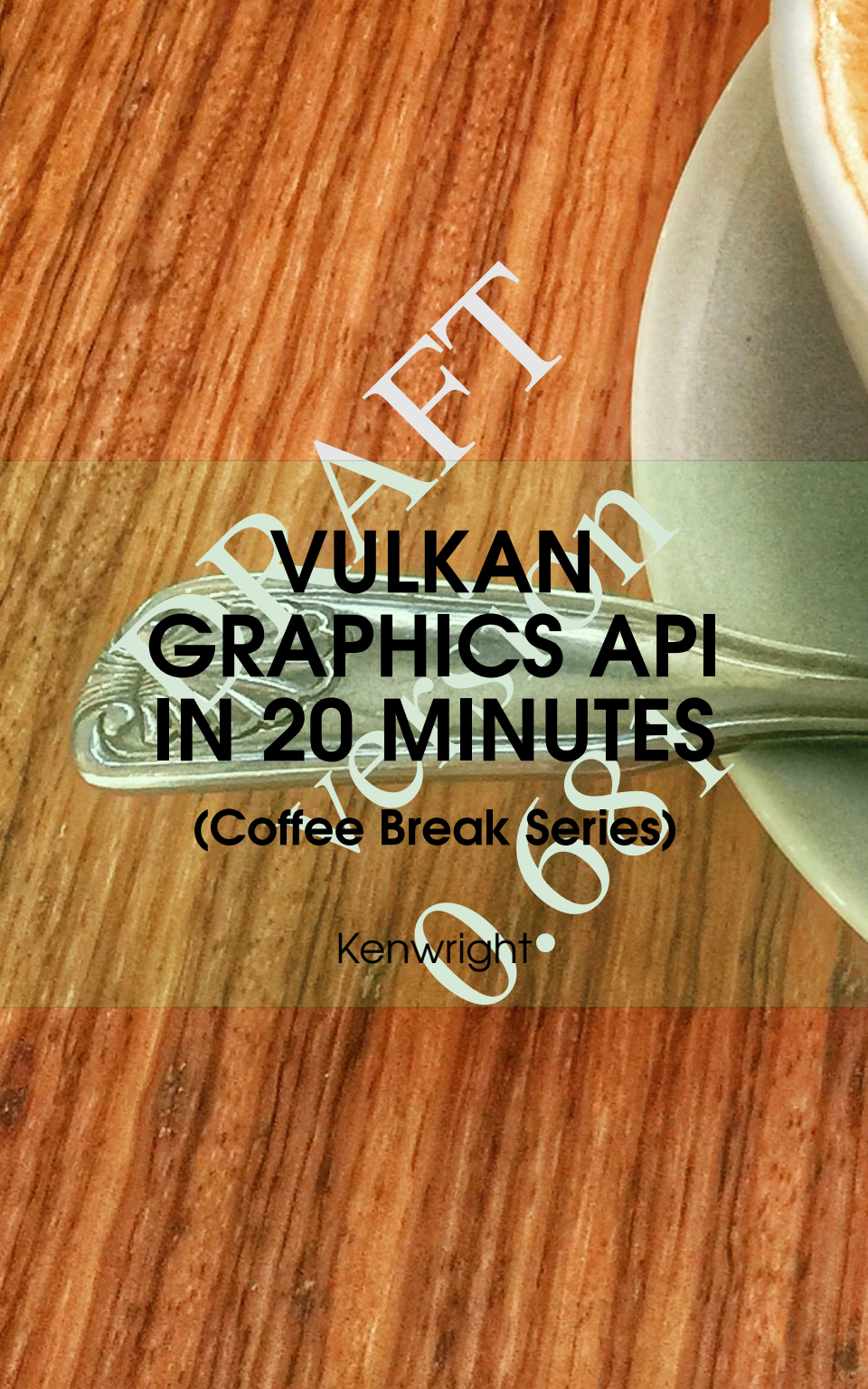

VULKAN GRAPHICS API IN 20 MINUTES

(Coffee Break Series)

Kenwright

BOOK TITLE:
VULKAN GRAPHICS API IN 20 MINUTES
ISBN-13: 978-1535124850
ISBN-10: 1535124857

Edition: 021042016

Table of Contents

1. Introduction

This book introducing the reader (you) to the Vulkan cross platform 3D graphics API (a taste of Vulkan) - including simple tutorials and samples. We address questions, such as, do we need another graphics API? what is special about Vulkan? how is Vulkan different from previous DirectX and OpenGL libraries? and how do we initialize and setup a basic Vulkan program in C++?

1.1 What is Vulkan?

To begin with, let's talk about what Vulkan is and what it has to do with computer graphics. As you'll see (or know), you typically use a set of graphical API to control your graphical

output - the days of writing your own driver for every computer/platform has gone. With today's huge range of constantly changing devices (and hardware) means it would be crazy to even try! Hence, a common, scalable, modern graphics API lets us get things done quickly with less errors. Vulkan is one of these API. Not just any API - but a low-overhead, cross-platform 3D graphics and compute API. As you'll discover, the Vulkan API enables us to unlock a lot of performance even with current-generation hardware. So learning Vulkan will give you more control and power - enabling you to take existing, current, and future hardware to new heights due to its fresh forward thinking design.Vulkan opens the door to improved multi-core 'CPU' usage, increased 'on-screen' detail, higher framerates (for both applications and video games - ultimately producing smoother gameplay), more efficient communications between the hardware (e.g., CPU and GPU), and reduced system power usage. These features allow you to create new applications (video games and tools) that were previously beyond your reach - unleashing a new era of possibilities (making the impossible - possible).

1.2 What is different about Vulkan?

Unlike previous graphical predecessors (e.g,. OpenGL and DirectX), Vulkan is designed from the ground up to run on diverse platforms, ranging from mobiles and tablets, to gaming consoles and high-end desktops. The underlying design of the API is layered (to be more structured and modular), so it enables the creation of a common, yet extensible architecture for code validation, debugging, and profiling, without impacting performance. Krhonos (the original developer of Vulkan) claims the layered approach delivers a lot more flexibility, catalysing stronger innovations in cross-vendor GPU tools, and provides more direct GPU control (which sophisticated game engines demand). As Vulkan has less latency and overhead than older OpenGL or DirectX (or Direct3D)

API - to enable you and your system to reach better levels of performance. In a nutshell, Vulkan helps us (and developers) avoid bottlenecks (through more explicit control) to unleash each systems full potential.

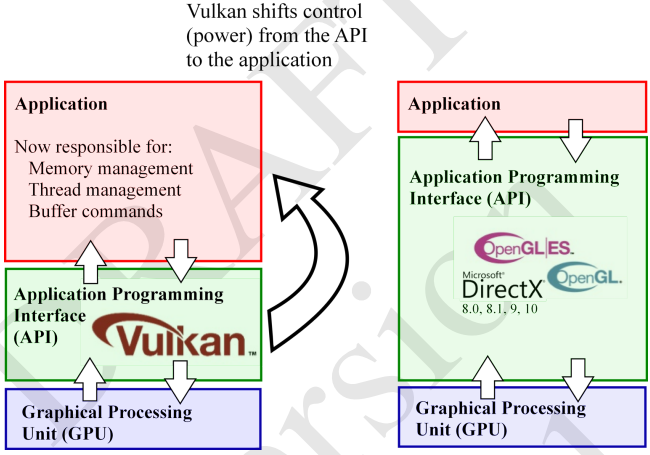

Figure 1.1: **More Explicit Control** - How Vulkan compares to traditional graphical API (more power to the developer).

In years gone by, OpenGL and DirectX drivers have been highly optimized for specific architectures (GPU). You'd find it difficult to get more out of these current drivers. However, with DirectX 12 and Vulkan the approach to how software and hardware are done has changed (requires the re-thinking of how to take advantage of the explicit APIs). Develop a new balance between the high and low level capabilities to exploit changes in technology trends (both CPU and GPU parallelism - effective and efficient synchronization of resources). Not so easy - can't just switch over - lots of thought is needed to exploit the new APIs way of working (e.g., multi-queue support requires additional work). Designed for the future and the next generation of hardware.

1.2.1 With great power comes great ...

The Vulkan API opens up new doors of possibility - providing greater control and customizability. However, this extra flexibility comes at a cost. The problems originally hidden away in the driver, are now the issues of the application developer (you), such as, synchronization (barriers) and memory management (uploads and residency). Of course, it is a double sided sword, you get the power (and responsibility) to fix these issues and solve other unforeseen complications or bottlenecks.

Modern CPUs are changing (i.e., more cores and more parallelism). The Vulkan API/drivers for CPU/GPU communication and management was designed around this core concept. Ultimately reducing bottlenecks and allowing more flexibility (low-level control) - direct access to hardware to recover lost performance. Vulkan has effectively removed the veil which was common in previous generation graphics API (masking and hiding the internal mechanics and implicitly making assumptions).

1.3 Should you bother learning Vulkan?

Like OpenGL, Vulkan targets high-performance real-time 3D graphics applications such as games and interactive media across all platforms, and offers higher performance and lower CPU usage, much like Direct3D 12 and Mantle. In addition to its lower CPU usage, Vulkan is also able to better distribute work amongst multiple CPU cores. Vulkan targets applications such as games and interactive media across multiple platforms to provide higher performance and lower CPU usage. The core advantages of the Vulkan graphics API, is it (should) be possible to have one set of game code that will run on any hardware that's compliant with the API (one API to rule them all).

1.4 Do any games support Vulkan yet?

While Vulkan is relatively new, you'll be happy to hear that it has been successfully employed in a number of real-world applications. Some examples of Vulkan used commercially for video games are:

- The Talos Principle – The first video game with Vulkan rendering support (Feb 2016)
- Dota 2 – Vulkan support released (May 2016)

Interesting, those new to software development and graphics the Vulkan API can be a bit intimidating compared to other graphics API, since it can typically takes over '500' lines of C++ code just to display a simple 'Triangle' (as you'll discover).

1.5 What this book is NOT

This book does not introduce graphical concepts, such as, matrices, lighting equations, shader languages, or programming languages (C/C++/Win32). However, there are an abundance of books available on these subjects - we encourage the reader (you) to look up concepts and material (i.e., to complement and develop the principles and samples given in this short introductory text).

Vulkan is developed to be cross-platform. However, in some specific chapters you'll use the Win32 API. Yet it should be easy for the you (the reader) to port these few lines of code across to other platforms. Remember, the Vulkan API will be changing constantly (trust me) - over the next few years, so you can expect to see an assortment of customizations and modifications. So keep up to date.

You need to have a basic grounding in existing APIs (e.g., the

workings of the graphics pipeline and traditional OpenGL/DirectX) - concepts such as, multi-threading, staging resources, synchronisation and so on. You'll discuss how these are now implemented in Vulkan. You'll take a brief whirlwind tour of why, what and how the main Vulkan concepts look like and how to get something up and running quickly.

This book isn't intended to be comprehensive text (for that you should read the specification), nor is it heavy in background or justification. However, hopefully by the end of this book, you should be able to read the specifications or library headers and have an idea of how to get a 'simple' Vulkan application up and running.

While you consider basic error handling, you'll skip over lots of query calls in our basic examples - which would be necessary to determine each systems capabilities and limits. A real world application would need to respect and implement these query calls. In light of the fact that you'll excluded a large number of details, this was necessary to keep the text as reasonably compact and simple as possible. In summary, after reading this book, you should have a good beginners start using to understanding and using Vulkan.

As you work through the different sections, remember the some key points:

1. Think in parallel (with both the CPU and the GPU)
2. Avoid allocating and releasing at runtime
3. Group (group command buffer submissions, barriers, and batch rendering)
4. Manage the memory efficiently (Don't over engineer the problem)
5. Still apply traditional optimisation logic (no one solution fits every problem)

1.6 Summary

Vulkan is an open source alternative to Microsoft DirectX on the PC and is the successor to OpenGL. The Vulkan API is derived from AMD's Mantle's API. Eventually, you'll see it supported on everything from phones to PCs. Similar to Microsoft's DirectX 12, it claims to make graphical applications such as games run faster! Yet Vulkan won't be limited to Windows 10 (like DirectX 12), instead, it will be supported right back to Windows 7 (not to mention Linux and Android support). A critical factor for Vulkan developers – is Vulkan aim to give fast low-level API access that was previously unavailable. Importantly, because it's open source, it won't cost you anything to use and develop for. What is more, it will run on any of AMD's graphics cards right back to Radeon HD 7000 series and Nvidia GeForce 600 series boards and newer. To get started - all you need is to update your graphics card driver.

1.7 Test Yourself

Here are some multiple-choice questions to help you identify your baseline knowledge of the material. Answers appear at the end of the test.

1.7.1 Question 1

Vulkan API is a low-overhead, cross-platform 3D graphics and compute API:

a false
b true

1.7.2 Question 2

Vulkan is an abbreviation for 'Validated GPU Levels Khronos Applications'?

a true
b false

1.7.3 Question 3

The Vulkan Window's driver only support the 'Windows 10' operating system?

a true
b false

1.7.4 Question 4

Which of the following are correct:

a Vulkan, like other low-overhead APIs, improves performance by providing more direct access to the GPU
b Vulkan is a next generation API for high-efficiency graphics and computing
c Vulkan is an open API
d all over the above

(Q1: Answer b; Q2: Answer b; Q3: Answer b; Q4: Answer d;)

2. Getting Started

2.1 Setup

Before you even think about developing any Vulkan C++ programs - you'll need to install the necessary drivers and SDK libraries/header files.

1. Install the Vulkan SDK
2. Install a working Vulkan graphics driver

2.2 Installing Drivers

The Vulkan driver is essential if you want to run your Vulkan applications i.e., you must setup your graphical display driver. However, don't worry, the process is extremely simple:

Download the NVidia Vulkan Driver:

Get and install the NVidia Vulkan drivers (http://developer.nvidia.com/vulkan-driver)

Download AMD Vulkan Driver:

Get and install the AMD Vulkan drivers (http://www.amd.com/en-gb/innovations/software-technologies/technologies-gaming/vulkan)

After installing the Vulkan driver, you should have a '**vulkan-1**' file on your computer (e.g., '**vulkan-1.dll**' in your System32 folder on Windows). Similarly, for Linux or Android operating systems, you should be able to find and install the necessary driver and locate the binaries - which you'll link with your application and Vulkan SDK.

2.3 LunarG Vulkan SDK

The LunarG Vulkan SDK provides the development and run-time components for building, running, and debugging Vulkan applications (which will help you to get up and running quickly and easily). The SDK is comprehensive, such that, it includes the Vulkan loader, Vulkan layers, debugging tools, SPIR-V tools, the Vulkan run time installer, documentation, samples, and demos.

Download the LunarG Vulkan SDK:

Get and install Vulkan SDK (Windows/Linux) (https://lunarg.com/vulkan-sdk/)

Interestingly, when you download and install the Vulkan SDK, you'll find it provides a wealth of documentation, such as:

- Getting started with the Vulkan SDK
- Vulkan tools framework (Loaders, Layers, Validation Layer Details)
- Tools
- Advanced topics
- Vulkan docs (Specifications, Samples)
- FAQ
- SDK Installers

This documentation provides additional facts on the API functions and procedures. So if you are unsure about anything - you can consult the SDK documentation or online articles for further information.

2.4 Practical Roadmap

After you've set everything up, e.g., drivers and sdk - and you're able to compile and run Vulkan programs - you're ready to start your journey. The chapters follow a sequential order and build upon one another (i.e., each chapter adds further features/operations). The eventual aim is to have a simple application up and running that outputs geometry (i.e., a color triangle moving around the screen). Hence, the following chapters include:

- Initializing Vulkan
- Surfaces (you get surface information for the device - so you're able to make a decisions on which device is most compatible)
- Device Enumeration
- Device Creation
- Swapping Chains
- Device Queues (Command Buffers)

- Framebuffer
- Presenting (Clearing Screen)
- Triangle Data (Buffers)
- Shaders (Loading/Linking)
- Descriptors (Defining/Binding Shader, Data and API)
- Pipeline (Triangle Output)

2.5 Summary

There is a massive range of resources online to support the Vulkan API. While you get you up and running with the basic Vulkan API features - you'll also find solutions to a whole variety of problems online as well. However, remember, the API is in its infancy, so don't expect it to remain constant over the coming years - as technology trends change - the API will be revamped and rewritten (e.g., version 1.1, 1.5, 2.0, and 3.0). The Vulkan API will and does change often. So, some portions of this book may become obsolete quickly. If you do find so, please leave me a message (e.g., email me) so that later revisions may be updated to reflect these updates. Hence, each revision includes modifications and corrections (not to mention your name in the credits if you are the first to point out something). All the best and enjoy the Vulkan API journey.

