Vulkan

Emanuele Franchi

February 4, 2022

Abstract

Thesis about Vulkan

Dedication

Bla Bla Bla

Acknowledgments

I want to thank...

Contents

List of Figures

Listings

Chapter 1

Vulkan

1.1 What is Vulkan?

Figure 1.1: Vulkan logo

Vulkan is a modern graphics API. It is maintained by the Khronos Group. Vulkan is meant to abstract how modern GPUs work. Using Vulkan, the programmer can write more performant code. The better performance comes at the cost of having a more verbose and low level API compared to other existing APIs such as OpenGL or Direct3D 11 and prior. Vulkan is not the only modern graphics API, other such APIs are Direct3D 12 and Metal. Nonetheless, Vulkan has the advantage of being fully cross platform.

1.2 What problems does Vulkan solve?

Figure 1.2: OpenGL logo

Common graphics APIs like OpenGL or Direct3D were developed during the 1990s. At that time, graphics card hardware was very limited not only in terms of computational power but also from a functionality standpoint. As time progressed, graphics card architectures continued to evolve, offering new functionalities. All these new functionalities had to be integrated with the old existing APIs. The more new functionalities were integrated, the more the GPU's driver complexity grew. Such complicated GPU drivers are inefficient and are also the cause of many inconsistencies between implementations of the same graphics API but on different GPUs.

1.3 How does Vulkan solve these problems?

Vulkan doesn't suffer from the problems we saw above because it has been designed from scratch and with modern GPU's architecture in mind. It reduces the driver overhead by being more verbose and low level. It is also designed to be multithreaded, allowing the programmer to submit GPU commands from different threads. This is very beneficial to performance, since modern CPUs usually have more than one core.

Chapter 2

Initializing Vulkan

TODO: chapter introduction ...

2.1 Create Vulkan Instance

To access any of the functionalities offered by Vulkan we first have to create a
Vulkan instance. To do this we call vkCreateInstance.

```
1  VkInstance instance = VK_NULL_HANDLE;
2  vkCreateInstance(&createInfo, nullptr, instance);
```

Listing 2.1: Create Vulkan instance

2.1.1 VkInstanceCreateInfo

To call vkCreateInstance we need to pass a pointer to a VkInstanceCreateInfo
struct. This struct collects all the information needed to configure our Vulkan
instance.

```
1  VkInstanceCreateInfo createInfo = {};
2  createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
3  createInfo.pApplicationInfo = &appInfo;
4  createInfo.enabledLayerCount = layerCount;
5  createInfo.ppEnabledLayerNames = layers;
6  createInfo.enabledExtensionCount = extensionCount;
7  createInfo.ppEnabledExtensionNames = extentions;
```

Listing 2.2: VkInstanceCreateInfo initialization

2.1.2 VkApplicationInfo

We can see that the VkInstanceCreateInfo struct is not the only thing we need.
We have to specify a pointer to a VkApplicationInfo struct. Such struct de-
scribes our Vulkan application.

```
1  VkApplicationInfo appInfo = {};
2  appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
3  appInfo.pApplicationName = "Vulkan example";
4  appInfo.apiVersion = VK_API_VERSION_1_2;
```

Listing 2.3: VkApplicationInfo initialization

2.1.3 Layers

While we initialize our VkInstanceCreateInfo struct, we can specify the layers that we want to enable. The specified layers will be loaded after the Vulkan instance creation.

Layers are optional components that hook into Vulkan. Layers can intercept, evaluate and modify existing Vulkan functions. Layers are implemented as libraries and are loaded during instance creation.

If we want to enable error checking, we need to load a layer that provides such functionality. This kind of layer is know as validation layer. There are different validation layers. Here follows an example. Since validation layers cause overhead, we can disable them when we build the application in release mode.

```
1  const char* const layers[] =
2  {
3      #ifdef _DEBUG
4      "VK_LAYER_KHRONOS_validation",
5      #endif
6      // other layers ...
7  };
```

Listing 2.4: Enabling the Khronos validation layer

Checking whether our layers are supported

Before creating our Vulkan instance, we should check if the layers we require are actually supported. To do this we use vkEnumerateInstanceLayerProperties. This function returns all the layers supported by our Vulkan installation. If all the layers we require are present, then we can proceed to create our Vulkan instance.

