# Vulkan-Edu: An Educational Library For Vulkan

UNDERGRADUATE HONOURS THESIS
FACULTY OF SCIENCE (COMPUTER SCIENCE)
ONTARIO TECH UNIVERSITY

Ibrahim Mushtaq

SUPERVISORS:
Jeremy S. Bradbury
Mark Green

April 23, 2020

## Abstract

Vulkan is a modern low-level graphics and computational API, developed by Khronos Group. Vulkan's primary uses are towards gaming and highly intensive workload, where CPU utilization might become a bottleneck. Vulkan gives many controls to the developer eliminating the driver overhead with guessing work. With the removal of overhead results in a much better performance improvement compared to retained-mode graphics APIs such as OpenGL or DirectX 11. The API becomes hard to program for due to many low-level calls, which made the codebase larger. The more extensive code provides hindrance to the newer developer who wants to learn this technology as one mistake can make the application have run time errors. In this paper, we present Vulkan-Edu, a library to manage Vulkans low-level calls, as well as boilerplate codes, that can be a problem for students that are learning vulkan for the first time. To facilitate the use of Vulkan-Edu, we devesided a set of labs for students in Advance Computer Graphics attending Ontario Tech University that will help the learning of some Vulkan API. The current state of the labs are 7 out of 10 labs are done. Lab 7-9 has minor bugs that needs to be fixed. There are no results as the library had bugs that need to get fixed and, we are hoping to test the library and the lab for the 2021 fall semester for that course.

# Contents

## List of Figures

## List of Tables

# 1 Introduction

## 1.1 General Overview

Vulkan is a modern graphical and computes Application Programming Interface or API, which offers more performance and speed [9]. It is platform-agnostic, which allows the application to work regardless of the target platform. It also is low level and explicit, which enables the developer to have full control over the hardware. Commercial game engines such as Unity 3D [2] or Unreal Engine [14] has Vulkan implemented. It is also used in academia since it provides the low-level context of how the GPU and CPU communicate. There are other graphics pipelines, such as OpenGL or DirectX; however, these choices provide some limitations despite their excellence. OpenGL is available to many platforms, provided that the manufacturer has incorporated them; however, OpenGL uses a single state machine with all the call being sequential [7]. DirectX is only available on the Microsoft Windows Operating System but not available to other platforms such as Apple Mac OS or Linux distros.

## 1.2 Problem and Motivation

Vulkan is not used everywhere despite being a powerful high-performance graphic API. It takes more time and resources to implement new technology within the software. Vulkan is not an easy API to use since there is very little validation on the driver; the developer would have to take care of driver validation, which increases the code complexity. Numerous API calls get reused throughout the life span of the code. It has more extended boilerplate code, and it takes more time to learn the new concept on how this API does its call. To write a simple program that displays an object, it would take about 2500 to 3000 lines of code, with 2/3 of the code being boilerplate. It provides a barrier to the new developer that wants to use the API since it is easier to mess up in coding.

## 1.3 Goal

The goal of this paper is to aid students in learning the Vulkan library. To support this goal, we presented a library, Vulkan-Edu, to manage low-level Vulkan call as well as handling boilerplate code. The library will be used in the Advanced Computer Graphics course at Ontario Tech University and should be able to facilitate the learning of Vulkan. The library achieves this by creating different procedures/methods that each does an initialization of some sort for Vulkan API. The library would give the student more time to learn what is necessary code to run Vulkan. There are sets of labs that the students can do using the given library, which gradually assisted the students in learning more about Vulkan and its API. Essentially it is a prototyping library but mainly focuses on learning the vital part of the code to make Vulkan run.

## 1.4 Contribution

Two main contributions from this paper:

1. An opensource library that supports novice user
2. A set of 10 labs that can be completed using the basic library function

They are both available publicly at `https://github.com/sqrlab/vulkan-edu`

## 2 Related Work

Vulkan is adopted early on by many game developers like DOTA 2, Mad Max, Doom Eternal with game engines such as Unity 3D, Unreal Engine and Source 2.0 [6]. However, all the example exploits Vulkan for the 3D rendering pipeline. For this case, only a few are working on a similar project of creating a rapid prototype library, the library that has the purpose of teaching, and a tutorial for using such API such as:

- Vulkan Cauldron, which is developed and used internally at AMD to showcase prototypes, demos and SDK samples [3].
- Sacha Williams Code which provides an extensive example of how many could achieve different way on programming in Vulkan. The whole GitHub repository is not dedicated to rapid prototyping, but the backend code that was written can be used as a rapid prototyping system [13].
- Vulkan Tutorial provided by LunarG explores the idea of teaching Vulkan in a layered approach; however, it only explains the bare minimum on using Vulkan [8].
- Vulkan-Tutorial provided by Alexander Overvoorde, which discusses Vulkan in more detailed then LunarG tutorial [10].

# 3 The Vulkan-Edu Library

The library that this paper refers to as ”Vulkan-Edu” has many ways to reduce the amount of code needed to at least use Vulkan in a bare task. Which includes many procedures in layers that allow the user to call API in layers. The use of Vulkan-Edu focuses on students attending the course Advance Graphics.

## 3.1 Design Assumption

To make the library, we made a few assumptions:

- This library is used in a course setting and should work with what the course is teaching
- The user should be able not to use a few procedure calls but would need to assign the appropriate variable

The library makes a couple of assumptions.

- When creating the instance, it assumes that no other Layers for Vulkan is needed.
- It assumes that the user will pick the first GPU in the list
- The user would want two images presented, colour and depth images. It builds from that assumption to create many primitives synchronized object such as Semaphores and Fences
- If an error occurs through the initialization process, the program will immediately print where the potential failure and closes the program

## 3.2 Library Implementation

Vulkan-Edu works by wrapping each API call into specific procedure calls. Each call represented by the function names will do specific initialization tasks. There is around 12 procedure calls that the user can call to fully initialize Vulkan such that they can program the critical part. Each procedure call takes In a parameter called LHContext. LHContext holds all the necessary information regarding Vulkan's initialization. The code block below provides an example of how a user should use the library. The user should not program the struct and all the methods below; Vulkan-Edu handles this. It is mainly there to show steps of the procedure to allow initialization of Vulkan

```
struct LHContext context = {};
createInstance(context);
createDeviceInfo(context);
createWindowContext(context, 1280, 720);
createSwapChainExtention(context);
createDevice(context);
createDeviceQueue(context);
createSynchObject(context);
createCommandPool(context);
createSwapChain(context);
createCommandBuffer(context);
createSynchPrimitive(context);
createDepthBuffers(context);
createRenderPass(context);
createPipeLineCache(context);
createFrameBuffer(context);
prepareSynchronizationPrimitives(context);
```

The user would first need to initialize the ”LHContext,” which holds all the required information and states to create a full Vulkan Instance. The user would then need to enter a set of procedural function that will initialize the instance.

Vulkan-Edu also provides many helper functions, which alleviates the repetition of code. There are around 12 methods implemented to aid users, such as creating buffers or binding the memory location of the GPU to the buffers. It also provides multiple ways of loading in the shaders. The user should only call the methods when needed.

