UWB-VK Guide

By Nicholas Carpenetti

uwb-vk is a capstone project demonstrating the use of the Vulkan rendering API, and has resulted in a framework for developing Vulkan applications. At first glance, the project may not be easy to understand. This document is intended to alleviate this problem by guiding the reader through the project structure and providing advice on how to catch up to speed with the Vulkan API.

Learning Resources

During the initial development process, several online resources were used to learn how to use Vulkan, and may prove invaluable to your understanding as well. They are as follows:

Alexander Overvoorde's Vulkan Tutorial (link)

A fantastic tutorial that will get anyone with a reasonable understanding of C++ and graphics concepts off the ground. This tutorial was the basis of the first half of the project, although uwb-vk deviated over time to become a framework instead.

Sascha Willems' Vulkan examples (link)

Throughout the project, this was occasionally used a reference to see how one might achieve a specific objective with Vulkan. As opposed to the aforementioned Vulkan Tutorial, Willems implements many more features, and you may find this more useful in the future. He uses his own framework, however, and it will take some time to understand how it works

LearnOpenGL (link)

OpenGL resources such as this one also helped, particularly when it comes to higher-level concepts. In particular, I used this tutorial to learn how to implement lighting. While it cannot demonstrate how to specifically do something with Vulkan, some openGL actions can be broken down or translated into equivalent Vulkan code. In addition, shader code mostly carries over, as only minimal changes need to be make to be compatible with Vulkan

Documentation

Documentation for each of the classes and contained methods was created using Doxygen and is contained within the "Docs" directory

How to Use The Framework

Using the Framework can be broken down into two broad categories

Initialization

Initialization of the RenderSystem is simple. First create your GLFW window. Then simply call **RenderSystem::initialize(...)**, and pass in a pointer to the GLFW window you wish to render to, along with the name of you wish to give the application.

```
void VkApp::initialize(const std::string& appName)
{
    glfwInit();
    createWindow();
    mRenderSystem.initialize(mWindow, appName);
    mInputSystem.initialize(mWindow);
```

Scene Setup

Resource Creation

Before you can create the actual Renderable object, you need to create the resources it is composed of. Creation of them is as follows:

Meshes

Meshes are created using the **RenderSystem::createMesh()** method, passing in a std::shared_ptr<Mesh>*, and the path to load the mesh file from. This file will use a limited version of the Wavefront .obj format (triangles only, no built-in materials)

**std::shared_ptr is used so that RenderSystem can manage the object, including freeing of Vulkan resources. This also allows you to use the same Texture object in multiple Renderables. There may be a better way of making sure this is done.*

Textures

Textures can be created using the **RenderSystem::createTexture()** method, passing in a std::shared_ptr<Texture>, and the path to load the texture file from.

Shaders & ShaderSets

Shaders are created using the **RenderSystem::createShader()** method, passing in a std::shared_ptr<Shader>, the path to the shader, and the stage it corresponds to. This file must compiled into .spv format.

ShaderSets are a collection of std::shared_ptr<Shader>'s, with one corresponding to each potential shader stage. A default nullptr value indicates that stage is not used. Renderables will actually use this, and not individual shaders.

UBOs

Uniform Buffer Objects, or UBOs, correspond to resources your shader needs to access from your application. They are created using the **RenderSystem::createUniformBuffer<T>()**, passing in an std::shared_ptr<T>, and the number of objects (1 if not an array)

An example of how to use these is as follows:

```
void VkApp::createCube()
{
    //start by creating the component resources
    std::shared_ptr<Mesh> cubeMesh;
    mRenderSystem.createMesh(cubeMesh, BOX_MODEL_PATH, true);

    std::shared_ptr<Texture> boxDiffuseMap;
    mRenderSystem.createTexture(boxDiffuseMap, BOX_DIFFUSE_PATH);

    std::shared_ptr<Texture> boxNormalMap;
    mRenderSystem.createTexture(boxNormalMap, BOX_NORMAL_PATH);

    std::shared_ptr<Texture> boxSpecularMap;
    mRenderSystem.createTexture(boxSpecularMap, BOX_SPECULAR_PATH);

    ShaderSet boxShaderSet;
    mRenderSystem.createShader(boxShaderSet.vertShader, BOX_VERT_SHADER_PATH, VK_SHADER_STAGE_VERTEX_BIT);
    mRenderSystem.createShader(boxShaderSet.fragShader, BOX_FRAG_SHADER_PATH, VK_SHADER_STAGE_FRAGMENT_BIT);

    mRenderSystem.createUniformBuffer<MVPMatrices>(mCubeMVPBuffer, 1);
```

Renderable Creation

Once the Necessary resources are created, you can use them to create your renderable. This is done by first creating a new Renderable Object with **RenderSystem::createRenderable()**, passing in a std::shared_ptr<Renderable> to be created.

