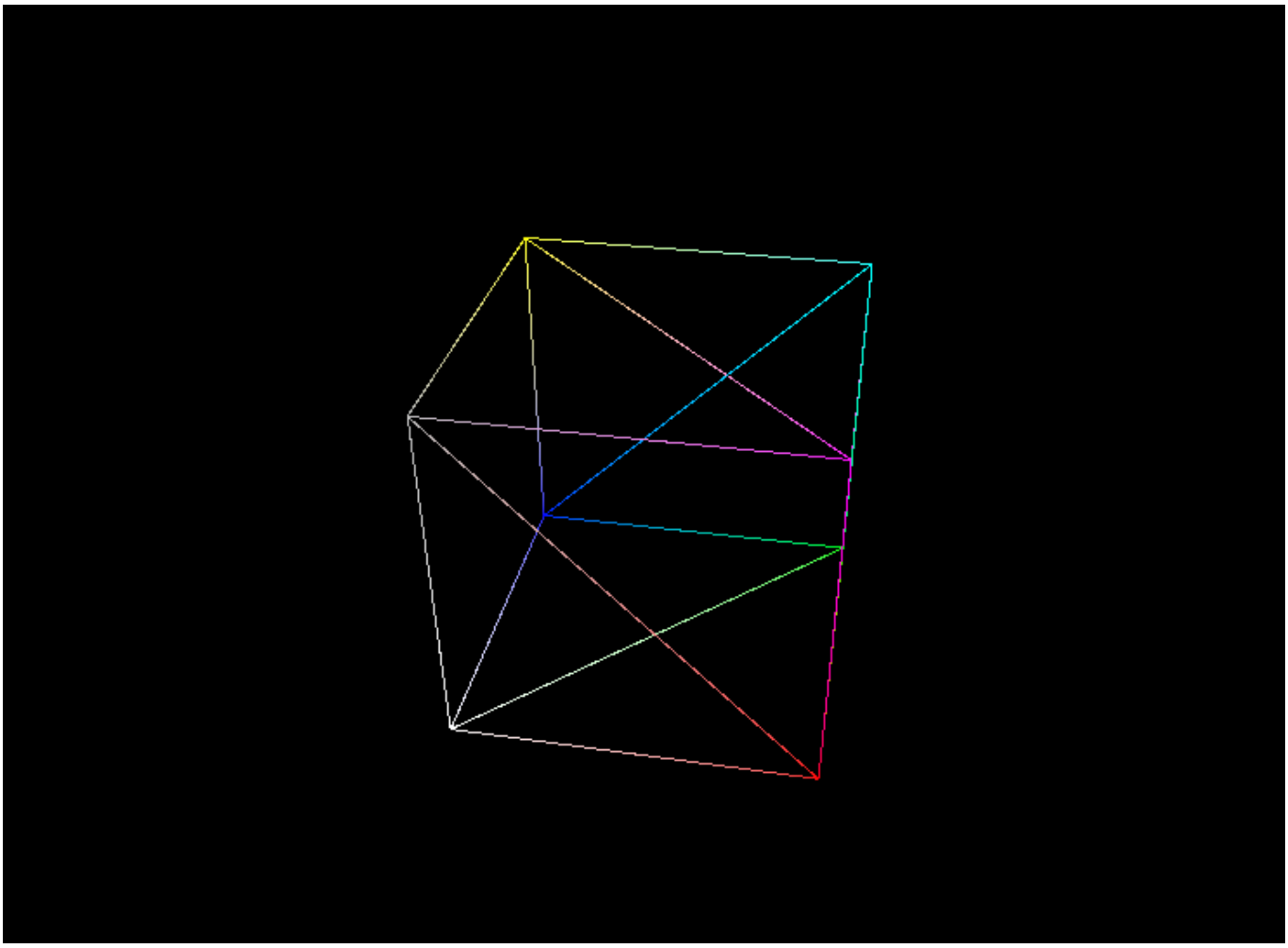
It would be helpful to get some help on how to position the camera to get a better view of the cube.