2.1.4 Extensions

While we initialize our VkInstanceCreateInfo struct, we can specify the instance extensions that we want to enable. The specified instance extensions will be loaded after the Vulkan instance creation.

Extensions are additional features that Vulkan implementations may provide. Extensions add new functions and structs to the API. Extensions may also change some of the behavior of existing functions. We can either enable extensions at an instance level or at a device level.

We can use an extension to provide a callback to handle the debug messages generated by the validation layers.

```
1  const char* const* extensions[] =
2  {
3      #ifdef _DEBUG
4      VK_EXT_DEBUG_UTILS_EXTENSION_NAME,
5      #endif
6      // Other extensions ...
7  };
```

Listing 2.5: Enabling an extention to handle validation layer debug messages

We specify one callback that handles messages generated by instance creation and destruction. We also specify another callback that handles all other API

debug messages.

```
1  #ifdef _DEBUG
2  VkDebugUtilsMessengerCreateInfoEXT dbgInfo = {};
3  dbgInfo.sType =
       VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
4  dbgInfo.messageSeverity = severity;
5  dbgInfo.messageType = type;
6  dbgInfo.pfnUserCallback = VulkanDebugCallback;
7  #endif
8
9  VkInstanceCreateInfo createInfo = {};
10 #ifdef _DEBUG
11 createInfo.pNext = (VkDebugUtilsMessengerCreateInfoEXT*)(dbgInfo);
12 #endif
13
14 // ... after instance creation
15
16 // Enabling debug callback for all other API functions
17 #ifdef _DEBUG
18 VkDebugUtilsMessengerEXT debugMessenger = VK_NULL_HANDLE;
19 CreateDebugUtilsMessengerEXT(instance, &dbgInfo, nullptr, &
       debugMessenger)
20 #endif
```

Listing 2.6: Setting up debug extension callbacks

The function that creates the VkDebugUtilsMessengerEXT object comes from the extension we have enabled. Because of this, we have to load it manually into our address space using vkGetInstanceProcAddr. An elegant way to solve this issue us to create a proxy function that handles this matter for us.

```
1  static VkResult CreateDebugUtilsMessengerEXT
2  (
3      VkInstance instance,
4      const VkDebugUtilsMessengerCreateInfoEXT* pCreateInfo,
5      const VkAllocationCallbacks* pAllocator,
6      VkDebugUtilsMessengerEXT* pDebugMessenger
7  )
8  {
9      PFN_vkCreateDebugUtilsMessengerEXT f = (
       PFN_vkCreateDebugUtilsMessengerEXT)(vkGetInstanceProcAddr(
       instance, "vkCreateDebugUtilsMessengerEXT"));
10     return f(instance, pCreateInfo, pAllocator, pDebugMessenger);
11 }
```

Listing 2.7: Extension function proxy

Checking whether our extensions are supported

Before creating our Vulkan instance, we should check if the instance extensions we require are actually supported. To do this we use vkEnumerateInstance-ExtensionProperties. This function returns all the instance extensions that are supported by our Vulkan installation. If all the instance extensions we require are present, then we can proceed to create our Vulkan instance.

2.1.5 Vulkan Instance Cleanup

When our application is shutting down, we destroy the debug messenger and destroy our vulkan instance. DestroyDebugUtilsMessengerEXT is an extension

function proxy.

```
1  #ifdef _DEBUG
2  DestroyDebugUtilsMessengerEXT(instance, debugMessenger, nullptr);
3  #endif
4
5  vkDestroyInstance(instance, nullptr);
```

Listing 2.8: Vulkan Instance Cleanup

2.2 Open A Window

After creating our Vulkan instance we open a window. To do this we have two options. We can use a cross platform library that will do all the heavy lifting for us, so that we don't have to worry about directly interacting with the OS, freeing us from the burden of knowing how its windowing API works. We can also decide to not use a library and opening the window ourselves. We will do the latter, since it's interesting to know how things work under the surface.

Since I'm on Windows, I'll be dealing with the Win32 API. We won't go in depth about the specifics of this API since it's beyond our scope.