## 3.3 Library Interface

The user should only need to program code to display models or to manipulate the display, such as creating multiple viewports. The user would need to program the main driver for the code such as:

- Loading in the relevant model information and mapping the data.
- Create different uniform buffers.
- Create descriptor sets layout, sets, and pool, which describes how the uniform works.
- Creating the pipeline for each object and the command buffer

The user can use the context that was initialized earlier to help create this set of codes.

| Lab Numbers | Lab Objectives |
|---|---|
| 1 | Setup the library and draw a cube (shaders and shape data are provided) |
| 2 | Load models from OBJ and program shaders |
| 3 | Add new uniform variables to the shaders |
| 4 | Add more objects to the scene |
| 5 | Add new shaders through different descriptors set |
| 6 | Add texture to multiple objects |
| 7 | Create a new depth buffer (start of shadow mapping) |
| 8 | Retrieve the depth buffer (continue of shadow mapping) |
| 9 | Finish shadow mapping |
| 10 | Create multiple view ports |

**Table 1: Lab objectives**

## 4 Lab based learning with Vulkan-Edu

### 4.1 Lab Overview

The main objective approach of this thesis has been to use the lab alongside the library to help the students learn Vulkan. The new sets of labs are designed to work alongside the library and build off from the previous labs. The labs are not a learn all solution but a way to learn different ways of tackling Vulkan. Table 1 describes what the objective for each lab is.

The lab structure provides a more intuitive understanding of how each module works. For example, Lab 2 starts with how to load your model and map it. The user would need to understand how to pack in a model and map different attributes.

The labs will not include the use of new technology; it will still use the same library that the previous course, Computer Graphic and Visualization, used, such as the TinyObjLoader[12], GLFW[4], FreeImage[1].

Lab 7 to 9 does not work due to some bugs within the solution code. However, labs 1 to 6 and 10 work and is in Appendix B if one wants to see the solution

## 4.2 Lab 1

What this lab does is mainly to help set up the library. It has the hard-coded data to draw the cube. The sample code is provided by the instructor to help students use the library. Students can learn how a basic Vulkan application can run with the sample code. The key area of interest is where the loading of vertices happens for the meshes, where the code initializes the uniform buffers. The students would also need to look at how different descriptors are made, such as descriptor sets layout, sets, and pools. Lastly, they would need to look at the pipeline and command buffer creations.

To achieve the goal of the lab, the students would just need to set up the environment correctly, as the lab builds sequentially with other labs. The user will need to see if a coloured cube is displayed to fully complete the lab.

## 4.3 Lab 2

Lab 2 extends from lab 1, with the goal now being loading a custom model from a file. The student would achieve this by modifying wherever the vertex was initialized in the previous lab and use the TinyObJLoader provided during the last course, Computer Graphics and Visualization, to load in the file. From there, the students would need to map the file to the GPU with various methods. The objective emphasizing creating multiple vertex attributes for a model, and to calculate the actual data stride value.

The general technique for this lab is simple, just change a function to accept models and revise some of its code to allow different vertex attributes. The students would also need to modify the shader to take in those attributes slightly.

## 4.4 Lab 3

Lab 3 gets a bit difficult, as it can be mistaken for the future lab as a copy-paste and increase the uniform size; however, it is still straight forward. The students can finish this lab by creating a buffer and binding the buffer to a memory location on the GPU. They would need to add another descriptor set layout for the new buffer and tell it to point to wherever the uniform should be in the shader, such as vertex shaders or fragment shader. They then need to focus on creating a new descriptor set with the relevant binding and the shader type. The students then would need to create a new descriptor type within the creation of descriptor pools. Lastly, changes are required in terms of drawing the model, as the current method is hardcoded.

The goal for this lab is to create new uniforms for any type of shader as it allows for CPU to GPU communication on a different value. The general technique on how a student should finish this lab is by looking at how the previous code handles one uniform buffer and build upon it.

## 4.5 Lab 4

The goal for this lab is to make students load in two or more objects. They can achieve this by looking at how a model was loaded. They would see that all the data gets mapped into a memory location on the GPU and such they need to do replicate the example.

After looking at how the model is loaded, they would need to have a look at where the pipeline creation happens. Each model should have its pipeline, and just need to initialize a new pipeline variable to the object. The last area of focus is to look at the command buffer and just replicate the code that draws for the objects

## 4.6 Lab 5

When loading in multiple models, the chances are that each model should have a separate shader as each object can represent something like a lamp or a reflective surface. With the current solution, each model vertices are loaded in, but they are using the same shaders. The goal for this lab is to load in multiple shader files and have them mapped to each model.

The area of focus for this lab is not creating new uniforms buffers, but the creation of the descriptors and how the pipeline creation handles it. Within the descriptor sets production, the preferred way of loading in the number of shaders is in arrays. Each index array handles different shaders. The descriptor layout set can remain the same if there is a uniform buffer for vertex and fragment shaders. The only changes left within the descriptor is the number of sets that can be allocated by the pool, which is the amount model with different shaders.

## 4.7 Lab 6

Lab 6 is the hardest lab within the whole selection of labs. It requires the user to write more code then just changing or duplicating a few lines of codes. The goal of this lab is to add textures to each of the models. With the help of the texture.h code, the students can achieve the outcome by creating buffers for the images and an image sampler for the uniforms. However, the students should also try to have this set of codes in a function for reusability purposes. Since the whole point of this lab is to program the texture creation, the entire learning objective is to map the data onto the GPU efficiently. The students should know that there are two ways to achieve this, which are using the optimal tile layer or successive layers.

### 4.8 Lab 7

This lab mainly focuses on creating an image for the shadow and storing the images into the buffer. A lot of the code that the students will write is a duplicate copy for creating the colour and depth images within the library. To achieve the outcome of the lab, the user would need to create a new image and use some of the code within lab 6 to make the sampler 2D code. However, the students will not have a complete lab as the amount of work for this lab is enough. The next lab is retrieving the buffer such that it can be displayed.

### 4.9 Lab 8

The lab continues by making the user create the render pass. Since it is necessary as the shadow images use a different format than what the render pass provided by the library does. The students need to create a new render pass that handles the shadows to achieve the labs. The students would need to modify significantly the pipeline creation such that it treats the shadows as a separate model and such need its pipeline. However, this does not fully conclude the shadow mapping lab as it continues to lab 9.

### 4.10 Lab 9

The goal of this lab is to display the shadows. To achieve this, the user would need to modify the command buffer to handle two render passes and would need to change the shader such that it can display the shadows. The learning outcome for this lab is how one would tackle creating shadows or any other types of images and show it on the models. This lab out of the rest does not work due to some bug when making the lab solution. Due to this, lab 7-9 is not working.

### 4.11 Lab 10

The last lab ties everything up by introducing new viewports creation. It requires multiple cameras, and each respective camera positioned correctly. It also uses a new set of shaders called the geometry shaders. This shader handles the geometry placement per camera. To achieve this lab, the students would need to create two viewports within the command buffers and figure out where the camera gets placed and create the geometry shaders.

The general technique starts with updating the uniform buffers to accept an array of variables such that it can hold multiple matrices for the position of each camera, and model. The next step is to modify the pipeline to accept the geometry shader. Follow by creating numerous views and scissors in the command buffer. An implemented geometry shader would mark the end of the lab.