3. Initializing Vulkan

3.1 Your First Vulkan Program

There is no global state in Vulkan; all application state is stored in a **vkInstance** object. Creating a **vkInstance** object initializes the Vulkan library and allows application to pass information about itself to the implementation.

To create an instance you'll also need a **vkInstanceCreateInfo** object controlling the creation of the instance and a **vkAllocationCallback** to control host memory allocation for the instance. For now you'll ignore **vkAllocationCallback** and use **NULL** which will use the system-wide allocator. More on **vkAllocationCallback** later.

```
/**
*
* **1** Filling out application description:
```

```
*
**/
vkApplicationInfo    applicationInfo;
// sType is mandatory
applicationInfo.sType = ↩
    VK_STRUCTURE_TYPE_APPLICATION_INFO;
// pNext is mandatory
applicationInfo.pNext = NULL;
// The name of our application
applicationInfo.pApplicationName = "Hello Vulkan";
// The name of the engine (e.g: Game engine name)
applicationInfo.pEngineName = NULL;
// The version of the engine
applicationInfo.engineVersion = 1;
// The version of Vulkan we're using for this
// application
applicationInfo.apiVersion = VK_API_VERSION_1_0;

/**
*
* **2** Filling out instance description:
*
**/
vkInstanceCreateInfo instanceInfo;
// sType is mandatory
instanceInfo.sType = ↩
    VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
// pNext is mandatorly set
instanceInfo.pNext                   = NULL;
// flags is mandatory set
instanceInfo.flags                   = 0;
// The application info structure is then passed
// through the instance
instanceInfo.pApplicationInfo        = &↩
    applicationInfo;
// Don't enable and layer
instanceInfo.enabledLayerCount       = 0;
instanceInfo.ppEnabledLayerNames     = NULL;
// Don't enable any extensions
instanceInfo.enabledExtensionCount   = 0;
instanceInfo.ppEnabledExtensionNames = NULL;

/**
*
* **3** Now create the desired instance
*
**/
vkInstance instance = VK_NULL_HANDLE;

vkResult result =
vkCreateInstance(&instanceInfo, NULL, &instance);

DBG_ASSERT_VULKAN_MSG(result,
  "Failed to create vulkan instance");
```

```
// To Come Later
// ...
// ...

/**
*
* **4** Never forget to free resources
*
**/
vkDestroyInstance(instance, NULL);
```

sType is used to describe the type of the structure. It must be filled out in every structure. **pNext** must be filled out too. The idea behind **pNext** is to store pointers to extension-specific structures. Valid usage currently is to assign it a value of **NULL**. The same goes for flags.

In the first chunk of code you setup an application description structure which will be a required component for our instance info structure. In Vulkan you're expected to describe what your application is, which engine it uses (or **NULL**). This is useful information for Vulkan to have as driver vendors may want to apply engine or game specific features/fixes to the application code. Traditionally this sort of technique was supported in much more complicated and unsafe manners. Vulkan addresses the problem by requiring upfront information.

The second part creates an instance description structure which will be used to actually initialize an instance. This is where you'd request extensions or layers. Extensions work in the same way as GL did extensions, nothing has changed here and it should be familiar. A layer is a new concept Vulkan has introduced. Layers are techniques you can enable that insert themselves into the call chain for Vulkan commands the layer is inserted in. They can be used to validate application behavior during development. Think of them as decorators to commands. In your simple example, you don't bother with extensions or layers, but they must be filled out.

From here it's as trivial as calling **vkCreateInstance** to create an instance. On success this function will return **VK_SUCCESS**. When you're done with your instance you destroy it with **vkDestroyInstance** (tidying up).

3.1.1 Debugging

You'll end up writing a large amount of code - and errors happen - either typos or initialization states fail. Avoid having the program simply exit without any notifications. Leaving you wondering 'what happened' - or searching through log files for an error print - it might be more useful to define a custom assert macro. When debugging you are able to have your program stop on the line or bring up a message box.

```
// Saving debug information  - e.g., log file,
// debugger, messagebox, etc.

// For variable argument functions ,e.g., dprintf(..)
#include <stdlib.h>
#include <stdarg.h>
#include <stdio.h>  // vsprintf_s

inline
void dprintf(const char *fmt, ...)
{
  va_list parms;
  static char buf[2048];

  // Try to print in the allocated space.
  va_start(parms, fmt);
  // format data into a string buffer
  vsprintf_s(buf, fmt, parms);
  va_end(parms);

  // Write the information out to a txt file
  #if 0
  // open file for writing
  FILE *fp = fopen("log.txt", "a+");
  // write to file
  fprintf(fp, "%s",  buf);
  // close file
  fclose(fp);
  #endif

  // Output to the visual studio window
```

```
  OutputDebugStringA( buf );
}// End dprintf(..)

// Debug Break (win32) causes Visual Studio to halt
// on the problem line so we can analyse the issue
// in real-time

#if defined(_WIN32)
#define DBG_ASSERT(f) { if(!(f)) \
   { __debugbreak(); }; }
#else
#define DBG_ASSERT(f) { if(!(f)) \
   { assert(0); }; }
#endif

#define DBG_VALID(f)  { if( (f)!=(f) ) \
   {DBG_ASSERT(false);} }

#define DBG_ASSERT_MSG( val, errmsg ) \
   dprintf( errmsg ); \
   DBG_ASSERT( val )

#define DBG_ASSERT_VULKAN( val ) \
   DBG_ASSERT( VK_SUCCESS == val )

#define DBG_ASSERT_VULKAN_MSG( val, errmsg ) \
   dprintf( errmsg ); \
   DBG_ASSERT( VK_SUCCESS == val )
```

Furthermore, for different builds, you can have the errors handled in different ways (e.g., log file in release, message box popup, or debug assert). The assert also helps provide insight for developers on what is happening (e.g., what is expected and what isn't expected).

```
#include <stdio.h>

dprintf ("Debug Assert - line %d of file %s (function↩
     %s)\n",
   __LINE__, __FILE__, __func__);
```

3.2 Windows (Win32)

As you progress through the examples, you'll move towards outputting to the screen. For the graphical output examples

that depend on platform specific information, you'll use the Win32 API (i.e., WinMain and Windows Callback Message Loop). A skeleton Win32 listing is shown below. You have three main 'Vulkan' functions - for the creation and deletion of Vulkan resources when the application starts and closes - and the render loop - which constantly gets called (over and over again) to update the screen as fast as possible.

```
// Vulkan will run within a Win32 application
#include <windows.h>

// Store the window handle - as we use
// it later on for setting up Vulkan Surfaces
HWND g_windowHandle = NULL;

void VulkanCreate()
{
  // Vulkan setup/initialization
}// End VulkanCreate(..)

void VulkanRelease()
{
  // Tidy up and release any Vulkan resources
  // before closing down
}// End VulkanRelease(..)

void VulkanRender()
{
  // Render loop - refreshes the graphical output
}// End VulkanRender(..)

// Win32 message callback
LRESULT CALLBACK WindowProc( HWND hwnd,
                             UINT uMsg,
                             WPARAM wParam,
                             LPARAM lParam )
{
    // choose what to do with the message
    switch( uMsg )
    {
        // called when we close the application
        case WM_CLOSE:
        {
            // posts a WM_QUIT message
            PostQuitMessage( 0 );
        }
        break;

        // windows lets us know we need to redraw
        case WM_PAINT:
```

```
        {
            // Override so drawing is managed
            // by us
            VulkanRender();
        }
        break;

        // all other messages
        default:
        {
        }
        break;
    }// End switch(..)

    // a pass-through for now. We will return to this
    // callback
    return DefWindowProc(hwnd, uMsg, wParam, lParam);
}// End WindowProc(..)


// Win32 Program Entry Point
int WINAPI WinMain( HINSTANCE hInstance,
          HINSTANCE hPrevInstance,
          LPSTR lpCmdLine,
          int   nCmdShow )
{
  // defines our window properties
  WNDCLASSEX windowClass    = {};
  // size of structure in bytes
  windowClass.cbSize        = sizeof(WNDCLASSEX);
  // window styles
  windowClass.style         = CS_OWNDC | CS_VREDRAW |↩
     CS_HREDRAW;
  // call back function for message handling
  windowClass.lpfnWndProc   = WindowProc;
  // file handle
  windowClass.hInstance     = hInstance;
  // class name
  windowClass.lpszClassName = "VulkanWindowClass";
  // register our new class with windows
  RegisterClassEx( &windowClass );

  g_windowHandle =
  CreateWindowEx(
    // no additional window styles
    NULL,
    // name of our class from above
    "VulkanWindowClass",
    // application name
    "Hello Vulkan",
    // window appearance styles
    WS_OVERLAPPEDWINDOW | WS_VISIBLE,
    // x,y top left corner
```

```
    100, 100,
    // width and height of window
    width, height,
    // no parents
    NULL,
    // no popup menus
    NULL,
    // executable file handle
    hInstance,
    // no parameters to pass
    NULL );

  // check we where successful
  DBG_ASSERT(g_windowHandle!=NULL);

  // initialize and setup Vulkan
  VulkanCreate();

  MSG msg;
  while( true )
  {
    // Continually force the window to be
    // redrawn as long as no other Win32
    // messages are pending
    PeekMessage( &msg, NULL, NULL, NULL, PM_REMOVE );

    // exit the while loop if quit
    if( msg.message == WM_QUIT )
      break;

    // translates any virtual-key messages
    TranslateMessage( &msg );
    // executes the appropriate function
    DispatchMessage( &msg );

    // We constantly tell windows to refresh the
    // screen - i.e., to send a WM_PAINT message
    RedrawWindow( g_windowHandle, NULL, NULL, ↩
    RDW_INTERNALPAINT );
  }

  // release any Vulkan resources we allocated
  VulkanRelease();

  // return the last message we got from PeekMessage
  // before quitting
  return msg.wParam;;
}// End WinMain(..)
```

3.2.1 WinMain

The program entry point is called 'WinMain' - we also need to include the windows header file 'Windows.h' (for the various macros and handle defines, such as, HINSTANCE, HWND, and WINAPI).

```
// program entry point in Win32
int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPSTR lpCmdLine,
                   int nCmdShow)
```

The Win32 API function calling convention is **__stdcall** which is defined by the **CALLBACK** and **WINAPI** macros.

- hInstance - handle to current instance
- hPrevInstance - handle to previous instance
- lpCmdLine - cmd line arguments
- nCmdShow - how should the window appear when created

3.2.2 Window Creation

WNDCLASSEX - struct that holds window class information. We set the window style to include '**WS_VISIBLE**', so the window is shown as soon as it's created (so we don't need to call '**ShowWindow(..)** function).

For the window class style (**WNDCLASSEX**), we need to include the style **CS_OWNDC**. This is important when we have multiple windows, since without the **CS_OWNDC** flag the windows would not interact with one another correctly when dynamically moved about (e.g., dragged around the screen).

```
WNDCLASSEX wc;
// same as memset(&wc, 0, sizeof(WNDCLASSEX)
ZeroMemory(&wc, sizeof(WNDCLASSEX));
```

```
// size
wc.cbSize = sizeof(WNDCLASSEX);
// window style: redraw after horizont
wc.style = CS_OWNDC | CS_VREDRAW | CS_HREDRAW;
// what function shall we use to handle draws
wc.lpfnWndProc = WindowProc;
// application instance
wc.hInstance = hInstance;
// cursor type
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
// prefered window color
wc.hbrBackground = (HBRUSH)COLOR_WINDOW;
// name of the window class
wc.lpszClassName = L"MyWindowClass";
// register the window class
RegisterClassEx(&wc);
```

We have defined our window - now we call the **CreateWindowEx** to actually have Windows create our window and return the all important **HWND** window handle that we use later on in our Vulkan API (i.e., to get surface/size information).

```
HWND CreateWindowEx(
    // specifies the extended style of the window
    DWORD dwExStyle,
    // specifies the window class name
    LPCTSTR lpClassName,
    // window title (displayed in the title bar)
    LPCTSTR lpWindowName,
    // style of the window being created
    DWORD dwStyle,
    // initial horizontal position of the window
    int x,
    // initial vertical position of the window
    int y,
    // width, in device units, of the window
    int nWidth,
    // height, in device units, of the window
    int nHeight,
    // handle to the parent or owner window
    HWND hWndParent,
    // overlapped or pop-up window
    HMENU hMenu,
    // handle to application instance
    HINSTANCE hInstance,
    // usually NULL
    LPVOID lpParam)
```

CreateWindowEx will return our window handle if we're successful or NULL if a failure occurred - we need to store the handle - as we'll pass it to Vulkan when we're setting up the graphical device (e.g., render surface).

3.2.3 Message Processing

```
LRESULT CALLBACK WindowProc(
    // window handle - very important
    HWND hWnd,
    // message identifier
    UINT message,
    // message information
    WPARAM wParam,
    // more information about the message
    LPARAM lParam
    );
```

What happens when a Windows event occurs? We have to:

- use an appropriate function that gets message from the queue
- call **TranslateMessage()** translates any virtual-key messages
- call **DispatchMessage()** executes the appropriate WindowProc function

Processing Windows messages - we want to avoid 'blocking call'. Hence, be aware that the standard 'GetMessage' Win32 API function is a 'blocking' function.

```
BOOL GetMessage(
     // pointer to message struct
     LPMSG lpMsg,
     // window handle
     HWND hWnd,
     // lowest msg id value
     UINT wMsgFilterMin,
     // highest msg id value
     UINT wMsgFilterMax
     );
// blocking methods are bad - and should be
```

```
// avoided so we use the 'PeekMessage' function
// instead of 'GetMessage'.
BOOL PeekMessage(
       // pointer to message structure
       LPMSG lpMsg,
       // window handle
     HWND hWnd,
     // params
     UINT wMsgFilterMin,
     UINT wMsgFilterMax,
       // PM_REMOVE/PM_NOREMOV
     UINT wRemoveMsg);
```

Passing (0,0) in (**wMsgFilterMin**, **wMsgFilterMax**) means we don't care what we get.

The main message loop:

```
MSG msg = {0};
while(TRUE)
{
  // non-blocking 'peek' function
  if(PeekMessage(&msg, NULL, 0, 0))
  {
    if(msg.message == WM_QUIT)
      break;

  // translates any virtual-key messages
  TranslateMessage( &msg );
  // pass along to the system for processing
  DispatchMessage( &msg );
  }

  // rendering and other operations
  // (physics, AI etc.)

  // We constantly tell windows to refresh the
  // screen - i.e., to send a WM_PAINT message
  RedrawWindow( g_windowHandle, NULL, NULL, ↩
    RDW_INTERNALPAINT );

}// End While(TRUE)
```

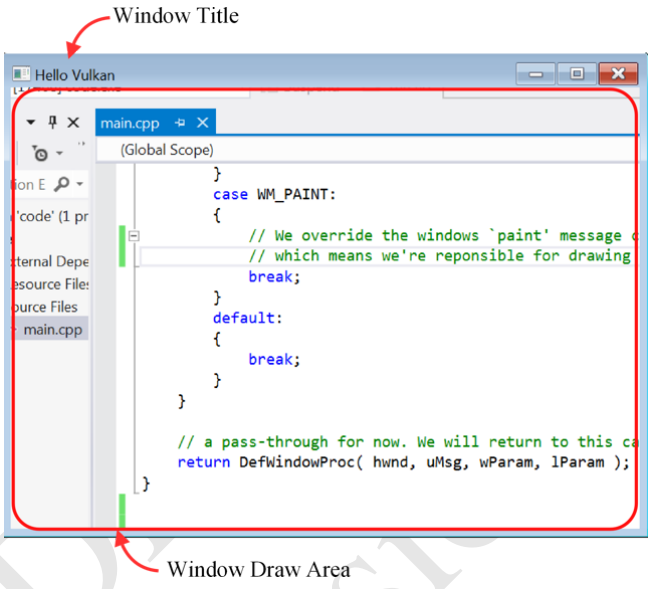

Figure 3.1: **Win32 Paint Message** - Overriding the **WM_PAINT** message in the callback function - means we're responsible for drawing the window. If you run the minimal Win32 code sample - you'll see the window draw area doesn't refresh. This is because, we'll pass the window handle (**HWND**) to Vulkan later on - who will manage the drawing.

3.3 Summary

You should have a simple Vulkan application up and running. Of course, it doesn't do much. Simply initializes and de-initialized Vulkan - but it gets you on the first step of the Vulkan ladder. In a few chapters, you'll have everything setup to run a simple graphical application (hello triangles).

3.4 Review

Remember, the Vulkan API programming methods typically start with the prefix **vk** (e.g., **vkCreateInstance(..)**). Both lowercase **vk** for methods and uppercase first letter for variable structures **Vk**. When checking the method return value for success - remember to use **VK_SUCCESS** - otherwise, the return value is a number to help diagnose the failure.