2.2.1 Create Window Handle

To create a handle to a window we use CreateWindowEx. We use windowStyle and windowExtendedStyle variables to configure how we want our window.

```
1  DWORD windowStyle = (WS_OVERLAPPEDWINDOW | WS_VISIBLE | WS_CAPTION)
       & (~WS_THICKFRAME) & (~WS_MINIMIZEBOX) & (~WS_MAXIMIZEBOX);
2  DWORD windowExtendedStyle = 0;
3
4  HWND handle = CreateWindowEx(
5      windowExtendedStyle,
6      WINDOW_CLASS_NAME,
7      name,
8      windowStyle,
9      CW_USEDEFAULT, CW_USEDEFAULT,
10     windowWidth, windowHeight,
11     0,
12     0,
13     GetModuleHandle(0),
14     0
15 );
```

Listing 2.9: Creating a window handle using Win32 API

2.2.2 Computing Window Dimensions

Before creating our window, we need to compute its width and height. This is due to the fact that a window comprises of a client area and a non client area. We usually want our client area to be of a certain size, but CreateWindowEx takes the whole window width and the whole window height as arguments.
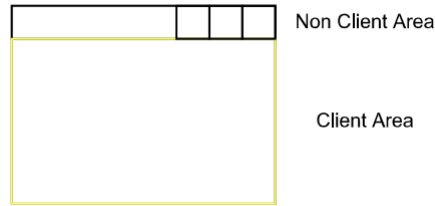
Figure 2.1: Anatomy of a Win32 Window

```
1  RECT windowDimensions = { 0, 0, clientWidth, clientHeight };
2  AdjustWindowRectEx(&windowDimensions, windowStyle, false,
       windowExtendedStyle);
3  i32 windowWidth = windowDimensions.right - windowDimensions.left;
4  i32 windowHeight = windowDimensions.bottom - windowDimensions.top;
```
Listing 2.10: Compute window width and height

2.2.3 Register Window Class

Before creating our window, we need to register its window class. To do this we use RegisterClassEx. This function takes a pointer to a WNDCLASSEX struct. This struct is used to configure our window class;

```
1  WNDCLASSEX windowClass = {};
2  windowClass.cbSize = sizeof(windowClass);
3  windowClass.style = CS_HREDRAW | CS_VREDRAW;
4  windowClass.lpfnWndProc = WindowProcedure;
5  windowClass.hInstance = GetModuleHandle(0);
6  windowClass.hIcon = LoadIcon(0, IDI_APPLICATION);
7  windowClass.hCursor = LoadCursor(0, IDC_ARROW);
8  windowClass.lpszClassName = WINDOW_CLASS_NAME;
9  windowClass.hIconSm = LoadIcon(0, IDI_APPLICATION);
10
11 RegisterClassEx(&windowClass);
```
Listing 2.11: Register Window Class

2.2.4 Window Procedure

While filling in our WNDCLASSEX struct, we also passed a WindowProcedure. This is a callback function that we have to define. We use this function to handle the events that our window will receive during the lifespan of our application.

The Win32 API also provides a default window procedure. Our custom window procedure will call this default procedure when we don't want to handle particular events ourselves.

```
1  static LRESULT CALLBACK WindowProcedure(HWND hwnd, UINT msg, WPARAM
       wparam, LPARAM lparam)
2  {
3      LRESULT result = 0;
4      switch (msg)
5      {
6      case WM_QUIT:
7      case WM_CLOSE:
8      case WM_DESTROY: { PostQuitMessage(0); } break;
9      default: { result = DefWindowProcA(hwnd, msg, wparam, lparam);
       } break;
10     };
11
12     return result;
13 }
```

Listing 2.12: Window Procedure

2.2.5 Window Cleanup

When our application is shutting down, we destroy our window and unregister its class.

```
1  DestroyWindow(handle);
2  UnregisterClass(WINDOW_CLASS_NAME, GetModuleHandle(0));
```

Listing 2.13: Window Cleanup

2.3 Create A Presentation Surface

We must link our newly created window to our Vulkan instance. To do this we create a presentation (or window) surface. This operation is platform specific. Since we are using Windows, to create our presentation surface we need to use vkCreateWin32SurfaceKHR.

```
1  VkSurfaceKHR surface = VK_NULL_HANDLE;
2  vkCreateWin32SurfaceKHR(instance, &createInfo, nullptr, &surface);
```