EXERCISE 4: WIREFRAME RENDERING

Goal: Render the cube from the previous exercise as a wireframe, showing only its edges.

Solution:

```
rasterizer.polygonMode = VK_POLYGON_MODE_LINE;
```



Reflection: This exercise was very straightforward, I just had to change the polygon mode to line. But seeing the cube in wireframe mode helped me understand better how the cube is constructed from its vertices and edges.

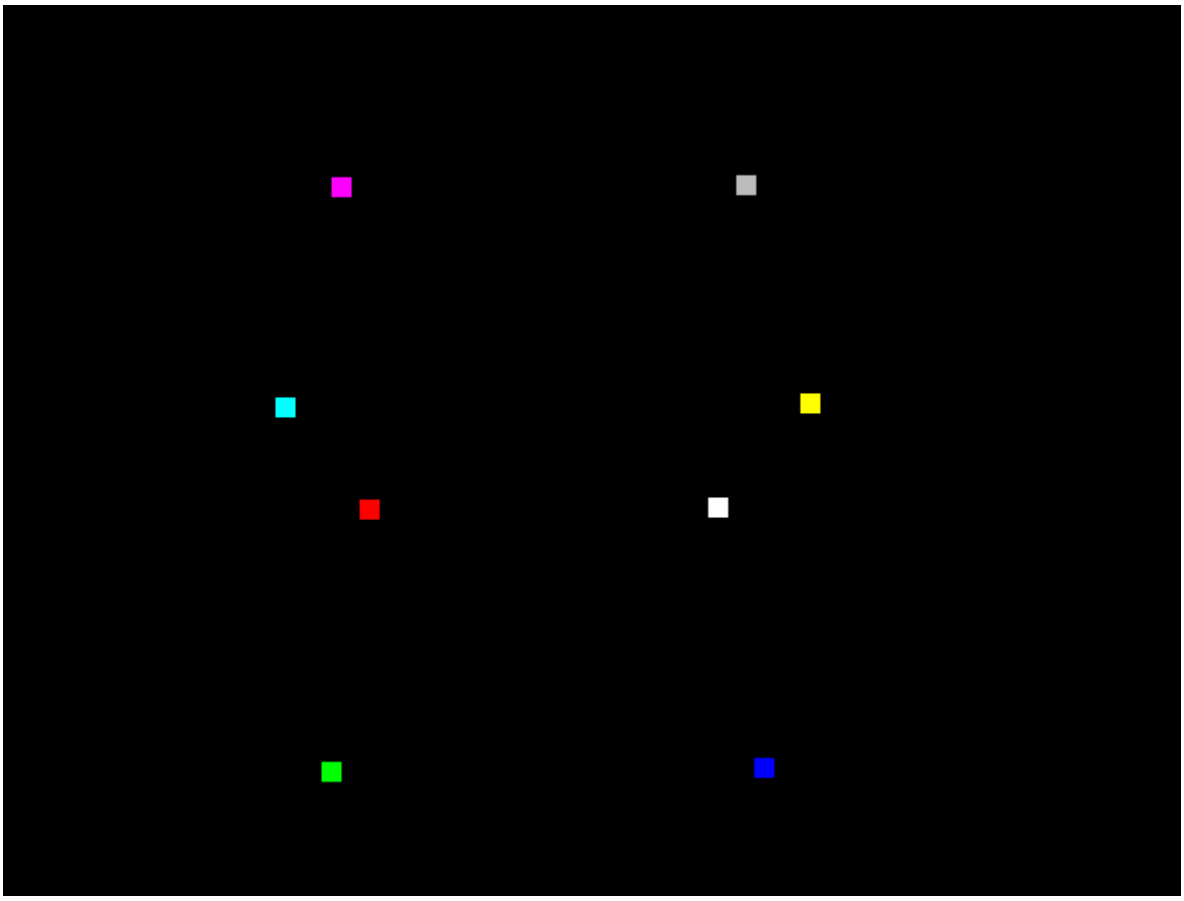
EXERCISE 5: RENDER THE CUBE'S VERTICES AS POINTS

Goal: Render only the eight vertices of the cube as individual points.

