UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

KHIN EMMANUEL RODRIGUES BAPTISTA

Shader Tutor: An Interactive Tool for Learning and Exploring Shader Programming

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr. Manuel Menezes de Oliveira
Neto

Porto Alegre
July 2018

ABSTRACT

Learning shader programming often requires the novice to create an entire host application, having to manually load polygonal models and textures, and to deal with graphics API details. This process can be quite discouraging, deviating one's attention from actual shader programming to the development of application infrastructure. To alleviate this load, shader development environments have been created by graphics cards manufacturers, such as NVIDIA's FX Composer and AMD's RenderMonkey, but both have long been discontinued, not supporting modern shading languages. Available on-line resources, like Shadertoy and Shdr, can be valuable tools in assisting the learning of shader programming, but are often difficult to use, have limited feature sets, and/or lack proper documentation to get beginners started. We present a complete environment for learning and exploring shader programming using GLSL. Our environment, developed using Vulkan, can suit the needs of both beginners and advanced users. Its user-friendly interface allows one to effortlessly load 3D models and images and modify variables and textures, applying the changes in real time.

Keywords: Shader programming. GLSL. vulkan. computer graphics.

Uma Ferramenta Interativa para o Aprendizado e Exploração de Programação de Shaders

RESUMO

O aprendizado de programação de shaders comumente requer que o novato crie uma aplicação hospedeira completa, carregando manualmente modelos poligonais e texturas e lidando com detalhes da API gráfica. Esse processo pode ser bastante desencorajador, desviando a atenção da programação de shaders para o desenvolvimento da infraestrutura da aplicação. Para aliviar esse trabalho, ambientes de desenvolvimento de shaders foram criados por fabricantes de GPUs, como o FX Composer da NVIDIA e o RenderMonkey da AMD, mas ambos foram descontinuados há bastante tempo, e não oferecem suporte a linguagens modernas de shading. Recursos disponíveis on-line, como Shadertoy e Shdr, podem ser ferramentas valiosas para ajudar no aprendizado de programação de shaders, mas costumam ser difíceis de utilizar, possuem um conjunto de funcionalidades limitado e/ou não oferecem documentação adequada para iniciantes. Nós apresentamos um ambiente completo para o aprendizado e exploração de programação de shaders utilizando GLSL. Nosso ambiente, desenvolvido usando Vulkan, é capaz de suprir as necessidades de usuários iniciantes bem como experientes. Sua interface amigável permite que o usuário carregue modelos 3D e imagens sem dificuldade, além de modificar variáveis e texturas, aplicando as mudanças em tempo real.

Palavras-chave: Programação de shaders. GLSL. vulkan. computação gráfica .

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

GPU	Graphics Processing Unit
API	Application Programming Interface
SDK	Software Development Kit
OpenGL	Open Graphics Library
GLSL	OpenGL Shading Language
HLSL	High-Level Shading Language
SPIR	Standard Portable Intermediate Representation
GLFW	Graphics Library Framework

CONTENTS

1 INTRODUCTION

Shader programming is an essential part of modern computer graphics. However, in order to write her/his first shader program, one is often faced with the need to write an entire host application that will supply data for the shader. This includes establishing associations between variables from the host program and from the shader, loading polygonal models, setting parameters, initializing textures, etc. All this can be quite discouraging for a novice, deviating one's attention from actual shader programming to the development of application infrastructure.

To make it easier for newcomers, we have built an application that provides a complete environment for shader programming. With this tool, one will be able to write shaders without having to worry about such infrastructure details. By being able to focus on the shaders themselves and get results faster, we hope to encourage more students to learn computer graphics. Developing such a tool also had another important personal goal: serve as an opportunity for learning Vulkan, a modern graphics API.

Our application allows students to easily prototype shaders using an integrated source code editor, and to view the results in real time in the visualization window, which can include any objects the user wants to add, including custom meshes that can be imported. While implementing this tool, we were able to understand the inner workings of Vulkan, an API which is growing fast and becoming more present in the industry of computer graphics.

This thesis is organized as follows: Chapter 2 presents the reader with a base knowledge about computer graphics and how the modern graphics applications perform their functions, as well as introduces Godot Engine, the tool used to create our application. Chapter 3 introduces important details of the Vulkan API and how it differs from OpenGL. Chapter 4 describes the methods applied in the implementation of our tool, the architecture of our application and how to use it from a user perspective. Chapter 5 presents the results we were able to obtain using our solution. Chapter 6 concludes this work, presenting our view of the tool developed. Chapter 7 discusses how our application could be improved in the future.

1.1 Related work

As we mentioned, many attempts were made to create development environments for shaders.

RenderMonkey, developed by AMD, aimed to provide tools for both programmers and artists. It featured a complete shader source editor and preview window, but also a workspace specific for artists, where much of the shader programming was hidden from the user, widgets used to adjust parameter values. It presented support for HLSL, GLSL and GLES shading languages (AMD, 2008b). The latest version of the software was released in 2008 (AMD, 2008a) and AMD's website no longer displays any information about the software or that it ever existed.

FX Composer, developed by NVIDIA, featured an integrated source code editor, a scene visualizer which could use DirectX 10, DirectX 9 or OpenGL, widgets for editing parameters, particle systems, material manager, material preview and much more (NVIDIA, 2008a). The software has not been updated since 2008 (NVIDIA, 2008b), presenting many usability issues and instability, such as debug messages pointing to errors that were not from the user code but from the software architecture. This is part of the motivation to create this work.

Shadertoy is an online resource which allows users to write only fragment shaders to create effects. **Shdr** is another online resource to write shaders, both vertex and fragment shaders, but the user is not allowed to define uniform variables in the shaders (parameters to change the results generated by the shaders). The lack of features on both applications prevents the user from creating a large variety of effects.

ShaderLabFramework (TOISOUL; RUECKERT; KAINZ, 2017) is a tool created for the undergraduate Computer Graphics course at the Imperial College London, UK, and was tailored to their course syllabus. It implements a two-pass rendering pipeline structure, allowing users to render to texture and applying post-processing effects. This application was implemented in C++ using the Qt framework for user interface, and uses OpenGL to render the visualization scene.

Unlike the above tools, Shader Tutor allows the user to load arbitrary geometry and define a variety of shader parameters and textures for both the vertex and the fragment shaders, all of which are displayed in the control window for the user to change. It is implemented using the Vulkan graphics API, and provides an integrated GLSL code editor and compiler. The editor was built using Godot instead of frameworks such as Qt.

2 BACKGROUND

In this chapter, we will introduce the reader to Godot, the tool used to create our application, to GLFW, a supporting library used in our scene visualization window, and to important concepts of computer graphics required for the complete understanding of this thesis. Concepts from linear algebra like vector maths, matrix operations and geometric transformations are presumed to be known.

2.1 Godot Engine

Godot Engine is a 2D and 3D game engine, free and open source under the MIT license. It provides a set of common tools for game development, including many GUI widgets, like buttons, menus, panels and text editors, which is exactly what we needed to build the control window of our application in a way that is very easy to use and to script, while being able to deploy to different platforms. The engine uses its own programming language for scripts, called GDScript, which is similar to Python and allowed us to script the behavior of the control window without worrying about including new libraries to the project.

2.1.1 GDNative

GDNative is a module of Godot Engine which adds a new "scripting language" to it (Thomaz Herzog, 2017). In reality, it is not a language, but a precompiled dynamic library that will be loaded into the Godot application. GDNative allows developers to access the Godot API and create scripts using different languages, for example C and C++, and use these scripts inside Godot as if they were scripts native to the engine.

There are two big use cases for GDNative:

- To allow developers to write **performance critical code** in more efficient languages than GDScript (the scripting language used by Godot Engine). GDScript is a good scripting language, but it was not built for performance. An example task that fits this scenario is generating terrain procedurally;
- To allow developers to **bring third-party code to Godot**. This is useful in game development in general to integrate useful libraries like Steamworks and Google

Play Services. In our case, we used it to bring Vulkan code into the application.

GDNative is provided as a C API. In order to use other languages, bindings to the original API must be created.

2.2 GLFW

GLFW is an Open Source, multi-platform library for OpenGL, OpenGL ES and Vulkan development on the desktop. It provides a simple API for creating windows, contexts and surfaces, receiving input and events (GLFW, 2016).

2.3 Graphics pipeline

In the field of computer graphics, the **graphics pipeline** is used to generate 2D images out of 3D geometric information in real time applications (SHIRLEY, 2002). Figure 2.1 shows a diagram describing the stages of the graphics pipeline. Each stage of the pipeline is responsible for a specific operation; the output of each stage is passed to the next stage to be used as input.

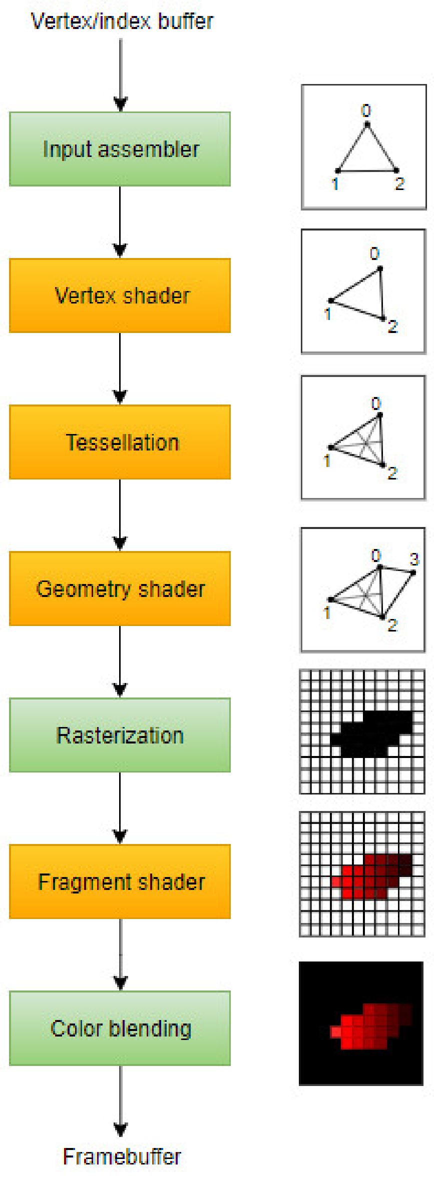
The function performed by each of the stages in the pipeline are described as follows (Alexander Overvoorde, 2016):

- Input assembler: Collects the raw vertex data from the vertex buffer and may also use an index buffer to repeat certain elements without having to duplicate the vertex data itself;
- Vertex shader: Executed for every vertex in the object, generally applies transformations to turn vertex positions from model space to screen space;
- Tesselation: Optional stage which subdivides geometry based on certain rules to increase mesh quality;
- Geometry shader: Optional stage executed for every primitive (point, line, triangle) which is able to discard or create new primitives;
- Rasterization: Breaks down the primitives into fragments, which are the pixel elements that will make up the rendered image. Fragments located outside the screen space are discarded, and vertex data is interpolated across the fragments;
- Fragment shader: Invoked for every fragment, determines its color based on the

interpolated vertex data outputted by the rasterization stage;

- Color blending: Applies operations to mix different fragments that map to the same pixel in the resulting image, usually based upon transparency.

Figure 2.1: Graphics pipeline



Source: (Alexander Overvoorde, 2016)

Certain stages can only be controlled by changing their parameters, but the function performed is always the same. These are known as *fixed function stages* and are the ones in green in Figure 2.1.

The other stages, in orange, are known as the *programmable stages*, which means developers can upload source code to the graphics card to apply exactly the operations they intended for that stage.

From the programmable stages, tesselation shaders and geometry shaders are optional, meaning that their functions can be skipped and the pipeline can still work properly. On the other hand, the *vertex shader* and the *fragment shader* are required from the developer and the graphics pipeline cannot function without them.

Shaders always receive data from the previous stage of the pipeline, but they can also define additional parameters to be used in their computations. These parameters are called *uniform variables*. Uniform variables can have their values changed from one draw call to the next, but for every time the shader is executed, the uniform variable will have the exact same value. These variables can be used to, for example, tell the shader where the lights are located in the scene, or provide an image to be used as texture. These values must be supplied from the host application, which manages the graphics pipeline.