Listing 2.14: Create Presentation Surface

2.3.1 VkWin32SurfaceCreateInfoKHR

When we call vkCreateWin32SurfaceKHR, we need to pass a pointer to a Vk-Win32SurfaceCreateInfoKHR struct. Such struct lets us configure our presentation surface creation.

```
1  VkWin32SurfaceCreateInfoKHR createInfo = {};
2  createInfo.sType = VK_STRUCTURE_TYPE_WIN32_SURFACE_CREATE_INFO_KHR;
3  createInfo.hinstance = GetModuleHandleA(0);
4  createInfo.hwnd = handle;
```

Listing 2.15: Filling in a VkWin32SurfaceCreateInfoKHR struct

2.3.2 Required Instance Extensions

Vulkan, being cross platform, cannot interact directly with the OS windowing system. To do this we use extensions.

The first extension that we enable is the instance level KHR surface extension. This extension exposes a VkSurfaceKHR object that represents a surface to present rendered images to. This surface will be backed by the window we have created.

The second extension we enable is platform specific and is needed to create our VkSurfaceKHR object. In our case, since we are using Windows, we enable the instance level KHR win32 surface extension.

```
1  #define VK_USE_PLATFORM_WIN32_KHR
2  #include "Vulkan.h"
3
4  const char* const* extensions[] =
5  {
6      VK_KHR_SURFACE_EXTENSION_NAME,
7      VK_KHR_WIN32_SURFACE_EXTENSION_NAME,
8      // ... other extensions
9  }
```

Listing 2.16: Presentation Surface Extensions

Notice the define preprocessor directive right before including our Vulkan header. We do this to access our native platform functions.

2.3.3 Presentation Surface Cleanup

When our application is shutting down, we destroy our presentation surface.

```
1  vkDestroySurfaceKHR(instance, surface, nullptr);
```

Listing 2.17: Presentation Surface Cleanup

2.4 Pick A Physical Device

Now that we have a Vulkan instance and a presentation surface, we select a physical device (a GPU) that supports the features we need. The selected GPU will be the one that will be used by our application.

2.4.1 Listing Available Physical Devices

We first get a list of all the physical devices that are available on the system. To do this we use vkEnumeratePhysicalDevices. These physical devices can either be integrated or dedicated GPUs.

2.4.2 Finding A Suitable Physical Device

Now that we have a list of all the physical devices, we can select one of them. We could, for example, automatically pick the first one without doing any kind of checking. This approach is doable if we don't have any particular requirement for our physical devices.

Usually we have a set of specific physical device features that are mandatory for our application to run. Hence, in our list, some physical devices will be suitable for our application, while others won't.

The approach we take here is to iterate through the list of all physical devices and pick the first one that is suitable for our application. One question still remains: how can we tell whether a physical device is suitable or not?

Spport Grpahics Operations

To check if our physical device supports graphics operations we list all the queue families of our physical device. To do this we use vkGetPhysicalDeviceQueue-FamilyProperties. Then we check if at least one queue family supports graphics operations.

```
1  for (u32 i = 0; i < queueFamilyCount; i++)
2  {
3      VkQueueFamilyProperties queueFamily = queueFamilies[i];
4      if (queueFamily.queueFlags & VK_QUEUE_GRAPHICS_BIT)
5      {
6          // graphics operations supported and i is the index
7          // of a queue family that supports such operations
8      }
9  }
```

Listing 2.18: Check for graphics operations support

Spport Present Operations

To check if our physical device supports present operations we list all the queue families of our physical device. To do this we use vkGetPhysicalDeviceQueue-FamilyProperties. Then we check if at least one queue family supports present operations.

```
1   for (u32 i = 0; i < queueFamilyCount; i++)
2   {
3       VkBool32 presentSupport = false;
4       vkGetPhysicalDeviceSurfaceSupportKHR(physicalDevice, i, surface
        , &presentSupport);
5       if (presentSupport)
6       {
7           // present operations are supported and i is the index
8           // of a queue family that supports such operations
9       }
10  }
```

