API without Secrets: Introduction to Vulkan* Part 1

Table of Contents

Tutorial 1: Vulkan* – The Beginning

We start with a simple application that unfortunately doesn't display anything. I won't present the full source code (with windowing, rendering loop, and so on) here in the text as the tutorial would be too long. The entire sample project with full source code is available in a provided example that can be found at https://github.com/gametechdev/IntroductionToVulkan. Here I show only the parts of the code that are relevant to Vulkan itself. There are several ways to use the Vulkan API in our application:

1. We can dynamically load the driver's library that provides Vulkan API implementation and acquire function pointers by ourselves from it.
2. We can use the Vulkan SDK and link with the provided Vulkan Runtime (Vulkan Loader) static library.
3. We can use the Vulkan SDK, dynamically load Vulkan Loader library at runtime, and load function pointers from it.

The first approach is not recommended. Hardware vendors can modify their drivers in any way, and it may affect compatibility with a given application. It may even break the application and requiredevelopers writing a Vulkan-enabled application to rewrite some parts of the code. That's why it's better to use some level of abstraction.

The recommended solution is to use the Vulkan Loader from the Vulkan SDK. It provides more configuration abilities and more flexibility without the need to modify Vulkan application source code. One example of the flexibility is Layers. The Vulkan API requires developers to create applications that strictly follow API usage rules. In case of any errors, the driver provides us with little feedback, only some severe and important errors are reported (for example, out of memory). This approach is used so the API itself can be as small (thin) and as fast as possible. But if we want to obtain more information about what we are doing wrong we have to enable debug/validation layers. There are different layers for different purposes such as memory usage, proper parameter passing, object life-time checking, and so on. These layers all slow down the application's performance but provide us with much more information.

We also need to choose whether we want to statically link with a Vulkan Loader or whether we will load it dynamically and acquire function pointers by ourselves at runtime. This choice is just a matter of personal preference. This paper focuses on the third way of using Vulkan: dynamically loading function pointers from the Vulkan Runtime library. This approach is similar to what we had to do when we wanted to use OpenGL* on a Windows* system in which only some basic functions were provided by the default implementation. The remaining functions had to be loaded dynamically using wglGetProcAddress() or standard windows GetProcAddress() functions. This is what wrangler libraries such as GLEW or GL3W were created for.

Loading Vulkan Runtime Library and Acquiring Pointer to an Exported Function

In this tutorial we go through the process of acquiring Vulkan functions pointers by ourselves. We load them from the Vulkan Runtime library (Vulkan Loader) which should be installed along with the graphics driver that supports Vulkan. The dynamic library for Vulkan (Vulkan Loader) is named vulkan-1.dll on Windows* and libvulkan.so on Linux*.

From now on, I refer to the first tutorial's source code, focusing on the Tutorial01.cpp file. So in the initialization code of our application we have to load the Vulkan library with something like this:

```
#if defined(VK_USE_PLATFORM_WIN32_KHR)
VulkanLibrary = LoadLibrary( "vulkan-1.dll" );
#elif defined(VK_USE_PLATFORM_XCB_KHR) || defined(VK_USE_PLATFORM_XLIB_KHR)
VulkanLibrary = dlopen( "libvulkan.so", RTLD_NOW );
#endif

if( VulkanLibrary == nullptr ) {
  printf( "Could not load Vulkan library!\n" );
  return false;
}
return true;
```

1. Tutorial01.cpp, function LoadVulkanLibrary()

VulkanLibrary is a variable of type HMODULE in Windows or just void* in Linux. If the value returned by the library loading function is not 0 we can load all exported functions. The Vulkan library, as well as Vulkan implementations (every driver from every vendor), are required to expose only one function that can be loaded with the standard techniques our OS possesses (like the previously mentioned GetProcAddress() in Windows or dlsym() in Linux). Other functions from the Vulkan API may also be available for acquiring using this method but it is not guaranteed (and even not recommended). The only function that must be exported is **vkGetInstanceProcAddr()**.