Solution:

```
inputAssembly.topology = VK_PRIMITIVE_TOPOLOGY_POINT_LIST;
```

```
vkCmdDraw(commandBuffer, 8, 1, 0, 0); // 8 for the vertex points i want to
visualise for exercise 5
```



Reflection: This exercise was also straightforward, I just had to change the topology to point list and adjust the draw call to render only 8 points. But what I learned from this exercise is how to manipulate the topology to achieve different rendering effects and also, altering the shader to make the points more visible.

EXERCISE 6: RENDER THE CUBE'S EDGES AS LINES

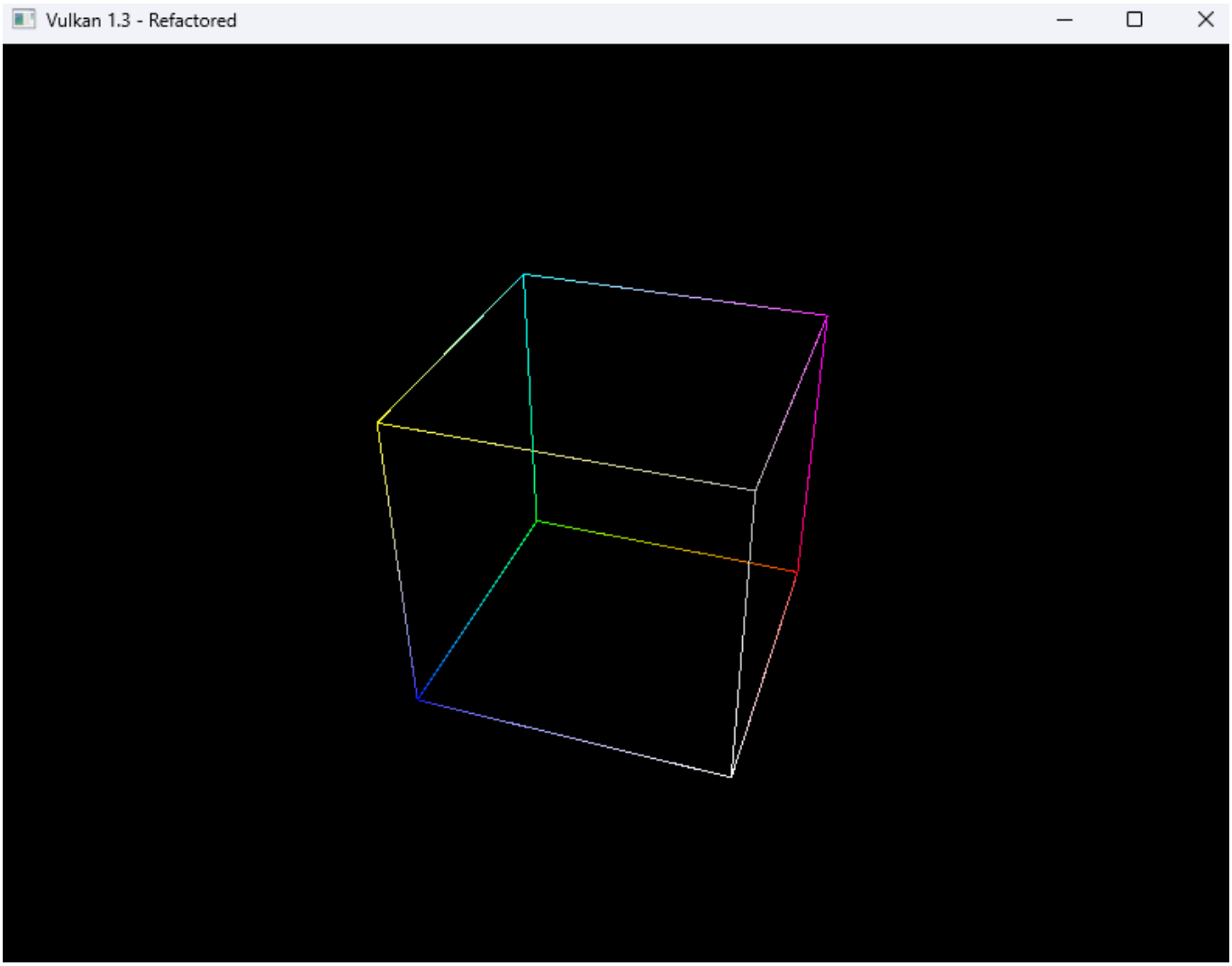
Goal: Render the 12 edges of the cube using line segments.