Applying a Shader Set

Applying a ShaderSet to the Renderable is achieved with the **Renderable::applyShaderSet()** method, passing in a ShaderSet object

Shader Bindings

Setting up shader bindings will describe which types of resources the Renderable's ShaderSet will expect, and at what binding number. For example, to tell the Renderable to expect a UBO in the vertex shader at binding zero, write the following line:

```
mCube->addShaderBinding(VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, VK_SHADER_STAGE_VERTEX_BIT, 0, 1);
```

Binding

To actually bind the resource, call **Renderable::bindUniformBuffer()**, or **Renderable::bindTexture(),** passing in the resource to bind, and the slot to bind at. This slot number needs to correspond to a shader binding of the same type (uniform buffer or Texture).

```
mCube->bindUniformBuffer(mBoxMVPBuffer, 0);
```

Instantiating

To instantiate a renderable, call **RenderSystem::instantiateRenderable()** and pass in the Renderable object.

```
//create a renderable and make the appropriate attachments
mRenderSystem.createRenderable(mCube);

//setup the shaders and note the bindings they will use
mCube->applyShaderSet(boxShaderSet);
mCube->addShaderBinding(VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, VK_SHADER_STAGE_VERTEX_BIT, 0, 1);              //MVP
mCube->addShaderBinding(VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER, VK_SHADER_STAGE_FRAGMENT_BIT, 1, 1);            //lights
mCube->addShaderBinding(VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_SHADER_STAGE_FRAGMENT_BIT, 2, 1);    //diffuse map
mCube->addShaderBinding(VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_SHADER_STAGE_FRAGMENT_BIT, 3, 1);    //normal map
mCube->addShaderBinding(VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER, VK_SHADER_STAGE_FRAGMENT_BIT, 4, 1);    //specular map

//set the mesh we will use
mCube->setMesh(cubeMesh);

//bind resources
mCube->bindUniformBuffer(mCubeMVPBuffer, 0);
mCube->bindUniformBuffer(mLightUBOBuffer, 1);
mCube->bindTexture(boxDiffuseMap, 2);
mCube->bindTexture(boxNormalMap, 3);
mCube->bindTexture(boxSpecularMap, 4);

//finally, instantiate
mRenderSystem.instantiateRenderable(mCube);
```

Updating & Rendering

Updating UBOs

While running the main loop you may want to update the UBO's. To do so, call **RenderSystem::updateUniformBuffer<T>()**, passing in a dereferenced UBO object, data to assign to the UBO, and the index at which to assign it (0 if not applicable). For example, a method that assigns a new MVP (model-view-projection matrices) buffer is as follows:

```
void VkApp::updateMVPBuffer(const UBO& mvpBuffer,
                            const Renderable& renderable,
                            const Transform& renderableXForm,
                            const Camera& camera)
{
    MVPMatrices mvp = {};
    mvp.model = renderableXForm.getModelMatrix();
    mvp.view = camera.viewMat;
    mvp.projection = camera.projMat;
    mvp.normalMat = glm::transpose(glm::inverse(mvp.view * mvp.model));

    mRenderSystem.updateUniformBuffer<MVPMatrices>(mvpBuffer, mvp, 0);
}
```

Project Breakdown

Uwb-vk was written in c++ With Visual Studio 2017. It requires at least c++14 to run. Each project (with the exception of project 0) in the Solution builds upon the previous to culminate in project 12-ShadowMapping, which is the project the final documentation focuses on.

Libraries Used

Several libraries were used to create the project. They are as follows:

- **Vulkan**: This one is self-explanatory. The Vulkan API is the driving force behind the entire project. Currently the project uses Version 1.1 link
- **GLFW**: GLFW is a library used to provide a window and input support link
- **GLM**: This a library that provides vector, quaternion, and matrix math support link
- **stb_image.h**: This a header-only library that provides the ability to load image files link

All of the necessary libraries to compile should be in the “Dependencies” directory.

Project 0

This project is used for compiling shaders from GLSL into SPIR-V so they can be used by uwb-vk applications. If any shaders are out-of-date, be sure to build this project.

Please Note: This project currently relies on Python 3 to run a script compiling all shaders in the Resources/shaders directory. If you do not have Python 3 installed, the project will not work.

Projects 1 - 4 : Following the Tutorial

The first 4 projects are mostly in line with www.vulkan-tutorial.com. Projects 1, 2, 3 and 4 cover the tutorial chapters “Drawing a triangle”, “Vertex Buffers”, “Uniform Buffers”, and “Texture Mapping”, respectively.

There are some notable differences, however. The code that would be in the primary class in the tutorial is instead contained within the **RenderSystem** class, with a new **InputSystem** class to handle keyboard controls. Both are managed by a master **VkApp** class, which will see much more use in later projects. Other pieces related to file input, for example, were also moved to their own files.

New Classes (beyond the tutorial):

VkApp

This class is the highest level object in the application. Setting up of the primary systems resides here, along with any application-specific logic.