This function is used to load all other Vulkan functions. To ease our work of obtaining addresses of all Vulkan API functions it is very convenient to place their names inside a macro. This way we won't have to duplicate function names in multiple places (like definition, declaration, or loading) and can keep them in only one header file. This single file will be used later for different purposes with an #include directive. We can declare our exported function like this:

```
#if !defined(VK_EXPORTED_FUNCTION)
#define VK_EXPORTED_FUNCTION( fun )
#endif

VK_EXPORTED_FUNCTION( vkGetInstanceProcAddr )

#undef VK_EXPORTED_FUNCTION
```

2. ListOfFunctions.inl, -

Now we define the variables that will represent functions from the Vulkan API. This can be done with something like this:

```
#include "vulkan.h"

#define VK_EXPORTED_FUNCTION( fun ) PFN_##fun fun;
#define VK_GLOBAL_LEVEL_FUNCTION( fun ) PFN_##fun fun;
#define VK_INSTANCE_LEVEL_FUNCTION( fun ) PFN_##fun fun;
#define VK_DEVICE_LEVEL_FUNCTION( fun ) PFN_##fun fun;

#include "ListOfFunctions.inl"
```

3. VulkanFunctions.cpp

Here we first include the vulkan.h file, which is officially provided for developers that want to use Vulkan API in their applications. This file is similar to the gl.h file in the OpenGL library. It defines all enumerations, structures, types, and function types that are necessary for Vulkan application development. Next we define the macros for functions from each "level" (I will describe these levels soon). The function definition requires providing function type and a function name. Fortunately, function types in Vulkan can be easily derived from function names. For example, the definition of **vkGetInstanceProcAddr()** function's type looks like this:

```
typedef PFN_vkVoidFunction (VKAPI_PTR *PFN_vkGetInstanceProcAddr)(VkInstance instance,
const char* pName);
```

4. Vulkan.h

The definition of a variable that represents this function would then look like this:

```
PFN_vkGetInstanceProcAddr vkGetInstanceProcAddr;
```

-

This is what the macros from VulkanFunctions.cpp file expand to. They take the function name (hidden in a "fun" parameter) and add "PFN_" at the beginning. Then the macro places a space after the type, and adds a function name and a semicolon after that. Functions are "pasted" into the file in the line with the #include "ListOfFunctions.inl" directive.

But we must remember that when we want to define Vulkan functions' prototypes by ourselves we must define the VK_NO_PROTOTYPES preprocessor directive. By default the vulkan.h header file contains definitions of all functions. This is useful when we are statically linking with Vulkan Runtime. So when we add our own definitions, there will be a compilation error claiming that the given variables (for function pointers) are defined more than once (since we would break the One Definition rule). We can disable definitions from vulkan.h file using the mentioned preprocessor macro.

Similarly we need to declare variables defined in the VulkanFunctions.cpp file so they would be seen in all other parts of our code. This is done in the same way, but the word "extern" is placed before each function. Compare to the VulkanFunctions.h file.

Now we have variables in which we can store addresses of functions acquired from the Vulkan library. To load the only one exported function, we can use the following code:

```
#if defined(VK_USE_PLATFORM_WIN32_KHR)
#define LoadProcAddress GetProcAddress
#elif defined(VK_USE_PLATFORM_XCB_KHR) || defined(VK_USE_PLATFORM_XLIB_KHR)
#define LoadProcAddress dlsym
#endif

#define VK_EXPORTED_FUNCTION( fun )                                          \
if( !(fun = (PFN_##fun)LoadProcAddress( VulkanLibrary, #fun )) ) {   \
printf( "Could not load exported function: " #fun "!\n" );           \
return false;                                                        \
}

#include "ListOfFunctions.inl"

return true;
```

5. Tutorial01.cpp, function LoadExportedEntryPoints()

This macro takes the function name from the "fun" parameter, converts it into a string (with #) and obtains its address from VulkanLibrary. The address is acquired using the GetProcAddress() (on Windows) or dlsym() (on Linux) function and is stored in the variable represented by fun. If this operation fails and the function is not exposed from the library, we report this problem by printing the proper information and returning false. The macro operates on lines included from ListOfFunctions.inl. This way we don't have to write the names of functions multiple times.

Now that we have our main function-loading procedure, we can load the rest of the Vulkan API procedures. These can be divided into three types:

- Global-level functions. Allow us to create a Vulkan instance.
- Instance-level functions. Check what Vulkan-capable hardware is available and what Vulkan features are exposed.
- Device-level functions. Responsible for performing jobs typically done in a 3D application (like drawing).

We will start with acquiring instance creation functions from the global level.