The following code represents the basic operation of a vertex shader (in the GLSL language):

```
uniform mat4 modelViewProjection;
in vec4 position;
out vec4 clip_position;
void main() {
    clip_position = modelViewProjection * position;
}
```

The first line defines a uniform variable which is a matrix that will transform vertices from local space to clipping space (a normalized space required by the next stages of the pipeline). This matrix accumulates model, view and the projection matrices. The model matrix applies rotation, scale and translation to the object's vertices, transforming them from local space to world space. The view matrix transforms the vertex positions from world space to view space, where the camera is at the origin. Finally, the projection matrix will deform the geometry to apply perspective or orthogonal projections.

The second line describes the vertex attribute that the vertex shader expects to receive from the input assembler. This simple example expects just the vertex position in object space, but it could require other attributes, such as normal vectors or texture coordinates.

The third line describes the output of this shader. Since it is a vertex shader, the vertex position in clip space is required to be an output. Other output variables can be defined, and these variables will be interpolated by the rasterizer and delivered to the fragment shader.

The following code represents the basic operation of a fragment shader:

```
uniform vec4 frag_color;
out vec4 final_color;
void main() {
    final_color = frag_color;
}
```

The purpose of the fragment shader is to output the color of the fragment. This sample code does just that: a uniform variable is used as color for the fragment. The `vec4` variable type can be used to represent colors because it holds four floating point values, which are interpreted as red, green, blue and alpha channels.

2.4 Graphics APIs

As we have mentioned, graphics applications require a host application to manage its operation. The means used to do that is through a graphics API. Graphics APIs are a set of routines implemented by graphics cards manufacturers that allow developers to control the hardware. Examples of graphics APIs include Direct3D, OpenGL, Metal and Vulkan. Each API defines their own set of methods and how they are used, and it is up for hardware manufacturers to support the APIs. Some APIs, such as Direct3D and Metal, are proprietary to enterprises (Microsoft and Apple, respectively) and can only be used on their platforms. Others, like OpenGL and Vulkan, are open standards, both provided by the Khronos group, an industry consortium.

Vulkan is a relatively new technology, released version 1.0 in 2016. Since then, several games have been ported to this API, and many game engines have developed rendering backends using Vulkan (Khronos, 2018b). This indicates that Vulkan may become the standard graphics API to real-time graphics applications. This work uses the Vulkan API to fulfill a personal goal of creating a functional piece of software using this technology.

3 VULKAN

In this chapter, we will introduce the reader to the Vulkan API, its motivations and goals, as well as present the objects of the API that are fundamental to building not only Shader Tutor, but any application using Vulkan.

Vulkan is a new generation graphics and compute API that provides high-efficiency, cross-platform access to modern GPUs used in a wide variety of devices from PCs and consoles to mobile phones and embedded platforms (Khronos, 2018b). Vulkan is a much lower level API compared to OpenGL (SELLERS, 2016). It requires the developer to deal with a lot of different hardware related aspects, from enabled GPU features to video memory management.

Vulkan also presents some very interesting features designed to solve problems of older APIs. For example, an OpenGL application has to explicitly tell the GPU what operations are executed to render the scene every frame (Segal, M. Akeley, K., 2010), which consumes a lot of valuable CPU time that could be used for other purposes depending on the application's nature (for example, a game could use this process time to compute artificial intelligence behavior for non-playable characters). Vulkan improves this by requiring the application to record all drawing commands in advance and possibly using multiple threads, and storing the commands in immutable objects which can be stored in memory optimal for reading. Then, each frame, the application just needs to tell the GPU to execute the commands in those objects, saving CPU time.

3.1 Debugging: validation layers

Because graphics operations are executed on the GPU, debugging becomes a difficult task. Device drivers for older APIs included extensive error checking mechanisms in order to produce meaningful error messages to help developers. The Vulkan API, however, is designed around the idea of minimal driver overhead and one of the manifestations of that goal is that there is very limited error checking in the API by default. Even mistakes as simple as setting enumerations to incorrect values or passing null pointers to required parameters are generally not explicitly handled and will simply result in crashes or undefined behavior (Alexander Overvoorde, 2016). Instead, Vulkan comes with a set of validation and debug layers as part of the Vulkan SDK and, when any subset of these layers are enabled, they insert themselves automatically into the call-chain of every Vulkan

API call issued by the application to perform their job. Validation layers can also report warnings about potential incorrect or dangerous use of the API, and are even capable of reporting performance warnings that allow developers to identify places where the API is used correctly but not used in the most efficient way (Daniel Rakos, 2016). This allows developers to enable validity checks in debug builds to make sure that the application is built correctly, but disable them for release builds, allowing the API to run without doing any checks, improving performance.

3.2 Compiling shaders

Up until OpenGL 4.6, applications had to compile shader source code at run-time, passing a string of high-level source code to the graphics driver. While this allowed the application to run on different hardware, this also forced each GPU manufacturer to provide a GLSL compiler with the device driver. The development of the updated drivers to support new versions of the APIs would take longer, or, in some cases, support for newer versions was simply not provided (HELPS, 2018).

3.2.1 SPIR-V

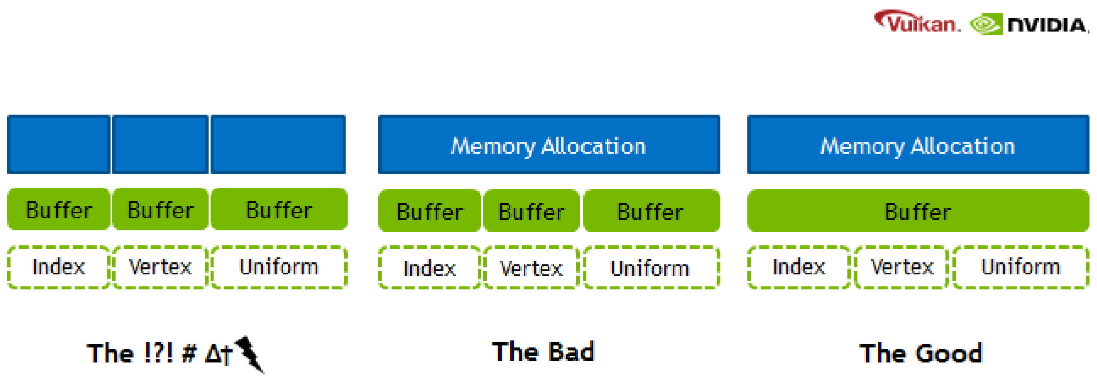
The "Standard Portable Intermediate Representation" version "V" (as in "Vulkan") is a binary representation of shader code which device drivers can parse more easily. Application developers, having their shaders written, must compile them into SPIR-V format before loading them in their applications. Support for SPIR-V shaders is included in OpenGL 4.6 and Vulkan 1.0 (KHRONOS, 2018).

Vulkan is not concerned about the language which was used to write the shader, as long as the provided SPIR-V code is valid. This allows shaders to be written in any shading language as long as it can be compiled into SPIR-V, which gives more flexibility to the API.

3.3 Memory management

While OpenGL drivers manage memory allocations transparently to the programmer, Vulkan not only allows but also requires that programmers allocate device memory and bind the memory to the resources used in the application. While this may increase the application's performance if the programmer takes advantage of it, it also requires the programmer to manually manage buffer alignments and aliasing. In Vulkan, a buffer object represents a linear array of data which can be used for various purposes, and require memory to be allocated and bound to it. Figure 3.1 shows three memory allocation strategies possible with Vulkan (NVIDIA, 2016).

Figure 3.1: Vulkan memory allocation strategies



Source: Adapted from (NVIDIA, 2016)

The first to the left is the naive approach: for each buffer it uses a "dedicated" memory allocation. This approach is the easiest to implement, since every object allocates, binds and frees their own memory, without taking into account other objects. It is also the least optimized approach, given that it most likely will not be cache-friendly. It is also important to note that Vulkan devices have a maximum number of memory allocations that can exist simultaneously.

The approach shown in the middle ("The Bad") shows a single memory allocation with various buffers bound to it. This approach is more cache-friendly than the previous one, because with a single allocation it is more likely to have different buffers loaded in the GPU cache, but it requires the programmer to deal with memory aliasing, offsets and buffer alignments.

The rightmost ("The Good") is the recommended approach. It represents a single memory allocation bound to a single buffer, with different data loaded to different areas of the buffer. This approach is the most cache-friendly, because a single buffer is more likely to load different types of data to the CPU cache, and the most optimized for performance in highly-dynamic scenes, but also the most difficult to implement. This setup is possible thanks to Vulkan low-level control of offsets even inside a single buffer, allowing the developer to define "virtual buffers".

The tool developed in this work was intended to allow the user to write shaders with few objects, for the purposes of visualization only. For this reason, we opted to use the naive approach. Implementing a memory management module would not have a perceived impact on the performance of the application.

3.4 Host application

Vulkan is a low-level API and, as such, requires developers to setup a large amount of objects and configuration values before displaying anything on screen. In this section we will explain the purpose of each of these objects.

3.4.1 Vulkan instance

There is no global state in Vulkan and all per-application state is stored in a 'VkInstance' object. Creating this object initializes the Vulkan library and allows the application to pass information about itself to the implementation (Khronos, 2018a). Applications can even define and manage multiple instances depending on their requirements.

In order to initialize a Vulkan instance object, we can first create a 'VkApplicationInfo' object, which will describe our application for the driver. This structure holds the name and version of the application, and also the name and version of the engine used, if any. This data is technically optional, but it may provide some useful information to the driver to optimize for specific applications (Alexander Overvoorde, 2016). Unlike the 'VkApplicationInfo', the 'VkInstanceCreateInfo' structure object is not optional. It describes the instance extensions and debug layers we want to enable in our application. To make sure our application can be executed, we can check the available extensions with the function 'vkEnumerateInstanceExtensionProperties' to compare against our required extensions.

During the Vulkan instance object creation we can also specify validation layers to be enabled. Available validation layers can be queried with the 'vkEnumerateInstanceLayerProperties' function. The Vulkan SDK provides a standard validation layer, "VK_LAYER_LUNARG_standard_validation", which implicitly enables a range of useful diagnostic layers. In order to receive the debug messages from the validation layers, however, the application must also enable the "VK_EXT_debug_report" extension, which will allow the developer to setup a callback function that will be called whenever a debug message is issued by the validation layers. To setup the debug callback function, the developer has to get a pointer to the ‘vkCreateDebugReportCallbackEXT’, which is an extension function and therefore not included in the base API, using the ‘vkGetInstance-ProcAddr’ function.

3.4.2 Physical device

The function ‘vkEnumeratePhysicalDevices’ returns a list of available Vulkan devices installed in the system. Basic device properties like the name, type and supported Vulkan version can be queried using ‘vkGetPhysicalDeviceProperties’. The support for optional features like texture compression, 64 bit floats and multi viewport rendering can be queried using ‘vkGetPhysicalDeviceFeatures’. Device features must be enabled before they are used by the application.

Almost every operation in Vulkan, from drawing to uploading textures, requires commands to be submitted to a queue. A queue belongs to a *queue family*, and each queue family supports certain types of commands. To query queue families available in the device, there is a ‘vkGetPhysicalDeviceQueueFamilyProperties’ function.

3.4.3 Device

The function ‘vkCreateDevice’ takes a physical device queried and a ‘VkDevice-CreateInfo’ object as arguments. The latter describes which physical device features we want to enable from our selected physical device and specifies the queue family index and how many queues we want to create from each family. The application does not need to explicitly create the queue objects; after the device is created, the application can call the ‘vkGetDeviceQueue’ function to get the ‘VkQueue’ objects to which commands will be passed into for execution.

3.4.4 Window surface

In order to display images, the Vulkan API must interface with the platform specific window system. Also, since Vulkan is also a compute API, this functionality is not in its core specification, but is part of the "VK_KHR_surface" extension. Window systems also vary for each platform so, in order to create the proper surface object, we also need to enable the platform-specific surface extension ("VK_KHR_win32_surface" for Windows, for example).