Solution:

```
const std::vector<uint16_t> cube_edge_indices = {
    // back face (-Z)
    0,1,  1,2,  2,3,  3,0,
    // front face (+Z)
    4,5,  5,6,  6,7,  7,4,
    // side edges (connecting back/front)
    0,4,  1,5,  2,6,  3,7
};
```

```
inputAssembly.topology = VK_PRIMITIVE_TOPOLOGY_LINE_LIST;
```

```
vkCmdDrawIndexed(commandBuffer, 24, 1, 0, 0, 0);// 24 for the line list indices
for exercise 6
```



Reflection: When compiled my initial attempt to render the cube's edges as lines, I encountered an issue where the lines on one face was not visible. After some investigation, I realized that the problem was due to me not pointing to the correct indices.

EXERCISE 7: TRIANGLE STRIPS

Goal: Render the cube using the VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP topology.

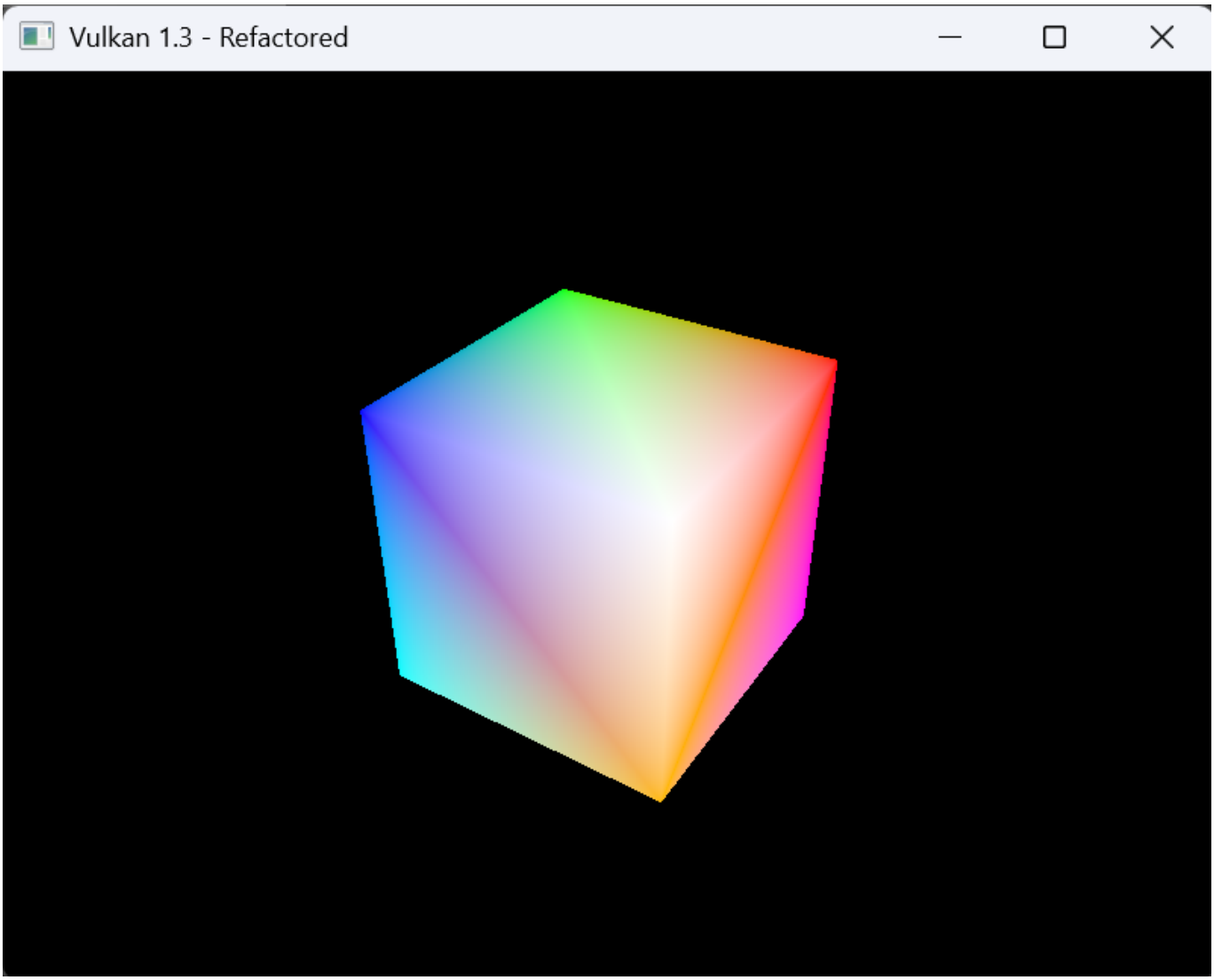
Solution:

```
const std::vector<Vertex> Cube_vertices = {

    {{-1.0f, -1.0f, +1.0f}, {1.0f, 0.0f, 0.0f}}, // 0
    {{+1.0f, -1.0f, +1.0f}, {0.0f, 1.0f, 0.0f}}, // 1
    {{+1.0f, +1.0f, +1.0f}, {0.0f, 0.0f, 1.0f}}, // 2
    {{-1.0f, +1.0f, +1.0f}, {1.0f, 1.0f, 1.0f}}, // 3

    {{-1.0f, -1.0f, -1.0f}, {1.0f, 0.0f, 1.0f}}, // 4
    {{+1.0f, -1.0f, -1.0f}, {1.0f, 1.0f, 0.0f}}, // 5