3.5 Test Yourself

Here are some multiple-choice questions to help you identify your baseline knowledge of the material. Answers appear at the end of the test.

3.5.1 Question 1

To initialize a Vulkan instance, which API method do we use:

a vkCreateInstance
b CreateInstance
c vkInstance
d VulkanCreateInstance
e vkNewInstance

3.5.2 Question 2

When completing the **vkApplicationInfo** structure, is the **sType** element mandatory?

a true
b false

3.5.3 Question 3

For the **vkInstanceCreateInfo** structure, what would you initialize the **sType** element to?

a **VK_STRUCTURE_TYPE_APPLICATION_INFO**
b **VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO**
c **VK_STRUCTURE_TYPE_TEST**
d **VK_UNDEFINED**
e **-1**

3.5.4 Question 4

The Vulkan API return value for identifying if a failure happened is:

a **NULL**
b **VK_DBG_SUCCESS**
c **0x10000000**
d **VK_OK**
e **VK_SUCCESS**

(Q1: Answer a; Q2: Answer a; Q3: Answer b; Q4: Answer e;)

4. Surfaces

4.1 Surfaces (Screen & Format)

You have created Vulkan (initialized it) - but you need to gather some information about our system. For example, the device, screen size and color format. Since you're going to use surfaces and Window for the graphical output, you'll need to modify your Vulkan initialization implementation. You need to tell Vulkan you'll be using a surfaces and the type of surfaces (i.e., Win32 screen). This information is embedded within an 'extensions' list (i.e., list of strings). What is more, you're now going to output something to the screen. So you need a 'layer'. Hence, you tell Vulkan when you initialize your instance that you need some layers, also the layers name. You've cut through a lot of red-tape here - but you'd want to iterate over the systems capabilities initially and extract

the layer and extension names using the available 'Get' API within Vulkan (e.g., **vkEnumerateInstanceExtensionProperties**).

Below shows the setup code:

```
// Store Vulkan instance handle
VkInstance   g_instance = NULL;

// Initialize VULKAN
const char *layers[] = { "VK_LAYER_NV_optimus" };
const char *extensions[] = { "VK_KHR_surface",
                             "VK_KHR_win32_surface" ↩
    };
{
// information about the application you
// want to pass to the Vulkan driver
VkApplicationInfo ai = { };
// manditory
ai.sType            = ↩
    VK_STRUCTURE_TYPE_APPLICATION_INFO;
// application name
ai.pApplicationName = "Hello Vulkan";
// a version number
ai.engineVersion    = 1;
// SDK version
ai.apiVersion       = VK_API_VERSION_1_0;


// Vulkan instance creation data
VkInstanceCreateInfo ici   = { };
// manditory type
ici.sType                   = ↩
    K_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
// reserved for future use
ici.flags                   = 0;
// pointer to an extension-specific structure
ici.pNext                   = NULL;
// application info from above
ici.pApplicationInfo        = &ai;
// number of layers we want to use
ici.enabledLayerCount       = 1;
// specify the layer names
ici.ppEnabledLayerNames     = layers;
// number of extensions
ici.enabledExtensionCount   = 2;
// the extension names
ici.ppEnabledExtensionNames= extensions;


// create Vulkan instance
```

```
VkResult result =
vkCreateInstance( &ici, NULL, &g_instance );

// where we successful
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to create vulkan instance." );

// check we have a valid handle
DBG_ASSERT(g_instance!=NULL);
}
```

If our Vulkan instance initialized without any problems, you're ready to create a surface (note - you need the Win32 windows handle for the surface compatibility search - the operating system tell you about the screen/window which helps you choose the correct graphics device). You've hardcoded the layer name (VK_LAYER_NV_optimus) in the above sample - however, depending upon your driver and graphics card, you may have to search the available layer names.

```
// Handle to our surface
VkSurfaceKHR g_surface  = NULL;

// We need to define what type of surface we'll be
// rendering to - this will depend on our computer
// and operating system
{
// setup parameters for our new windows
// surface we'll render into:
VkWin32SurfaceCreateInfoKHR sci = {};
// surface type (win32)
sci.sType      =
  VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR;
// calling GetModuleHandle with NULL returns the
// file handle used to create the calling process
sci.hinstance = GetModuleHandle(NULL);
// We should use our 'created' window handle (HWND)
sci.hwnd       = g_windowHandle;


// create surface
VkResult result =
vkCreateWin32SurfaceKHR(
  // instance
  g_instance,
  // pCreateInfo
  &sci,
  // pAllocator
  NULL,
```

```
  // pSurface
  &g_surface );

// check if successful
DBG_ASSERT_VULKAN_MSG( result,
   "Could not create surface." );
DBG_ASSERT(g_surface!=NULL);
}
```

4.2 Summary

The surface provides essential information that you'll use throughout our Vulkan application. In the next Chapter, you'll search for which 'physical' devices are available on our machine (e.g., our PC may have multiple graphics cards). You'll need to check if what the capabilities are and if the physical device support your screen/window surface properties.

4.3 Test Yourself

Here are some multiple-choice questions to help you identify your baseline knowledge of the material. Answers appear at the end of the test.

4.3.1 Question 1

To initialize and create Vulkan ('Win32') surface, which API method do we use:

a vkCreateSurface
b CreateSurface
c vkSurfaceWin32
d VulkanCreateSurfaceWin32

e vkCreateWin32SurfaceKHR

4.3.2 Question 2

When completing the **VkWin32SurfaceCreateInfoKHR** structure, is the **sType** element mandatory?

a true
b false

4.3.3 Question 3

For the **VkWin32SurfaceCreateInfoKHR** structure, what would you initialize the **sType** element to?

a **NULL**
b **VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR**
c **VK_STRUCTURE_TYPE_TEST**
d **VK_UNDEFINED**
e **-1**

(Q1: Answer e; Q2: Answer b; Q3: Answer b;)

5. Devices Enumeration

5.1 Flexibility of Vulkan

Once you've initialised Vulkan and you have a valid **VkInstance**, you're ready to get a device. In Vulkan there are two types of devices: physical and logical. A physical device, or **VkPhysicalDevice**, is a handle to an actual piece of hardware (hence the "physical"). A logical device, or **VkDevice**, is an abstraction of the underlying hardware and is what the application sees and interacts with.

Now that you have an instance, you need to a way to associate the instance with the hardware. In Vulkan there is no notion of a singular GPU, instead you enumerate physical devices and choose. This allows you to use multiple physical devices at the same time for rendering or compute.

In order to get a logical **VkDevice**, you must first get a physical device. To do this you first poll Vulkan to get a count of compatible devices on the system. This can be achieved by calling the function:

```
vkEnumeratePhysicalDevices()
```

The **vkEnumeratePhysicalDevices** function allows you to both query the count of physical devices present on the system and fill out an array of **vkPhysicalDevice** structures representing the physical devices.

```
// keep a copy of the handle to
// the physical device we use
VkPhysicalDevice g_device = VK_NULL_HANDLE;

{
// temporary variable for determining how
// many devices are present in the system
uint32_t deviceCount = 0;

// query how many devices are present
VkResult result =
vkEnumeratePhysicalDevices(
  // instance
  g_instance,
  // pPhysicalDeviceCount
  &deviceCount,
  // pPhysicalDevices
  NULL);

// check if our call was successful
DBG_ASSERT_VULKAN_MSG(result,
  "Failed to query the number of physical devices ↩
    present");

// There has to be at least one device present
DBG_ASSERT_MSG(0 != deviceCount,
  "Couldn't detect any device present with Vulkan ↩
    support");

// array to store our list of physical devices
vector<VkPhysicalDevice> physicalDevices(deviceCount)↩
    ;
// get the physical devices
result =
vkEnumeratePhysicalDevices(
  // instance
  g_instance,
```

```
  // pPhysicalDeviceCount
  &deviceCount,
  // pPhysicalDevices
  &physicalDevices[0]);

// check if we were successful
DBG_ASSERT_VULKAN_MSG(result,
  "Faied to enumerate physical devices present");
// check we have at least one physical device
DBG_ASSERT(physicalDevices.size()>0);

// Use the first available device
g_physicalDevice = physicalDevices[0];
}
```

Once you've got a physical device; you can fetch the properties of that physical device using **vkGetPhysicalDeviceProperties** which will fill out a **vkPhysicalDeviceProperties** structure:

```
// Enumerate all physical devices and print out the ↩
    details
for (uint32_t i = 0; i < deviceCount; ++i)
{
  // struct that holds the device details
  VkPhysicalDeviceProperties deviceProperties;
  memset(&deviceProperties, 0, sizeof ↩
    deviceProperties);

  // fill structure with info about the device
  vkGetPhysicalDeviceProperties(physicalDevices[i], &↩
    deviceProperties);

  dprintf("Driver Version: %d\n",
    deviceProperties.driverVersion);
  dprintf("Device Name:    %s\n",
    deviceProperties.deviceName);
  dprintf("Device Type:    %d\n",
    deviceProperties.deviceType);
  dprintf("API Version:    %d.%d.%d\n",
    (deviceProperties.apiVersion>>22)&0x3FF,
    (deviceProperties.apiVersion>>12)&0x3FF,
    (deviceProperties.apiVersion&0xFFF));
}//End for i
```

An example output for a single GPU system would be:

```
Device Name:    GT 750
Device Type:    2
API Version:    1.0.8
```

In Vulkan, the API version is encoded as a 32-bit integer with the major and minor version being encoded into bits 31-22 and 21-12 respectively (for 10 bits each.); the final 12-bits encode the patch version number. These handy macros should help with fetching some human readable digits from the encoded API integer.

```
#define VK_MAKE_VERSION(major, minor, patch) \
    (((major) << 22) | ((minor) << 12) | (patch))
```

Taking the device count provided by the enumerate function, you can correctly allocate enough **VkPhysicalDevice** structs for every device on the system. To do this you call the enumerate function once again, this time passing a pointer to the allocated array. You also pass the same unsigned integer containing the device count. When said integer does not equal zero, the function treats the value as the size of the passed device array. Since you've got this value directly from Vulkan, you can assume it's correct. However, to be sure, you can check this value again after the function call as Vulkan will change this value to the amount of device structs actually written into memory.

A call to **vkGetPhysicalDeviceProperties** can be useful if you are interested in retrieving information about the physical devices in the system. It will tell you API version, driver version, limitations, and sparse properties. For example:

```
// Vulkan 1.0 version number
#define VK_API_VERSION_1_0 VK_MAKE_VERSION(1, 0, 0)


// We are able to extract the parameters
// from the version number:
#define VK_VER_MAJOR(X) ((((uint32_t)(X))>>22)&0x3FF)
#define VK_VER_MINOR(X) ((((uint32_t)(X))>>12)&0x3FF)
#define VK_VER_PATCH(X) ((( uint32_t)(X)) & 0xFFF)
```

You should now have an array of **VkPhysicalDevice**'s, with each element containing the handle to each compatible piece of hardware on the system. Using this array, you can get the low-down on each device by using the function:

```
vkGetPhysicalDeviceProperties()
```

This function takes two parameters: a **VkPhysicalDevice** struct to get properties from, and a pointer to a **VkPhysicalDeviceProperties** struct in which to store the data. Using the device count, you can allocate a sufficient amount of property structs for all devices on the system. After allocating said property structs, you can then iterate through each **VkPhysicalDevice** and get their properties by enclosing the function inside a loop.

5.2 Summary

Each system configuration will be different (i.e., desktop computer to a mobile tablet). The ability to identify and extract hardware information is crucial - especially with multiple CPU/GPU configurations. You're able to support a larger range of different hardware configurations (e.g., discreet GPU and integrated GPU) through explicit access to multiple devices. Not to mention support different hardware generations of the future (forward thinking).

5.3 Test Yourself

Here are some multiple-choice questions to help you identify your baseline knowledge of the material. Answers appear at the end of the test.

5.3.1 Question 1

Which function do we use to determine the number of 'physical' devices available on the system?

a vkCreateDevices
b VulkanCountDevices
c VkGetNumberOfDevices
d vkEnumPhyDevices
e vkEnumeratePhysicalDevices

5.3.2 Question 2

What are the three parameters passed to **VK_MAKE_VERSION(..)**?

a day, month, year
b minor, major, version
c major, minor, patch
d platform, release, undefined

5.3.3 Question 3

We want the physical device's name, which Vulkan API function do we use:

a **VkGetDeviceName**
b **vkGetPhysicalDeviceProperties**
c **GetPhysicalProperties**
d **VulkanGetPropertiesForDevice**

(Q1: Answer e; Q2: Answer c; Q3: Answer b;)

6. Device Creation

6.1 Devices

So far you've have the ability to get the physical devices present on the system, create an instance and query the queue families supported by the physical devices. Vulkan does not operate directly on a **VkPhysicalDevice**. Instead it operates on views of a **VkPhysicalDevice** which it represents as a **VkDevice** and calls a logical device. This additional layer of abstraction is what allows us to tie together everything into an abstract usable solution.

Like the other structures you've filled out previously, **sType**, **pNext**, and **flags** are mandatory here.

Once your desired physical device has been chosen, you can begin polling it for more information needed in creating a log-

ical device. The first step in this process is getting the number of available queue families on the device. A queue is where our GPU commands will go in order to be executed by the hardware. There are various types of queue families such as Graphics or Compute. The amount and type of queue families may differ among different devices, but they all contain at least the Graphics queue family. Most operations, graphics or no, can be performed in the Graphics queue family, so the other families are simply more descriptive family identifiers.

VkDeviceCreateInfo depends on **VkQueueCreateInfo**, you'll make that struct first. You'll only be creating a single queue to start with so the parameters of the **VkQueueCreateInfo** struct are pretty straightforward. The system only has a single family so the family index will be zero and the queue count is one. The queue priorities parameter takes a pointer to an array of floats with the execution priority for each queue. Since there is only one queue, you simply pass this a single float value of 1.0. According to the specification, queue priority can range from 0.0 to 1.0, with 1.0 being higher priority.

The last struct you'll need to build is the **VkDeviceCreateInfo** struct. As mentioned earlier, this struct contains values for any layers, extensions, or device features you wish to enable for the new device. For now you'll not be using any of these, you'll see their importance later on as you start to add more features. This struct also needs a count of how many **VkQueueCreateInfo** structs, you'll be passing, and a pointer to an array of those structs. Since you only have the one queue info struct, you'll assign it to these parameters.

At last you're now ready to create our logical **VkDevice**. To do this you'll will use the function:

```
vkCreateDevice()
```

The sample listing will create and setup our **VkDevice**:

```
VkDevice g_device = NULL;
{
// Physical device memory properties structure
VkPhysicalDeviceMemoryProperties memoryProperties;
// Fill structure with information
vkGetPhysicalDeviceMemoryProperties(
// physicalDevice
g_physicalDevice,
// pMemoryProperties
&memoryProperties );

// Here's where we initialize our queues
VkDeviceQueueCreateInfo queueCreateInfo = {};
// manditory stype
queueCreateInfo.sType = ↩
    VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
// Use the first queue family in the family list
queueCreateInfo.queueFamilyIndex = 0;
queueCreateInfo.queueCount       = 1;
float queuePriorities[]          = { 1.0f };
queueCreateInfo.pQueuePriorities = queuePriorities;

VkDeviceCreateInfo dci = {};
// manditory stype
dci.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
// Set queue(s) into the device
dci.queueCreateInfoCount    = 1;
dci.pQueueCreateInfos       = &queueCreateInfo;
dci.enabledLayerCount       = 1;
dci.ppEnabledLayerNames     = layers;
dci.enabledExtensionCount   = 2;
dci.ppEnabledExtensionNames = extensions;

VkPhysicalDeviceFeatures features = {};
features.shaderClipDistance = VK_TRUE;
dci.pEnabledFeatures = &features;

// Ideally, we'd want to enumerate and find the best
// device, however, we just use the first device
// 'physicalDevices[0]' for our sample, which we
// stored in the previous chapter
VkResult result =
vkCreateDevice(
// physicalDevice
g_physicalDevice,
// pCreateInfo
&dci,
// pAllocator
NULL,
// pDevice
&g_device );
```

```
// check if we were successful
DBG_ASSERT_VULKAN_MSG( result,
"Failed to create logical device!" );
}
```

To get the data about the different available queue families and their properties you can call the function:

```
vkGetPhysicalDeviceQueueFamilyProperties()
```

This function has three parameters: a **VkPhysicalDevice** for our chosen device, a pointer to an unsigned integer, and a pointer to an array of **VkQueueFamilyProperties** structs. This function operates just like the enumerate function from earlier. You'll call it once, passing our **VkInstance** and an integer address, with the array pointer as **NULL** and the total available family count will be written into our integer. Again you'll use that information to allocate enough **VkQueueFamilyProperties** structs for each of the families available. Then, like before, you call the function a second time including the now allocated property array and Vulkan will fill said array with the desired data.