3.4.5 Swap chain

Unlike OpenGL, Vulkan does not have the concept of a "default framebuffer", hence it requires a structure to own the buffers where images will be rendered before they are displayed on screen. This structure is the swap chain, and it must be explicitly created. This requires the "VK_KHR_swapchain" device extension enabled. The swap chain is essentially a queue of images that are waiting to be presented on the screen.

Each physical device presents different capabilities in its swap chain. These include the maximum number of images the swap chain can hold, the pixel formats supported and presentation modes. The presentation modes are (Alexander Overvoorde, 2016):

- `VK_PRESENT_MODE_IMMEDIATE_KHR`: Images submitted by your application are transferred to the screen right away. May result in tearing;
- `VK_PRESENT_MODE_FIFO_KHR`: The swap chain is a queue where the display takes an image from the front of the queue when the display is refreshed and the program inserts rendered images at the back of the queue. If the queue is full then the program has to wait. This is most similar to vertical sync as found in modern games. This is the only mode that is guaranteed to be supported;
- `VK_PRESENT_MODE_FIFO_RELAXED_KHR`: This mode only differs from the previous one if the application is late and the queue was empty at the last vertical blank. Instead of waiting for the next vertical blank, the image is transferred right away when it finally arrives. This may result in visible tearing;
- `VK_PRESENT_MODE_MAILBOX_KHR`: This is another variation of the second mode. Instead of blocking the application when the queue is full, the images that

are already queued are simply replaced with the newer ones. This mode can be used to implement triple buffering, which avoids tearing with significantly less latency issues than standard vertical sync that uses double buffering.

3.4.6 Depth resources

In Vulkan, the depth buffer is an image object which the developer has to create explicitly using special formats that are defined for depth images. Device memory must be allocated for the image and bound to it, and an image view also needs to be created to describe how the image will be used. After creation, the image must also be transitioned to a layout optimal for depth attachment before it can be used as depth buffer.

3.4.7 Render pass

A render pass represents a collection of attachments, subpasses, and dependencies between the subpasses, and describes how the attachments are used over the course of the subpasses (Khronos, 2018a).

3.4.8 Framebuffer

The attachments specified in the render pass creation are bound into a framebuffer object. This object references all the image view objects that represent the attachments in the render pass. Since the swap chain has many different images (and image views), applications must also create a framebuffer for each image view of the swap chain.

3.4.8.1 Image vs. Image View vs. Framebuffer

In Vulkan, an **image** represent multidimensional arrays of data which can be used for various purposes. An **image view** represent contiguous ranges of an image object and contain additional metadata to allow image objects to be accessed by pipeline shaders for reading or writing image data. A **framebuffer** represents a collection of specific memory attachments that a render pass instance uses.

3.4.9 Command buffers and command pools

As we mentioned, operations like drawing and memory transfers are not executed with function calls in Vulkan. Instead, the commands are recorded in command buffers, which are then submitted to a queue for execution.

Command buffers are not created directly, but they are allocated from command pools which will manage the memory used by the command buffers. Each command pool can only execute commands on a single queue family; command buffers allocated from a pool cannot simply execute any command, because the commands will be executed by a queue family which may not support the operations.

3.4.10 Buffers

Data is passed from the host application to the graphics pipeline via buffer objects. Buffers must have device memory allocated and bound to them in order to work, and will be used to pass vertex attributes, vertex index data when available, and uniform values. Uniform buffers are generally updated every frame.

3.4.11 Images, image views and samplers

Much like buffers, in order to be accessed by shaders, images must be created and device memory must be allocated and bound to them. There is also the need to create an image view, an object that describes how to access the image and which part of the image to access. Sampler objects will determine how the shaders will sample the pixels of the image, determining filtering and repeat mode, for example. In Vulkan, images and samplers are distinct objects, which allows the same image to be accessed in shader code with different samplers, using different configuration values. This feature can be used to reduce memory usage.

3.4.12 Descriptor set layout, descriptor pool and descriptor set

The descriptor set layout describes the bindings that are used in the shaders, whether they bind a uniform buffer object or an image sampler, and at which stages each binding is available.

The descriptor pool holds a pool of descriptor sets to be allocated; it requires the number of uniform buffers and image samplers, but is not concerned about the size of these structures.

The descriptor set layout is just a description of what types of descriptors can be bound, but no uniform buffers are referenced. The descriptor set specifies a buffer to be bound in the uniform buffer descriptor.

3.4.13 Pipeline

In order to create the 'VkPipeline' object, one first needs to create a 'VkPipelineLayout' object. The pipeline layout defines the interface of the pipeline object with our application, that is, it holds a reference to the descriptor set layout object. Developers also create objects that will configure every stage in the graphics pipeline:

- `VkPipelineVertexInputStateCreateInfo`
- `VkPipelineInputAssemblyStateCreateInfo`
- `VkPipelineViewportStateCreateInfo`
- `VkPipelineRasterizationStateCreateInfo`
- `VkPipelineMultisampleStateCreateInfo`
- `VkPipelineDepthStencilStateCreateInfo`
- `VkPipelineShaderStageCreateInfo`
- `VkPipelineColorBlendStateCreateInfo`

Each of the structures mentioned above have a set of possible configurations which will determine, for example, the topology used when interpreting the vertex data (points, lines, triangles), which polygon mode to use in rasterization (points, lines, fill), line width, back face culling, depth comparison operation and alpha blending.

4 METHOD

In this chapter we explain how we used the technologies and tools described so far to create our application. We start by describing the architecture of our software and how it works internally. Then, we describe how the functions implemented are presented to the final user such that it helps to learn shader programming.

4.1 Architecture

In this work, the most important decision of the architecture was the choice of using Vulkan for the graphics API. Using Vulkan has been a learning process, and the architecture chosen reflects it.

In order to have our Vulkan code respond to changes in the control window, some of the classes in our architecture are exposed to GDScript via GDNative. GDNative requires the developer to register each method and property that will be made available for engine scripts to call, which allows our application to provide a minimal interface, keeping the implementation encapsulated, separated from the GUI in the control window.

Since GDNative loads a precompiled dynamic library, it is required for the developer to provide the library for the platforms to which the project is to be deployed. The development of our application was completely done in Linux, and thus it is the only platform the library has been compiled into, but our code does not rely on any platform-dependent libraries, so compiling to Windows or MacOS should be straightforward.

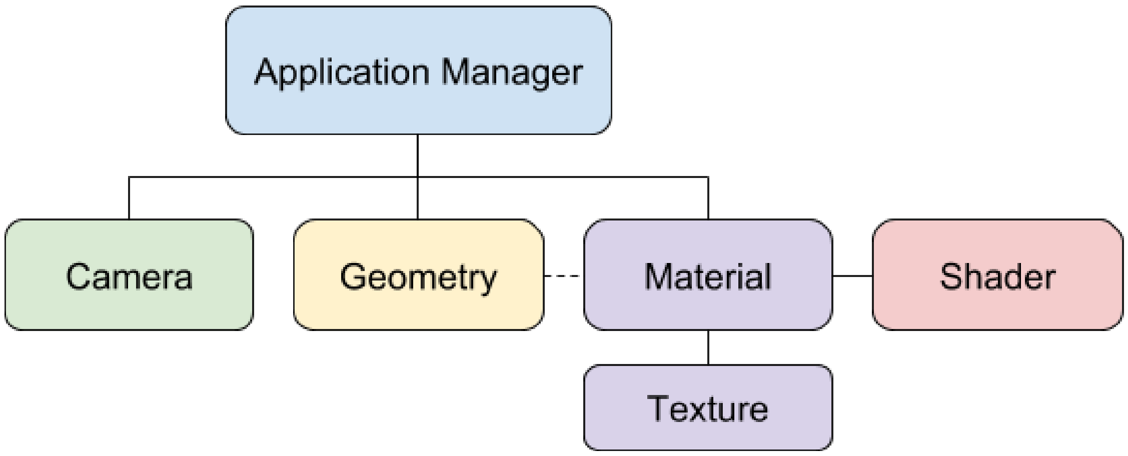
The architecture of Shader Tutor is depicted in Figure 4.1. The purpose of each module is described in the following sections.

4.1.1 Vulkan Application Module

This module is responsible for creating the objects which are global to the application. First, it creates the GLFW window that we will use to render the Vulkan scene. It also registers the mouse button callback and the mouse motion callback, which will be called by the GLFW library so that our application can react to user interaction, mostly to move the camera.

To create the Vulkan instance, the extensions our application will enable are those

Figure 4.1: Shader Tutor architecture



Source: the author

listed by ‘glfwGetRequiredInstanceExtensions’, a convenient method provided by GLFW to query the required extensions for our window to work properly. We do not need any other instance extensions.

With a list of the available devices, their features and queue families properties, we can select a ‘VkPhysicalDevice’ and create a logical device to interface with it. For a rendering application like ours, we need a physical device which presents a queue family able to render graphics, and another able to do presentation (displaying images in the window). These queues could be the same, if the physical device presents a single queue family with both graphics and presentation capabilities.

Creating the window surface is easy thanks to GLFW, which enables the required extensions with ‘glfwGetRequiredInstanceExtensions’, and provides a simple ‘glfwCreateWindowSurface’ function which simply returns a surface appropriate for the current platform.

For swap chain creation, we select the mailbox present mode if available; if not, we use the FIFO, which is the only setting guaranteed to be supported. We also create the depth resources, render pass and framebuffers. After the command pool is created, our application can allocate command buffers from it.

For the drawing commands, we will have to issue a command to bind the correct framebuffer object and, since we have more than one framebuffer (one for each image in the swap chain), we also have to allocate multiple command buffers, and record the drawing operations in all of them.

Whenever a new object is added to the scene, or a material is assigned to an object, the drawing operations must be re-recorded. To record the drawing commands, we will begin the command buffer with the flag "command buffer usage: simultaneous use" set, which allows our application to request execution of the command buffer before a previous execution has finished. With the command buffer initialized (after a command buffer begin function call), we begin a render pass, which will refer to the render pass object we created beforehand, the framebuffer which will be used, and the clear values used for the color and depth attachments. Then, the render pass can begin (‘vkCmdBeginRenderPass’ function). The drawing commands are then recorded for each geometry in the scene. After the commands are recorded, the render pass must be ended, as well as the command buffer itself.

4.1.1.1 Application Loop

For every frame, there is a sequence of commands we have to execute. Since this module manages the Vulkan window, we first need to handle any input events from the window. We also poll for certain keyboard events to move the camera. We do this instead of registering a callback with GLFW because keyboard events have delay and repeat configurations, which makes the camera movement stiff and unpleasant; the poll approach allows us to check keys’ status every frame, creating a smooth camera movement.

After our application handles the input events, certain variables are updated to be passed to the shaders as uniform variables. These include the camera view and projection matrices and their inverted counterparts. These variables must be passed to all the objects rendered in the scene so that their uniform buffers will be up to date for rendering.

Vulkan rendering is comprised of two different steps: first, we have to render to one of the swap chain images, and then we have to display it in the window surface. To begin rendering, we have to acquire the next image available from the swap chain. Since the command executions happen in the GPU, rendering is asynchronous, which requires us to use a semaphore (provided by the API) which will be signaled when there is an image available for rendering. We then use the graphics queue to execute the command buffer of the image acquired from the swap chain.

After rendering, we will use the present queue to display the image on the window surface. Again, rendering happens in the GPU and our application must make sure the image is available for presentation, so we have to use a second semaphore, signaled when rendering is finished in the graphics queue, to which the present queue will wait before executing.

4.1.1.2 Single Time Commands

Certain common operations are executed in different parts of code, like copying buffer data, transferring buffer data to image and transitioning image layout. These operations, in Vulkan, must be executed in a queue by a command buffer. The commands must be recorded in advance, and then the command buffer can be executed. These operations, however, are not part of the render loop; they are required to take place during initialization of resources. For this end, since command buffers must be allocated from a command pool which is managed by the Vulkan application module, there are commands to setup a command buffer to perform any of these commands, and then to submit them to the graphics queue for execution before they are freed.