Listing 2.19: Check for present operations support

Support Presentation To A Surface

Not only our physical device must support present operations. It must also be able to present images to the screen. Image presentation is tied to the window and consequently to the surface associated with it. For this reason, image presentation to the screen is not part of Vulkan. We have to enable the KHR swapchain device extension to support such operation. We need this particular extension because image presentation to a surface is achieved using a swapchain.

```
1  const char* const* deviceExtensions[] =
2  {
3      VK_KHR_SWAPCHAIN_EXTENSION_NAME,
4      // ... other device extensions
5  };
```
Listing 2.20: Device extension for image presentation to the screen

As we have seen earlier, before enabling an extension, we should check for its support. To check whether our physical device supports one or more device extensions we use vkEnumerateDeviceExtensionProperties. This function returns a list of all the extensions supported by our physical device. Then, we simply check whether all the extensions we require are present in the list.

Support A Present Mode

Checking if a swapchain is supported is not sufficient. Even if it's supported, it may not be compatible with our presentation surface. We need to check whether our physical device supports at least one present mode for our presentation surface. We can do this using vkGetPhysicalDeviceSurfacePresentModesKHR. This functions returns a list of present modes supported by our physical device that are compatible with our presentation surface. If there is at least one present mode in the list, then we are good to go.

2.5   Create A Logical Device

To interact with the physical device we have selected we need to create a logical device.

```
1  VkPhysicalDevice physicalDevice = VK_NULL_HANDLE;
2  u32 graphicsQueueFamilyIndex;
3  u32 presentQueueFamilyIndex;
4
5  // ... selecting physical device
6
7  VkDevice device = VK_NULL_HANDLE;
8  vkCreateDevice(physicalDevice, &createInfo, nullptr, &device)
```
Listing 2.21: Create a logical device

2.5.1   VkDeviceCreateInfo

When we call vkCreateDevice, we need to pass a pointer to a VkDeviceCreateInfo struct. Such struct lets us configure the device we are about to create.

During physical device picking, we saved two queue family indices. The first for a queue family that supports graphics operations. The second for a queue family that supports present operations. The ways we populate our VkDeviceCreateInfo struct is different based on whether these two indices are equal or not.

```
1  // Specify requested device features here
2  VkPhysicalDeviceFeatures deviceFeatures = {};
3  f32 queuePriority = 1.0f;
4
5  VkDeviceQueueCreateInfo queueCreateInfo = {};
6  queueCreateInfo.sType = VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
7  queueCreateInfo.queueFamilyIndex = graphicsQueueFamilyIndex;
8  queueCreateInfo.queueCount = 1;
9  queueCreateInfo.pQueuePriorities = &queuePriority;
10
11 VkDeviceCreateInfo createInfo = {};
12 createInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
13 createInfo.queueCreateInfoCount = 1;
14 createInfo.pQueueCreateInfos = &queueCreateInfo;
15 createInfo.enabledExtensionCount = deviceExtensionCount;
16 createInfo.ppEnabledExtensionNames = deviceExtensions;
17 createInfo.pEnabledFeatures = &deviceFeatures;
```

Listing 2.22: Create info struct when queue families are the same

```
1  // Specify requested device features here
2  VkPhysicalDeviceFeatures deviceFeatures = {};
3  f32 queuePriority = 1.0f;
4
5  VkDeviceQueueCreateInfo graphicsQueueCreateInfo = {};
6  graphicsQueueCreateInfo.sType =
       VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
7  graphicsQueueCreateInfo.queueFamilyIndex = graphicsQueueFamilyIndex
       ;
8  graphicsQueueCreateInfo.queueCount = 1;
9  graphicsQueueCreateInfo.pQueuePriorities = &queuePriority;
10
11 VkDeviceQueueCreateInfo presentQueueCreateInfo = {};
12 presentQueueCreateInfo.sType =
       VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO;
13 presentQueueCreateInfo.queueFamilyIndex = presentQueueFamilyIndex;
14 presentQueueCreateInfo.queueCount = 1;
15 presentQueueCreateInfo.pQueuePriorities = &queuePriority;
16
17 VkDeviceQueueCreateInfo queueCreateInfos[] =
18 {
19     graphicsQueueCreateInfo,
20     presentQueueCreateInfo,
21 };
22
23 VkDeviceCreateInfo createInfo = {};
24 deviceInfo.sType = VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO;
25 createInfo.queueCreateInfoCount = arraysize(queueCreateInfos);
26 createInfo.pQueueCreateInfos = queueCreateInfos;
27 createInfo.enabledExtensionCount = deviceExtensionCount;
28 createInfo.ppEnabledExtensionNames = deviceExtensions;
29 createInfo.pEnabledFeatures = &deviceFeatures;
```