You can create many instances of the same queue family and set multiple queues into a **VkDeviceCreateInfo** structure. Just be sure to set the **queueCount** correctly. In Vulkan you can also control the priority of each queue with an array of normalized floats. A value of 1.0 has highest priority.

With that you should have a logical device setup from a physical device with your associated queues containing your application-provided information. From here you are now ready to move onto the following chapters to create the appropriate swap-chains and rendering components.

6.2 Summary

You should have our logical device (i.e., **VkDevice**). You are ready to start configuring the device resources (e.g., memory, framebuffer and swap-chains).

6.3 Test Yourself

Here are some multiple-choice questions to help you identify your baseline knowledge of the material. Answers appear at the end of the test.

6.3.1 Question 1

You create a logical device using the function:

a vkCreateLogicalDevices
b VkDevice
c VkGetDevices
d VulkanDevice

6.3.2 Question 2

Which function gets the data about the different available queue families and their properties for the device:

a VkGetPhysicsDeviceProperites
b vKGetDeviceFamilyProperties
c vkDeviceQueue
d vkGetPhysicalDeviceQueueFamilyProperties

(Q1: Answer b; Q2: Answer d;)

7. Swap-Chains

7.1 Buffering & Synchronization

Up until now everything you've done with Vulkan has been setting the ground work with no real end-goal of how this is all going to connect to the renderer. You're now going to use the results of previous chapters to build a swap-chain which plays a crucial part in rendering.

So what is a swap-chain? A swap-chain is effectively a circular buffer of images (and image-views) for presenting your graphical output to the window. It specifies how your buffer will be rendered (choice of double, triple, or more, buffering); as well as the synchronization mode (swap, tear, or vertical synchronization).

7.1.1 Swap-chain

This is where you begin to write your core graphical components. Each of these graphical components will typically depend upon each other (i.e., each chapter will build upon the previous chapter). For example, setting up the physical device (**VkPhysicalDevice**) and window surface (**VkSurfaceKHR**) was covered previously and is used here. So if you didn't create a **VkPhysicalDevice**, you won't be able to create a swap-chain. The basic setup of a double buffered swap-chain listing is shown below.

```
// swap-chain handle
VkSwapchainKHR g_swapChain = NULL;

// Create swap chain
{
// swap chain creation:

// structure listing surface capabilities
VkSurfaceCapabilitiesKHR surfaceCapabilities = {};
// fill structure with data
vkGetPhysicalDeviceSurfaceCapabilitiesKHR(
  // physicalDevice
  g_physicalDevice,
  // surface
  g_surface,
  // pSurfaceCapabilities
  &surfaceCapabilities );


VkExtent2D surfaceResolution =
  surfaceCapabilities.currentExtent;

g_width  = surfaceResolution.width;
g_height = surfaceResolution.height;

VkSwapchainCreateInfoKHR ssci = {};
// manditory stype
ssci.sType            =
  VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
ssci.surface          = g_surface;
// We'll use 2 for 'double' buffering
ssci.minImageCount    = 2;
ssci.imageFormat      = VK_FORMAT_B8G8R8A8_UNORM;
ssci.imageColorSpace  =
  VK_COLORSPACE_SRGB_NONLINEAR_KHR;
ssci.imageExtent      = surfaceResolution;
ssci.imageArrayLayers = 1;
```

```
ssci.imageUsage        =
  VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
ssci.imageSharingMode = VK_SHARING_MODE_EXCLUSIVE;
ssci.preTransform      = ↩
    VK_SURFACE_TRANSFORM_IDENTITY_BIT_KHR;
ssci.compositeAlpha    =
  VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
ssci.presentMode       = VK_PRESENT_MODE_MAILBOX_KHR;
// If we want clipping outside the extents
ssci.clipped           = true;
ssci.oldSwapchain      = NULL;

VkResult result =
vkCreateSwapchainKHR(
  // device
  g_device,
  // pCreateInfo
  &ssci,
  // pAllocator
  NULL,
  // pSwapchain
  &g_swapChain );

// check if we were successful
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to create swapchain." );
}
```

You're almost there, now all that's left is to do, is call the function for actually creating the swap-chain resources and making it active for a **VkCommandBuffer**. You'll be exploring for the first time how to utilize the physical device surface properties and formats to guide your creation of the swap-chain. Like all creation functions in Vulkan, you'll be filling out a creation info structure **VkSwapchainCreateInfoKHR** to actually create our swap-chain.

One thing, before you actually do that, you also need to touch on presentation modes. In Vulkan you get to choose how you want frames presented to the swap-chain. The presentation modes to choose from are:

```
VK_PRESENT_MODE_MAILBOX_KHR
VK_PRESENT_MODE_IMMEDIATE_KHR
VK_PRESENT_MODE_FIFO_RELAXED_KHR
VK_PRESENT_MODE_FIFO_KHR
```

All compliant implementations of Vulkan must support **VK_PRESENT_MODE_FIFO_KHR**. The others are optional.

VK_PRESENT_MODE_MAILBOX_KHR Optimized v-sync technique, will not screen-tear. Has more latency than tearing. Generally speaking this is the preferred presentation mode if supported for it is the lowest latency non-tearing presentation mode.

VK_PRESENT_MODE_IMMEDIATE_KHR does not vertical synchronize and will screen-tear if a frame is late. This is basically useless unless you're doing one-off rendering to a surface where the result is not immediately needed.

VK_PRESENT_MODE_FIFO_RELAXED_KHR normally vertical synchronizes but will screen-tear if a frame is late. This is the same as 'late swap tearing' extensions in GL/D3D. The idea behind this is to allow late swaps to occur without synchronization to the video frame. It does reduce visual stuffer on late frames and reduces the stall on subsequent frames.

VK_PRESENT_MODE_FIFO_KHR will vertical synchronizes and won't screen-tear. This is your standard vertical synchronization.

7.2 Creating Images

Swap-chains do not contain any images - they just manage them. Hence, we need to create images for our 'double' buffered screen output.

```
// Store a pointer to the created images
VkImage* g_presentImages = NULL;

// Create two images for 'double' buffering
{
```

```
// temporary variable
uint32_t imageCount    = 0;
// ask how many images we need to create
vkGetSwapchainImagesKHR( g_device, g_swapChain, &↩
    imageCount, NULL );
// we should have 2 images for double buffering
DBG_ASSERT(imageCount==2);

// this should be 2 for double-buffering
g_presentImages = new VkImage[imageCount];

// link the images to the swapchain
VkResult result =
vkGetSwapchainImagesKHR(
  // device
  g_device,
  // swapchain
  g_swapChain,
  // pSwapchainImageCount
  &imageCount,
  // pSwapchainImages
  g_presentImages );

// check if we were successful
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to create swap-chain images" );
}
```

You can ask your swap-chain how many images it holds. For our simple double buffered example, this should be 2. However, you have the flexibility to use as many images as you want (e.g., triple buffered, quad-triple). You can then go ahead and allocate the necessary images (i.e., **VkImage**) and attach them to your swap-chain.

7.3 Image Views

Interestingly, you can't directly access an 'Image' - instead you must create an 'ImageView'. This provides a 'handle' for the image, as shown below:

```
// Imageview handle
VkImageView *g_presentImageViews = NULL;
// Create image view and attach them to the images
```

```
{
// We have 2 for double buffering
g_presentImageViews = new VkImageView[2];

// create image view for each image
for( uint32_t i = 0; i < 2; ++i )
{
  // create VkImageViews for our swap chain
  // VkImages buffers:
  VkImageViewCreateInfo ivci = {};
  // manditory
  ivci.sType                =
    VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
  ivci.viewType         = VK_IMAGE_VIEW_TYPE_2D;
  ivci.format           = VK_FORMAT_B8G8R8A8_UNORM;
  ivci.components.r     = VK_COMPONENT_SWIZZLE_R;
  ivci.components.g     = VK_COMPONENT_SWIZZLE_G;
  ivci.components.b     = VK_COMPONENT_SWIZZLE_B;
  ivci.components.a     = VK_COMPONENT_SWIZZLE_A;
  ivci.subresourceRange.aspectMask     = ↩
    VK_IMAGE_ASPECT_COLOR_BIT;
  ivci.subresourceRange.baseMipLevel   = 0;
  ivci.subresourceRange.levelCount     = 1;
  ivci.subresourceRange.baseArrayLayer = 0;
  ivci.subresourceRange.layerCount     = 1;
  ivci.image                           = ↩
    g_presentImages[i];

  VkResult result =
  vkCreateImageView(
    // device
    g_device,
    // pCreateInfo
    &ivci,
    // pAllocator
    NULL,
    // pView
    &g_presentImageViews[i] );

  // check everything went okay
  DBG_ASSERT_VULKAN_MSG( result,
    "Could not create ImageView." );
}// End for i
}
```

7.4 Summary

In this chapter, you've got a little closer to actually rendering something on screen. For example, at this stage, you've initialized Vulkan, created a compatible surface, and now, a swap-chain structure with images. Your swap-chain will be responsible for managing your front and back buffer. You'll need the your swap-chain later on in the main render loop, where you'll have swap-chain images passed into the render pipeline (e.g. you'll get the image index and set it to be drawn/updated and presented).

7.5 Test Yourself

Here are some multiple-choice questions to help you identify your baseline knowledge of the material. Answers appear at the end of the test.

7.5.1 Question 1

We create a swap-chain using the function:

a vkCreateLogicalChain
b VkSwapChain
c CreateSwapChain
d vkCreateSwapchainKHR

7.5.2 Question 2

When we create **VkImageView**'s, we define a **VkImageViewCreateInfo** structure, what is the value for the 'sType'?

a VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO
b VK_STRUCTURE_TYPE_IMAGE_VIEW_INFO
c NULL
d -1
e STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO

7.5.3 Question 3

The **VK_PRESENT_MODE_MAILBOX_KHR** define is:

a Does not vertical synchronize and will screen-tear if a frame is late.
b Optimized v-sync technique that will not screen-tear (but has more latency than tearing).
c Will vertical synchronizes and won't screen-tear.

(Q1: Answer d; Q2: Answer a; Q3: Answer b;)

8. Device Queues

8.1 Commands

Your computer typically has a CPU with 4, 8 or more cores. However, traditional graphics API are only able to utilize a single core - so only a single CPU core is able to speak to the GPU at any one time. This was a serious bottleneck that Vulkan addressed through the use of the device queue and custom command buffers. You're now able to take full advantage of both the CPU and GPU - use all of their resources (multi-core scaling for future hardware). With Vulkan you can now have 'all' your CPU cores speak to your GPU simultaneously (remember, you may have multiple CPUs and GPUs - you have to think of the future). Previously, you'd not be able to do amazing things on the GPU beacuse the CPU could not feed the GPU fast enough - however, with every single core of the GPU beging pushed to its limits - things have changed.

The Vulkan API means you're no longer limited by the CPU - instead the limitation is determined by the systems power (GPU) and your software abilities.

For your simple program, you need to give your device commands. These commands get queued and executed (depending upon the specified order and dependencies). Commands are recorded into command buffers ahead of execution time. These same buffers are then submitted to queues for execution. Each physical devices provides a family of queues to choose from. The choice of the queue depends on the task at hand.

A Vulkan queue can support one or more of the following operations (in order of most common):

graphic	**VK_QUEUE_GRAPHICS_BIT**
compute	**VK_QUEUE_COMPUTE_BIT**
transfer	**VK_QUEUE_TRANSFER_BIT**
sparse memory	**VK_QUEUE_SPARSE_BINDING_BIT**

This is encoded in the queueFlags field of the **vkQueueFamilyProperties** structures filled out by **vkGetPhysicalDeviceQueueFamilyProperties**. Which, like **vkEnumeratePhysicalDevices** can also be used to query the count of available queue families.

While the queue support bits are pretty straightforward; something must be said about **VK_QUEUE_SPARSE_BINDING_BIT**. If this bit is set, it indicates that the queue family supports sparse memory management operations. Which means you can submit operations that operate on sparse resources. If this bit is not present, submitting operations with sparse resource is undefined. Sparse resources will be covered in later chapters as they are an advanced topic.

The listing below allows you to output your device memory capabilities:

```
// temporary variable
uint32_t queueFamilyCount = 0;

// query the device for the family count
vkGetPhysicalDeviceQueueFamilyProperties(
  // physicalDevice
  physicalDevice,
  // pQueueFamilyPropertyCount
  &queueFamilyCount,
  // pQueueFamilyProperties
  NULL);

// create array of the size of the family count
vector<VkQueueFamilyProperties> familyProperties(↩
    queueFamilyCount);
// retrieve properties
vkGetPhysicalDeviceQueueFamilyProperties(
  // physicalDevice
  physicalDevice,
  // pQueueFamilyPropertyCount
  &queueFamilyCount
  // pQueueFamilyProperties
  &damilyProperties[0]);

// Print the families
for (uint32_t i = 0; i < deviceCount; ++i)
{
 for (uint32_t j = 0; j < queueFamilyCount; ++j)
 {
  dprintf("Count of Queues: %d\n", familyProperties[j↩
    ].queueCount);
  dprintf("Supported operationg on this queue:\n");

  if (familyProperties[j].queueFlags & ↩
    VK_QUEUE_GRAPHICS_BIT)
    dprintf("\t\t Graphics\n");
  if (familyProperties[j].queueFlags & ↩
    VK_QUEUE_COMPUTE_BIT)
    dprintf("\t\t Compute\n");
  if (familyProperties[j].queueFlags & ↩
    VK_QUEUE_TRANSFER_BIT)
    dprintf("\t\t Transfer\n");
  if (familyProperties[j].queueFlags & ↩
    VK_QUEUE_SPARSE_BINDING_BIT)
    dprintf("\t\t Sparse Binding\n");
 }// End for j
}// End for i
```

Example output for your system, might be:

```
Count of Queues: 16
  Supported operationg on this queue:
  Graphics
  Compute
  Transfer
  Sparse Binding

Count of Queues: 1
  Supported operationg on this queue:
  Transfer
```

The actual operations within the **VkCommandBuffer** should not sound too unfamiliar. A RenderPass is similar to framebuffer-object binding, and a DescriptorSet handles uniform bindings (buffer, texture), which you'll cover more later on.

- VkPhysicalDevice: The device is used to query information, and to create most of Vulkan's API objects
- Queue: A device can expose multiple queues. For example, there can be dedicated queue to copying data, or the compute and/or graphics queue. Operations on a single queue are typically processed in-order, but multiple queues can overlap in parallel
- VkCommandBuffer: Here we record the general commands such as setting state, executing work like drawing from vertex-buffers, dispatching compute grids, copying between buffers... function wise nothing fundamentally different. While there are still costs for building, the submission to the queue will be rather quick

```
// Store our command and queue handles
VkCommandBuffer g_drawCmdBuffer = NULL;
VkQueue         g_presentQueue  = NULL;

// Give our device some commands (orders)
{
// we can now get the device queue we will
// be submitting work to:
vkGetDeviceQueue(
  // device
  g_device,
  // queueFamilyIndex
```

```
  0,
  // queueIndex
  0,
  // pQueue
  &g_presentQueue );

// create our command buffers:
VkCommandPoolCreateInfo cpci = {};
// manditory
cpci.sType = ↩
    VK_STRUCTURE_TYPE_COMMAND_POOL_CREATE_INFO;
cpci.flags = ↩
    VK_COMMAND_POOL_CREATE_RESET_COMMAND_BUFFER_BIT;
cpci.queueFamilyIndex = 0;

// store query result
VkCommandPool commandPool;

VkResult result =
vkCreateCommandPool(
  // device
  g_device,
  // pCreateInfo
  &cpci,
  // pAllocator
  NULL,
  // pCommandPool
  &commandPool );

// check if successful
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to create command pool." );

VkCommandBufferAllocateInfo cbai = {};
cbai.sType       = ↩
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
cbai.commandPool = commandPool;
cbai.level       = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
cbai.commandBufferCount = 1;


result =
vkAllocateCommandBuffers(
  // device
  g_device,
  // pAllocateInfo
  &cbai,
  // pCommandBuffers
  &g_drawCmdBuffer );

// check if successful
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to allocate draw command buffer." );
}
```

8.2 Summary

Since GPU performance is increasing faster than single/multi core CPU performance, we can end up GPU bound (GPU is waiting for the CPU). Hence, optimising the command buffer, we allow multi-threaded command buffer recording and queues to gain greater throughput. While the new API requires additional work, it offers significant performance improvements in real-time applications that want to push the limits of the CPU and GPU. In conclusion, we now have support for multi-threaded command buffer recording, low-level memory management, and multiple asynchronous queues.