4.1.1.3 Cleaning Up

During cleanup, a lot of Vulkan resources have to be appropriately freed and destroyed, respecting their dependencies. First, the application must wait for the device to become idle, i.e., there are not any pending operations to be executed. Then, resources can be freed. Our application begins destroying the depth image view, depth image, and then freeing the device memory used by the depth image. Next, the semaphores used to signal when an image is ready and when a presentation is done are destroyed. Then, the command pool is destroyed, which automatically destroys the command buffers allocated from it. Next, the framebuffers are destroyed, followed by the render pass object and image views of the swap chain images. The swap chain is then destroyed as well, followed by the debug callback (when the application is compiled in debug mode), the window surface and finally the Vulkan instance. Last, the GLFW window is destroyed before the application is terminated.

4.1.2 Shader Module

A shader object represents one stage of the programmable pipeline stages. This module has a type indication, either vertex or fragment, and is capable of loading GLSL source files and compiling them into SPIR-V binary using the ShaderC library, developed by Google (GOOGLE, 2018). The compiled code is then enclosed by a ‘VkShaderModule’ object which will be used in a material’s graphics pipeline. The type of the shader is inferred when a GLSL shader file is loaded based on the extension of the shader file,

‘.vert’ for a vertex shader and ‘.frag’ for a fragment shader. This module also provides ways to check the compilation status and get any compilation error messages.

4.1.3 Material Module

A Material is a combination of a vertex and a fragment shader, plus the values of the uniform variables defined in those shaders. It is responsible for the creation of the Vulkan graphics pipeline object, based on the shaders loaded into it and other configuration values.

4.1.3.1 Shader Parsing

The shaders added to the material will have their SPIR-V code parsed by the *SPIRV Cross* library. Our application will use this data to load the appropriate vertex attributes, uniform objects and sampler objects used by the shaders.

Shaders define which vertex attributes will be used as input, and uniform variables can serve as parameters for the material. All vertex attributes must be recognized by our application, following the defined names and types defined in table 4.1. The user can also query the application for certain uniform variables, following the names and types defined in Table 4.2.

Table 4.1: Recognized vertex attributes

Type	*Name*	*Description*
vec3 or vec4	`position`	Vertex position
vec3 or vec4	`normal`	Normal vector of the surface
vec3 or vec4	`tangent`	Tangent vector to the surface
vec2	`uv`	Texture coordinate

If the `position` attribute is a vec4, the fourth component will be set to one; If the `normal` is a vec4, the fourth component will be set to zero; If the `tangent` is a vec4, the fourth component will be either set do 1 or -1, indicating the direction pointed by the binormal vector, used in normal mapping.

Note that the normal matrix is a 3x3 matrix, unlike the other ones. This is because the correct matrix to transform the normal vector is the transpose of the inverse of the 3x3 submatrix from the model view matrix (LIGHTHOUSE3D, 2011).

The shader code can also define arbitrary uniform variables, which will be initialized to a default value and exposed in the control window so the user can change the

Table 4.2: Recognized uniform variables

Type	*Name*	*Description*
mat4	`model`	Model matrix
mat4	`view`	View matrix
mat4	`projection`	Projection matrix
mat4	`modelView`	Model-View matrix
mat4	`modelViewProjection`	Model-View-Projection matrix
mat4	`modelInverse`	Model matrix inverse
mat4	`viewInverse`	View matrix inverse
mat4	`projectionInverse`	Projection matrix inverse
mat4	`modelViewInverse`	Model-View matrix inverse
mat4	`modelViewProjectionInverse`	Model-View-Projection matrix inverse
mat3	`normalMatrix`	Matrix to transform the normal vectors of the object; Corresponds to the inverse transposed 3x3 model-view submatrix

values manually. Numeric values are initialized to 1 to avoid division by zero. Image samplers defined in code will be initialized to a white texture.

4.1.3.2 Descriptor set layout and descriptor pool

After the shaders are parsed, the Material module will create the descriptor set layout object ('VkDescriptorSetLayout') and descriptor pool. In our case, every uniform buffer object and image sampler is available in all graphics stages (vertex and fragment). For this reason, the shaders must declare the binding of each uniform buffer object and image sampler used in GLSL code, and the binding numbers cannot be reused. This is done in GLSL by using the layout qualifier in the declaration of the uniforms.

For our application, the size of the descriptor pool translates into how many different objects can have the same material applied at a given time.

4.1.4 Geometry Module

The geometry module represents an object in the scene. It holds all the vertices attributes (vertex position, normal, tangent, texture coordinates and indices, if available); a transformation, which defines the rotation, scale and translation applied to the object; and a reference to a material object which will be used to render the geometry in the scene.

When an object is added to the scene, it must be filled with vertex data and then a material has to be assigned to the object. Then, the object will get a list of the required vertex attributes from the material, and create the vertex buffer accordingly. If the vertex attributes include index data, the index buffer is created, too. Then, the uniform buffers are created, based on the uniform buffer objects in the material assigned.

For the uniform buffer to be passed to the shaders, we need to allocate a descriptor set from the pool defined in the material assigned to the geometry.

4.1.4.1 Recording Drawing Commands

Since all drawing operations are recorded in advance in Vulkan, this module also provides a method for recording the drawing commands in a command buffer previously initialized.

- `VkCmdBindPipeline`: Binds the graphics pipeline of the material applied to the object;
- `VkCmdBindVertexBuffer`: Binds the vertex buffer of the object, which contains the vertex attributes;
- `VkCmdBindDescriptorSets`: Binds the descriptor set which contains the uniform variables data;
- `VkCmdBindIndexBuffer`: If the object presents index data, binds the buffer containing the indices;
- `VkCmdDrawIndexed`: Draw the object using the index buffer;
- `VkCmdDraw`: Draw object without indices.

4.1.4.2 Updating Uniform Buffers

The function to update uniform buffers is called by the Vulkan application module, and expects a structure containing certain uniform values that are not dependent on the object, such as the camera's view and projection matrices. The other matrices are calculated and loaded into the appropriate buffers.

The material can define arbitrary uniform variables, and those must be updated as well. The material holds an array of dictionaries that describe each of the user-defined variables, their types, sizes, bindings, offsets and values. With this information, our application is able to load the variable value into the correct buffer in the correct offset.

4.1.5 Texture Module

The texture module handles the passage of image data for the shaders. This means creating an image, allocating device memory for the image, creating the image view and the sampler.

Image files are loaded by Godot, the engine used to create the user interface, and their data is passed to the texture module. Our application then has to pass this data into the device memory bound to the image. However, there is no way to write directly to the image memory, which forces us to use a *staging buffer*. The staging buffer can be mapped to CPU memory and accessed by our application. In order to transfer this data to the image, the image must have its layout transitioned to optimal for transfer destination. This is done using a single time command buffer, which is allocated, recorded, executed and freed, sequentially. Single time command buffers are managed by the Vulkan application module. Once the image is ready to receive a data transfer and the staging buffer is loaded, we can use another single time command buffer to copy the buffer contents into the image. After this transfer is complete, the image must be transitioned again to a layout optimal for shader read only. Then, the staging buffer can be destroyed, and the device memory can be freed.

4.1.6 Camera Module

The camera module represents a virtual camera. It has a position, target and up vectors for reference and an angle for the field of view of the projection. This module provides the view and projection matrices used to transform the objects in the shaders. It also presents methods for moving and turning, used by the Vulkan application module.

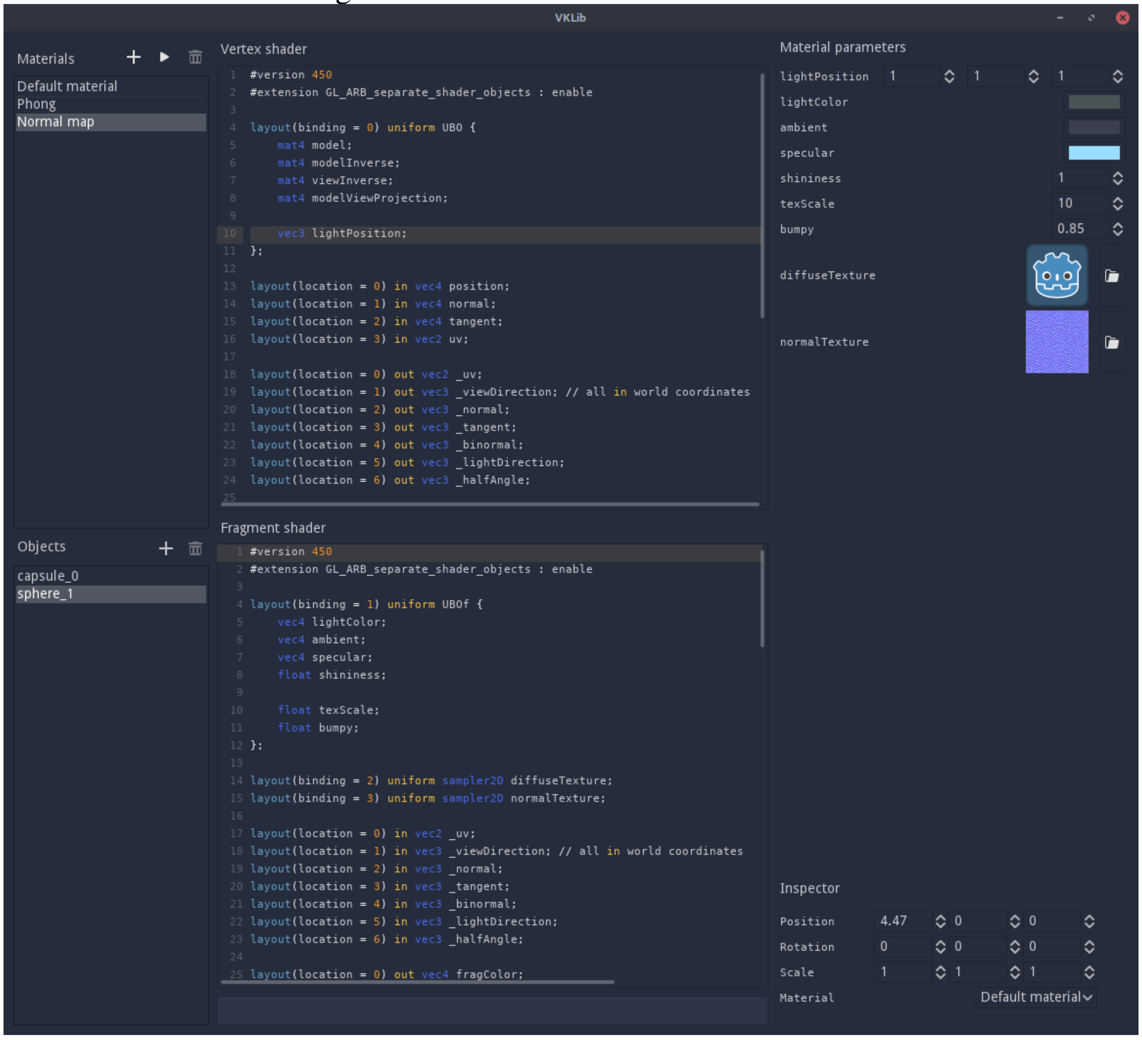
4.2 User Interface

In this section we will describe how our application works from a user perspective, explaining the different panels, their responsibilities and how to use them.