# 5 Conclusion

## 5.1 Conclusion

The thesis explores the idea of developing a library to assist the new programmer in learning the Vulkan API. The main paper aim was toward students in the Advanced Computer Graphics course taught Ontario Tech University and was supplemented with a series of labs. Each lab has an objective that can be solved using the Vulkan API alongside the library. The first part explores the library and how one would use the library to initialize an instance. The second part explores each lab, with the primary objective, end goal, and brief description on how to achieve the overall labs. However, the thesis does not have any results as there are minor bugs that need to be patched before actual use. It also doesn't have a full lab solution, with lab 7-9 currently does not work. Due to that, there was not any lab trial to test it if it works or not. However, the need for educational material for Vulkan is greater, and this is one starting point to help the cause.

## 5.2 Limitation

The whole thesis does have some flaws that may impede students with learning of Vulkan. These includes:

- The library is not working on another platform. The current work was done using the Windows OS. Some codes support multi-platforming; however, those weren't tested due to not having the equipment and time constraint.
- The library is not a library but a framework. This means students would need all the code to use the library, which results in more considerable compile time.

- The current code base is not efficient, as mapping data on the GPU linearly would take more space. There are ways to improve the code further, but those are not presented in the library.
- The library that relies on using the TinyObjLoader has a limitation on how much vertices can be loaded. This may be the way of mapping data onto the GPU
- Not able to test the library and the labs

## 5.3 Future Work

For future work, our primary goal is to have this checked. Preferable in the fall semester of 2021, where students can use the library alongside the lab while collecting feedback. With the feedback, we will improve the overall structure of the library and labs. There is another area that requires work, such as fixing the many fixable limitations presented in the paper. There is another area that will improve ease of use for students, such as creating an initializer for all the struct that the Vulkan API provides. Since many of the structs can use the default value, the students would just need to call those and move on. Another ease of use is to create more helper function as it can reduce the code size quite a bit, allowing the developer to move on to other parts of the code. The last area that needs to be explored is using newer technology within the library, such as adding ray tracing support, adding debugging by introducing different layers, adding 3D texture or cube mapping support, and many more.

## 6 Reference

## References

[1] What is FreeImage? `http://freeimage.sourceforge.net/`. accessed: 2020-04-12.

[2] Introducing the Vulkan renderer preview. `https://blogs.unity3d.com/2016/09/29/introducing-the-vulkan-renderer-preview/`, Oct 2016. accessed: 2020-04-12.

[3] GPUOpen-LibrariesAndSDKs/Cauldron. `https://github.com/GPUOpen-LibrariesAndSDKs/Cauldron`, Dec 2019. accessed: 2020-04-12.

[4] BERGLUND, C. An OpenGL library. `https://www.glfw.org/`. accessed: 2020-04-12.

[5] BOURKE, P. Calculating Stereo Pairs. `http://paulbourke.net/stereographics/stereorender/`, Jul 1999. accessed: 2020-04-12.

[6] GAMBHIR, M., PANDA, S., AND BASHA, S. J. Vulkan Rendering Framework for Mobile Multimedia. In *SIGGRAPH Asia 2018 Posters* (New York, NY, USA, 2018), SA '18, Association for Computing Machinery. accessed: 2020-04-12.

[7] JOSEPH, S. A. An exploratory study of high performance graphics. `https://scholar.utc.edu/cgi/viewcontent.cgi?article=1592&context=theses`, Mar 2016. accessed: 2020-04-12.

[8] LUNARG. Vulkan Samples Tutorial. https://vulkan.lunarg.com/doc/sdk/1.0.57.0/windows/tutorial/html/index.html, Aug 2016. accessed: 2020-04-12.

[9] OLSON, T., ZDRAVKOVIC, A., NGUYEN, H., BUTLER, M., PEARCE, L., DIERCKS, D., AND WARDELL, B. Vulkan - Industry Forged . https://www.khronos.org/vulkan/, Mar 2015. accessed: 2020-04-12.

[10] OVERVOORDE, A. Vulkan Tutorial. https://vulkan-tutorial.com/. accessed: 2020-04-12.

[11] TAN, S. L. E. Stereo Geometry in OpenGL. http://www.orthostereo.com/geometryopengl.html. accessed: 2020-04-12.

[12] TINYOBJLOADER. tinyobjloader/tinyobjloader. https://github.com/tinyobjloader/tinyobjloader, Mar 2020. accessed: 2020-04-12.

[13] WILLEMS, S. SaschaWillems/Vulkan. https://github.com/SaschaWillems/Vulkan/, Apr 2020. accessed: 2020-04-12.

[14] WILSON, J. Unreal Engine 4.21 Released. https://www.unrealengine.com/en-US/blog/unreal-engine-4-21-released, Nov 2018. accessed: 2020-04-12.

## A Appendix: Lab

Each lab provides a new problem that can be solved using the Vulkan API. Students are expected to complete the lab alongside the library at a reasonable time. The expectation of how a student can complete the lab can be seen below:

### A.1 Lab 1

Lab 1 is not just a lab to set up this library. It is also a lab that teaches what each procedure does and why it is there. The library uses GLFW to provide cross-platform functionality since writing system code for one system would void the cross-platform functionality. The first lab is also a learning opportunity to see how one can load in a model and just display it. The code block below is a good start for the user to program as it passes the context and the state that the user should create. Each state will differ for each lab but provides some type of structured variable.

```
 void prepareVertices(struct LHContext& context, struct appState& state, bool
useStagingBuffers)
```

The user should have in this function is a hard-coded model data and some type of mapping of the vertex data to the GPU memory as well as index position data to the GPU memory. The user can use the helper function to reduce the amount of line that is needed while still achieving the same goal. After mapping, the user would need to deal with mapping the model attribute. These attributes describe how the GPU will render the point through shaders. The attribute can contain the vertex position, or the vertex normal as well as the vertex colour. In this case, since the user is drawing a simple cube, the attributes would be the position and the colour. The code block below

Is what the user should write.

```
state.vertexInputBinding.binding = 0;
state.vertexInputBinding.stride = sizeof(g_vb_solid_face_colors_Data[0]);
state.vertexInputBinding.inputRate = VK_VERTEX_INPUT_RATE_VERTEX;
//layout (location = 0) in vec3 inPos;
//layout (location = 1) in vec3 inColor;
//Attribute location 0: Position
state.vertexInputAttributs[0].binding = 0;
state.vertexInputAttributs[0].location = 0;
// Position attribute is three 32-bit signed (SFLOAT) floats (R32 G32 B32)
state.vertexInputAttributs[0].format = VK_FORMAT_R32G32B32_SFLOAT;
state.vertexInputAttributs[0].offset = 0;
// Attribute location 1: Color
state.vertexInputAttributs[1].binding = 0;
state.vertexInputAttributs[1].location = 1;
// Color attribute is three 32-bit signed (SFLOAT) floats (R32 G32 B32)
state.vertexInputAttributs[1].format = VK_FORMAT_R32G32B32_SFLOAT;
state.vertexInputAttributs[1].offset = 16;
```

This code tells the vertex shader which binding it should use to particularly access, and with that tell how the model should behave once drawn on the screen.