Acquiring Pointers to Global-Level Functions

Before we can create a Vulkan instance we must acquire the addresses of functions that will allow us to do it. Here is a list of these functions:

- vkCreateInstance
- vkEnumerateInstanceExtensionProperties

- vkEnumerateInstanceLayerProperties

The most important function is **vkCreateInstance()**, which allows us to create a "Vulkan instance." From application point of view Vulkan instance can be thought of as an equivalent of OpenGL's rendering context. It stores per-application state (there is no global state in Vulkan) like enabled instance-level layers and extensions. The other two functions allow us to check what instance layers are available and what instance extensions are available. Validation layers are divided into instance and device levels depending on what functionality they debug. Extensions in Vulkan are similar to OpenGL's extensions: they expose additional functionality that is not required by core specifications, and not all hardware vendors may implement them. Extensions, like layers, are also divided into instance and device levels, and extensions from different levels must be enabled separately. In OpenGL, all extensions are (usually) available in created contexts; using Vulkan we have to enable them before the functionality exposed by them can be used.

We call the function **vkGetInstanceProcAddr()** to acquire addresses of instance-level procedures. It takes two parameters: an instance, and a function name. We don't have an instance yet so we provide "null" for the first parameter. That's why these functions may sometimes be called null-instance or no-instance level functions. The second parameter required by the **vkGetInstanceProcAddr()** function is a name of a procedure address of which we want to acquire. We can only load global-level functions without an instance. It is not possible to load any other function without an instance handle provided in the first parameter.

The code that loads global-level functions may look like this:

```
#define VK_GLOBAL_LEVEL_FUNCTION( fun )                                    \
if( !(fun = (PFN_##fun)vkGetInstanceProcAddr( nullptr, #fun )) ) {   \
  printf( "Could not load global level function: " #fun "!\n" );     \
  return false;                                                            \
}

#include "ListOfFunctions.inl"

return true;
```

6. Tutorial01.cpp, function LoadGlobalLevelEntryPoints()

The only difference between this code and the code used for loading the exported function (**vkGetInstanceProcAddr()** exposed by the library) is that we don't use function provided by the OS, like GetProcAddress(), but we call **vkGetInstanceProcAddr()** where the first parameter is set to null.

If you follow this tutorial and write the code yourself, make sure you add global-level functions wrapped in a properly named macro to ListOfFunctions.inl header file:

```
#if !defined(VK_GLOBAL_LEVEL_FUNCTION)
#define VK_GLOBAL_LEVEL_FUNCTION( fun )
#endif

VK_GLOBAL_LEVEL_FUNCTION( vkCreateInstance )
VK_GLOBAL_LEVEL_FUNCTION( vkEnumerateInstanceExtensionProperties )
VK_GLOBAL_LEVEL_FUNCTION( vkEnumerateInstanceLayerProperties )

#undef VK_GLOBAL_LEVEL_FUNCTION
```

7. ListOfFunctions.inl

Creating a Vulkan Instance

Now that we have loaded global-level functions, we can create a Vulkan instance. This is done by calling the **vkCreateInstance()** function, which takes three parameters.

- The first parameter has information about our application, the requested Vulkan version, and the instance level layers and extensions we want to enable. This all is done with structures (structures are very common in Vulkan).
- The second parameter provides a pointer to a structure with list of different functions related to memory allocation. They can be used for debugging purposes but this feature is optional and we can rely on built-in memory allocation methods.
- The third parameter is an address of a variable in which we want to store Vulkan instance handle. In the Vulkan API it is common that results of operations are stored in variables we provide addresses of. Return values are used only for some pass/fail notifications. Here is the full source code for instance creation:

```
   VkApplicationInfo application_info = {
     VK_STRUCTURE_TYPE_APPLICATION_INFO,             // VkStructureType            sType
     nullptr,                                        // const void                *pNext
     "API without Secrets: Introduction to Vulkan",  // const char
*pApplicationName
     VK_MAKE_VERSION( 1, 0, 0 ),                     // uint32_t
applicationVersion
     "Vulkan Tutorial by Intel",                     // const char
*pEngineName
     VK_MAKE_VERSION( 1, 0, 0 ),                     // uint32_t
engineVersion
     VK_API_VERSION                                  // uint32_t
apiVersion
   };

   VkInstanceCreateInfo instance_create_info = {
     VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO,         // VkStructureType            sType
     nullptr,                                        // const void*               pNext
     0,                                              // VkInstanceCreateFlags      flags
     &application_info,                              // const VkApplicationInfo
*pApplicationInfo
     0,                                              // uint32_t
enabledLayerCount
     nullptr,                                        // const char * const
*ppEnabledLayerNames
     0,                                              // uint32_t
enabledExtensionCount
     nullptr                                         // const char * const
*ppEnabledExtensionNames
   };

   if( vkCreateInstance( &instance_create_info, nullptr, &Vulkan.Instance ) !=
VK_SUCCESS ) {
     printf( "Could not create Vulkan instance!\n" );
     return false;
   }
   return true;
```