Figure 4.3 shows the different panels of Shader Tutor:

1. **Material list**: Lists all the materials loaded. Allows the user to load shaders and compile them, which will prepare the material to be used in the scene;

Figure 4.2: Shader Tutor control window



```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(binding = 0) uniform UBO {
    mat4 model;
    mat4 modelInverse;
    mat4 viewInverse;
    mat4 modelViewProjection;

    vec3 lightPosition;
};

layout(location = 0) in vec4 position;
layout(location = 1) in vec4 normal;
layout(location = 2) in vec4 tangent;
layout(location = 3) in vec2 uv;

layout(location = 0) out vec2 _uv;
layout(location = 1) out vec3 _viewDirection; // all in world coordinates
layout(location = 2) out vec3 _normal;
layout(location = 3) out vec3 _tangent;
layout(location = 4) out vec3 _binormal;
layout(location = 5) out vec3 _lightDirection;
layout(location = 6) out vec3 _halfAngle;
```

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(binding = 1) uniform UBOf {
    vec4 lightColor;
    vec4 ambient;
    vec4 specular;
    float shininess;

    float texScale;
    float bumpy;
};

layout(binding = 2) uniform sampler2D diffuseTexture;
layout(binding = 3) uniform sampler2D normalTexture;

layout(location = 0) in vec2 _uv;
layout(location = 1) in vec3 _viewDirection; // all in world coordinates
layout(location = 2) in vec3 _normal;
layout(location = 3) in vec3 _tangent;
layout(location = 4) in vec3 _binormal;
layout(location = 5) in vec3 _lightDirection;
layout(location = 6) in vec3 _halfAngle;

layout(location = 0) out vec4 fragColor;
```

lightPosition	1	1	1
lightColor			
ambient			
specular			
shininess			1
texScale			10
bumpy			0.85
diffuseTexture			
normalTexture			

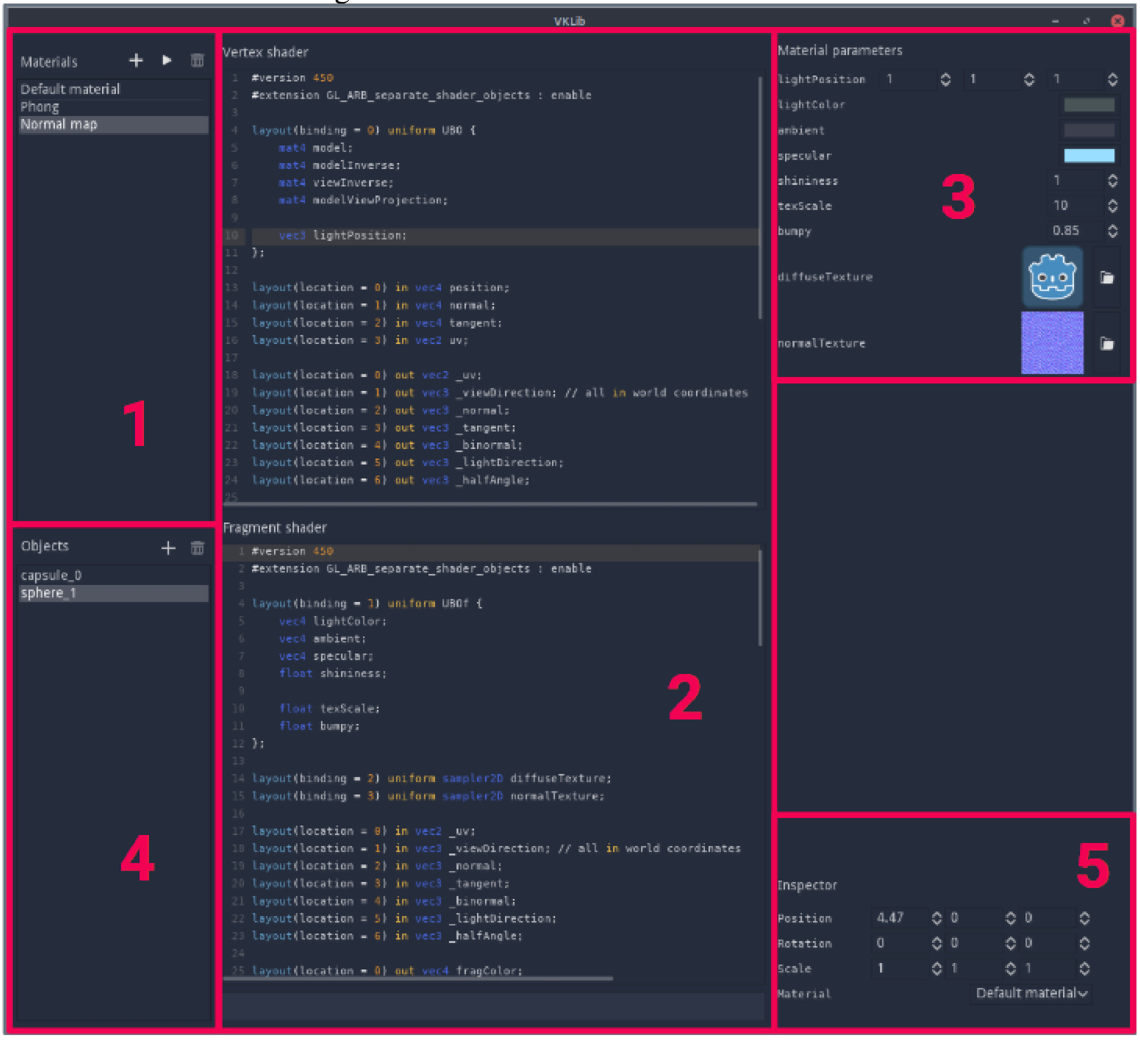
Position	4.47	0	0
Rotation	0	0	0
Scale	1	1	1
Material	Default material		

Source: the author

2. **Shader editor**: Source code editor featuring GLSL syntax highlighting, line numbers and other useful features. Also contains a multiline shader compilation status bar to display any errors in the shader code found during compilation;
3. **Uniform variables editor**: Uniform variables defined in shader code will be displayed in this panel as widgets which allow the user to change the values of the variables. The changes are applied instantly in the visualization window;
4. **Object list**: List of the objects in the scene. Allows the user to create new objects, either from primitives (capsule, cube, cylinder, plane, prism or sphere) or loaded from a file;
5. **Inspector**: Displays the transformation applied to the selected object in the object list (translation, rotation and scale) and the material used to display it.

Figure 4.4 shows that the visualization window, which displays the scene, is a

Figure 4.3: Panels of the control window



```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(binding = 0) uniform UBO {
    mat4 model;
    mat4 modelInverse;
    mat4 viewInverse;
    mat4 modelViewProjection;

    vec3 lightPosition;
};

layout(location = 0) in vec4 position;
layout(location = 1) in vec4 normal;
layout(location = 2) in vec4 tangent;
layout(location = 3) in vec2 uv;

layout(location = 0) out vec2 _uv;
layout(location = 1) out vec3 _viewDirection; // all in world coordinates
layout(location = 2) out vec3 _normal;
layout(location = 3) out vec3 _tangent;
layout(location = 4) out vec3 _binormal;
layout(location = 5) out vec3 _lightDirection;
layout(location = 6) out vec3 _halfAngle;
```

```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(binding = 1) uniform UBOf {
    vec4 lightColor;
    vec4 ambient;
    vec4 specular;
    float shininess;

    float texScale;
    float bumpy;
};

layout(binding = 2) uniform sampler2D diffuseTexture;
layout(binding = 3) uniform sampler2D normalTexture;

layout(location = 0) in vec2 _uv;
layout(location = 1) in vec3 _viewDirection; // all in world coordinates
layout(location = 2) in vec3 _normal;
layout(location = 3) in vec3 _tangent;
layout(location = 4) in vec3 _binormal;
layout(location = 5) in vec3 _lightDirection;
layout(location = 6) in vec3 _halfAngle;

layout(location = 0) out vec4 fragColor;
```

```
lightPosition   1   1   1
lightColor
ambient
specular
shininess       1
texScale        10
bumpy           0.85
diffuseTexture
normalTexture
```

```
Position   4.47   0   0
Rotation   0      0   0
Scale      1      1   1
Material   Default material
```

Source: the author

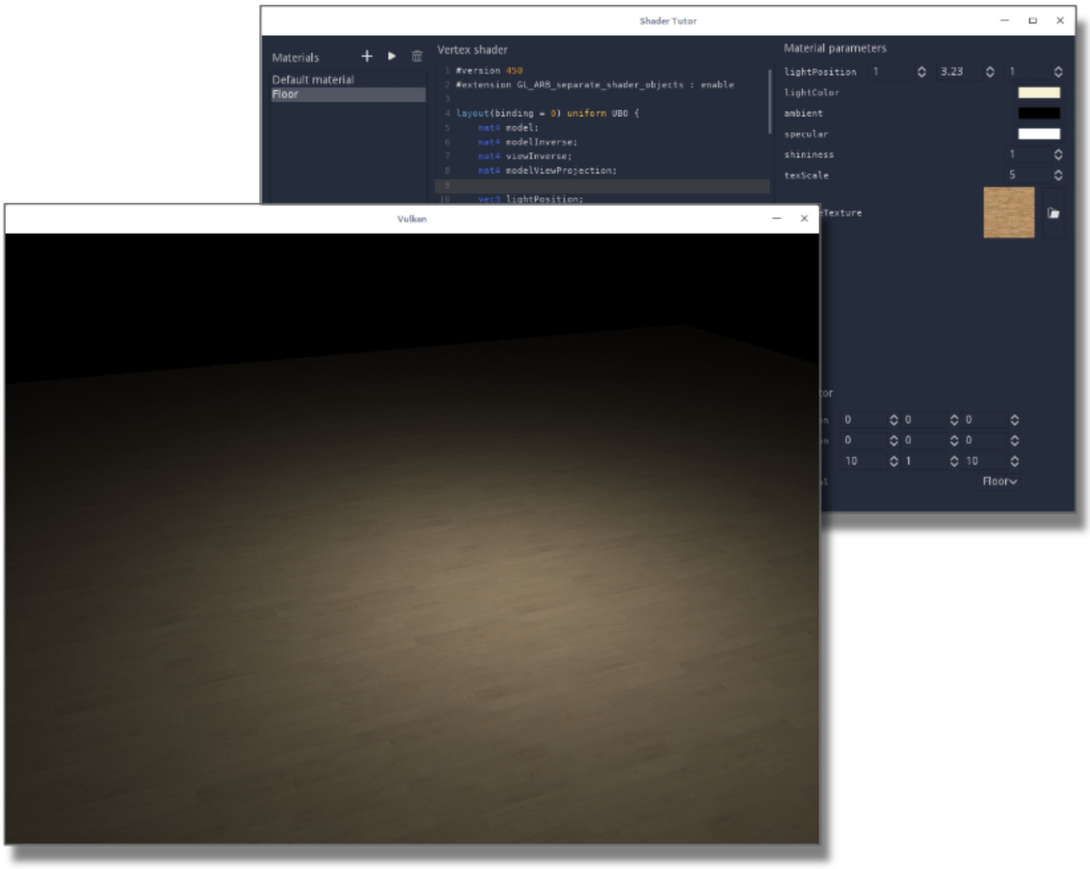
different window from the control window described above.

4.2.1 Materials and Shaders

When the application is first executed, a default material is loaded. This is a simple unshaded material that will be assigned to new objects when they are created. This material does not require the geometry to have any any attributes other than position, which ensures this material can be applied to any loaded mesh.

The *create button* (labeled with a plus sign) allows the user to load shaders and create a new material via the *create material* popup window (Figure 4.5). The material name must be unique and the vertex and fragment shaders source files must have a ".vert" and ".frag" extension, respectively. When the material is created, the contents of the

Figure 4.4: Visualization and control windows