The next important set of code is dealing with Uniform Buffer creation. The user should write a function like the code block below. The function:

```
void prepareUniformBuffers(struct LHContext& context, struct appState& state)
```

just merely creates the Buffer with the size of what the Uniform variable holds. However, this function does not describe how does this buffer work in the shader such as pointing the Uniform to the Vertex Shader or Fragment Shader. The user would need to write a couple of more function such as:

```
void setupDescriptorSetLayout(struct LHContext& context, struct appState&
state)
```

Within this function, it creates a Descriptor Set Layout which provides a layout template on describing what the Uniform should do. The Descriptor Set tells how the uniform variable works and maps the uniform buffer to the set itself.

```
// Binding 0 : Uniform buffer
writeDescriptorSet.sType = VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
writeDescriptorSet.dstSet = state.descriptorSet;
writeDescriptorSet.descriptorCount = 1;
writeDescriptorSet.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
writeDescriptorSet.pBufferInfo = &state.uniformBufferVS.descriptor;
// Binds this uniform buffer to binding point 0
writeDescriptorSet.dstBinding = 0;
```

The Descriptor Set Layout and Descriptor Set would be the first point of interest if there are multiple shaders that are different and are being applied to an object

The pipeline function deals with how the object should be rendered out as. Mainly most of the code is boilerplate but could be change. The pipeline stores info such as how the GPU should draw the model through different topology, to the rasterization states and many more. The last segment that the user would need to deal is telling the GPU the commands for drawing the object. The command should accommodate the view port size, the Vertex and Index buffer as well as a way to draw the object itself

```
 vkCmdBeginRenderPass(context.cmdBuffer[i], &renderPassBeginInfo,
VK_SUBPASS_CONTENTS_INLINE);
 VkViewport viewport = {};
 createViewports(context, context.cmdBuffer[i], viewport);
 VkRect2D scissor = {};
 createScisscor(context, context.cmdBuffer[i], scissor);
 vkCmdBindDescriptorSets(context.cmdBuffer[i], VK_PIPELINE_BIND_POINT_GRAPHICS,
  state.pipelineLayout, 0, 1, &state.descriptorSet, 0, nullptr);
 vkCmdBindPipeline(context.cmdBuffer[i], VK_PIPELINE_BIND_POINT_GRAPHICS, state
.pipeline);
 VkDeviceSize offsets[1] = { 0 };
 vkCmdBindVertexBuffers(context.cmdBuffer[i], 0, 1, &state.v.buffer, offsets);
 vkCmdDraw(context.cmdBuffer[i], 12 * 3, 1, 0, 0);
```

The user is provided with the draw function along with the render loop, which deals with telling the GPU to draw it and making sure that the image the GPU draws get displayed on the scene. The

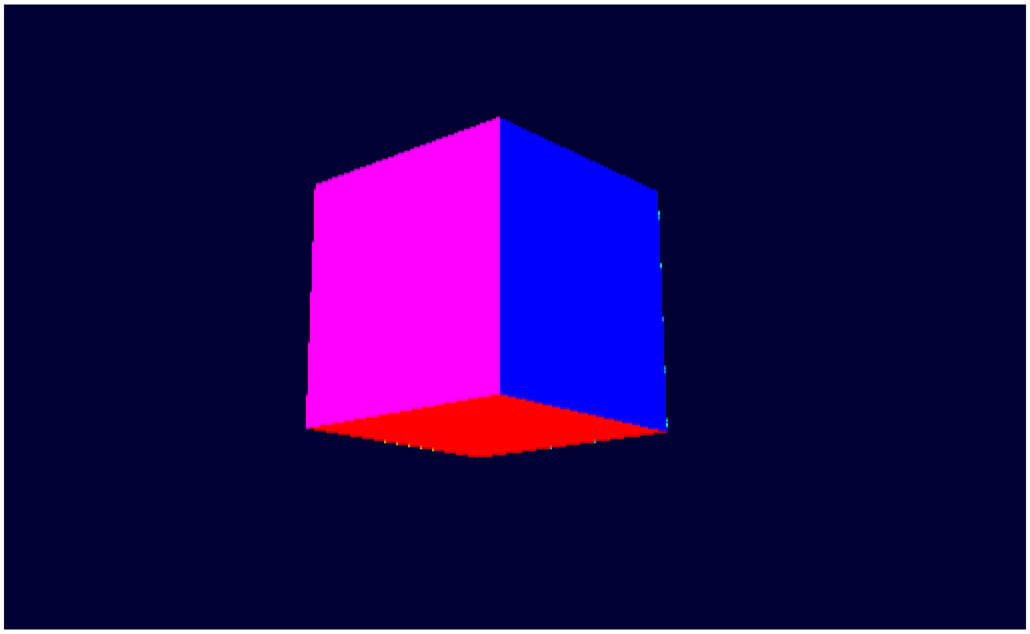

**Figure 1: A cube drawn as an example for Lab 1**

user should have something like Figure 1 The user should get accustomed on making function as it provides code reusability

## A.2 Lab 2

Lab 2 is an extension of lab 1. Its main goal is to start loading in a custom model using OBJ files. The lab uses the same code for loading in Obj provided by the course instructor. Wherever the user-programmed, a function such as "prepareVertices" should amend the code to take in the file location of the object. The code below is what the user should add to the appropriate function.

```
std::string err = tinyobj::LoadObj(shapes, materials, "sphere.obj", 0);

/*  Retrieve the vertex coordinate data */
nv = (int)shapes[0].mesh.positions.size();
vertices = new GLfloat[nv];
for (i = 0; i < nv; i++) {
    vertices[i] = shapes[0].mesh.positions[i];
}

/*  Retrieve the vertex normals */
```

```
nn = (int)shapes[0].mesh.normals.size();
normals = new GLfloat[nn];
for (i = 0; i < nn; i++) {
    normals[i] = shapes[0].mesh.normals[i];
}

/*  Retrieve the triangle indices */
ni = (int)shapes[0].mesh.indices.size();
triangles = ni / 3;
indices = new uint32_t[ni];
for (i = 0; i < ni; i++) {
    indices[i] = shapes[0].mesh.indices[i];
}

state.vBuffer = new float[nv + nn];
int k = 0;
for (i = 0; i < nv / 3; i++) {
    state.vBuffer[k++] = vertices[3 * i];
    state.vBuffer[k++] = vertices[3 * i + 1];
    state.vBuffer[k++] = vertices[3 * i + 2];
    state.vBuffer[k++] = normals[3 * i];
    state.vBuffer[k++] = normals[3 * i + 1];
    state.vBuffer[k++] = normals[3 * i + 2];
}

uint32_t dataSize = (nv + nn) * sizeof(state.vBuffer[0]);
uint32_t dataStride = 6 * (sizeof( float));
```

The user would then need to fix the attribute code to take in normal attribute rather than taking in a generic attribute. The lab also provides shader code that the user can play around with. It takes a uniform with variables:

```
mat4 projectionMatrix;
mat4 modelMatrix;
mat4 viewMatrix;
```

The user would need to use those variables to create Phong light shading. Luckily the Vulkan SDK provided a tool to convert GLSL or HLSL shader code to SPIR-V binary code. The Vulkan-Edu library has a helper function to turn shader code on run time. This, however, has a drawback of taking a long time to load the application.