8.3 Test Yourself

Here are some multiple-choice questions to help you identify your baseline knowledge of the material. Answers appear at the end of the test.

8.3.1 Question 1

Which best defines the CommandBuffer?

a Integrates the physical device and the swapchain.
b Controls the encoding and decoding of graphical data.
c Records general commands, such as, setting states, executing work like drawing from vertex-buffers, dispatching compute grids, and copying between buffers.

8.3.2 Question 2

What is the function for allocating a command buffer?

a vkCreateCommandBuffer
b AllocateCommandBuffer
c vkAllocateCommandBuffers
d vkCommandBufferAllocate

(Q1: Answer c; Q2: Answer c;)

9. Framebuffer

9.1 Framebuffer

The term 'framebuffer' is traditionally referred to as the video memory used to hold the image displayed on the screen. With the memory size for the image depending primarily on the resolution of the screen and the color depth used per pixel (8-bit, 16-bit, 32-bit). In Vulkan, we have a **VkFramebuffer**, which is a collection of VkImageViews. The thing to remember is, about the Vulkan framebuffer is you control the images you are rendering at any given point (i.e., not just a single image for the whole screen). As the **VkFramebuffer** can contain multiple images - more than the renderer would be able to render at once.

Below shows our basic framebuffer/renderpass setup code:

```
// Store handles to framebuffer
VkFramebuffer* g_frameBuffers = NULL;
VkRenderPass   g_renderPass   = NULL;

// Frame buffer
{
// define our attachment points
VkAttachmentDescription pass[1] = { };
pass[0].format         = VK_FORMAT_B8G8R8A8_UNORM;
pass[0].samples        = VK_SAMPLE_COUNT_1_BIT;
pass[0].loadOp         = VK_ATTACHMENT_LOAD_OP_CLEAR;
pass[0].storeOp        = VK_ATTACHMENT_STORE_OP_STORE↩
    ;
pass[0].stencilLoadOp  = ↩
    VK_ATTACHMENT_LOAD_OP_DONT_CARE;
pass[0].stencilStoreOp = ↩
    VK_ATTACHMENT_STORE_OP_DONT_CARE;
pass[0].initialLayout  = ↩
    VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
pass[0].finalLayout    = ↩
    VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;


VkAttachmentReference ar = {};
ar.attachment = 0;
ar.layout     = ↩
    VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;

// create the one main subpass of our renderpass:
VkSubpassDescription subpass    = {};
subpass.pipelineBindPoint       = ↩
    VK_PIPELINE_BIND_POINT_GRAPHICS;
subpass.colorAttachmentCount    = 1;
subpass.pColorAttachments       = &ar;
subpass.pDepthStencilAttachment = NULL;

// create our main renderpass:
VkRenderPassCreateInfo rpci = {};
rpci.sType           = ↩
    VK_STRUCTURE_TYPE_RENDER_PASS_CREATE_INFO;
rpci.attachmentCount = 1;
rpci.pAttachments    = pass;
rpci.subpassCount    = 1;
rpci.pSubpasses      = &subpass;


VkResult result =
vkCreateRenderPass(
  // device
  g_device,
  // pCreateInfo
  &rpci,
  // pAllocator
  NULL,
```

```
  // pRenderPass
  &g_renderPass );

// check if successful
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to create renderpass" );


// create our frame buffer:
VkImageView frameBufferAttachments[1] = {0};

VkFramebufferCreateInfo fbci = {};
fbci.sType = ↩
    VK_STRUCTURE_TYPE_FRAMEBUFFER_CREATE_INFO;
fbci.renderPass      = g_renderPass;
// must be equal to the attachment count
// on render pass
fbci.attachmentCount = 1;
fbci.pAttachments    = frameBufferAttachments;
fbci.width           = g_width;
fbci.height          = g_height;
fbci.layers          = 1;

// create a framebuffer per swap chain imageView:
g_frameBuffers = new VkFramebuffer[ 2 ];
for( uint32_t i = 0; i < 2; ++i )
{
  frameBufferAttachments[0] = g_presentImageViews[ i ↩
    ];

  result =
  vkCreateFramebuffer(
    // device
    g_device,
    // pCreateInfo
    &fbci,
    // pAllocator
    NULL,
    // pFramebuffer
    &g_frameBuffers[i] );

  // check if successful
  DBG_ASSERT_VULKAN_MSG( result,
    "Failed to create framebuffer.");
}// End for i

}
```

9.2 Summary

Vulkan's approach to rendering and the framebuffer is more explicit - denoting every detail, rather than letting your driver/API make assumptions. This added control and information, will aid you and everyone in the future, for example, it helps with tile based solutions, so you have direct control over rendering and the dependencies.

9.3 Test Yourself

Here are some multiple-choice questions to help you identify your baseline knowledge of the material. Answers appear at the end of the test.

9.3.1 Question 1

What is the name of the function for creating a framebuffer?

a vkCreateFramebuffer
b CreateFramebuffer
c vkNewFramebuffer
d VulkanCreateFramebuffer

9.3.2 Question 2

The Vulkan **VkFramebuffer** is a collection of **VkImageView**s?

a true
b false

(Q1: Answer b; Q2: Answer a;)

10. Displaying (Presenting)

10.1 Presenting

Finally, you'll have setup your Vulkan graphical components - but you haven't got any output - how do you know everything is connected and working together? At this point, you've written a lot of code - and yet to see anything in return - which can be a bit disheartening. You'll do a simple render test in this chapter so you can check you're on track. The sample in this chapter, will let you clear the screen using the Vulkan API. The 'VulkanRender' function below is constantly called (in the 'render-loop'). Each frame you'll clear the surface of the presentation screen to a changing color (magenta to white). Not a very sophisticated example, but it enables you to test that everything is running - compared to a blank screen.

```
// called each time we draw the screen
```

```
void VulkanRender( )
{
  // Render loop - refreshes the graphical output
  // clearing screen and gradually changing color

  // temp variable to get the image id from
  uint32_t nextImageIdx;

  // query the swap-chain for the image id
  vkAcquireNextImageKHR(
    // device
    g_device,
    // swapchain
    g_swapChain,
    // timeout
    UINT64_MAX,
    // semaphore
    VK_NULL_HANDLE,
    // fence
    VK_NULL_HANDLE,
    // pImageIndex
    &nextImageIdx );

  // buffer commands
  VkCommandBufferBeginInfo beginInfo = {};
  beginInfo.sType = ↩
    VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
  beginInfo.flags = ↩
    VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

  // start recording commands
  vkBeginCommandBuffer( g_drawCmdBuffer, &beginInfo )↩
    ;
  {
    // temporary float that oscilates between 0 and 1
    // to gradually change the color on the screen
    static float aa = 0.0f;
    // slowly increment
    aa += 0.001f;
    // when value reaches 1.0 reset to 0
    if ( aa >= 1.0 ) aa = 0;


    // activate render pass:
    // clear color (r,g,b,a)
    VkClearValue clearValue[] = {
      { 1.0f, aa, 1.0f, 1.0f },  //color
      { 1.0, 0.0 } };            //depth stencil

    // define render pass structure
    VkRenderPassBeginInfo renderPassBeginInfo = {};
    renderPassBeginInfo.sType       = ↩
    VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
```

```
    renderPassBeginInfo.renderPass  = g_renderPass;
    // which image to draw on (use the id from
    // the swap-chain lookup we did at the start)
    renderPassBeginInfo.framebuffer = g_frameBuffers[↩
      nextImageIdx ];
    VkOffset2D a = {0, 0};
    VkExtent2D b = {g_width, g_height};
    VkRect2D   c = {a,b};
    renderPassBeginInfo.renderArea      = c;
    renderPassBeginInfo.clearValueCount = 2;
    renderPassBeginInfo.pClearValues    = clearValue;

    // command to start a render pass
    vkCmdBeginRenderPass(
      g_drawCmdBuffer,
      &renderPassBeginInfo,
      VK_SUBPASS_CONTENTS_INLINE );
    {
    // to come later - call draw commands
    // to present geometry data/triangles
    }

    // command to end the render pass
    vkCmdEndRenderPass( g_drawCmdBuffer );

}
// end recording commands
vkEndCommandBuffer( g_drawCmdBuffer );


// present:
// create a fence to inform us when the GPU
// has finished processing our commands

// setup the type of fence
VkFence renderFence;
VkFenceCreateInfo fenceCreateInfo = {};
fenceCreateInfo.sType = ↩
  VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;

// create the fence
vkCreateFence(
  // device
  g_device,
  // pCreateInfo
  &fenceCreateInfo,
  // pAllocator
  NULL,
  // pFence
  &renderFence );

// configure the queue submit structure
VkSubmitInfo si = {};
```

```
  si.sType                = ↩
    VK_STRUCTURE_TYPE_SUBMIT_INFO;
  si.waitSemaphoreCount   = 0;
  si.pWaitSemaphores      = VK_NULL_HANDLE;
  si.pWaitDstStageMask    = NULL;
  si.commandBufferCount   = 1;
  si.pCommandBuffers      = &g_drawCmdBuffer;
  si.signalSemaphoreCount = 0;
  si.pSignalSemaphores    = VK_NULL_HANDLE;

  // submit the command queue (notice, we
  // also pass the fence, to let us know the
  // gpu has finished)
  vkQueueSubmit(
    // queue
    g_presentQueue,
    // submitCount
    1,
    // pSubmits
    &si,
    // fence
    renderFence );

  // wait until the GPU has finished processing
  // the commands
  vkWaitForFences(
    g_device,
    1,
    &renderFence,
    VK_TRUE,
    UINT64_MAX );

  vkDestroyFence(
    g_device,
    renderFence,
    NULL );

  // present the image on the screen (flip the
  // swap-chain image)
  VkPresentInfoKHR pi = {};
  pi.sType            = ↩
    VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
  pi.pNext            = NULL;
  pi.waitSemaphoreCount = 0;
  pi.pWaitSemaphores = VK_NULL_HANDLE;
  pi.swapchainCount   = 1;
  pi.pSwapchains      = &g_swapChain;
  pi.pImageIndices    = &nextImageIdx;
  pi.pResults         = NULL;
  vkQueuePresentKHR( g_presentQueue, &pi );

}// End VulkanRender(..)
```

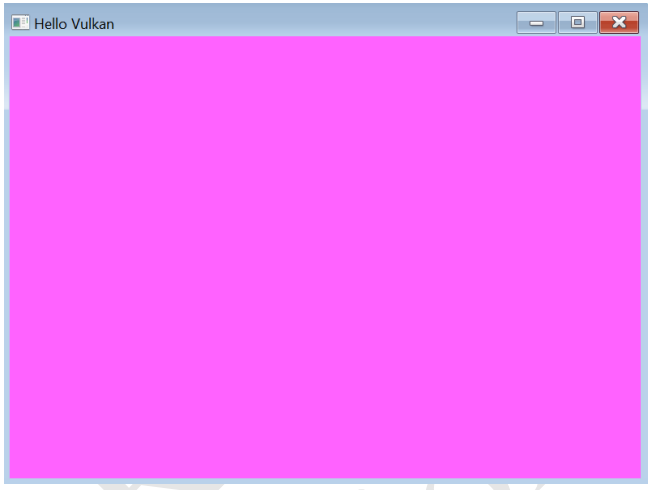

Figure 10.1: **Changing Color** - While our simple clear-screen example only displays a window that is changing color - it helps show us that our Vulkan components are actually working together.

10.2 Summary

At the end of this chapter, you should have a 'minimal' Vulkan program working - a skeleton configuration. Now you've connected the bones together to get the Vulkan program up and running quickly - you can work on fleshing it out and adding the meat (i.e., some actual graphics). Remember, your implementation still lacks lots of checking and device enumeration - but you should start to be getting a feel for how things work. So what else is missing? To get triangles on the screen (i.e., a full 3D renderer configuration), you need to add the shader buffer and some vertex data; not to mention a 'depth buffer'. The following chapters will now take you a step further to get geometry on screen - 'hello triangle'.

10.3 Test Yourself

Here are some multiple-choice questions to help you identify your baseline knowledge of the material. Answers appear at the end of the test.

10.3.1 Question 1

What two arguments does **vkBeginCommandBuffer** take?

a VkCommandBuffer, VkCommandBufferBeginInfo
b VkCommandBuffer, VkImage
c VkRenderPass, VkCommandBufferBeginInfo
d VkImage, VkImage

10.3.2 Question 2

For the **VkSubmitInfo** structure, what is the '**sType**' parameter?

a STRUCTURE_TYPE_SUBMIT_INFO
b VK_STRUCTURE_TYPE_SUBMIT_INFO
c NULL
d -1
e VK_STRUCTURE_INFO

(Q1: Answer a; Q2: Answer b;)

11. Triangle Data

11.1 Vertices & Buffers

In this chapter, you'll create a vertex buffer for a basic 3-dimensional colored triangle. Once you get your triangle on screen, you'll create and modify a matrix transform (i.e., view and projection matrices) to move the triangle around in real-time. Once you have one triangle on screen, you can expand the sample code to include more triangles (complex meshes) and multiple buffers and optimisations.

Here is the listing for the creation of the buffer and vertex data:

```
// Handle to our vertex buffer data
VkBuffer g_vertexInputBuffer = NULL;

// Create vertex buffer setup its configuration
{
```

```
// Vertex info
struct vertex
{
  float x, y, z, w; // position
  float r, g, b;    // color
};

// create our vertex buffer:
VkBufferCreateInfo vertexInputBufferInfo = {};
// manditory (type of buffer)
vertexInputBufferInfo.sType  = ↩
    VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
// size in bytes of our data
vertexInputBufferInfo.size   = sizeof(vertex) * 3;
// what the buffer will hold
vertexInputBufferInfo.usage  = ↩
    VK_BUFFER_USAGE_VERTEX_BUFFER_BIT;
// sharing/access level
vertexInputBufferInfo.sharingMode = ↩
    VK_SHARING_MODE_EXCLUSIVE;
// ignore for now
vertexInputBufferInfo.queueFamilyIndexCount = 0;
vertexInputBufferInfo.pQueueFamilyIndices   = NULL;


// create the vertex buffer
VkResult result =
vkCreateBuffer(
  // device
  g_device,
  // pCreateInfo
  &vertexInputBufferInfo,
  // pAllocator
  NULL,
  // pBuffer
  &g_vertexInputBuffer );

// check if successful
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to create vertex input buffer." );

// allocate memory for buffers:

// get the memory information - type of memory ↩
    required
// for our vertex buffer (we can't put the data just ↩
    anywhere)
VkMemoryRequirements vertexBufferMemoryRequirements =↩
    {};
vkGetBufferMemoryRequirements(
  // device
  g_device,
  // buffer
```

```
  g_vertexInputBuffer,
  // pMemoryRequirements
  &vertexBufferMemoryRequirements );

// specify our memory allocation details
// (i.e., the size and type of memory, which we
// pass to the allocator when we want to create a
// chunk of memory)
VkMemoryAllocateInfo bufferAllocateInfo = {};
// manditory structure define
bufferAllocateInfo.sType = ↩
    VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
// how much memory in bytes
bufferAllocateInfo.allocationSize = ↩
    vertexBufferMemoryRequirements.size;
// ignore
bufferAllocateInfo.pNext = NULL;

// the memory our vertex needs
uint32_t vertexMemoryTypeBits = ↩
    vertexBufferMemoryRequirements.memoryTypeBits;

// we need to search through the physical
// devices memory and find the index of the
// memory index that we need

// set the search flag to a 'default'
VkMemoryPropertyFlags vertexDesiredMemoryFlags =   ↩
    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT;

// read the device memory properties
VkPhysicalDeviceMemoryProperties memoryProperties;
vkGetPhysicalDeviceMemoryProperties( g_physicalDevice↩
    , &memoryProperties );

// check each of the memory types and store
// the one we want
for( uint32_t i = 0; i <VK_MAX_MEMORY_TYPES; ++i )
{
  VkMemoryType memoryType = memoryProperties.↩
    memoryTypes[i];
  // is this the memory type we are looking for?
  if( ( memoryType.propertyFlags & ↩
    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT ) )
  {
    // save location
    bufferAllocateInfo.memoryTypeIndex = i;
  // exit loop
  }
}// End for i

// allocate memory using the allocator info
VkDeviceMemory vertexBufferMemory;
```

```
result = vkAllocateMemory(
  g_device,
  &bufferAllocateInfo,
  NULL,
  &vertexBufferMemory );

// check if successful
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to allocate buffer memory." );

// we'll set buffer contents

// temp pointer
void *mapped;
// lock memory and set the temp pointer to point
// to the memory we want to write to
result =
vkMapMemory(
  // device
  g_device,
  // memory
  vertexBufferMemory,
  // offset
  0,
  // size
  VK_WHOLE_SIZE,
  // flags
  0,
  // ppData
  &mapped );

// check if successful
DBG_ASSERT_VULKAN_MSG( result,
"Failed to mapp buffer memory." );

// cast our temp pointer to a vertex type
vertex *triangle = (vertex *) mapped;

// define our triangle vertices
// facing down the z-axis
vertex v1 =
{   -1.0f, -1.0f, 0.0f, 1.0f,   // position
     0.0f,  1.0f, 0.0f};        // color
vertex v2 =
{   1.0f, -1.0f, 0.0f, 1.0f,
    1.0f,  0.0f, 0.0f};
vertex v3 =
{   0.0f,  1.0f, 0.0f, 1.0f,
    0.0f,  0.0f, 1.0f};

// set triangle vertices on our
// locked allocated buffer
triangle[0] = v1;
```

```
triangle[1] = v2;
triangle[2] = v3;

// unlock the buffer so the GPU can use it
vkUnmapMemory( g_device, vertexBufferMemory );

// connect the buffer with the shader
result =
vkBindBufferMemory(
  // device
  g_device,
  // buffer
  g_vertexInputBuffer,
  // memory
  vertexBufferMemory,
  // memoryOffset
  0 );

// check if successful
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to bind buffer memory." );
}
```

The 'vertexbuffer' holds our triangle information (i.e., vertices) which you'll use in your render loop command buffer (**vkCmdBindVertexBuffers**).