```
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(binding = 0) uniform UBO {
    mat4 model;
    mat4 modelInverse;
    mat4 viewInverse;
    mat4 modelViewProjection;

    vec3 lightPosition;
```

lightPosition	1	3.23	1
lightColor			
ambient			
specular			
shininess			1
texScale			5

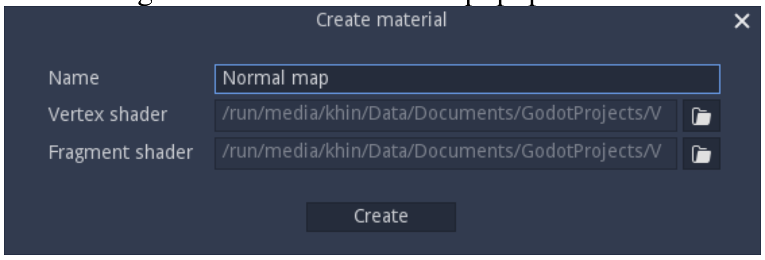
0	0	0
0	0	0
10	1	10

Floor

Source: the author

shader files are loaded in the shader editor (Figure 4.6). At this point, the material is not ready for usage yet.

Figure 4.5: Create material popup window



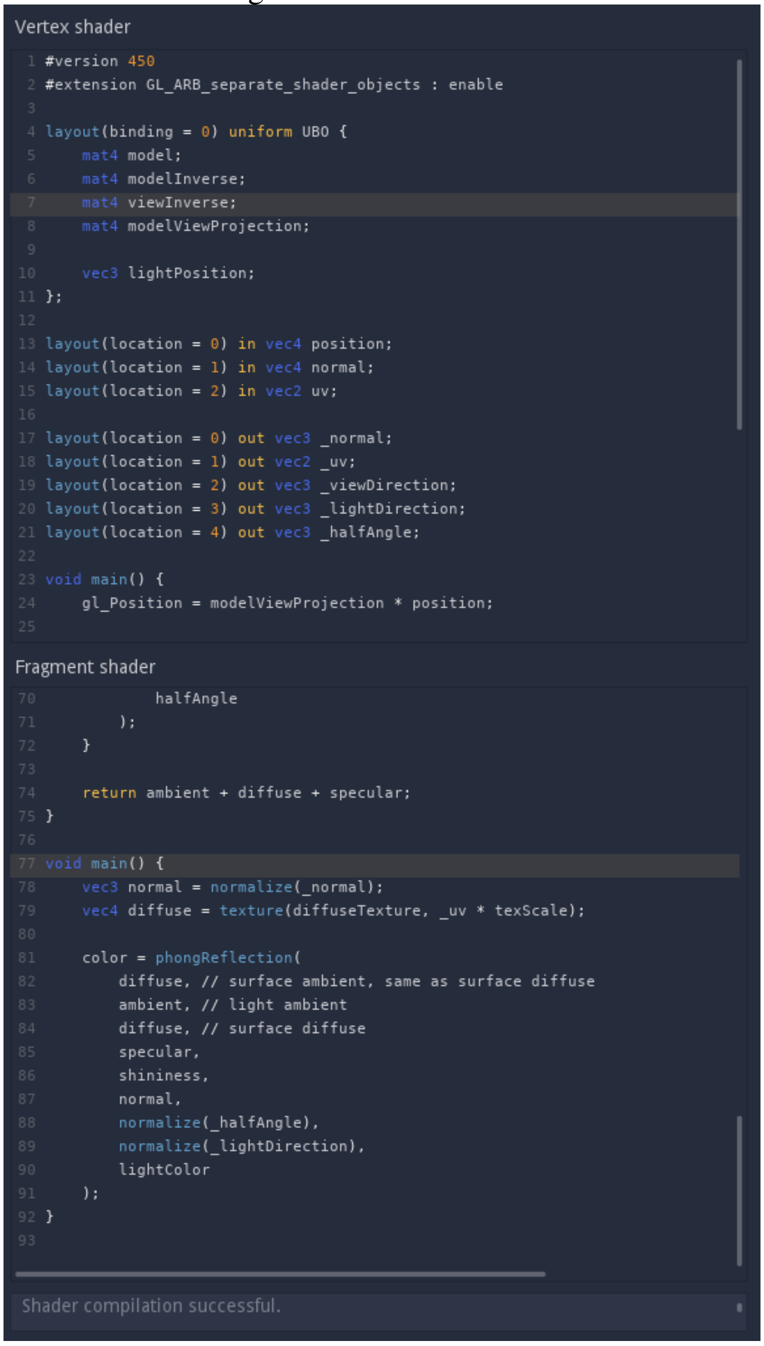
Name	Normal map
Vertex shader	/run/media/khin/Data/Documents/GodotProjects/V
Fragment shader	/run/media/khin/Data/Documents/GodotProjects/V

Source: the author

The GLSL source code for the vertex and fragment shaders can be altered in the shader editor, which features syntax highlighting, line numbers, highlight selected word occurrences and a shader compilation status bar.

To compile the shaders and setup the material, the user can click on the execute button in the material list panel (the button with the "play" icon). The shaders are then compiled and loaded in the material. If the compilation is unsuccessful, the status bar,

Figure 4.6: Shader editor



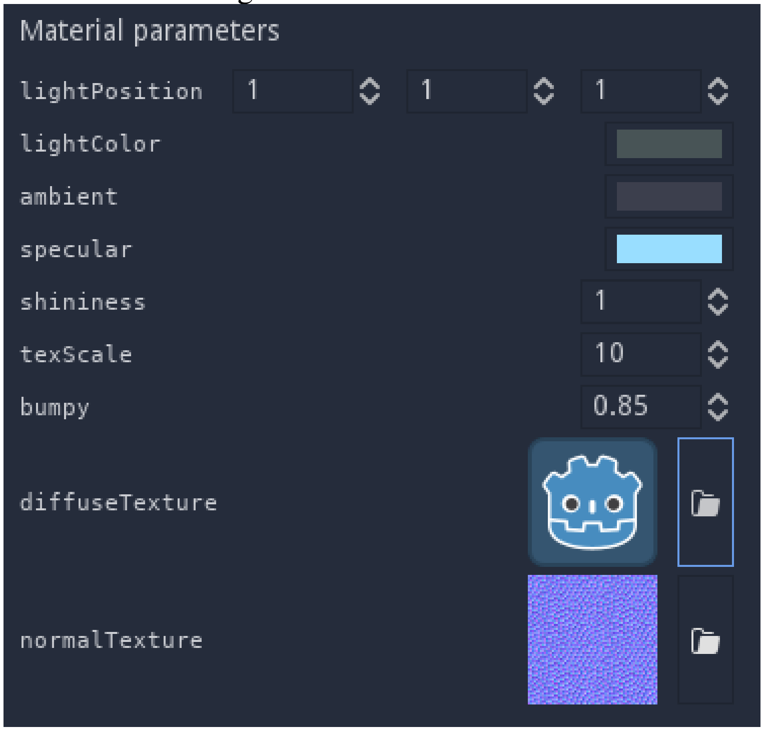
```glsl
#version 450
#extension GL_ARB_separate_shader_objects : enable

layout(binding = 0) uniform UBO {
    mat4 model;
    mat4 modelInverse;
    mat4 viewInverse;
    mat4 modelViewProjection;

    vec3 lightPosition;
};

layout(location = 0) in vec4 position;
layout(location = 1) in vec4 normal;
layout(location = 2) in vec2 uv;

layout(location = 0) out vec3 _normal;
layout(location = 1) out vec2 _uv;
layout(location = 2) out vec3 _viewDirection;
layout(location = 3) out vec3 _lightDirection;
layout(location = 4) out vec3 _halfAngle;

void main() {
    gl_Position = modelViewProjection * position;

```

```glsl
            halfAngle
        );
    }

    return ambient + diffuse + specular;
}

void main() {
    vec3 normal = normalize(_normal);
    vec4 diffuse = texture(diffuseTexture, _uv * texScale);

    color = phongReflection(
        diffuse, // surface ambient, same as surface diffuse
        ambient, // light ambient
        diffuse, // surface diffuse
        specular,
        shininess,
        normal,
        normalize(_halfAngle),
        normalize(_lightDirection),
        lightColor
    );
}

```