RenderSystem

This class was made to contain all rendering logic. In the case of the tutorials, most of the work done as part of the tutorials belong here.

InputSystem

To help demonstrations, input handling with GLFW was introduced and placed in this class. After initialization, InputSystem can be queried for the state of the keyboard (and later mouse), including down, press, and release events.

Texture

This class was made to combine and manage four Vulkan objects that are frequently used together: VkDeviceMemory, VkImage, VkImageView, and VkSampler.

Projects 5 - 7 : Branching out

These project continue with the same goals as in www.vulkan-tutorial.com, however the architecture is in a rapid transition period.

Project 5 (BigRefactor) is a large refactor meant to reduce the messy code bloat in the RenderSystem class, as it was becoming difficult to navigate.

Project 6 (DepthBuffer) adds a depth map component to the RenderPass. This allows the API to only draw the fragments that are closest to the viewpoint. This solves an issue where later fragments were always in front. This corresponds to the “Depth Buffering” chapter in www.vulkan-tutorial.com

Project 7 (ObjLoading) corresponds to the “Loading Models” chapter in the tutorial, but instead, I wrote my own method for loading and parsing the Wavefront .obj format. It can be found in FileIO.h

Note: The loader is limited in its functionality. Notably, it only loads a single mesh, only supports triangles, and does not support materials. As such, I have renamed the extension to “.mesh”

New Classes:

VulkanContext

This class contains all the very core Vulkan constructs and their initialization. In particular, **VkDevice** and other objects that have a lifetime that spans nearly the whole application are used frequently in multiple classes such as the BufferManager and ImageManager classes. **VulkanContext** makes it easy to share these objects.

CommandPool

As the name implies, **CommandPool** holds the **VkCommandPool** object, but in addition, contains functionality for allocating and freeing command buffers, and starting and ending command buffers.

BufferManager

The BufferManager class handles the allocation and freeing of **VkBuffer**s, along with some methods for creating specific types of Buffers.

ImageManager

This class handles allocation and freeing of Image resources. In addition, it allows for operations such as image copying, and transitioning image layouts.

Swapchain

This class contains the main vulkan constructs related to the swapchain, including the associated **VkImage**s and **VkImageView**s.

Projects 8 - 12 : Self-directed

From this point onward, the projects are not driven by www.vulkan-tutorial.com.

Project 8 (Multiple Models) came out of a desire to further generalize the framework, as up until that point, it had been designed with the assumption that only one thing was being rendered to the screen. To remedy this, A model struct was made to contain vertex and index buffers, textures, UBOs and the associated descriptor sets, as well as a pseudo-transform. This was later changed to the Renderable class

Projects 9(Tessellation) and 10(Geometry) are similar in that they explore previously unused shaders in the pipeline. These new shaders prompted the creation of the ShaderSet class, which is used in the pipeline creation method.

Project 11 (Illumination) explores the Phong Lighting Model, along with the use of directional, point, and spot lights. This does not require any significant changes to the RenderSystem, but a Light class was created to contain the data passed to the shaders.

Finally, Project 12 (Shadow Mapping) Explores shadows using a Shadow Map. This is achieved by Adding a second renderpass that runs before the standard pass. This renders the scene from the perspective of the light source, and records a depth map. This map is compared against in the second pass to determine how the surface is lit. For more information look here:

Note: This project is limited to only one light source generating shadows, the framework will have to be modified if you want more shadow-projecting sources.

New Classes:

Renderable

To allow for multiple renderable objects in a scene, this class was created to contain all that could be unique between renderables. This includes:

- A mesh
- A ShaderSet
- Descriptor sets
- A Pipeline layout
- A Pipeline

Mesh

This class was made to hold and manager vertex index buffers as a single unit, as well as handle loading from a “.mesh” file.

Shader

Vulkan Shaders were given their own class including methods for loading the files and getting Specific info needed by the **createPipeline** method.

ShaderSet

A ShaderSet is a collection of shaders used within a specific pipeline. It contains a pointer to a Shader object for each possible stage used, as well as a method for generating a vector of shaderStageInfo objects for use in the **createPipeline** method.

UBO

A Class for managing Uniform Buffer objects. The user can create this with the templated method "**createUnfiormBuffer()**", and it will allocate the **VkBuffer** object properly to handle a **UBO** of the type indicated using the **BufferManager** class.

Camera

Cameras are not an inherent concept to the **RenderSystem**, but is invaluable for demonstrating it. This class contains position and orientation information, and can be queried for view and projection matrices to place in an MVP buffer.

Light

While not critical to the engine, this is a core part of one of the applications. It contains all of the information need to calculate any type of light in a shader. 4 bytes of padding was added to the end to help with alignment issues.

ShadowMap

A class for holding Shadow Map-related vulkan objects. Similar data to a Texture class, but handled differently, so this gets its own class.