Listing 2.23: Create info struct when queue families are different

2.5.2 Retrieve Queue Handles

After creating our logical device, it's useful to retrieve the handles to the queues we will use during our application. In our case we need to submit both graphics and present commands to our GPU. Hence, we need a handle to a graphics

queue and a handle to a present queue.

```
1  VkQueue graphicsQueue = VK_NULL_HANDLE;
2  vkGetDeviceQueue(device, graphicsQueueFamilyIndex, 0, &
       graphicsQueue);
3
4  VkQueue presentQueue = VK_NULL_HANDLE;
5  vkGetDeviceQueue(device, presentQueueFamilyIndex, 0, &presentQueue)
       ;
```

Listing 2.24: Retrieve queue handles

2.6 Create A Swapchain

After having created our logical device, we can create a swapchain object. We
need a swapchain to handle the logic for image presentation.

```
1  VkSwapchainKHR swapchain = VK_NULL_HANDLE;
2  vkCreateSwapchainKHR(device, &createInfo, nullptr, &swapchain);
```

Listing 2.25: Create a swapchain

2.6.1 VkSwapchainCreateInfoKHR

When we call vkCreateSwapchainKHR, we need to pass a pointer to a Vk-
SwapchainCreateInfoKHR struct. Such struct lets us configure the swapchain
we are about to create.

```
1   VkSwapchainCreateInfoKHR createInfo = {};
2   createInfo.sType = VK_STRUCTURE_TYPE_SWAPCHAIN_CREATE_INFO_KHR;
3   createInfo.surface = surface;
4   createInfo.minImageCount = swapchainMinImageCount;
5   createInfo.imageFormat = swapchainImageFormat.format;
6   createInfo.imageColorSpace = swapchainImageFormat.colorSpace;
7   createInfo.imageExtent = swapchainImageExtent;
8   createInfo.imageArrayLayers = 1;
9   createInfo.imageUsage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT;
10  createInfo.preTransform = surfaceCapabilites.currentTransform;
11  createInfo.compositeAlpha = VK_COMPOSITE_ALPHA_OPAQUE_BIT_KHR;
12  createInfo.presentMode = swapchainPresentMode;
13  createInfo.clipped = VK_TRUE;
14  createInfo.oldSwapchain = VK_NULL_HANDLE;
```

Listing 2.26: Configure our swapchain

We have to additionally provide other data that depends on whether or not
we use the same queue for graphics and present operations.

```
1   u32 queueFamilyIndices [] =
2   {
3       graphicsQueueFamilyIndex ,
4       presentQueueFamilyIndex ,
5   };
6
7   if ( graphicsQueueFamilyIndex != presentQueueFamilyIndex )
8   {
9       // Using the concurrent sharing mode we don't need to worry
10      // about resource queue ownership transitions
11      swapchainCreateInfo . imageSharingMode =
        VK_SHARING_MODE_CONCURRENT;
12      swapchainCreateInfo . queueFamilyIndexCount = arraysize (
        queueFamilyIndices );
13      swapchainCreateInfo . pQueueFamilyIndices = queueFamilyIndices ;
14  }
15  else
16  {
17      // We use a single queue , thus it can exclusively own the
18      // swapchain images that will be created .
19      // This is more efficient
20      swapchainCreateInfo . imageSharingMode =
        VK_SHARING_MODE_EXCLUSIVE;
21  }
```