```
Shader compilation successful.
```

Source: the author

Figure 4.7: Uniform editor


Material parameters

lightPosition	1	1	1
lightColor			
ambient			
specular			
shininess			1
texScale			10
bumpy			0.85
diffuseTexture			
normalTexture			

Source: the author

right under the fragment shader editor, displays the error messages. When the compilation succeeds, the uniform variables defined in the shaders' source code are exposed in the material parameters panel (Figure 4.7).

The editor creates appropriate widgets for each uniform variable defined in the shaders based on their types. Supported types are described in Table 4.3.

4.2.2 Objects

The plus button on the objects panel allows the user to create a new object, selecting a primitive geometry from a list or loading a custom mesh file (in Wavefront ".obj" format). If the file defines multiple groups inside the object, the groups will be merged and loaded as a single group. The geometry primitives provided are: capsule, cube, cylinder, plane, prism and sphere.

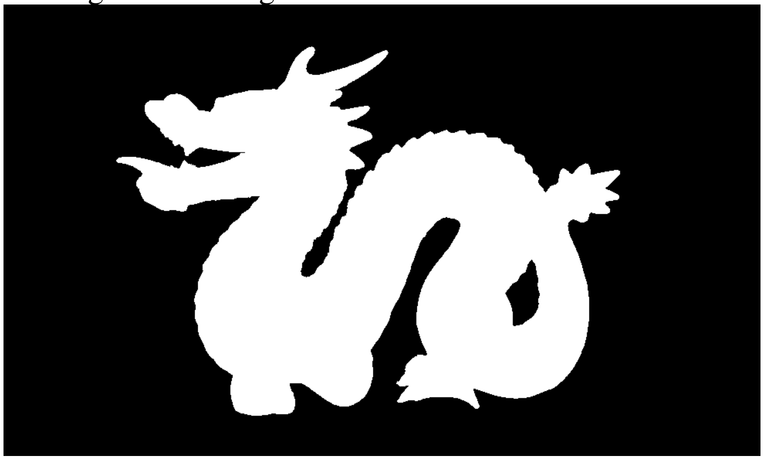
Newly created objects are given a name based on their shape (if the object is a primitive) or their file name (if it was created from a mesh file), and their creation order. New objects are assigned the default material upon creation, and instantly appear in the scene (Figure 4.8).

When an object is selected, their transformation and currently assigned material

Table 4.3: Recognized types for uniform variables

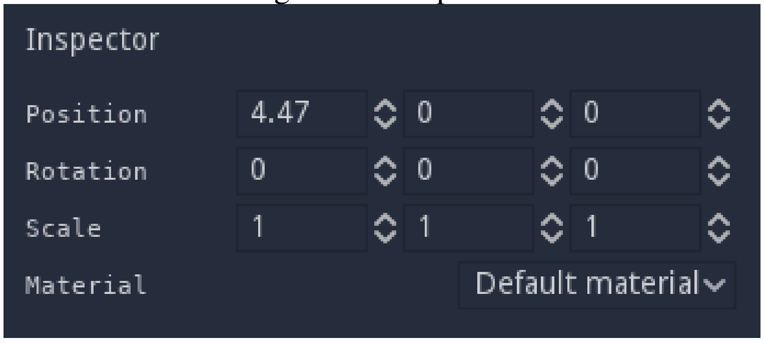
Type name	Widget created
`int`	Single numerical input widget. Accepts values from -10000 to 10000. Fractions are rounded.
`uint`	Single numeric input widget. Accepts values between 0 and 10000. Fractions are rounded.
`float`	Single numeric input field. Accepts values between -10000.0 and 10000.0. Uses 2 decimal digits.
`ivec2`	Two numerical input widgets. Each widget follows the same input rules as `int`.
`uvec2`	Two numerical input widgets. Each widget follows the same input rules as `uint`.
`vec2`	Two numerical input widgets. Each widget follows the same input rules as `float`.
`ivec3`	Three numerical input widgets. Each widget follows the same input rules as `int`.
`uvec3`	Three numerical input widgets. Each widget follows the same input rules as `uint`.
`vec3`	Three numerical input widgets. Each widget follows the same input rules as `float`.
`vec4`	Color picker widget

Figure 4.8: Dragon mesh with the default material



Source: the author

Figure 4.9: Inspector

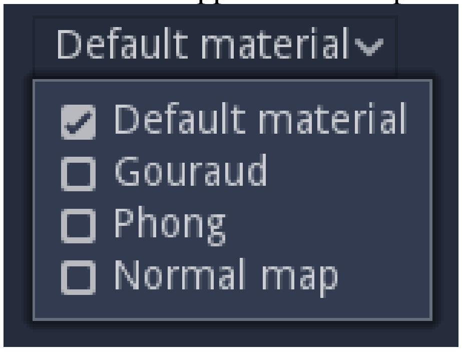


Inspector

Position	4.47	0	0
Rotation	0	0	0
Scale	1	1	1
Material			Default material

Source: the author

Figure 4.10: Available materials appear in the inspector drop-down menu



Default material

- [✓] Default material
- [ ] Gouraud
- [ ] Phong
- [ ] Normal map

Source: the author

are displayed on the inspector panel, below the material parameters panel (Figures 4.7 and 4.9). Changes made to the inspector and the material parameters take effect immediately.

Table 4.4 shows the available commands to navigate through the scene.

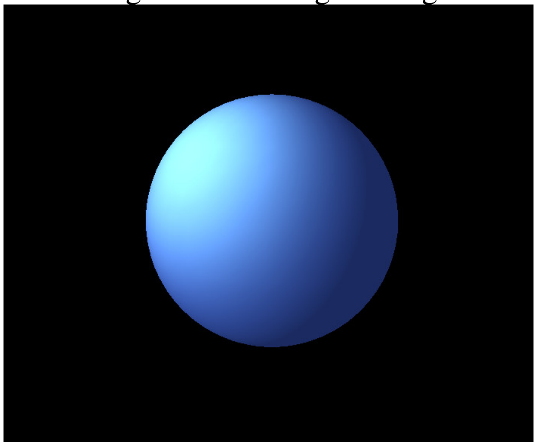
Table 4.4: Scene navigation controls

Input	Action
Mouse right click	When pressed, enables movement
Mouse movement	When movement is enabled, rotates the camera around its own center, looking around
’W’ key	Move the camera forward, relative to its current orientation
’A’ key	Move the camera to the left, relative to its current orientation
’S’ key	Move the camera backward, relative to its current orientation
’D’ key	Move the camera to the right, relative to its current orientation

5 RESULTS

The application allows the user to create and visualize different shaders in multiple objects, define an arbitrary number of parameters for each material and also use multiple textures to create the desired visuals. In Figure 5.1, we see a sphere illuminated using the Phong illumination model and shading (illumination is calculated per pixel).

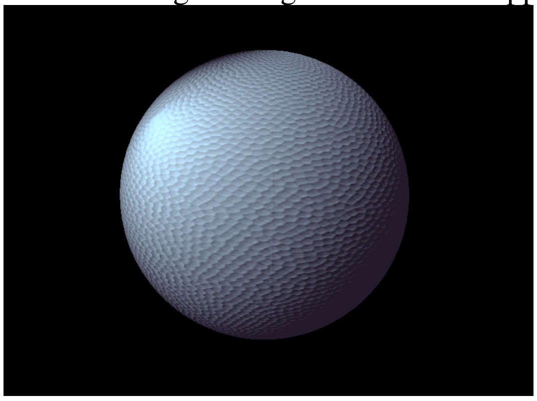
Figure 5.1: Phong shading



Source: the author

In Figure 5.2, the same shading model is applied, but the illumination computation uses normal mapping to perturb the normal vectors of the sphere.

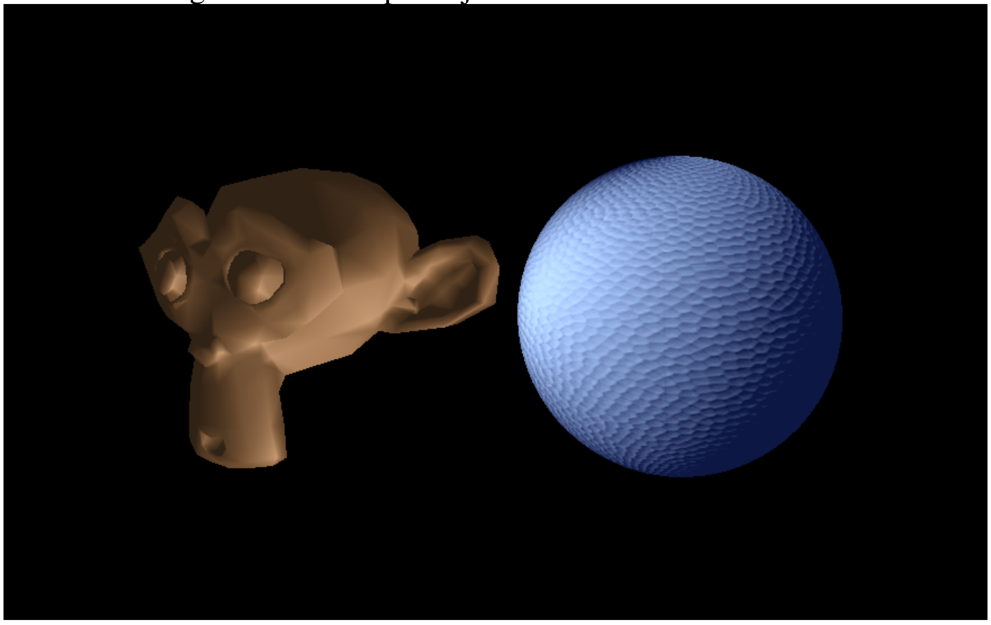
Figure 5.2: Phong shading with normal mapping



Source: the author

Figure 5.3 shows multiple objects using different materials, instanced in the same scene. Each material defines its own parameters. Figure 5.4 shows a scene with a variety of objects. The sky is rendered by applying a texture to a capsule. The ground, sphere, and torus have normal mapping applied (most visible in Figures 5.6 and 5.7) and the dragon uses Phong shading (Figure 5.5).

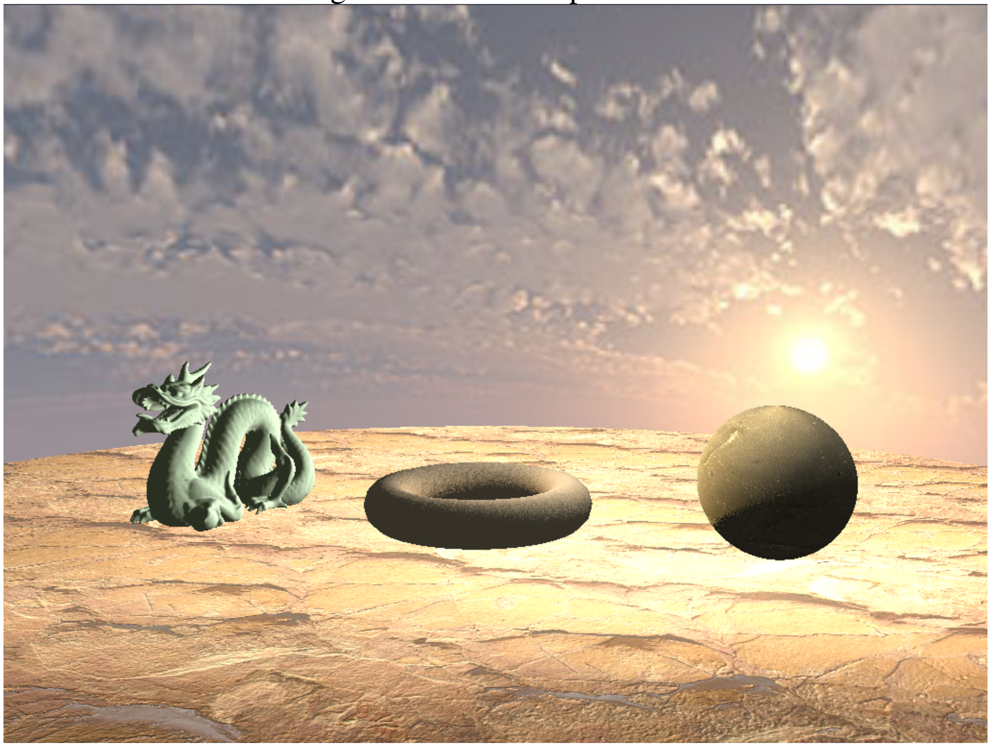
Figure 5.3: Multiple objects with different materials



Source: the author

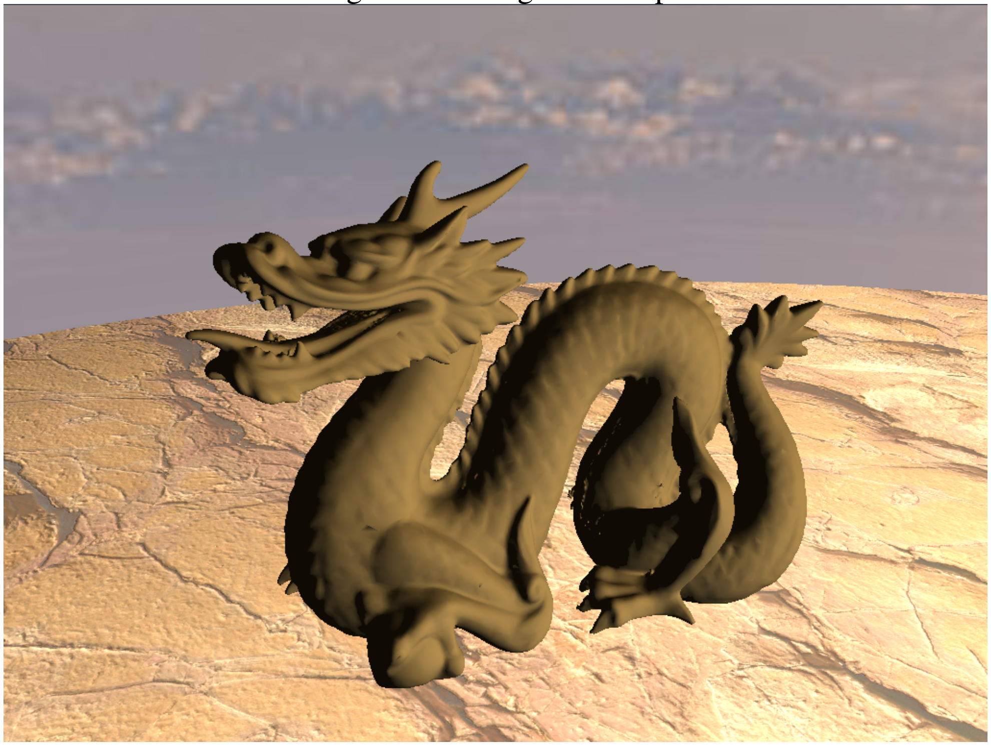
Figure 5.8 shows a scene rendered with an illumination model which accounts for distance from the light source, creating a more realistic effect.

Figure 5.4: An example scene



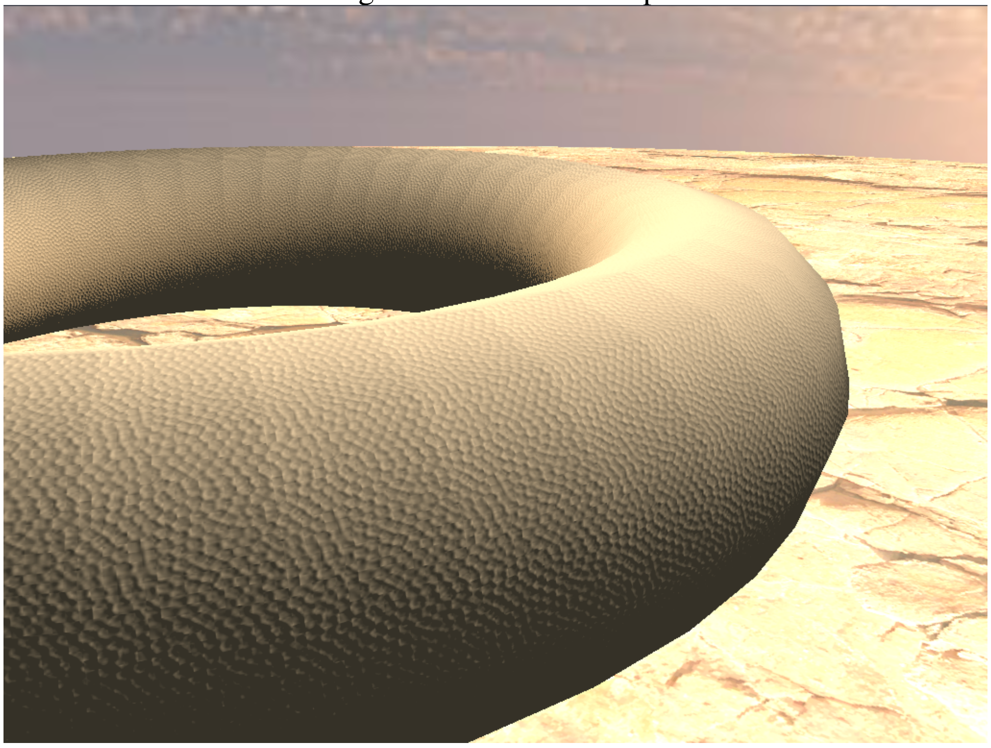
Source: the author

Figure 5.5: Dragon closeup



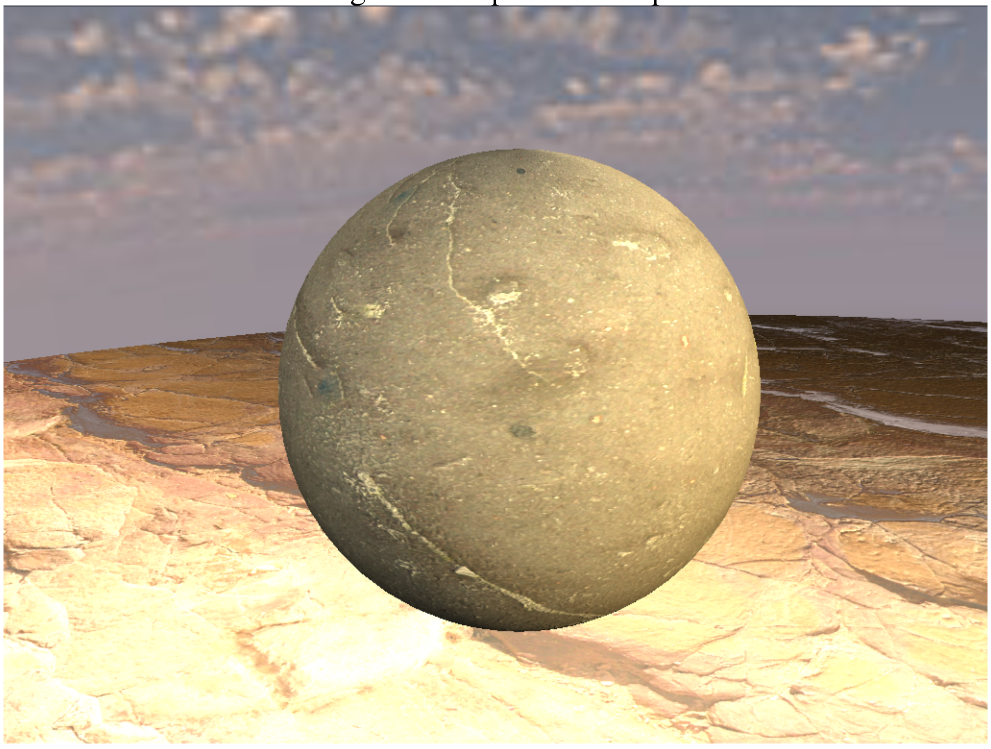
Source: the author

Figure 5.6: Torus closeup



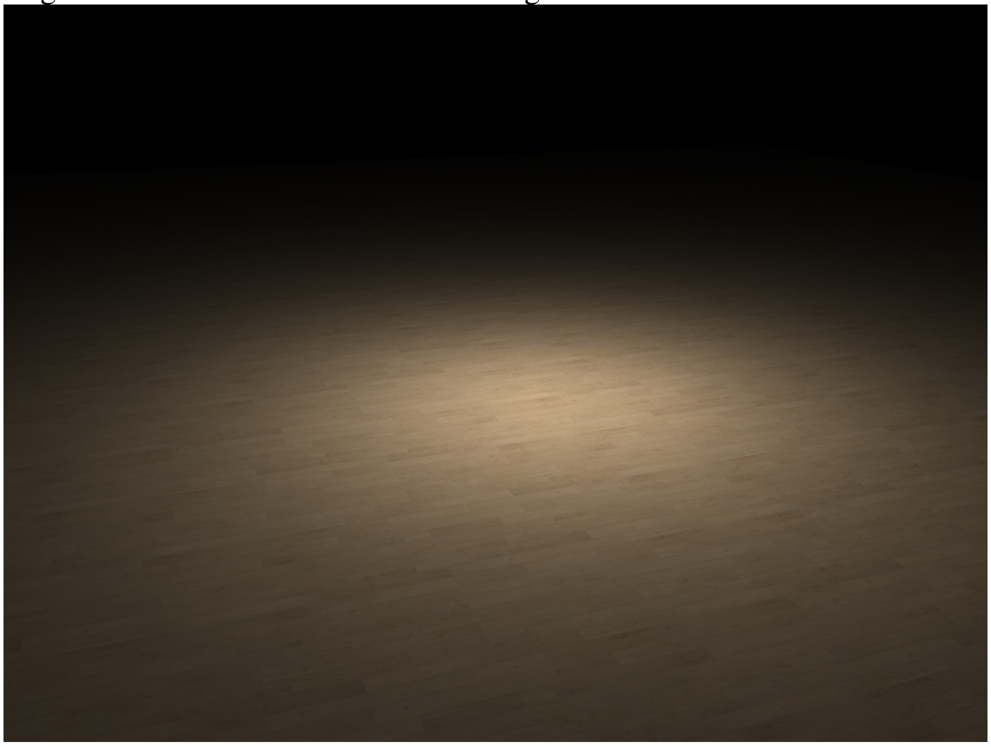
Source: the author

Figure 5.7: Sphere closeup



Source: the author

Figure 5.8: Floor illuminated with Phong and attenuation based on distance



Source: the author

6 CONCLUSION

We presented a complete environment for learning and exploring shader programming using GLSL. Our environment, developed using Vulkan, can suit the needs of both beginners and advanced users. Its user-friendly interface allows one to effortlessly load 3D models and images and modify variables and textures, applying the changes in real time. This was also an important learning experience. Developing a real application using Vulkan provided the opportunity to learn such new graphics API.