    {{+1.0f, +1.0f, -1.0f}, {0.0f, 1.0f, 1.0f}}, // 6
```

```
    {{-1.0f, +1.0f, -1.0f}, {1.0f, 0.5f, 0.0f}}, // 7
};
```

```
const std::vector<uint16_t> Cube_edge_indices = {
    // Face 1: Front (+Z)
    0, 1, 3, 2,

    // Degenerate
    2, 1,

    // Face 2: Right (+X)
    1, 5, 2, 6,

    // Degenerate
    6, 5,

    // Face 3: Back (-Z)
    5, 4, 6, 7,

    // Degenerate
    7, 4,

    // Face 4: Left (-X)
    4, 0, 7, 3,

    // Degenerate
    3, 2,

    // Face 5: Top (+Y)
    3, 2, 7, 6,

    // Degenerate
    6, 5,

    // Face 6: Bottom (-Y)
    5, 1, 4, 0
};
```

```
inputAssembly.topology = VK_PRIMITIVE_TOPOLOGY_TRIANGLE_STRIP;
```



Reflection: After the initial attempt to render the cube using triangle strips, I noticed that some faces (top and bottom) were not visible. I had thought I did it wrong, but then I realised the issue was I wasn't using a deptth buffer. Without a depth buffer, the triangles are drawn in the order they are processed, which can lead to some faces getting overdawned by others, resulting in them not being visible.

Question: I would like to understand if my understanding is correct, and also how to implement a depth buffer in Vulkan to properly render 3D objects like this cube.