The triangle data is stored in an allocated buffer. This buffer can't be used immediately after creation as no memory has been allocated initially (i.e., you create the buffer but it's just a handle - it has no size).

Different memory types:

```
VK_MEMORY_PROPERTY_DEVICE_LOCAL_BIT  = 0x00000001,
VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT  = 0x00000002,
VK_MEMORY_PROPERTY_HOST_COHERENT_BIT = 0x00000004,
VK_MEMORY_PROPERTY_HOST_CACHED_BIT   = 0x00000008,
...
```

You need to understand what type of memory you're dealing with - as it will influence how you create and update it. The available memory is exposed to applications by the **vkGetPhysicalDeviceMemoryProperties()**. It reports one or more memory heaps of given sizes, and one or more memory types

with given properties. Each memory type comes from one heap - so a typical example for a discrete GPU on a PC would be two heaps - one for system RAM, and one for GPU RAM, and multiple memory types from each.

The memory types have different properties. Some will be CPU visible or not, coherent between GPU and CPU access, cached or uncached. You can find out all of these properties by querying from the physical device. This allows you to choose the memory type you want. For instance, if you stage your resources you'll need to host them so they're in visible memory, but if you render to an image you'll probably want to host them in device memory for optimal use. However there is also additional restriction on memory selection.

To allocate memory you call **vkAllocateMemory()** which requires your **VkDevice** handle and a description structure. The structure dictates which type of memory to allocate from which heap and how much to allocate, and returns a **VkDeviceMemory** handle.

Host visible memory can be mapped for update - with **vkMapMemory()** and **vkUnmapMemory()**. These familiar mapping functions are by definition persistent, and as long as you synchronise provide legal access to memory while in use by the GPU.

11.2 Summary

You'll have set-up your Vulkan data (triangle vertices) for the render loop, which you'll modify in later chapters. Previously, the render-loop was only responsible for clearing the screen - however, later you've now passed the vertex buffer information along to your render pipeline so you get a triangle on screen. After you've got one triangle working, it's only a matter of adding more triangles to display more complex

geometry.

11.3 Test Yourself

Here are some multiple-choice questions to help you identify your baseline knowledge of the material. Answers appear at the end of the test.

11.3.1 Question 1

What is the name of the Vulkan function to allocate memory?

a VulkanAllocateMemory
b vkMemoryAllocate
c vkNew
d vkAllocateMemory

11.3.2 Question 2

Which function is used to set the allocated Vulkan memory contents?

a Access memory directly
b vkMapMemory
c vkLinkMemory
d vkMemoryMap

(Q1: Answer d; Q2: Answer b;)

12. Shaders

The programmable stages of the pipeline are controlled by shaders. These shaders are written in a C-like shader language. While there are a variety of shader stages, such as, vertex, hull, domain, geometry, and pixel shader, you'll consider only the vertex and pixel shader here (keep it as simple as possible).

12.1 Vertex & Pixel Shaders

Developers are always trying to push the envelope for faster graphics rendering. Today's graphics hardware contains shaders (e.g., vertex and pixel shaders) which can be reprogrammed by the user. Vertex and pixel (or fragment) shaders allow almost arbitrary computations per vertex and per pixel. You need to use shaders to output graphics in Vulkan. Hence,

you'll create a simple vertex and pixel shader. Compile the shaders using the GLSL compiler (i.e., glslangValidator.exe). The compiled shader binaries is loaded and integrated into the Vulkan operations (e.g., command buffer).

12.1.1 Vertex Shader

Your Vertex Shader will handle the processing of your individual vertex data (i.e., positions and transforms - which you created in the previous chapter). For your minimal example, you'll simply transform your input vertices by the projection and view matrices. The color information will simply be passed straight along to the pixel shader.

Below shows a minimal vertex shader listing (vert.vs):

```
// Vertex Shader
// shader version
#version 400

// input matrix transforms (3D) passed
// in from our program
layout ( std140, set = 0, binding = 0 ) uniform ↩
    buffer
{
  mat4 projection_matrix;
  mat4 view_matrix;
  mat4 model_matrix;
} UBO;

// input a position and a colour
layout( location = 0 ) in vec4 pos;
layout( location = 1 ) in vec3 color;

// output a color
layout( location = 0 ) out struct vertex_out
{
  vec4 vColor;
} OUT;


// shader entry point
void main()
{
  // combine the matrices
  mat4 modelView = UBO.model_matrix * UBO.view_matrix↩
```

```
    ;

  // transform position by matrices
  gl_Position = pos * ( modelView * UBO.↩
    projection_matrix );

  // pass input colour to the next stage
  OUT.vColor = vec4(color, 1.0);
  // Or a fixed test color (RED)
  // OUT.vColor = vec4( 1.0, 0.0, 0.0, 1 );

}// End main(..)
```

12.1.2 Pixel (or Fragment) Shader

Your Pixel Shader is even simpler than your Vertex Shader - and is responsible for the 'per-pixel' processing. For your example, you'll simply pass the color information directly forwards.

Below shows a minimal fragment (pixel) shader listing (frag.ps):

```
// Fragement Shader or sometimes called the
// 'pixel shader'
// shader version
#version 400

// input (from our vertex shader above)
layout ( location = 0 ) in struct vertex_in
{
  vec4 vColor;
} IN;

// final screen output color
layout ( location = 0 ) out vec4 uFragColor;

// shader entry point
void main()
{
  // pass input color along without any
  // modifications (e.g., Phong lighting
  // calculations could be done here)
  uFragColor = IN.vColor;

}// End main(..)
```

12.1.3 Compiling Shader Binaries

The shaders are text files - which you have to compile to get the binary files. Compiling the shaders requires you to use the GLSL compiler (i.e., glslangValidator.exe). After compiling your shaders you'll have the 'vert.spv' and 'frag.spv' files which you'll load in using the standard system libraries.

Example batch script file (.bat) to compile your shaders:

```
rem buildshaders.bat
@echo off

rem Uncomment the following line and set it to the ↩
    path where you have your layers *.dll and *.json
rem set VK_LAYER_PATH=c:\path\to\vulkan\layers

glslangValidator -V simple.vert
glslangValidator -V simple.frag
```

You are able to get a full breakdown of the shader compiler properties (e.g., glslangValidator.exe):

12.1.4 Loading Shader Binaries

You have the compiled the vertex and pixel shader text files (i.e., binaries 'vert.spv' and 'frag.spv' files). You'll load them into your system memory using the standard file input/output functions:

```
// store the loaded shader modules for
// later use
VkShaderModule g_vertexShaderModule   = NULL;
VkShaderModule g_fragmentShaderModule = NULL;

{
// Simple Shaders (Vertex & Fragement)
// temp variable for the code size in file
uint32_t codeSize = 0;
// temp buffer to hold our file data
char *code        = new char[10000];
// handle for reading
FILE* fileHandle = 0;
```

```
Usage: glslangValidator [option]... [file]...

Where: each 'file' ends in .<stage>, where <stage> is one of
    .conf   to provide an optional config file that replaces the default configu
ration
            (see -c option below for generating a template)
    .vert   for a vertex shader
    .tesc   for a tessellation control shader
    .tese   for a tessellation evaluation shader
    .geom   for a geometry shader
    .frag   for a fragment shader
    .comp   for a compute shader

Compilation warnings and errors will be printed to stdout.

To get other information, use one of the following options:
Each option must be specified separately.
  -U          create SPIR-V binary, under Vulkan semantics; turns on -l;
              default file name is <stage>.spv (-o overrides this)
              (unless -o is specified, which overrides the default file name)
  -G          create SPIR-V binary, under OpenGL semantics; turns on -l;
              default file name is <stage>.spv (-o overrides this)
  -H          print human readable form of SPIR-V; turns on -V
  -E          print pre-processed GLSL; cannot be used with -l;
              errors will appear on stderr.
  -c          configuration dump;
              creates the default configuration file (redirect to a .conf file)
  -d          default to desktop (#version 110) when there is no shader #versior

              (default is ES version 100)
  -h          print this usage message
  -i          intermediate tree (glslang AST) is printed out
  -l          link all input files together to form a single module
  -m          memory leak mode
  -o  <file>  save binary into <file>, requires a binary option (e.g., -V)
  -q          dump reflection query database
  -r          relaxed semantic error-checking mode
  -s          silent mode
  -t          multi-threaded mode
  -v          print version strings
  -w          suppress warnings (except as required by #extension : warn)
```

Figure 12.1: **glslangValidator options** - Options

```
// load our vertex shader:
fileHandle = fopen( ".\\vert.spv", "rb" );

// did we successfully find file
DBG_ASSERT_MSG(fileHandle != NULL,
  "Failed to open shader file.");

// read the file contents
codeSize = fread(code, 1, 10000, fileHandle);
// close the file
fclose(fileHandle);
fileHandle = NULL;

// create shader module
VkShaderModuleCreateInfo vertexShaderCreationInfo = ↩
    {};
// define shader type
vertexShaderCreationInfo.sType    = ↩
    VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
```

```
// size of shader binary in bytes
vertexShaderCreationInfo.codeSize = codeSize;
// pointer to the loaded shader binary
vertexShaderCreationInfo.pCode    = (uint32_t *)code;

// create shader module and store the handle
VkResult result =
vkCreateShaderModule( g_device, &↩
    vertexShaderCreationInfo, NULL, &↩
    g_vertexShaderModule );

// check we were successfully
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to create vertex shader module." );

// load our fragment shader:
fileHandle = fopen( ".\\frag.spv", "rb" );

// did we successfully find file
DBG_ASSERT_MSG(fileHandle != NULL,
  "Failed to open shader file.");

// read the file contents
codeSize = fread(code, 1, 10000, fileHandle);
// close the file
fileHandle = NULL;

// fill the structure with the
// shader details
VkShaderModuleCreateInfo fragmentShaderCreationInfo =↩
    {};
// type of shader (module)
fragmentShaderCreationInfo.sType    = ↩
    VK_STRUCTURE_TYPE_SHADER_MODULE_CREATE_INFO;
// size of fragment shader binary (bytes)
fragmentShaderCreationInfo.codeSize = codeSize;
// pointer to fragment shader data
fragmentShaderCreationInfo.pCode    = (uint32_t *)↩
    code;

// create fragment shader module
// and store the handle to it
result =
vkCreateShaderModule(
  // device
  g_device,
  // pCreateInfo
  &fragmentShaderCreationInfo,
  // pAllocator
  NULL,
  // pShaderModule
  &g_fragmentShaderModule );
```

```
// check we were successful
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to create vertex shader module." );
}
```

12.1.5 Shader Data/Parameters

After the shaders are loaded in and you've got handles to them. You'll setup your shader parameters (globals/data mapping). You allocate memory and map/lock the memory to copy/update data (like matrices and positions) which are used by your shaders.

```
// Global parameters that we need to keep hold of
// to manage the shaders/update them in the render
// loop later on

// target distance from the camera location
float g_cameraZ          = 10.0f;
// direction
float g_cameraZDir       = -1.0f;
// three matrix pointers
float *g_projectionMatrix = NULL;
float *g_viewMatrix       = NULL;
float *g_modelMatrix      = NULL;
// shader memory accessors to set and
// get parameters on the shaders
VkBuffer    g_buffer  = NULL;
VkDeviceMemory  g_memory  = NULL;

// Setup and configure shader data
{
// Shader helper constants (connecting shader with ↩
    the data)
const double PI    = 3.14159265359f;
const double TORAD = PI/180.0f;

// perspective projection parameters:
float fov   = 45.0f;
float nearZ = 0.1f;
float farZ  = 1000.0f;
float aspectRatio = g_width / (float)g_height;
float t     = 1.0f / (float)tan( fov * TORAD * 0.5f )↩
    ;
float nf    = nearZ - farZ;

// our matrices (model-view-projection)
static
```

```
float lprojectionMatrix[16] = {
  t / aspectRatio, 0, 0, 0,
  0, t, 0, 0,
  0, 0, (-nearZ-farZ) / nf, (2*nearZ*farZ) / nf,
  0, 0, 1, 0 };

// view matrix looking down the z-axis
static
float lviewMatrix[16] = {
  1, 0, 0, 0,
  0, 1, 0, 0,
  0, 0, 1, 0,
  0, 0, 0, 1 };

// simple identity matrix for our
// model transform
static
float lmodelMatrix[16] = {
  1, 0, 0, 0,
  0, 1, 0, 0,
  0, 0, 1, 0,
  0, 0, 0, 1 };

// animate camera (globals):
g_cameraZ        = 10.0f;
g_cameraZDir     = -1.0f;
lviewMatrix[11] = g_cameraZ;

// store matrices in our uniforms
g_projectionMatrix = lprojectionMatrix;
g_viewMatrix       = lviewMatrix;
g_modelMatrix      = lmodelMatrix;

// create our uniforms buffers:
VkBufferCreateInfo bufferCreateInfo = {};
bufferCreateInfo.sType = ↩
    VK_STRUCTURE_TYPE_BUFFER_CREATE_INFO;
// size in bytes
bufferCreateInfo.size  = sizeof(float) * 16 * 3;
bufferCreateInfo.usage = ↩
    VK_BUFFER_USAGE_UNIFORM_BUFFER_BIT;
bufferCreateInfo.sharingMode = ↩
    VK_SHARING_MODE_EXCLUSIVE;

// create a handle for the storing
// the matrices on the gpu (only a
// handle - doesn't allocate memory)
VkResult result =
vkCreateBuffer(
  g_device,
  &bufferCreateInfo,
  NULL,
  &g_buffer );
```

```
// check if successful
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to create uniforms buffer." );

// allocate memory for buffer, we
// we create earlier
VkMemoryRequirements bufferMemoryRequirements = {};
vkGetBufferMemoryRequirements(
  // device
  g_device,
  // buffer
  g_buffer,
  // pMemoryRequirements
  &bufferMemoryRequirements );

VkMemoryAllocateInfo matrixAllocateInfo = {};
matrixAllocateInfo.sType          = ↩
    VK_STRUCTURE_TYPE_MEMORY_ALLOCATE_INFO;
matrixAllocateInfo.allocationSize = ↩
    bufferMemoryRequirements.size;

// struct holding memory properties
VkPhysicalDeviceMemoryProperties memoryProperties;

vkGetPhysicalDeviceMemoryProperties(
  // physicalDevice
  g_physicalDevice,
  // pMemoryProperties
  &memoryProperties );

for( uint32_t i = 0; i <VK_MAX_MEMORY_TYPES; ++i )
{
  VkMemoryType memoryType = memoryProperties.↩
    memoryTypes[i];
  // is this the memory type we are looking for?
  if( ( memoryType.propertyFlags & ↩
    VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT ) )
  {
    // save location
  matrixAllocateInfo.memoryTypeIndex = i;
  // exit loop
  }
}// End for i


result =
vkAllocateMemory(
  g_device,
  &matrixAllocateInfo,
  NULL,
  &g_memory );

// check if successful
```

```
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to allocate uniforms buffer memory." );

result =
vkBindBufferMemory(
  g_device,
  g_buffer,
  g_memory,
  0 );

// check if successful
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to bind uniforms buffer memory." );

// set buffer content
// (lock the memory for writing)
void *matrixMapped = NULL;

result = vkMapMemory(
  g_device,
  g_memory,
  0,
  VK_WHOLE_SIZE,
  0,
  &matrixMapped );

// check if successful
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to mapp uniform buffer memory." );

// copy the local (CPU) matrix values
// to the buffer memory on the gpu
// matrix is 16 floats
memcpy( ((float *)matrixMapped + 0,   // dst
       &lprojectionMatrix[0],         // src
       sizeof( lprojectionMatrix ) ); // size

memcpy( ((float *)matrixMapped + 16), // dst
        &lviewMatrix[0],              // src
        sizeof( lviewMatrix ) );      // size

memcpy( ((float *)matrixMapped + 32), // dst
        &lmodelMatrix[0],             // src
        sizeof( lmodelMatrix ) );     // size

// (unlock memory)
vkUnmapMemory( g_device, g_memory );
}
```

12.2 Summary

You should have compiled and loaded your shaders into Vulkan. However, your shaders are not yet integrated into your render pipeline (i.e., you haven't connected everything together - the data and drawing). You'll do that in the next couple of chapters.