The outcome of this lab should look like Figure 2

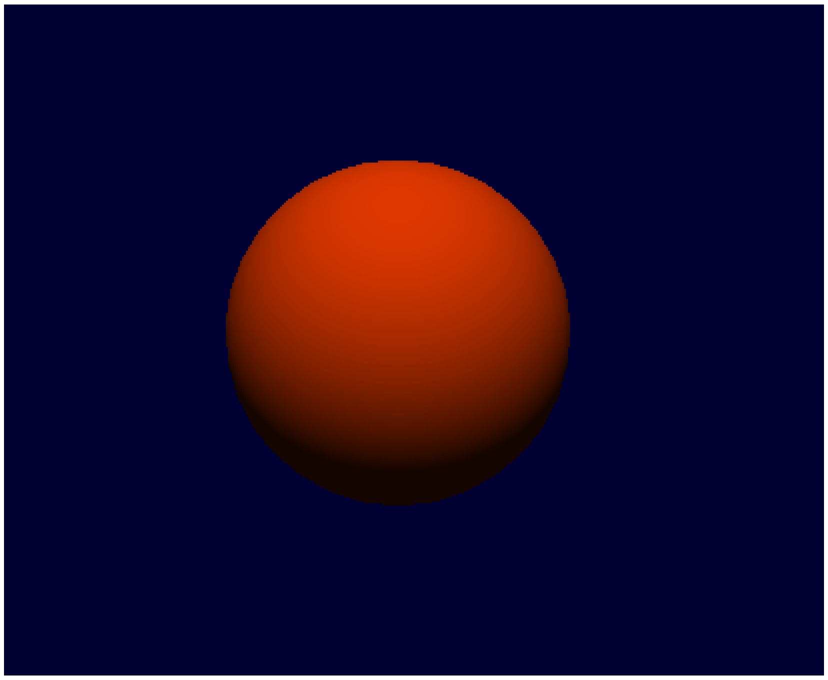

**Figure 2: A sphere with phong shading for Lab 2**

## A.3 Lab 3

Lab 3 starts with Lab 2. The general scope of this lab is to make a new uniform variable that can be used in different parts of the shader code. Currently, the previous lab has a uniform variable within the Vertex Shader. This uniform contains all the transformation matrix for the model. This is useful since it provides data from CPU to GPU on how the model should be displayed. However, there are many cases where more uniform variables are needed. An example is manipulating the position for the lighting or manipulate the light intensity.

The user would need to modify wherever they have initialized the uniform buffer such that another buffer can be made and have that buffer bind to the memory location on the GPU. An easier way to achieve this is to make an array that holds all the information for a uniform like so:

```
// uniform buffer block object
struct {
    VkDeviceMemory memory;
    VkBuffer buffer;
    VkDescriptorBufferInfo descriptor;
} uniformBuffer[2];
```

The user should have something like what this struct has since all the variable is organized and have the relevant variables for the uniforms. The user does not need to store the actual uniform data since all the data is in the device memory. After the creation of the new buffer and mapping the data onto the GPU, the user should look at where the descriptors were made.

The first point of interest is wherever the DescriptorSetLayout was made since it provides info on how and where each uniform works. The user would need to make the number of the set according to how much uniform buffers are there.

```
    // Binding 0: Uniform buffer (Vertex shader)
    std::array<VkDescriptorSetLayoutBinding , 2>layoutBinding = {};
    layoutBinding[0].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    layoutBinding[0].descriptorCount = 1;
    layoutBinding[0].binding = 0;
    layoutBinding[0].stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
    layoutBinding[0].pImmutableSamplers = nullptr;
    // Binding 0: Uniform buffer (Fragment shader)
    layoutBinding[1].descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    layoutBinding[1].descriptorCount = 1;
    layoutBinding[1].binding = 1;
    layoutBinding[1].stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
    layoutBinding[1].pImmutableSamplers = nullptr;
```

The next step that the user should look is where they program the DescriptorSet. The descriptor set holds information regarding about the location where the uniform is being held and binds it to a location. The number of DescriptorSet should be the same as the number of uniform buffers.

```
    VkDescriptorPoolSize typeCounts[2];
    // This example only uses one descriptor type (uniform buffer) and only
   requests
    one descriptor of this type
    typeCounts[0].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    typeCounts[0].descriptorCount = 1;
    typeCounts[1].type = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
    typeCounts[1].descriptorCount = 1
```

With all this, the user should be able to create more than one uniform. The user should still have something like Figure 2

### A.4 Lab 4

This lab is easy, as the students would generally need to make more copies of what the previous labs have done with loading in a model. The students would need to modify the prepare Vertices function such that it should accept the string for the file location as well as the buffers and memory variable for the index and vertex position.

The key area of focus is how a pipeline is created for a model. The students should look at where the pipeline creation info is and create a new pipeline variable for the model that is loaded in. In this case, it is only two models that are being drawn. The students should look closely where the vertex input state handles the vertex input binding and change it to what the model

```
 vertexInputState.pVertexBindingDescriptions = &state.vertexInputBinding[1];
 pipelineCreateInfo.pVertexInputState = &vertexInputState;

 // Vertex shader
 createShaderStage(context, "./shaders/shader.vert", VK_SHADER_STAGE_VERTEX_BIT
, state.shaderStages[0]);

 // Fragment shader
 createShaderStage(context, "./shaders/shader.frag",
VK_SHADER_STAGE_FRAGMENT_BIT, state.shaderStages[1]);

 // Create a rendering pipeline using the specified states
 res = (vkCreateGraphicsPipelines(context.device, context.pipelineCache, 1, &
pipelineCreateInfo, nullptr, &state.pipelineSphere));
```

The only job left to do is to tell the GPU to draw the other model. To do so, the student must look at the command buffer code, and simply copy the code that is drawing the model, and change

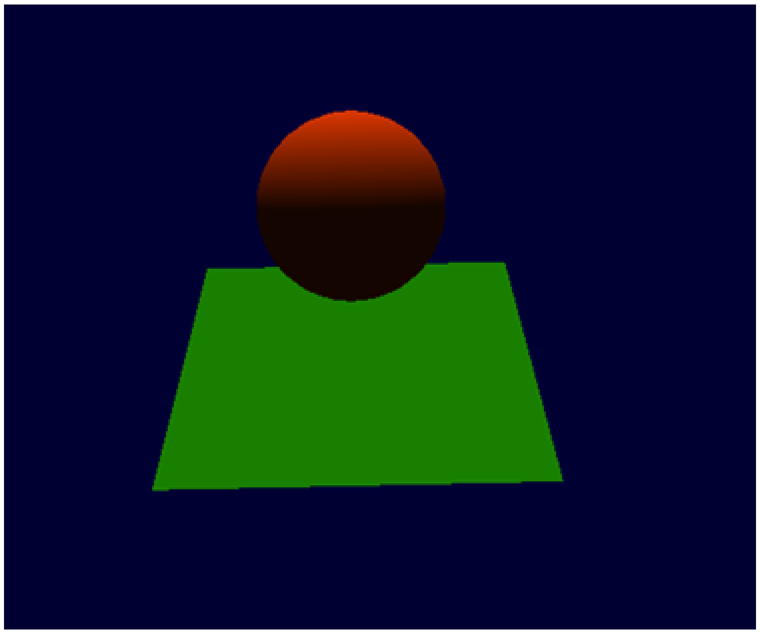

**Figure 3: A sphere and plane being drawn for Lab 2**

the variable to the other model. The outcome of the lab should have multiple models being drawn, such as Figure 3

## A.5 Lab 5

The chances are that students would want to load different shaders for a different model. For example, there might be a model that is just emitting light, while the other model reflects the light like a Sun or Earth. With the current set of labs, there is just a set of shaders with the same uniforms; however, one may want different uniforms. The general idea is to create uniform buffers, create a descriptor set layout and set for each different shader and change the max sets within the pool to how many sets of shaders you would have for each object.