EXERCISE 8: DRAWING MULTIPLE CUBES USING INSTANCED DRAWING

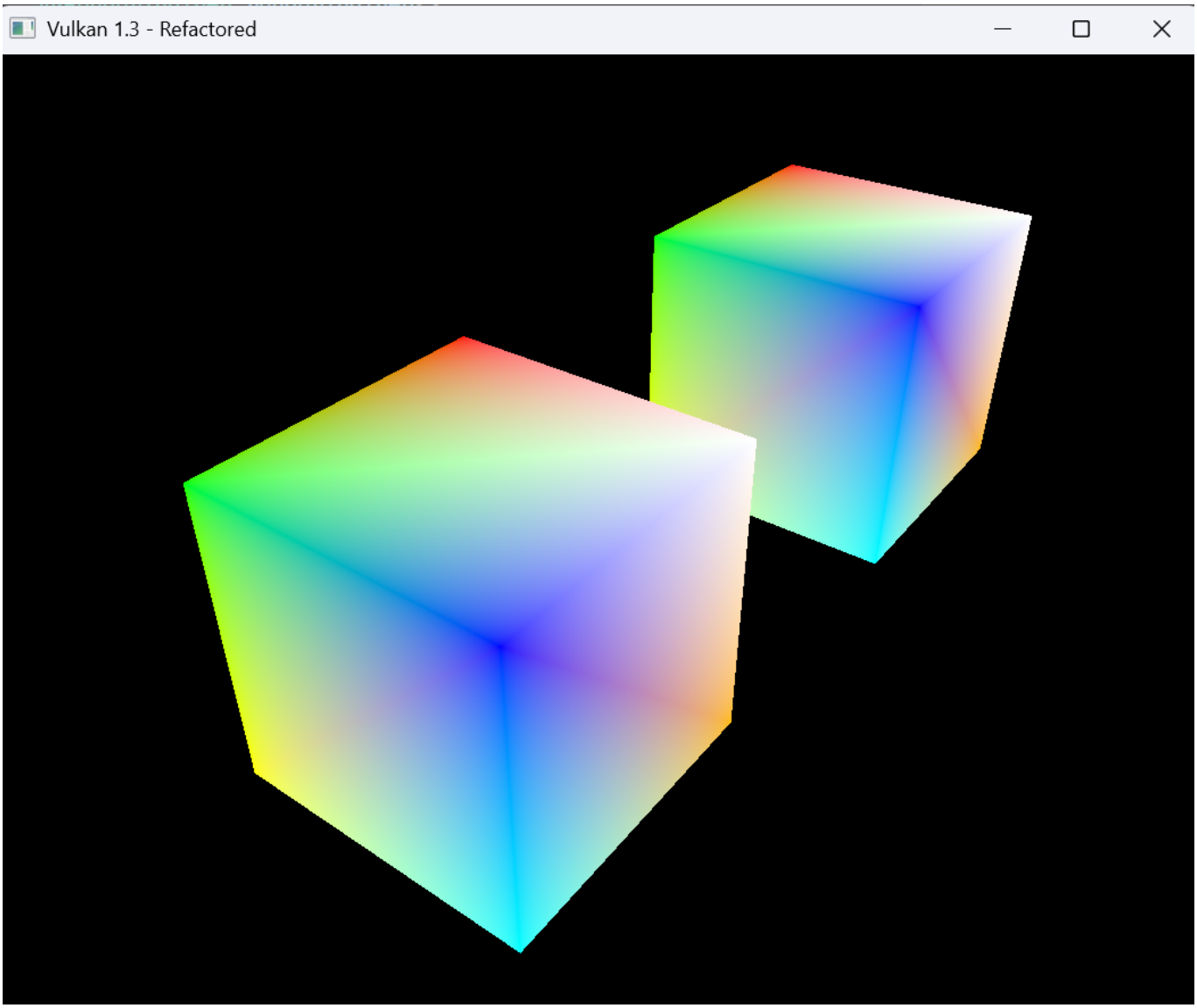
Goal: Render two distinct triangles instead of one quad using vkCmdDraw().