12.3 Test Yourself

Here are some multiple-choice questions to help you identify your baseline knowledge of the material. Answers appear at the end of the test.

12.3.1 Question 1

What is the function name to create a shader module?

a ShaderModuleNew
b CreateModule
c vkCreateShaderModule
d vkShaderModule

12.3.2 Question 2

Which function is used to bind the shader memory for configuration?

a vkMapBufferMemory
b vkBindBufferMemory
c BindBufferMemory
d vkBindMemory

(Q1: Answer c; Q2: Answer b;)

13. Descriptors & Binding

13.0.3 Descriptors & Binding

Traditional graphics API (prior to Vulkan) would allow each shader stage to have its own namespace, so pixel shader texture binding 0 is not vertex shader texture binding 0. Each resource type is namespaced apart, so constant buffer binding 0 is definitely not the same as texture binding 0. Resources are individually bound and unbound to slots (or at best in contiguous batches). In Vulkan, the base binding unit is a descriptor. A descriptor is an opaque representation that stores 'one bind'. For example, this could be an image, a sampler, or a uniform/constant buffer. It is even allowed to be an array - so you can have an array of images that can be different sizes, as long as they are all 2D floating point images.

Descriptors aren't bound individually, they are bound in

blocks in a **VkDescriptorSet** which each have a particular **VkDescriptorSetLayout**. The **VkDescriptorSetLayout** describes the types of the individual bindings in each **VkDescriptorSet**.

The easiest way to think about the concept, is to consider **VkDescriptorSetLayout** as being like a C struct type - it describes some members, each member having an opaque type (constant buffer, load/store image). The **VkDescriptorSet** is a specific instance of that type - and each member in the **VkDescriptorSet** is a binding you can update with whichever resource you want it to contain.

This is roughly how you create the objects too. You pass a list of the types, array sizes and bindings to Vulkan to create a **VkDescriptorSetLayout**, then you can allocate **VkDescriptorSets** with that layout from a **VkDescriptorPool**. The pool acts the same way as **VkCommandPool**, to let you allocate descriptors on different threads more efficiently by having a pool per thread.

```
VkDescriptorSet       g_descriptorSet = NULL;
VkDescriptorSetLayout g_setLayout     = NULL;

{
// Define and create our descriptors:
VkDescriptorSetLayoutBinding bindings[1];

// uniform buffer for our matrices:
bindings[0].binding            = 0;
bindings[0].descriptorType     = ↩
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
bindings[0].descriptorCount    = 1;
bindings[0].stageFlags         = ↩
    VK_SHADER_STAGE_VERTEX_BIT;
bindings[0].pImmutableSamplers = NULL;

VkDescriptorSetLayoutCreateInfo setLayoutCreateInfo =↩
    {};
setLayoutCreateInfo.sType        = ↩
    VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO↩
    ;
setLayoutCreateInfo.bindingCount = 1;
setLayoutCreateInfo.pBindings    = bindings;
```

```
VkResult result =
vkCreateDescriptorSetLayout(
  // device
  g_device,
  // pCreateInfo
  &setLayoutCreateInfo,
  // pAllocator
  NULL,
  // pSetLayout
  &g_setLayout );

// check if successful
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to create DescriptorSetLayout." );

// decriptor pool creation:
VkDescriptorPoolSize uniformBufferPoolSize[1];
uniformBufferPoolSize[0].type = ↩
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
uniformBufferPoolSize[0].descriptorCount = 1;

VkDescriptorPoolCreateInfo poolCreateInfo = {};
poolCreateInfo.sType         = ↩
    VK_STRUCTURE_TYPE_DESCRIPTOR_POOL_CREATE_INFO;
poolCreateInfo.maxSets       = 1;
poolCreateInfo.poolSizeCount = 1;
poolCreateInfo.pPoolSizes    = uniformBufferPoolSize;

VkDescriptorPool descriptorPool;
result = vkCreateDescriptorPool(
  // device
  g_device,
  // pCreateInfo
  &poolCreateInfo,
  // pAllocator
  NULL,
  // pDescriptorPool
  &descriptorPool );

// check if successful
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to create descriptor pool." );

// allocate our descriptor from the pool:
VkDescriptorSetAllocateInfo dsai = {};
dsai.sType              = ↩
    VK_STRUCTURE_TYPE_DESCRIPTOR_SET_ALLOCATE_INFO;
dsai.descriptorPool     = descriptorPool;
dsai.descriptorSetCount = 1;
dsai.pSetLayouts        = &g_setLayout;

result =
vkAllocateDescriptorSets(
```

```
  // device
  g_device,
  // pAllocateInfo
  &dsai,
  // pDescriptorSets
  &g_descriptorSet );

// check if successful
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to allocate descriptor sets." );

// When a set is allocated all values are undefined
// and all descriptors are uninitialized. We must
// init all statically used bindings:
VkDescriptorBufferInfo dbi = {};
dbi.buffer = g_buffer;
dbi.offset = 0;
dbi.range  = VK_WHOLE_SIZE;


VkWriteDescriptorSet wd = {};
wd.sType           = ↩
    VK_STRUCTURE_TYPE_WRITE_DESCRIPTOR_SET;
wd.dstSet          = g_descriptorSet;
wd.dstBinding      = 0;
wd.dstArrayElement = 0;
wd.descriptorCount = 1;
wd.descriptorType  = ↩
    VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
wd.pImageInfo      = NULL;
wd.pBufferInfo     = &dbi;
wd.pTexelBufferView = NULL;


vkUpdateDescriptorSets(
  // device
  g_device,
  // descriptorWriteCount
  1,
  // pDescriptorWrites
  &wd,
  // descriptorCopyCount
  0,
  // pDescriptorCopies
  NULL );
}
```

13.1 Summary

You should have setup your 'descriptors', which are structures that tell the shaders about the resources (i.e., data). In the final stage, you need to connect everything together (i.e., shaders, data, and a pipeline). The pipeline hasn't been discussed yet - this is done in the next chapter.

13.2 Test Yourself

Here are some multiple-choice questions to help you identify your baseline knowledge of the material. Answers appear at the end of the test.

13.2.1 Question 1

What function updates the descriptor?

a VulkanUpdateDescriptorSets
b vkDescriptorSets
c UpdateDescriptorSets
d vkUpdateDescriptorSets

13.2.2 Question 2

The base binding unit, in Vulkan, is a descriptor, and a descriptor is an opaque representation that stores 'one bind'.

a true
b false

(Q1: Answer d; Q2: Answer a;)

14. Pipeline

14.1 Connecting Everything

The pipeline is your final stage to getting your triangle on screen. All the previous chapters have been building up to this. Importantly, the this sample does not include depth buffering, however, this can be added in later. What you'll find is, as your scene becomes increasingly complex (i.e., the numbers of triangles and textures increases) - the extra work you've had to do to get started with Vulkan will only then become more useful.

Below shows the listing for the pipeline creation (also its connection with the other graphical elements from previous chapters):

```
// store pipeline handles (accessors for
```

```
// using the pipeline for drawing)
VkPipeline       g_pipeline       = NULL;
VkPipelineLayout g_pipelineLayout = NULL;

// Graphics Pipeline Setup:
{
// define the structure for the
// pipeline properties (layout)
VkPipelineLayoutCreateInfo layoutCreateInfo = {};
layoutCreateInfo.sType = ↩
    VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
layoutCreateInfo.setLayoutCount         = 1;
layoutCreateInfo.pSetLayouts            = &↩
    g_setLayout;
layoutCreateInfo.pushConstantRangeCount = 0;
layoutCreateInfo.pPushConstantRanges    = NULL;

// create pipeline layout
VkResult result =
vkCreatePipelineLayout(
  // device
  g_device,
  // pCreateInfo
  &layoutCreateInfo,
  // pAllocator
  NULL,
  // pPipelineLayout
  &g_pipelineLayout );

// check if successful
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to create pipeline layout." );


// setup shader stages in the pipeline:
VkPipelineShaderStageCreateInfo shaderStageCreateInfo↩
    [2] = {};
shaderStageCreateInfo[0].sType = ↩
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO↩
    ;
shaderStageCreateInfo[0].stage = ↩
    VK_SHADER_STAGE_VERTEX_BIT;
shaderStageCreateInfo[0].module = ↩
    g_vertexShaderModule;
// shader entry point function name
shaderStageCreateInfo[0].pName = "main";
shaderStageCreateInfo[0].pSpecializationInfo = NULL;

shaderStageCreateInfo[1].sType = ↩
    VK_STRUCTURE_TYPE_PIPELINE_SHADER_STAGE_CREATE_INFO↩
    ;
shaderStageCreateInfo[1].stage = ↩
    VK_SHADER_STAGE_FRAGMENT_BIT;
```

```
shaderStageCreateInfo[1].module = ↩
    g_fragmentShaderModule;
// shader entry point function name
shaderStageCreateInfo[1].pName = "main";
shaderStageCreateInfo[1].pSpecializationInfo = NULL;

// vertex input configuration:
VkVertexInputBindingDescription ↩
    vertexBindingDescription = {};
vertexBindingDescription.binding = 0;
vertexBindingDescription.stride = sizeof(vertex);
vertexBindingDescription.inputRate = ↩
    VK_VERTEX_INPUT_RATE_VERTEX;

VkVertexInputAttributeDescription ↩
    vertexAttributeDescritpion[2];

// position:
vertexAttributeDescritpion[0].location = 0;
vertexAttributeDescritpion[0].binding  = 0;
vertexAttributeDescritpion[0].format   = ↩
    VK_FORMAT_R32G32B32A32_SFLOAT;
vertexAttributeDescritpion[0].offset   = 0;

// colors:
vertexAttributeDescritpion[1].location = 1;
vertexAttributeDescritpion[1].binding  = 0;
vertexAttributeDescritpion[1].format   = ↩
    VK_FORMAT_R32G32B32_SFLOAT;
vertexAttributeDescritpion[1].offset   = 4 * sizeof(↩
    float);

VkPipelineVertexInputStateCreateInfo ↩
    vertexInputStateCreateInfo = {};
vertexInputStateCreateInfo.sType = ↩
    VK_STRUCTURE_TYPE_PIPELINE_VERTEX_INPUT_STATE_CREATE_INFO↩
    ;
vertexInputStateCreateInfo.↩
    vertexBindingDescriptionCount = 1;
// attribute indexing is a function of the vertex ↩
    index
vertexInputStateCreateInfo.pVertexBindingDescriptions↩
     = &vertexBindingDescription;
vertexInputStateCreateInfo.↩
    vertexAttributeDescriptionCount = 2;
vertexInputStateCreateInfo.↩
    pVertexAttributeDescriptions = ↩
    vertexAttributeDescritpion;

// vertex topology config:
VkPipelineInputAssemblyStateCreateInfo ↩
    inputAssemblyStateCreateInfo = {};
inputAssemblyStateCreateInfo.sType = ↩
    VK_STRUCTURE_TYPE_PIPELINE_INPUT_ASSEMBLY_STATE_CREATE_INFO↩
```

```
    ;
inputAssemblyStateCreateInfo.topology = ↩
    VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST;
inputAssemblyStateCreateInfo.primitiveRestartEnable =↩
     VK_FALSE;

// viewport config:
VkViewport viewport = {};
viewport.x        = 0;
viewport.y        = 0;
viewport.width    = (float)g_width;
viewport.height   = (float)g_height;
viewport.minDepth = 0;
viewport.maxDepth = 1;

VkRect2D scissors = {};
VkOffset2D k      = {0,0};
VkExtent2D m      = {g_width,g_height };
scissors.offset   = k;
scissors.extent   = m;

VkPipelineViewportStateCreateInfo viewportState = {};
viewportState.sType =
VK_STRUCTURE_TYPE_PIPELINE_VIEWPORT_STATE_CREATE_INFO↩
    ;
viewportState.viewportCount = 1;
viewportState.pViewports    = &viewport;
viewportState.scissorCount  = 1;
viewportState.pScissors     = &scissors;

// rasterization config:
VkPipelineRasterizationStateCreateInfo ↩
    rasterizationState = {};
// manditory structure type
rasterizationState.sType =
VK_STRUCTURE_TYPE_PIPELINE_RASTERIZATION_STATE_CREATE_INFO↩
    ;
rasterizationState.depthClampEnable       = VK_FALSE;
rasterizationState.rasterizerDiscardEnable= VK_FALSE;
rasterizationState.polygonMode            = ↩
    VK_POLYGON_MODE_FILL;
rasterizationState.cullMode               = ↩
    VK_CULL_MODE_NONE;
rasterizationState.frontFace              = ↩
    VK_FRONT_FACE_COUNTER_CLOCKWISE;
rasterizationState.depthBiasEnable        = VK_FALSE;
rasterizationState.depthBiasConstantFactor= 0;
rasterizationState.depthBiasClamp         = 0;
rasterizationState.depthBiasSlopeFactor   = 0;
rasterizationState.lineWidth              = 1;

// sampling config:
VkPipelineMultisampleStateCreateInfo multisampleState↩
     = {};
```

```
multisampleState.sType =
VK_STRUCTURE_TYPE_PIPELINE_MULTISAMPLE_STATE_CREATE_INFO;
multisampleState.rasterizationSamples  = VK_SAMPLE_COUNT_1_BIT;
multisampleState.sampleShadingEnable   = VK_FALSE;
multisampleState.minSampleShading      = 0;
multisampleState.pSampleMask           = NULL;
multisampleState.alphaToCoverageEnable = VK_FALSE;
multisampleState.alphaToOneEnable      = VK_FALSE;


// color blend config: (Actually off for tutorial)
VkPipelineColorBlendAttachmentState colorBlendAttachmentState = {};
colorBlendAttachmentState.blendEnable         = VK_FALSE;
colorBlendAttachmentState.srcColorBlendFactor = VK_BLEND_FACTOR_SRC_COLOR;
colorBlendAttachmentState.dstColorBlendFactor = VK_BLEND_FACTOR_ONE_MINUS_DST_COLOR;
colorBlendAttachmentState.colorBlendOp        = VK_BLEND_OP_ADD;
colorBlendAttachmentState.srcAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
colorBlendAttachmentState.dstAlphaBlendFactor = VK_BLEND_FACTOR_ZERO;
colorBlendAttachmentState.alphaBlendOp        = VK_BLEND_OP_ADD;
colorBlendAttachmentState.colorWriteMask      = 0xf;

VkPipelineColorBlendStateCreateInfo colorBlendState ={};
colorBlendState.sType =
VK_STRUCTURE_TYPE_PIPELINE_COLOR_BLEND_STATE_CREATE_INFO;
colorBlendState.logicOpEnable     = VK_FALSE;
colorBlendState.logicOp           = VK_LOGIC_OP_CLEAR;
colorBlendState.attachmentCount   = 1;
colorBlendState.pAttachments      = &colorBlendAttachmentState;
colorBlendState.blendConstants[0] = 0.0;
colorBlendState.blendConstants[1] = 0.0;
colorBlendState.blendConstants[2] = 0.0;
colorBlendState.blendConstants[3] = 0.0;

// configure dynamic state:
VkDynamicState dynamicState[2] = {
    VK_DYNAMIC_STATE_VIEWPORT,
    VK_DYNAMIC_STATE_SCISSOR };
VkPipelineDynamicStateCreateInfo
    dynamicStateCreateInfo = {};
dynamicStateCreateInfo.sType =
```

```
VK_STRUCTURE_TYPE_PIPELINE_DYNAMIC_STATE_CREATE_INFO;
dynamicStateCreateInfo.dynamicStateCount = 2;
dynamicStateCreateInfo.pDynamicStates    = ↩
    dynamicState;

// and finally, pipeline config and creation:
VkGraphicsPipelineCreateInfo pipelineCreateInfo = {};
pipelineCreateInfo.sType = ↩
    VK_STRUCTURE_TYPE_GRAPHICS_PIPELINE_CREATE_INFO;
pipelineCreateInfo.stageCount          = 2;
pipelineCreateInfo.pStages             = ↩
    shaderStageCreateInfo;
pipelineCreateInfo.pVertexInputState   = &↩
    vertexInputStateCreateInfo;
pipelineCreateInfo.pInputAssemblyState = &↩
    inputAssemblyStateCreateInfo;
pipelineCreateInfo.pTessellationState = NULL;
pipelineCreateInfo.pViewportState      = &↩
    viewportState;
pipelineCreateInfo.pRasterizationState = &↩
    rasterizationState;
pipelineCreateInfo.pMultisampleState   = &↩
    multisampleState;
pipelineCreateInfo.pDepthStencilState = NULL;
pipelineCreateInfo.pColorBlendState    = &↩
    colorBlendState;
pipelineCreateInfo.pDynamicState       = &↩
    dynamicStateCreateInfo;
pipelineCreateInfo.layout              = ↩
    g_pipelineLayout;
pipelineCreateInfo.renderPass          = g_renderPass;
pipelineCreateInfo.subpass             = 0;
pipelineCreateInfo.basePipelineHandle = NULL;
pipelineCreateInfo.basePipelineIndex   = 0;


result =
vkCreateGraphicsPipelines(
  // device
  g_device,
  // pipelineCache
  VK_NULL_HANDLE,
  // createInfoCount
  1,
  // pCreateInfos
  &pipelineCreateInfo,
  // pAllocator
  NULL,
  // pPipelines
  &g_pipeline );

// check if successful
DBG_ASSERT_VULKAN_MSG( result,
  "Failed to create graphics pipeline." );
```

```
}
```

14.1.1 RenderLoop (Revisited)

You need to add some additional commands to your RenderLoop so your triangle will draw. Previously, you cleared the screen and presented the buffer. Now you need to include the draw commands to push your vertices and data along the pipeline (i.e., the shader parameters, such as, the matrices).