Listing 2.27: Configure queue ownership over swapchain images

2.6.2 Select The Minimum Swapchain Image Count

Here we want to determine the minimum number of swapchain images to create. We can do this by querying the surface capabilities with vkGetPhysicalDeviceSurfaceCapabilitiesKHR.

```
1   u32 swapchainMinImageCount = capabilities ->minImageCount + 1;
2   // If maxImageCount is 0, there is no limit on the number of images
3   if (( capabilities ->maxImageCount > 0) && ( swapchainMinImageCount >
        capabilities ->maxImageCount ))
4   {
5       swapchainMinImageCount = capabilities ->maxImageCount ;
6   }
7
8   return swapchainMinImageCount ;
```

Listing 2.28: Select swapchain image count

Here we would like to use one more image than the bare minimum. This is due to the fact that, if we use the bare minimum number of images, we may have to wait for the driver to complete internal operations before we can acquire another swapchain image to render to.

Here we also be aware of the fact that there can be a maximum number of swapchain images we can require. Thus, we must be careful to cap the minimum number of images that we request to the nominal maximum.

2.6.3 Select The Swapchain Image Format

We must specify a proper format for our swapchain images. To do this, we first query for all image formats that are supported by our surface. We can do this using vkGetPhysicalDeviceSurfaceFormatsKHR.

Once we have a list of valid formats we could either pick one randomly or try to pick the one that we consider the best.

```
1   if ((formatCount == 1) && (formats[0].format == VK_FORMAT_UNDEFINED
        ))
2   {
3       // There is no preferred surface format
4       return { VK_FORMAT_R8G8B8A8_UNORM ,
        VK_COLORSPACE_SRGB_NONLINEAR_KHR };
5   }
6   else
7   {
8       // We have to pick a format from the list
9       // We search for a format that we like
10      for (u32 i = 0; i < formatCount; i++)
11      {
12          if (formats[i].format == VK_FORMAT_R8G8B8A8_UNORM)
13          {
14              return formats[i];
15          }
16      }
17
18      // We haven't found the format(s) that we were looking for
19      // Pick the first format
20      return formats[0];
21  }
```

Listing 2.29: Select swapchain image format

For the color space we'll use SRGB, if it is available, because it's pretty much the standard color space for images. Because of this we should prefer a SRGB color format.

2.6.4   Select The Swapchain Image Extent

We must specify the resolution for our swapchain images. This will almost always be equal to the resolution of our window. Some window systems allow us to differ, indicating that the current width and height are the maximum value of a an unsigned 32 bits integer. In this scenario, we have to pick the resolution that best matches the window within the bounds specified by our surface capabilities.

```
1   if (capabilities->currentExtent.width == 0xFFFFFFFF)
2   {
3       VkExtent2D extent = { windowWidth, windowHeight };
4       extent.width = clamp(extent.width, capabilities->minImageExtent
        .width, capabilities->maxImageExtent.width);
5       extent.height = clamp(extent.height, capabilities->
        minImageExtent.height, capabilities->maxImageExtent.height);
6       return extent;
7   }
8   else
9   {
10      // the current surface size is perfect for the job
11      return capabilities->currentExtent;
12  }
```

Listing 2.30: Select swapchain image extent

2.6.5 Select The Swapchain Presentation Mode

A presentation mode tells the conditions for showing our swapchain images to the screen.

We start by listing all the presentation modes that our physical device supports for presenting images to our surface. We can do this using vkGetPhysicalDeviceSurfacePresentModesKHR. We already did this while selecting our physical device. After that, we check whether the mailbox presentation mode is supported. We would like to use this present mode because it doesn't suffer from tearing and it's not locked to the screen refresh rate. If it's present we are good to go. Otherwise we select the presentation mode that is guaranteed to be always supported: `VK_PRESENT_MODE_FIFO_KHR`.

```
1   for (u32 i = 0; i < modeCount; i++)
2   {
3       if (modes[i] == VK_PRESENT_MODE_MAILBOX_KHR)
4       {
5           return VK_PRESENT_MODE_MAILBOX_KHR;
6       }
7   }
8
9   // Use FIFO since it's always supported (spec)
10  return VK_PRESENT_MODE_FIFO_KHR;
```

Listing 2.31: Select swapchain present mode

2.6.6 Retrieve The Swapchain Images

Now that we have created a swapchain we can retrieve the handles to the images in it. We use these images during rendering. We can do this using vkGetSwapchainImagesKHR;