Solution:

```
vkCmdDrawIndexed(commandBuffer, static_cast<uint32_t>(indices.size()), 2, 0, 0,
0);
```

```
void main() {
    // Per-instance offset along X: ..., -1.5, 1.5, 4.5, ...
    float idx = float(gl_InstanceIndex);
```

```
    vec3 offset = vec3(idx * 2.0 - 1.5, 0.0, 0.0);

    // Build translation matrix (column-major)
    mat4 T = mat4(1.0);
    T[3] = vec4(offset, 1.0);

    mat4 instanceModel = ubo.model * T;
    gl_Position = ubo.proj * ubo.view * instanceModel * vec4(inPosition, 1.0);
    fragColor = inColor;
}
```



Reflection: I was able to change the number of instances to 2 in the draw call, and then use the gl_InstanceIndex. I attempted to experimented with the multiplier and offset values to see how they affected the positioning of the cubes. So I was able to have a better understanding of how instanced drawing works in Vulkan. It would be interesting to explore how to make sure that no clipping occurs when rendering multiple instances, so I would like to understand how to adjust the view or projection matrices accordingly. By the end I understood, that this is an efficient way to limit overhead, when multiple instances of an object is required. Rather than calculating the vertex position for each object, it can be defined once, and matrix transform will be responsible for things like the new objects position in space, oritentation, etc.

EXERCISE 9: DRAWING TWO CUBES USING PUSH CONSTANTS

Goal: Render two distinct wireframe cubes side-by-side using the same vertex/index buffers, but with different cube positions

Solution:

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

layout(push_constant) uniform PushConstants {
    mat4 model;
} pushConstants;

layout(location = 0) in vec3 inPosition;
layout(location = 1) in vec3 inColor;

layout(location = 0) out vec3 fragColor;

void main() {
    fragColor = inColor;
    gl_Position = ubo.proj * ubo.view * pushConstants.model * vec4(inPosition,
1.0);
}
```

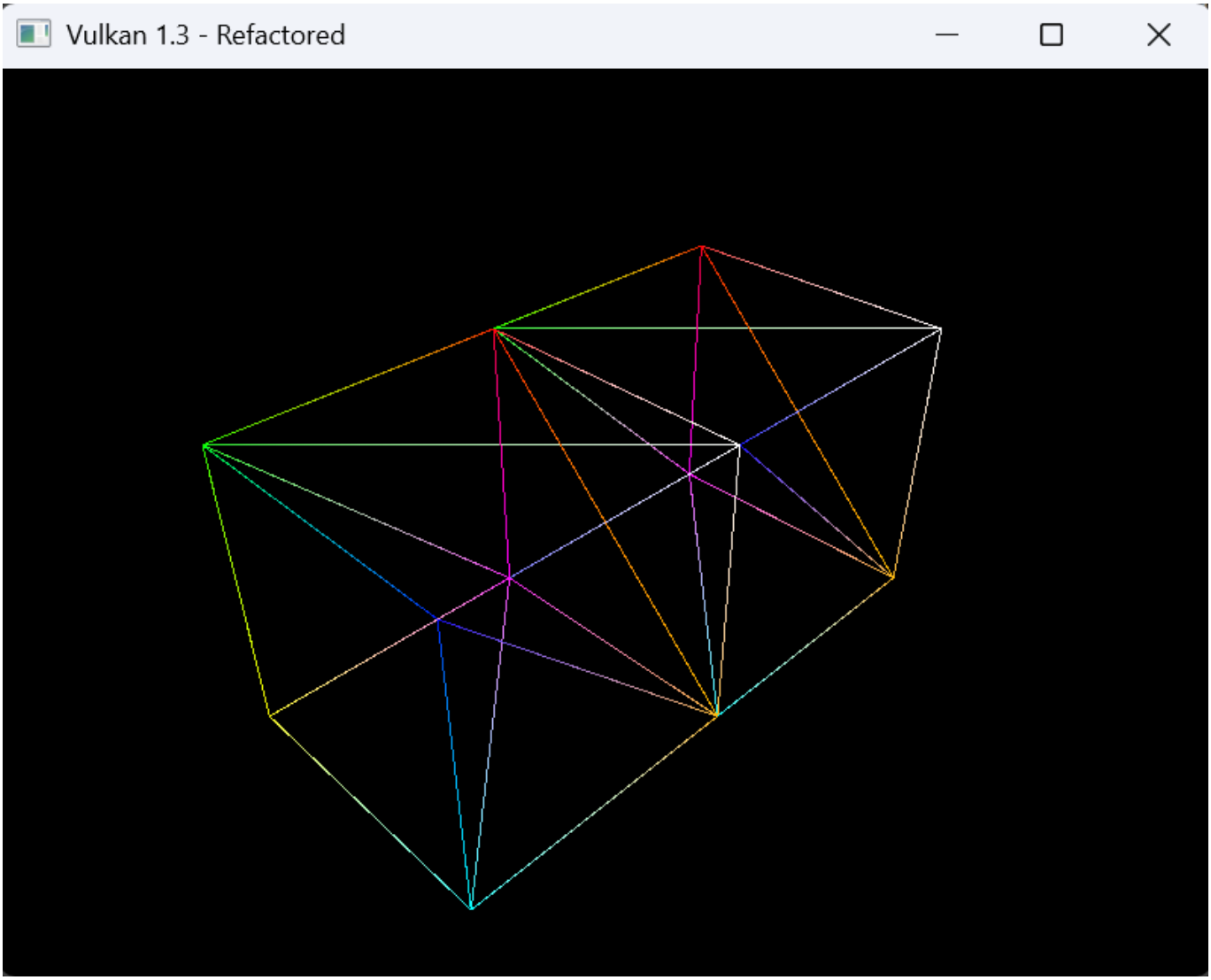
```
    VkPushConstantRange pushConstantRange{};
    pushConstantRange.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
    pushConstantRange.offset = 0;
    pushConstantRange.size = sizeof(ModelPushConstant);