8. Tutorial01.cpp, function CreateInstance()

Most of the Vulkan structures begin with a field describing the type of the structure. Parameters are provided to functions by pointers to avoid copying big memory chunks. Sometimes, inside structures, pointers to other structures, are also provided. For the driver to know how many bytes it should read and how members are aligned, the type of the structure is always provided. So what exactly do all these parameters mean?

- sType – Type of the structure. In this case it informs the driver that we are providing information for instance creation by providing a value of VK_STRUCTURE_TYPE_APPLICATION_INFO.

- pNext – Additional information for instance creation may be provided in future versions of Vulkan API and this parameter will be used for that purpose. For now, it is reserved for future use.
- flags – Another parameter reserved for future use; for now it must be set to 0.
- pApplicationInfo – An address of another structure with information about our application (like name, version, required Vulkan API version, and so on).
- enabledLayerCount – Defines the number of instance-level validation layers we want to enable.
- ppEnabledLayerNames – This is an array of enabledLayerCount elements with the names of the layers we would like to enable.
- enabledExtensionCount – The number of instance-level extensions we want to enable.
- ppEnabledExtensionNames – As with layers, this parameter should point to an array of at least enabledExtensionCount elements containing names of instance-level extensions we want to use.

Most of the parameters can be nulls or zeros. The most important one (apart from the structure type information) is a parameter pointing to a variable of type VkApplicationInfo. So before specifying instance creation information, we also have to specify an additional variable describing our application. This variable contains the name of our application, the name of the engine we are using, or the Vulkan API version we require (which is similar to the OpenGL version; if the driver doesn't support this version, the instance will not be created). This information may be very useful for the driver. Remember that some graphics card vendors provide drivers that can be specialized for a specific title, such as a specific game. If a graphics card vendor knows what graphics the engine game uses, it can optimize the driver's behavior so the game performs faster. This application information structure can be used for this purpose. The parameters from the VkApplicationInfo structure include:

- sType – Type of structure. Here VK_STRUCTURE_TYPE_APPLICATION_INFO, information about the application.
- pNext – Reserved for future use.
- pApplicationName – Name of our application.
- applicationVersion – Version of our application; it is quite convenient to use Vulkan macro for version creation. It packs major, minor, and patch numbers into one 32-bit value.
- pEngineName – Name of the engine our application uses.
- engineVersion – Version of the engine we are using in our application.
- apiVersion – Version of the Vulkan API we want to use. It is best to provide the version defined in the Vulkan header we are including, which is why we use VK_API_VERSION found in the vulkan.h header file.

So now that we have defined these two structures we can call the **vkCreateInstance()** function and check whether an instance was created. If successful, instance handle will be stored in a variable we provided the address of and VK_SUCCESS (which is zero!) is returned.

Acquiring Pointers to Instance-Level Functions

We have created a Vulkan instance. Next we can acquire pointers to functions that allow us to create a logical device, which can be seen as a user view on a physical device. There may be many different devices installed on a computer that support Vulkan. Each of these devices may have different features and capabilities and different performance, or may support different functionalities. When we want to use Vulkan, we must specify which device to perform the operations on. We may use many devices for different purposes (such as one for rendering 3D graphics, one for physics calculations, and one for media decoding). We must check what devices and how many of them are available, what their capabilities are, and what operations they support. This is all done with instance-level functions. We get the addresses of these functions using the **vkGetInstanceProcAddr()** function used earlier. But this time we will provide handle to a created Vulkan instance.

Loading every Vulkan procedure using the **vkGetInstanceProcAddr()** function and Vulkan instance handle comes with some trade-offs. When we use Vulkan for data processing, we must create a logical device and acquire device-level functions. But on the computer that runs our application, there may be many devices that support Vulkan. Determining

which device to use depends on the mentioned logical device. But **vkGetInstanceProcAddr()** doesn't recognize a logical device, as there is no parameter for it. When we acquire device-level procedures using this function we in fact acquire addresses of a simple "jump" functions. These functions take the handle of a logical device and jump to a proper implementation (function implemented for a specific device). The overhead of this jump can be avoided. The recommended behavior is to load procedures for each device separately using another function. But we still have to use the **vkGetInstanceProcAddr()** function to load functions that allow us to create such a logical device.