After creating the uniform buffers, the next step is creating a different set layout and bind it to a different bind layout. The same goes for descriptor sets. When creating the pipeline, the only

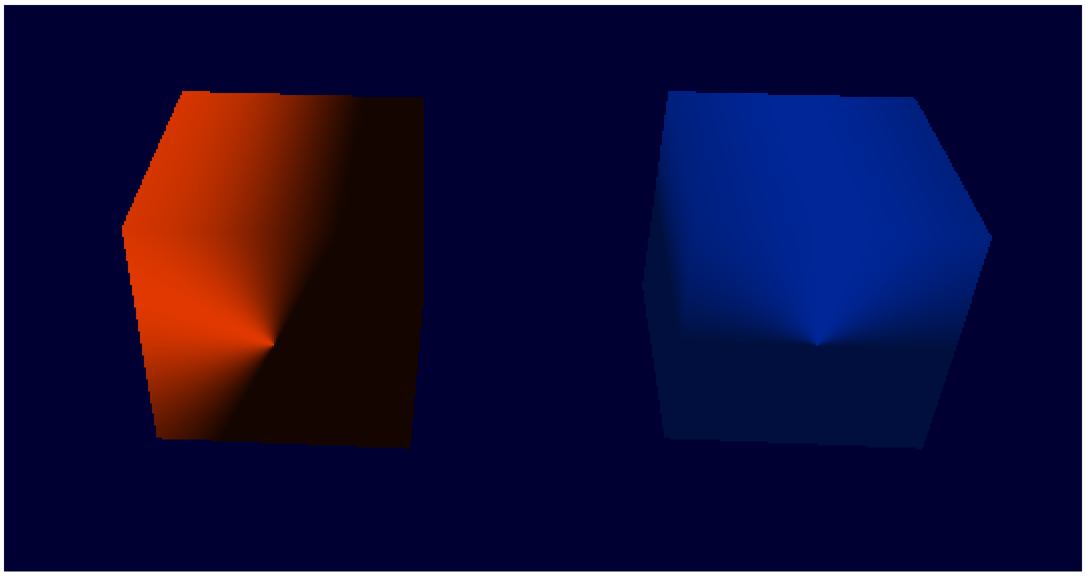

**Figure 4: Two cubes with different value in shaders**

variable that would change regarding different shaders is the descriptor set layouts. Student output can vary due to the shaders; an example output can be seen in Figure 4

## A.6 Lab 6

Lab 6 adds a new challenge of loading in the image and applying the image to a model. In other API such as OpenGL, this would be a trivial task as one would read in the image, call glTexStorage2D and load in the appropriate information. This is not the case for Vulkan as more work is needed to load in the image, map it onto the GPU and creating a Sampler2D uniform variable.

Students are recommended to create a method to handle many of the texture code as it becomes long and dry. Within this function, the user should read in the file using the provided library, FreeImage, to read in and store data. The professor for both Advance Computer Graphic and Computer Graphic and visualization courses have already made a wrapper code for FreeImage, and the user just needs to call it.

After calling the wrapper function, the students need to focus on creating the image info through VkImageCreateInfo, which holds the appropriate information on how the image is being presented. The students should have similar code to what is being shown:

```
 // Setup image memory barrier transfer image to shader read layout
 VkCommandBuffer copyCmd = {};
 VkCommandBufferAllocateInfo commandBufferAllocateInfo{};
 commandBufferAllocateInfo.sType =
VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
 commandBufferAllocateInfo.commandPool = context.cmd_pool;
 commandBufferAllocateInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
 commandBufferAllocateInfo.commandBufferCount = 1;
 res = (vkAllocateCommandBuffers(context.device, &commandBufferAllocateInfo, &
copyCmd));
 VkCommandBufferBeginInfo cmdBufInfo = {};
 cmdBufInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
 res = (vkBeginCommandBuffer(copyCmd, &cmdBufInfo));
```

The next part is to create an Image Subresource range as it describes the region of the image that will be a transition. This can be left as default value. Once that has been made, creating Image Barrier requires the image which was created when making the mappable memory, and many of the value can be left the same as such:

```
 VkImageMemoryBarrier imageMemoryBarrier{};
 imageMemoryBarrier.sType = VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
 imageMemoryBarrier.srcQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
 imageMemoryBarrier.dstQueueFamilyIndex = VK_QUEUE_FAMILY_IGNORED;
 imageMemoryBarrier.image = state.text[index].image;
 imageMemoryBarrier.subresourceRange = subresourceRange;
 imageMemoryBarrier.srcAccessMask = VK_ACCESS_HOST_WRITE_BIT;
 imageMemoryBarrier.dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
 imageMemoryBarrier.oldLayout = VK_IMAGE_LAYOUT_PREINITIALIZED;
 imageMemoryBarrier.newLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;
```

The user would need to submit a pipeline barrier with the image memory barrier and end the command buffer. They then would need to submit those to the queue. A uniform variable called sampler 2d must be created to use the texture. Luckily there is a sampler create info that can be used to help such as the code below

```
VkSamplerCreateInfo samplerCreateInfo = {};
samplerCreateInfo.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
samplerCreateInfo.magFilter = VK_FILTER_NEAREST;
samplerCreateInfo.minFilter = VK_FILTER_NEAREST;
samplerCreateInfo.mipmapMode = VK_SAMPLER_MIPMAP_MODE_NEAREST;
samplerCreateInfo.addressModeU = VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE;
samplerCreateInfo.addressModeV = VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE;
samplerCreateInfo.addressModeW = VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE;
samplerCreateInfo.mipLodBias = 0.0;
samplerCreateInfo.anisotropyEnable = VK_FALSE;
samplerCreateInfo.maxAnisotropy = 1;
samplerCreateInfo.compareOp = VK_COMPARE_OP_NEVER;
samplerCreateInfo.minLod = 0.0;
samplerCreateInfo.maxLod = 0.0;
samplerCreateInfo.compareEnable = VK_FALSE;
samplerCreateInfo.borderColor = VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE;
```

Lastly, this function is to create an image view. The image view provides additional information to the shaders.

```
VkImageViewCreateInfo view_info = {};
view_info.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
view_info.pNext = NULL;
view_info.image = state.text[index].image;
view_info.viewType = VK_IMAGE_VIEW_TYPE_2D;
view_info.format = VK_FORMAT_R8G8B8A8_SRGB;
view_info.components.r = VK_COMPONENT_SWIZZLE_R;
view_info.components.g = VK_COMPONENT_SWIZZLE_G;
view_info.components.b = VK_COMPONENT_SWIZZLE_B;
view_info.components.a = VK_COMPONENT_SWIZZLE_A;
view_info.subresourceRange.aspectMask = VK_IMAGE_ASPECT_COLOR_BIT;
view_info.subresourceRange.baseMipLevel = 0;
view_info.subresourceRange.levelCount = 1;
view_info.subresourceRange.baseArrayLayer = 0;
view_info.subresourceRange.layerCount = 1;
```

After all of this is set, update the descriptor such that it can handle samplers, and the output should look something similar to the one Figure 5.