    VkPipelineLayoutCreateInfo pipelineLayoutInfo{};
    pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
    pipelineLayoutInfo.setLayoutCount = 1;
    pipelineLayoutInfo.pSetLayouts = &descriptorSetLayout;
    pipelineLayoutInfo.pushConstantRangeCount = 1;
    pipelineLayoutInfo.pPushConstantRanges = &pushConstantRange;
```

```
    ModelPushConstant pushUBO{};
    pushUBO.model = glm::translate(glm::mat4(1.0f), glm::vec3(-1.0f, 0.0f, 0.0f));
    vkCmdPushConstants(
        commandBuffer,
        pipelineLayout,
        VK_SHADER_STAGE_VERTEX_BIT,
        0,
```

```
        sizeof(ModelPushConstant),
        &pushUBO
    );
    vkCmdDrawIndexed(commandBuffer, static_cast<uint16_t>(indices.size()), 3, 0,
0, 0);

    pushUBO.model = glm::translate(glm::mat4(1.0f), glm::vec3(1.0f, 0.0f, 0.0f));
    vkCmdPushConstants(
        commandBuffer,
        pipelineLayout,
        VK_SHADER_STAGE_VERTEX_BIT,
        0,
        sizeof(ModelPushConstant),
        &pushUBO
    );
    vkCmdDrawIndexed(commandBuffer, static_cast<uint16_t>(indices.size()), 3, 0,
0, 0);
```

```
struct UniformBufferObject {
    alignas(16) glm::mat4 view;
    alignas(16) glm::mat4 proj;
};

struct ModelPushConstant {
    glm::mat4 model;
};
```

```
ubo.view = glm::lookAt(glm::vec3(2.0f, 2.0f, 2.0f)*2.0f, glm::vec3(0.0f, 0.0f,
0.0f), glm::vec3(0.0f, 0.0f, 1.0f));
```



Reflection: During this exercise, I encountered a challenge when trying to implement push constants for rendering two cubes. After following the steps, I realized that I something had gone wrong because I was not seeing any cubes rendered on the screen. and the window would open and close immediately. I had to roll back to my previous working version and begin the exercise again from scratch. I was able to successfully implement push constants and render the two cubes side by side. This exercise shares a similaraity to exercise 8 because they both use an offset to calculate the new position, but it differs, because this is done with a new draw call.