Some of the instance level functions include:

- vkEnumeratePhysicalDevices
- vkGetPhysicalDeviceProperties
- vkGetPhysicalDeviceFeatures
- vkGetPhysicalDeviceQueueFamilyProperties
- vkCreateDevice
- vkGetDeviceProcAddr
- vkDestroyInstance

These are the functions that are required and are used in this tutorial to create a logical device. But there are other instance-level functions, that is, from extensions. The list in a header file from the example solution's source code will expand. The source code used to load all these functions is:

```
#define VK_INSTANCE_LEVEL_FUNCTION( fun )                                      \
if( !(fun = (PFN_##fun)vkGetInstanceProcAddr( Vulkan.Instance, #fun )) ) {   \
  printf( "Could not load instance level function: " #fun "\n" );            \
  return false;                                                              \
}

#include "ListOfFunctions.inl"

return true;
```

9. Tutorial01.cpp, function LoadInstanceLevelEntryPoints()

The code for loading instance-level functions is almost identical to the code loading global-level functions. We just change the first parameter of **vkGetInstanceProcAddr()** function from null to create Vulkan instance handle. Of course we also operate on instance-level functions so now we redefine the VK_INSTANCE_LEVEL_FUNCTION() macro instead of a VK_GLOBAL_LEVEL_FUNCTION() macro. We also need to define functions from the instance level. As before, this is best done with a list of macro-wrapped names collected in a shared header, for example:

```
#if !defined(VK_INSTANCE_LEVEL_FUNCTION)
#define VK_INSTANCE_LEVEL_FUNCTION( fun )
#endif

VK_INSTANCE_LEVEL_FUNCTION( vkDestroyInstance )
VK_INSTANCE_LEVEL_FUNCTION( vkEnumeratePhysicalDevices )
VK_INSTANCE_LEVEL_FUNCTION( vkGetPhysicalDeviceProperties )
VK_INSTANCE_LEVEL_FUNCTION( vkGetPhysicalDeviceFeatures )
VK_INSTANCE_LEVEL_FUNCTION( vkGetPhysicalDeviceQueueFamilyProperties )
VK_INSTANCE_LEVEL_FUNCTION( vkCreateDevice )
VK_INSTANCE_LEVEL_FUNCTION( vkGetDeviceProcAddr )
VK_INSTANCE_LEVEL_FUNCTION( vkEnumerateDeviceExtensionProperties )

#undef VK_INSTANCE_LEVEL_FUNCTION
```

10. ListOfFunctions.inl

Instance-level functions operate on physical devices. In Vulkan we can see “physical devices” and “logical devices” (simply called devices). As the name suggests, a physical device refers to any physical graphics card (or any other hardware component) that is installed on a computer running a Vulkan-enabled application that is capable of executing Vulkan commands. As mentioned earlier, such a device may expose and implement different (optional) Vulkan features, may have different capabilities (like total memory or ability to work on buffer objects of different sizes), or may provide different extensions. Such hardware may be a dedicated (discrete) graphics card or an additional chip built (integrated) into a main processor. It may even be the CPU itself. Instance-level functions allow us to check all these parameters. After we check them, we must decide (based on our findings and our needs) which physical device we want to use. Maybe we even want to use more than one device, which is also possible, but this scenario is too advanced for now. So if we want to harness the power of any physical device we must create a logical device that represents our choice in the application (along with enabled layers, extensions, features, and so on). After creating a device (and acquiring queues) we are prepared to use Vulkan, the same way as we are prepared to use OpenGL after creating rendering context.