2.6.7 Create Swapchain Image Views

Vulkan doesn't allow us to use images directly. Before using an image, we first have to create a view on it. This also applies to our swapchain images. Thus, for every image in the swapchain, we must create a corresponding image view for it.

```
1   VkImageViewCreateInfo createInfo = {};
2   createInfo.sType = VK_STRUCTURE_TYPE_IMAGE_VIEW_CREATE_INFO;
3   createInfo.image = swapchainImages[i];
4   createInfo.viewType = VK_IMAGE_VIEW_TYPE_2D;
5   createInfo.format = swapchainImageFormat.format;
6   createInfo.components =
7   {
8       VK_COMPONENT_SWIZZLE_IDENTITY,
9       VK_COMPONENT_SWIZZLE_IDENTITY,
10      VK_COMPONENT_SWIZZLE_IDENTITY,
11      VK_COMPONENT_SWIZZLE_IDENTITY,
12  };
13  createInfo.subresourceRange =
14  {
15      VK_IMAGE_ASPECT_COLOR_BIT,
16      0,
17      1,
18      0,
19      1,
20  };
21
22  VkImageView* swapchainImageViews = nullptr;
23  // .. allocate an array
24  vkCreateImageView(device, &createInfo, nullptr, &
        swapchainImageViews[i]);
```

Listing 2.32: Create swapchain image views

2.6.8 Cleanup

TODO: to be written ...

The swapchain images will be automatically destroyed when we destroy our swapchain.

Chapter 3

Clear The Window

Chapter 4

Our First Pipeline

Chapter 5

Vertex Buffer

Chapter 6

Staging Buffer

Chapter 7

Uniform Buffer

Chapter 8

Depth Buffer

Chapter 9

Setting Up A Simple Scene

Chapter 10

Blinn-Phong Lighting

Chapter 11

Multisample Anti Aliasing

Chapter 12

Conclusion

Appendix A

Appendix

Bibliography

[1] Creating a window - win32 apps — microsoft docs. ”https://docs.microsoft.com/en-us/windows/win32/learnwin32/creating-a-window”.

[2] Erika Johnson. Overview of vulkan loader and layers - LunarG. ”https://www.lunarg.com/tutorial-overview-of-vulkan-loader-layers”.

[3] Pawel Lapinski. Api without secrets: Introduction to vulkan part 1: The beginning. ”https://www.intel.com/content/www/us/en/developer/articles/training/api-without-secrets-introduction-to-vulkan-part-1.html”.

[4] Pawel Lapinski. Api without secrets: Introduction to vulkan part 2: Swap chain. ”https://www.intel.com/content/www/us/en/developer/articles/training/api-without-secrets-introduction-to-vulkan-part-2.html”.

[5] Pawel Lapinski. *Vulkan Cookbook: work through recipes to unlock the full potential of the next generation graphics API-Vulkan.* Packt, Birmingham, 2017.

[6] Alexander Overvoorde. Image views - Vulkan Tutorial. ”https://vulkan-tutorial.com/Drawing_a_triangle/Presentation/Image_views”.

[7] Alexander Overvoorde. Instance - Vulkan Tutorial. ”https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Instance”.

[8] Alexander Overvoorde. Introduction - Vulkan Tutorial. ”https://vulkan-tutorial.com/”.

[9] Alexander Overvoorde. Logical devices and queues - Vulkan Tutorial. ”https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Logical_device_and_queues”.

[10] Alexander Overvoorde. Physical devices and queue families - Vulkan Tutorial. ”https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Physical_devices_and_queue_families”.

[11] Alexander Overvoorde. Swap chain - Vulkan Tutorial. ”https://vulkan-tutorial.com/Drawing_a_triangle/Presentation/Swap_chain”.

[12] Alexander Overvoorde. Validation layers - Vulkan Tutorial. "https://vulkan-tutorial.com/Drawing_a_triangle/Setup/Validation_layers".

[13] Alexander Overvoorde. Window surface - Vulkan Tutorial. "https://vulkan-tutorial.com/Drawing_a_triangle/Presentation/Window_surface".

[14] Graham Sellers. *Vulkan programming guide: the official guide to learning Vulkan.* OpenGL series. Addison-Wesley, Boston, 2017.