## A.7 Lab 7

For this lab, it is encouraged to go back to the lab that starts with a multi rendering of the object. The first part of the lab describes making the image. The user would need to set up the offscreen

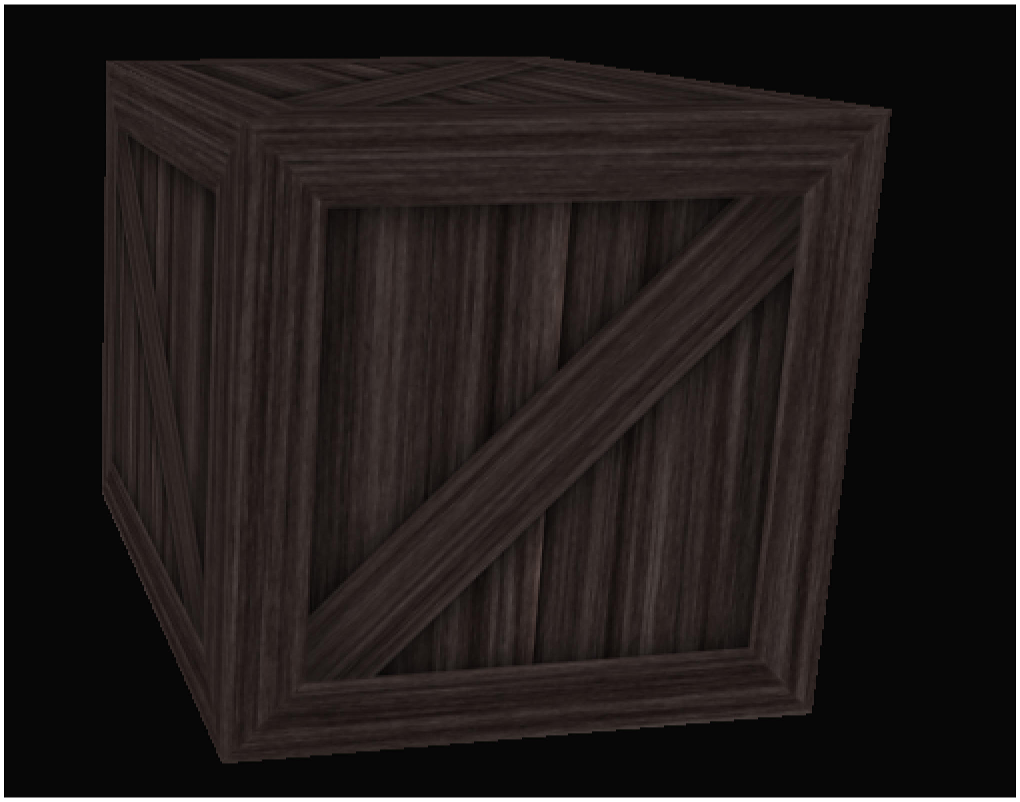

**Figure 5: Texture on a cube [13]**

framebuffer for rendering the scene from the light point of view. To start, the user would need to create image Create info; however, this info is for shadow mapping. The same goes for creating the image view and sampler. This is similar to labs six; however, some image value will differ like the border colour for the sampler being VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE or the shadow filter being SHADOWMAP_FILTER. The last part for this lab is to create a framebuffer with all the related variables. An example of this can be seen below:

```
VkImageCreateInfo image = {};
image.sType = VK_STRUCTURE_TYPE_IMAGE_CREATE_INFO;
image.imageType = VK_IMAGE_TYPE_2D;
image.extent.width = state.offscreenPass.width;
image.extent.height = state.offscreenPass.height;
image.extent.depth = 1;
image.mipLevels = 1;
image.arrayLayers = 1;
image.samples = VK_SAMPLE_COUNT_1_BIT;
image.tiling = VK_IMAGE_TILING_OPTIMAL;
image.format = VK_FORMAT_D16_UNORM;
image.usage = VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT |
VK_IMAGE_USAGE_SAMPLED_BIT;

VkImageViewCreateInfo depthStencilView = {};
depthStencilView.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
depthStencilView.viewType = VK_IMAGE_VIEW_TYPE_2D;
depthStencilView.format = VK_FORMAT_D16_UNORM;
depthStencilView.subresourceRange = {};
depthStencilView.subresourceRange.aspectMask = VK_IMAGE_ASPECT_DEPTH_BIT;
depthStencilView.subresourceRange.baseMipLevel = 0;
depthStencilView.subresourceRange.levelCount = 1;
depthStencilView.subresourceRange.baseArrayLayer = 0;
depthStencilView.subresourceRange.layerCount = 1;
depthStencilView.image = state.offscreenPass.depth.image;

VkSamplerCreateInfo sampler = {};
sampler.sType = VK_STRUCTURE_TYPE_SAMPLER_CREATE_INFO;
sampler.maxAnisotropy = 1.0f;
sampler.magFilter = VK_FILTER_LINEAR;
sampler.minFilter = VK_FILTER_LINEAR;
sampler.mipmapMode = VK_SAMPLER_MIPMAP_MODE_LINEAR;
sampler.addressModeU = VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE;
sampler.addressModeV = sampler.addressModeU;
sampler.addressModeW = sampler.addressModeU;
sampler.mipLodBias = 0.0f;
sampler.maxAnisotropy = 1.0f;
sampler.minLod = 0.0f;
sampler.maxLod = 1.0f;
sampler.borderColor = VK_BORDER_COLOR_FLOAT_OPAQUE_WHITE;

VkFramebufferCreateInfo fbufCreateInfo = {};
fbufCreateInfo.sType = VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
fbufCreateInfo.renderPass = state.offscreenPass.renderPass;
fbufCreateInfo.attachmentCount = 1;
fbufCreateInfo.pAttachments = &state.offscreenPass.depth.view;
fbufCreateInfo.width = state.offscreenPass.width;
fbufCreateInfo.height = state.offscreenPass.height;
fbufCreateInfo.layers = 1;
```

The user would need to update the descriptors to handle a sampler 2d for the shaders. Nothing should display as more tweaking is needed to make it fully work.