We show the updated draw loop below:

```
void RenderVulkan( )
{
// Oscillates the triangle backwards and
// forwards (towards and away from the
// camera)
if ( g_cameraZ <= 1 )
{
  g_cameraZ    = 1;
  g_cameraZDir = 1;
}
else if ( g_cameraZ >= 10 )
{
  g_cameraZ    = 10;
  g_cameraZDir = -1;
}

// update view (camera) distance
g_cameraZ += g_cameraZDir * 0.01f;

// update camera matrix (move around)
g_viewMatrix[11] = g_cameraZ;

// update shader uniforms:

// temp pointer to the mapped memory
void *matrixMapped;

// mapped the GPU memory so we can access it
// and update our variables
vkMapMemory( g_device,       // device
             g_memory,       // memory
             0,              // offset
             VK_WHOLE_SIZE,  // size
             0,              // flags
```

```
            &matrixMapped );// ppData

// copy our updated matrix data from CPU to GPU
// matrix is 16 floats
memcpy( ((float *)matrixMapped + 0),
        g_projectionMatrix,
        sizeof(float) * 16 );
memcpy( ((float *)matrixMapped + 16),
        g_viewMatrix,
        sizeof(float) * 16 );
memcpy( ((float *)matrixMapped + 32),
         g_modelMatrix,
         sizeof(float) * 16 );


VkMappedMemoryRange memoryRange = {};
memoryRange.sType  = ↩
    VK_STRUCTURE_TYPE_MAPPED_MEMORY_RANGE;
memoryRange.memory = g_memory;
memoryRange.offset = 0;
memoryRange.size   = VK_WHOLE_SIZE;

// ensure the memory is updated
// (not cached or wait for gpu)
vkFlushMappedMemoryRanges( g_device, 1, &memoryRange ↩
    );

// finished updating memory so unmap
vkUnmapMemory( g_device, g_memory );

// temp index to the image we're gonig to write to
uint32_t nextImageIdx;

// get the back buffer image id from the
// swap-chain
vkAcquireNextImageKHR(
  // device
  g_device,
  // swapchain
  g_swapChain,
  // timeout
  UINT64_MAX,
  // semaphore
  VK_NULL_HANDLE,
  // fence
  VK_NULL_HANDLE,
  // pImageIndex
  &nextImageIdx );

// configure buffer information
VkCommandBufferBeginInfo beginInfo = {};
beginInfo.sType =
  VK_STRUCTURE_TYPE_COMMAND_BUFFER_BEGIN_INFO;
```

```
beginInfo.flags =
  VK_COMMAND_BUFFER_USAGE_ONE_TIME_SUBMIT_BIT;

// start recording buffer commands
vkBeginCommandBuffer( g_drawCmdBuffer, &beginInfo );

// barrier for reading from uniform buffer after all
// writing is done:
VkMemoryBarrier uniformMemoryBarrier = {};
uniformMemoryBarrier.sType = ↩
    VK_STRUCTURE_TYPE_MEMORY_BARRIER;
uniformMemoryBarrier.srcAccessMask = ↩
    VK_ACCESS_HOST_WRITE_BIT;
uniformMemoryBarrier.dstAccessMask = ↩
    VK_ACCESS_UNIFORM_READ_BIT;


vkCmdPipelineBarrier(
  g_drawCmdBuffer,                  // commandBuffer
  VK_PIPELINE_STAGE_HOST_BIT,       // srcStageMask
  VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,// dstStageMask
  0,                              // dependencyFlags
  1, &uniformMemoryBarrier,       // memoryBarrierCount
  0, NULL,                        // bufferMemoryBarrier
  0, NULL );                      // imageMemoryBarrier

// change image layout
// structure for specifying the
// the image memory barrier.
VkImageMemoryBarrier layoutTransitionBarrier = {};
layoutTransitionBarrier.sType                =
  VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
layoutTransitionBarrier.srcAccessMask        =
  VK_ACCESS_MEMORY_READ_BIT;
layoutTransitionBarrier.dstAccessMask        =
  VK_ACCESS_COLOR_ATTACHMENT_READ_BIT |
  VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
layoutTransitionBarrier.oldLayout            =
  VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;
layoutTransitionBarrier.newLayout            =
  VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
layoutTransitionBarrier.srcQueueFamilyIndex =
  VK_QUEUE_FAMILY_IGNORED;
layoutTransitionBarrier.dstQueueFamilyIndex =
  VK_QUEUE_FAMILY_IGNORED;
layoutTransitionBarrier.image                =
  g_presentImages[ nextImageIdx ];


VkImageSubresourceRange resourceRange =
{ VK_IMAGE_ASPECT_COLOR_BIT, // aspectMask
  0,                         // baseMipLevel
  1,                         // levelCount
  0,                         // baseArrayLayer
```

```
  1 };                          // layerCount

layoutTransitionBarrier.subresourceRange = ↩
    resourceRange;

vkCmdPipelineBarrier(
  g_drawCmdBuffer,
  VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
  VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT,
  0,
  0, NULL,
  0, NULL,
  1, &layoutTransitionBarrier );

// activate render pass:
VkClearValue clearValue[] = {
     { 1.0f, 1.0f, 1.0f, 1.0f }, //color
     { 1.0, 0.0 } };             //depth stencil

VkRenderPassBeginInfo renderPassBeginInfo = {};
renderPassBeginInfo.sType = ↩
    VK_STRUCTURE_TYPE_RENDER_PASS_BEGIN_INFO;
renderPassBeginInfo.renderPass  = g_renderPass;
renderPassBeginInfo.framebuffer = g_frameBuffers[ ↩
    nextImageIdx ];
VkOffset2D a = {0, 0};
VkExtent2D b = {g_width, g_height};
VkRect2D   c = {a,b};
renderPassBeginInfo.renderArea      = c;
renderPassBeginInfo.clearValueCount = 2;
renderPassBeginInfo.pClearValues    = clearValue;

// start render commands
vkCmdBeginRenderPass(
  g_drawCmdBuffer,            // commandBuffer
  &renderPassBeginInfo,       // pipelineBindPoint
  VK_SUBPASS_CONTENTS_INLINE);// pipeline
{
// bind the graphics pipeline to the command buffer.
// Any vkDraw command afterwards is affected by this
// pipeline.
vkCmdBindPipeline(
  // commandBuffer
  g_drawCmdBuffer,
  // VkPipelineBindPoint
  VK_PIPELINE_BIND_POINT_GRAPHICS,
  // VkPipeline
  g_pipeline );


// take care of dynamic state:
VkViewport viewport = { 0, 0, (float)g_width, (float)↩
    g_height, 0, 1 };
```

```
vkCmdSetViewport( g_drawCmdBuffer, 0, 1, &viewport );

// scissor test determines if a fragment's
// framebuffer coordinates lie within the
// scissor rectangle corresponding to the viewport
VkRect2D scissor = { 0, 0, g_width, g_height };
vkCmdSetScissor(
  g_drawCmdBuffer,    // commandBuffer
  0,                  // firstScissor
  1,                  // scissorCount
  &scissor);          // pScissors


// bind our shader parameters:
vkCmdBindDescriptorSets(
  g_drawCmdBuffer,          // commandBuffer
  // pipelineBindPoint
  VK_PIPELINE_BIND_POINT_GRAPHICS,
  g_pipelineLayout,         // layout
  0,                        // firstSet
  1, &g_descriptorSet,      // descriptorSet
  0, NULL );                // dynamicOffset

// render the triangle:
VkDeviceSize offsets = { 0 };
vkCmdBindVertexBuffers(
  g_drawCmdBuffer,     // commandBuffer
  0,                   // firstBinding
  1,                   // bindingCount
  &g_vertexInputBuffer,// pBuffers
  &offsets );          // pOffsets


// command to draw our triangle
vkCmdDraw( g_drawCmdBuffer, // commandBuffer
           3,               // vertexCount
           1,               // instanceCount
           0,               // firstVertex
           0 );             // firstInstance

}
// stop render commands
vkCmdEndRenderPass( g_drawCmdBuffer );

// change layout back to ↩
    VK_IMAGE_LAYOUT_PRESENT_SRC_KHR
VkImageMemoryBarrier prePresentBarrier = {};
prePresentBarrier.sType          =
  VK_STRUCTURE_TYPE_IMAGE_MEMORY_BARRIER;
prePresentBarrier.srcAccessMask =
  VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
prePresentBarrier.dstAccessMask =
  VK_ACCESS_MEMORY_READ_BIT;
```

```
prePresentBarrier.oldLayout     =
  VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL;
prePresentBarrier.newLayout     =
  VK_IMAGE_LAYOUT_PRESENT_SRC_KHR; // *** change back↩
     ***
prePresentBarrier.srcQueueFamilyIndex =
  VK_QUEUE_FAMILY_IGNORED;
prePresentBarrier.dstQueueFamilyIndex =
  VK_QUEUE_FAMILY_IGNORED;

VkImageSubresourceRange d = {↩
    VK_IMAGE_ASPECT_COLOR_BIT, 0, 1, 0, 1};
prePresentBarrier.subresourceRange = d;
prePresentBarrier.image = g_presentImages[ ↩
    nextImageIdx ];


vkCmdPipelineBarrier(
  // commandBuffer
  g_drawCmdBuffer,
  // srcStageMask
  VK_PIPELINE_STAGE_ALL_COMMANDS_BIT,
  // dstStageMask
  VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT,
  0,                        // dependencyFlags
  0, NULL,                  // memoryBarrier
  0, NULL,                  // bufferMemoryBarrier
  1, &prePresentBarrier ); // imageMemoryBarrier

// finished recording commands
vkEndCommandBuffer( g_drawCmdBuffer );

// present to the display:
VkFence renderFence;
VkFenceCreateInfo fenceCreateInfo = {};
fenceCreateInfo.sType = ↩
    VK_STRUCTURE_TYPE_FENCE_CREATE_INFO;

vkCreateFence(
  // device
  g_device,
  // pCreateInfo
  &fenceCreateInfo,
  // pAllocator
  NULL,
  // pFence
  &renderFence );

VkPipelineStageFlags waitStageMash =
  { VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT };

// struct for information about
// the batch of work.
VkSubmitInfo submitInfo = {};
```

```
submitInfo.sType = VK_STRUCTURE_TYPE_SUBMIT_INFO;
submitInfo.waitSemaphoreCount = 0;
submitInfo.pWaitSemaphores = VK_NULL_HANDLE;
submitInfo.pWaitDstStageMask    = &waitStageMash;
submitInfo.commandBufferCount   = 1;
submitInfo.pCommandBuffers      = &g_drawCmdBuffer;
submitInfo.signalSemaphoreCount = 0;
submitInfo.pSignalSemaphores    = VK_NULL_HANDLE;

// submit the command queue (notice, we
// also pass the fence, to let us know when
// the gpu has finished)
vkQueueSubmit(
  // queue
  g_presentQueue,
  // submitCount
  1,
  // pSubmits
  &submitInfo,
  // fence
  renderFence );


// wait until the GPU has finished processing
// the commands
vkWaitForFences( g_device, 1, &renderFence, VK_TRUE, ↩
    UINT64_MAX );
vkDestroyFence( g_device, renderFence, NULL );

// structure for specifying the
// parameters of the image presentation.
VkPresentInfoKHR presentInfo = {};
presentInfo.sType          = ↩
    VK_STRUCTURE_TYPE_PRESENT_INFO_KHR;
presentInfo.pNext          = NULL;
presentInfo.waitSemaphoreCount = 0;
presentInfo.pWaitSemaphores = VK_NULL_HANDLE;
presentInfo.swapchainCount = 1;
presentInfo.pSwapchains    = &g_swapChain;
presentInfo.pImageIndices = &nextImageIdx;
presentInfo.pResults       = NULL;

vkQueuePresentKHR( g_presentQueue, &presentInfo );
}// End VulkanRender(..)
```

14.2 Putting it all Together (Simple Triangle)

The completed example (i.e., everything from creating a device through to swap-chains and framebuffers) is a considerable amount of code - while it may seem overwhelming initially, as you come to understand and appreciate each of the different parts of the implementation - you'll see the advantages and its elegance. Compared to previous OpenGL and DirectX samples, the minimal Vulkan demo opens up a doorway to future hardware customizations that were not possible previously - helping to reduce performance bottlenecks.

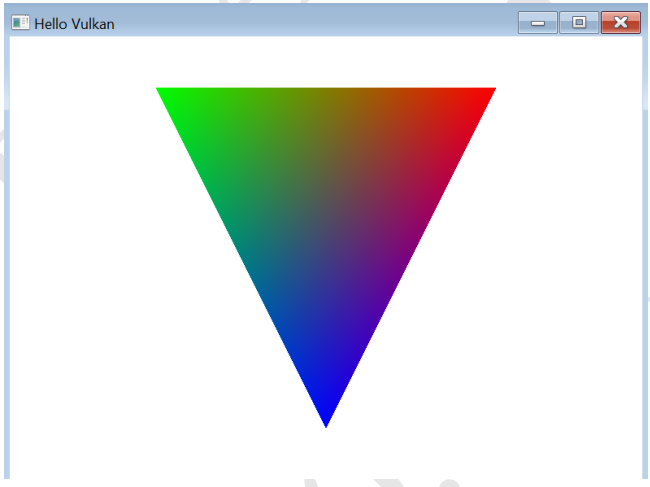

Figure 14.1: **Window Triangle** - Drawing a simple color triangle on screen.

14.2.1 Summary (What Next?)

This book has only scratched the surface of the Vulkan API - giving you (the reader) a taste. You are now ready to start writing and editing larger projects. As the examples/snippets in this text, were written to show the underlying raw Vulkan

API (minimalistic view). Once you start becoming more comfortable with the Vulkan API (through lots of trial and error - practising and experimenting) you'll have no problem with larger more intricate programs (you'll also have a feel for what wrapper classes are doing internally - since it's common practice to encapsulate a lot of the Vulkan API in classes to make it more manageable). Example tasks/exercises to get you started:

- Module programming (wrapper classes/structures/manage render engine)
- Textured geometry
- Batched rendering (switching commands)
- Profiling (analysing performance bottlenecks)
- Win32/Linux/Android builds
- Complex shaders (e.g., shadows, post processing, deferred rendering, ..)

14.3 Test Yourself

Here are some multiple-choice questions to help you identify your baseline knowledge of the material. Answers appear at the end of the test.

14.3.1 Question 1

What command function do you use for drawing geometry?

a vkCmdDraw
b vkCommandDraw
c VulkanCmdDraw
d CmdDraw

14.3.2 Question 2

Which function presents the pipeline in the render loop?

a vkQueuePresentPipeline
b QueuePresent
c vkQueuePresentKHR
d vkQueuePresentFlush

(Q1: Answer a; Q2: Answer c;)

Index