In order to provide a context for this work, the thesis also reviewed the concept of graphics pipeline and its stages, especially the programmable stages, all of which are key concepts in the field of computer graphics. We also introduced the tool used to build our application, Godot Engine, and the technology called GDNative, used in our application to integrate the Vulkan host application and the control window. We also talked about graphics APIs and how APIs have different characteristics that the developer must be aware of.

Then, we presented the details of the Vulkan API, used throughout this work. As we could see, this API burdens the developers with every detail of the graphics pipeline and configuration. While this empowers programmers to tailor the application that is most optimized, it can also be very difficult to do so.

We described the architecture and how our tool works from the inside, explaining each of its modules and how they work to build the best user experience possible. The user interface is also discussed in depth, showing the features from a user perspective and describing how to use each of them.

The application we created is a tool that allows students to write shader code without having to worry about the underlying graphics API or creating the application infrastructure just to see the shaders in action. The results show we were successful in creating such tool, rendering different scenes which demonstrate the flexibility achieved.

Creating this application has been a great learning exercise of the Vulkan API, and the efforts have resulted in a very interesting piece of software with a real-world use-case scenario, with lots of room to grow. Given the proper treatment, this application, which is currently presented as a proof of concept, could be really useful in computer graphics lectures and workshops, from fundamentals of computer graphics to advanced shader programming.

7 FUTURE WORK

We have some ideas as to how this project could be improved. For example, the definition of a project description file format would be of great benefit for students, allowing them to share projects between one another and the teacher. It may also be desirable to support more types of shaders, like geometry, tesselation control, and tesselation evaluation shaders. Although the graphics hardware supports cube map samplers, Shader Tutor does not parse uniform variables defined as sampler cubes, but doing so would be useful to create certain effects, such as environment mapping. Adding lights as entities of the application, broadcasting their attributes to all shaders that required them, instead of having each shader add their own light parameters. Finally, removing and renaming resources (materials and objects) could improve the experience of students using the application.

The overall implementation could also benefit from more rigid software engineering guidelines in order to separate the core functionality from the interface used to communicate with the control window (using the "Model-View-Controller" model, for example).

This application could also be expanded to help students understand general computer graphics topics instead of just shader programming. This could be achieved if the Vulkan API configuration values in its various structures were exposed to the user in an organized manner, so that the student could tweak parameters of the entire graphics pipeline and see the changes in real time. Going even further, this could evolve into a tool to learn the Vulkan API specifically, if the user were given the chance to, for example, see the available physical devices, their features and extensions, being able to enable and disable each of them at will. Such tool could be really enlightening to anyone using the API for the first time.

REFERENCES

Alexander Overvoorde. **Vulkan Tutorial**. 2016. Disponível em: <https://vulkan-tutorial.com>.

AMD. **RenderMonkey Release Notes**. 2008. Disponível em: <https://32ipi028l5q82yhj72224m8j-wpengine.netdna-ssl.com/wp-content/uploads/2017/01/rendermonkey-releasenotes-v.1.82.320.txt>.

AMD. **RenderMonkey Toolsuite**. 2008. Disponível em: <https://gpuopen.com/archive/gamescgi/rendermonkey-toolsuite/>.

Daniel Rakos. **Using the Vulkan Validation Layers**. 2016. Disponível em: <https://gpuopen.com/using-the-vulkan-validation-layers>.

GLFW. **GLFW: An OpenGL library**. 2016. Disponível em: <http://www.glfw.org/>.

GOOGLE. **ShaderC: A collection of tools, libraries and tests for shader compilation**. 2018. Disponível em: <https://github.com/google/shaderc>.

HELPS, A. **Why is OpenGL poorly supported on Mac OS X?** 2018. Disponível em: <https://www.quora.com/Why-is-OpenGL-poorly-supported-on-Mac-OS-X>.

KHRONOS. **Khronos SPIR-V Registry**. 2018. Disponível em: <https://www.khronos.org/registry/spir-v>.

Khronos. **Vulkan 1.1.78 - A Specification**. 2018. Disponível em: <https://www.khronos.org/registry/vulkan/specs/1.1-khr-extensions/html/vkspec.html>.

Khronos. **Vulkan Overview**. 2018. Disponível em: <https://www.khronos.org/vulkan>.

LIGHTHOUSE3D. **The Normal Matrix**. 2011. Disponível em: <http://www.lighthouse3d.com/tutorials/glsl-12-tutorial/the-normal-matrix/>.

NVIDIA. **FX Composer**. 2008. Disponível em: <https://developer.nvidia.com/fx-composer>.

NVIDIA. **FX COMPOSER 2.5 RELEASE NOTES**. 2008. Disponível em: <https://developer.nvidia.com/sites/default/files/akamai/gamedev/docs/FX_Composer_2.5_README.TXT>.

NVIDIA. **Vulkan Memory Management**. 2016. Disponível em: <https://developer.nvidia.com/vulkan-memory-management>.

Segal, M. Akeley, K., F. L. B. **The OpenGL® graphics system: A specification (version 4.0 (core profile) - march 11, 2010)**. 2010. Disponível em: <http://www.opengl.org/registry/doc/glspec40.core.20100311.pdf>.

SELLERS, G. **Vulkan Programming Guide: The Official Guide to Learning Vulkan**. First. [S.l.]: Addison-Wesley Professional, ISBN: 0134464540, 2016.

SHIRLEY, P. **Fundamentals of Computer Graphics**. First. [S.l.]: Taylor & Francis, ISBN: 1568811241, 9781568811246, 2002.

Thomaz Herzog. **GDNative is here!** 2017. Disponível em: <https://godotengine.org/article/dlscript-here>.

TOISOUL, A.; RUECKERT, D.; KAINZ, B. Accessible GLSL Shader Programming. In: BOURDIN, J.-J.; SHESH, A. (Ed.). **EG 2017 - Education Papers**. [S.l.]: The Eurographics Association, 2017. ISSN 1017-4656.