## A.8 Lab 8

After creation of a image such that it can render the light point of view, a renderpass is needed for the image since they use a different format then what the library provides. They would need to make a new Attachment Description to handle the new format. In this case the format is VK_FORMAT_D16_UNORM. An example of this creation be such:

```
VkAttachmentDescription attachmentDescription{};
attachmentDescription.format = VK_FORMAT_D16_UNORM;
attachmentDescription.samples = VK_SAMPLE_COUNT_1_BIT;
attachmentDescription.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
attachmentDescription.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
attachmentDescription.stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
attachmentDescription.stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
attachmentDescription.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
attachmentDescription.finalLayout =
VK_IMAGE_LAYOUT_DEPTH_STENCIL_READ_ONLY_OPTIMAL;
```

Many are just defaulting value, which can be used in this case. The user would need to create sub-pass dependencies for layout transition.

```
std::array<VkSubpassDependency, 2> dependencies;
dependencies[0].srcSubpass = VK_SUBPASS_EXTERNAL;
dependencies[0].dstSubpass = 0;
dependencies[0].srcStageMask = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
dependencies[0].dstStageMask = VK_PIPELINE_STAGE_EARLY_FRAGMENT_TESTS_BIT;
dependencies[0].srcAccessMask = VK_ACCESS_SHADER_READ_BIT;
dependencies[0].dstAccessMask = VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
dependencies[0].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;

dependencies[1].srcSubpass = 0;
dependencies[1].dstSubpass = VK_SUBPASS_EXTERNAL;
dependencies[1].srcStageMask = VK_PIPELINE_STAGE_LATE_FRAGMENT_TESTS_BIT;
dependencies[1].dstStageMask = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
dependencies[1].srcAccessMask = VK_ACCESS_DEPTH_STENCIL_ATTACHMENT_WRITE_BIT;
dependencies[1].dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
dependencies[1].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;
```

After the dependencies creation, its just a matter of creating the render pass. Within the pipeline, a few key areas must be added and changed. When creating the pipeline for the shadows, there is no colour blend state for the shadows; the rasterization states must have the depth Bias enabled.

```
 // Offscreen pipeline (vertex shader only)
 createShaderStage(context, "./shaders/shaderOffscree.vert",
VK_SHADER_STAGE_VERTEX_BIT, state.shaderStages[0]);
 assert(state.shaderStages[0].module != VK_NULL_HANDLE);
 pipelineCreateInfo.stageCount = 1;
 colorBlendState.attachmentCount = 0;
 depthStencilState.depthCompareOp = VK_COMPARE_OP_LESS_OR_EQUAL;
 rasterizationState.depthBiasEnable = VK_TRUE;
 dynamicStateEnables.push_back(VK_DYNAMIC_STATE_DEPTH_BIAS);

 dynamicState = {};
 dynamicState.sType = VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
```

```
    dynamicState.pDynamicStates = dynamicStateEnables.data();
    dynamicState.dynamicStateCount = dynamicStateEnables.size();
    dynamicState.flags = 0;

    pipelineCreateInfo.layout = state.pipelineLayouts.offscreen;
    pipelineCreateInfo.renderPass = state.offscreenPass.renderPass;
    res = (vkCreateGraphicsPipelines(context.device, context.pipelineCache, 1, &
   pipelineCreateInfo, nullptr, &state.pipelines.offscreen));
```

This part will not work yet; however, a few more step is needed to get it working. A lot of the code is generally meant for initialization.

## A.9 Lab 9

The last part of the lab is to implement shadows by telling the GPU to draw it. Looking at the command buffer, the student should partition the code such that one is part is meant for the shadows while the other part is meant for the 3d models. The shadow partition code should include the Set Dept Bias to avoid shadow mapping artifact. They should bind the pipeline and the descriptor sets. However, with the current lab solution implemented, the code isn't working due to some error.

## A.10 Lab 10

The student should use the lab four as its easier to start with. This lab introduces geometry shaders as having an array of the model matrix would not work properly with regular vertex shaders. A little math is needed to calculate the model position for the multviewports. This includes calculating the aspect ratio, finding the near clipping and far clipping plane. More of the math and implementation can be found in [5, 11]. The geometry shader code is quite simple. Have a loop that calculates each position and maps it. An example can be seen below

```
layout (binding = 0) uniform UBO {
    mat4 projectionMatrix[2];
    mat4 modelMatrix[2];
    mat4 viewMatrix;
} ubo;

layout (binding = 1) uniform UBOFS{
    vec4 lightPos;
    float ambientStrenght;
    float specularStrenght;
}uboFS;

layout (location = 0) in vec3 inNormal[];
layout (location = 0) out vec3 outNormal;
layout (location = 2) out vec3 outViewVec;
layout (location = 3) out vec3 outLightVec;
void main(void){
    for(int i = 0; i < gl_in.length(); i++){
        outNormal = mat3(ubo.modelMatrix[gl_InvocationID]) * inNormal[i];
        vec4 pos = gl_in[i].gl_Position;
        vec4 worldPos = (ubo.modelMatrix[gl_InvocationID] * pos);
        vec3 lPos = vec3(ubo.modelMatrix[gl_InvocationID] * uboFS.lightPos);
        outLightVec = lPos - worldPos.xyz;
        outViewVec = -worldPos.xyz;
        gl_Position = ubo.projectionMatrix[gl_InvocationID] * worldPos;
        // Set the viewport index that the vertex will be emitted to
        gl_ViewportIndex = gl_InvocationID;
        gl_PrimitiveID = gl_PrimitiveIDIn;
        EmitVertex();
    }
EndPrimitive();
}
```

The user would need to create two scissors and viewports within the command buffer. Each viewport can be placed wherever, for this example, it is just spitted in half

```
VkViewport viewport[2];
viewport[0] = { 0, 0, (float)context.width / 2.0f, (float)context.height, 0.0,
 1.0f };
viewport[1] = { (float)context.width / 2.0f, 0, (float)context.width / 2.0f, (
float)context.height, 0.0, 1.0f };
vkCmdSetViewport(context.cmdBuffer[i], 0, 2, viewport);

VkRect2D scissor[2];
scissor[0] = {};
scissor[0].extent.width = context.width / 2;
scissor[0].extent.height = context.height;
scissor[0].offset.x = 0;
scissor[0].offset.y = 0;
scissor[1] = {};
scissor[1].extent.width = context.width / 2;
scissor[1].extent.height = context.height;
scissor[1].offset.x = context.width / 2;
scissor[1].offset.y = 0;
vkCmdSetScissor(context.cmdBuffer[i], 0, 2, scissor);
```

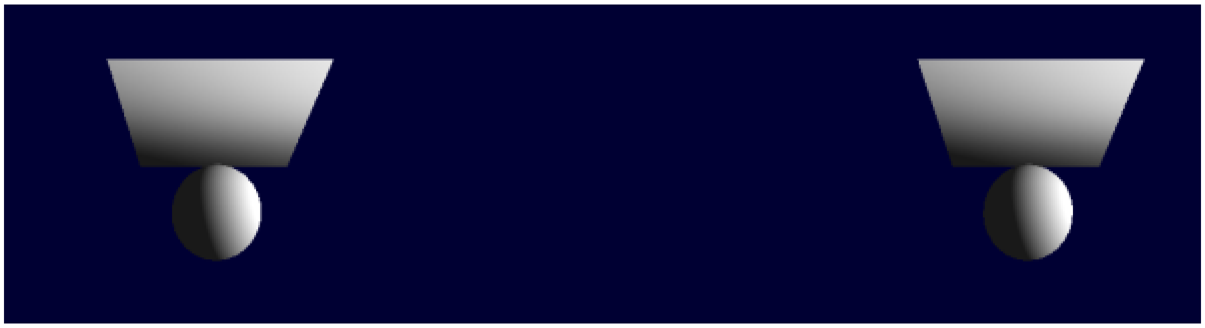
**Figure 6: Multiple view ports with plane and ball model**

The user should get something similar to figure 6.

## B Appendix: External Link

All the work can be found here: `https://github.com/sqrlab/vulkan-edu` Course content can be found here: `http://calendar.uoit.ca/preview_program.php?catoid=22&poid=4194&returnto=891`