Creating a Logical Device

Before we can create a logical device, we must first check to see how many physical devices are available in the system we execute our application on. Next we can get handles to all available physical devices:

```
    uint32_t num_devices = 0;
    if( (vkEnumeratePhysicalDevices( Vulkan.Instance, &num_devices, nullptr ) !=
VK_SUCCESS) ||
        (num_devices == 0) ) {
      printf( "Error occurred during physical devices enumeration!\n" );
      return false;
    }

    std::vector<VkPhysicalDevice> physical_devices( num_devices );
    if( vkEnumeratePhysicalDevices( Vulkan.Instance, &num_devices, &physical_devices[0] )
!= VK_SUCCESS ) {
      printf( "Error occurred during physical devices enumeration!\n" );
      return false;
    }
```

11. Tutorial01.cpp, function CreateDevice()

To check how many devices are available, we call the **vkEnumeratePhysicalDevices()** function. We call it twice, first with the last parameter set to null. This way the driver knows that we are asking only for the number of available physical devices. This number will be stored in the variable we provided the address of in the second parameter.

Now that we know how many physical devices are available we can prepare storage for their handles. I use a vector so I don’t need to worry about memory allocation and deallocation. When we call **vkEnumeratePhysicalDevices()** again, this time with all the parameters not equal to null, we will acquire handles of the physical devices in the array we provided addresses of in the last parameter. This array may not be the same size as the number returned after the first call, but it must hold the same number of elements as defined in the second parameter.

Example: we can have four physical devices available, but we are interested only in the first one. So after the first call we set a value of four in num_devices. This way we know that there is any Vulkan-compatible device and that we can proceed. We overwrite this value with one as we only want to use one (any) such device, no matter which. And we will get only one physical device handle after the second call.

The number of devices we provided will be replaced by the actual number of enumerated physical devices (which of course will not be greater than the value we provided). Example: we don’t want to call this function twice. Our application supports up to 10 devices and we provide this value along with a pointer to a static, 10-element array. The driver always returns the number of actually enumerated devices. If there is none, zero is stored in the variable address we provided. If there is any such device, we will also know that. We will not be able to tell if there are more than 10 devices.

Now that we have handles of all the Vulkan compatible physical devices we can check the properties of each device. In the sample code, this is done inside a loop:

```
    VkPhysicalDevice selected_physical_device = VK_NULL_HANDLE;
    uint32_t selected_queue_family_index = UINT32_MAX;
    for( uint32_t i = 0; i < num_devices; ++i ) {
      if( CheckPhysicalDeviceProperties( physical_devices[i], selected_queue_family_index
) ) {
        selected_physical_device = physical_devices[i];
      }
    }
```

12. Tutorial01.cpp, function CreateDevice()

Device Properties

I created the CheckPhysicalDeviceProperties() function. It takes the handle of a physical device and checks whether the capabilities of a given device are enough for our application to work properly. If so, it returns true and stores the queue family index in the variable provided in the second parameter. Queues and queue families are discussed in a later section.

Here is the first half of a CheckPhysicalDeviceProperties() function:

```
    VkPhysicalDeviceProperties device_properties;
    VkPhysicalDeviceFeatures   device_features;

    vkGetPhysicalDeviceProperties( physical_device, &device_properties );
    vkGetPhysicalDeviceFeatures( physical_device, &device_features );

    uint32_t major_version = VK_VERSION_MAJOR( device_properties.apiVersion );
    uint32_t minor_version = VK_VERSION_MINOR( device_properties.apiVersion );
    uint32_t patch_version = VK_VERSION_PATCH( device_properties.apiVersion );

    if( (major_version < 1) &&
        (device_properties.limits.maxImageDimension2D < 4096) ) {
      printf( "Physical device %p doesn't support required parameters!\n",
physical_device );
      return false;
    }
```

13. Tutorial01.cpp, function CheckPhysicalDeviceProperties()

At the beginning of this function, the physical device is queried for its properties and features. Properties contain fields such as supported Vulkan API version, device name and type (integrated or dedicated/discrete GPU), Vendor ID, and limits. Limits describe how big textures can be created, how many samples in anti-aliasing are supported, or how many buffers in a given shader stage can be used.

Device Features

Features are additional hardware capabilities that are similar to extensions. They may not necessarily be supported by the driver and by default are not enabled. Features contain items such as geometry and tessellation shaders multiple viewports, logical operations, or additional texture compression formats. If a given physical device supports any feature we can enable it during logical device creation. Features are not enabled by default in Vulkan. But the Vulkan spec points out that some features may have performance impact (like robustness).

After querying for hardware info and capabilities, I have provided a small example of how these queries can be used. I “reversed” the VK_MAKE_VERSION macro and retrieved major, minor, and patch versions from the apiVersion field of device properties. I check whether it is above some version I want to use, and also check whether I can create 2D textures of a given size. In this example I’m not using features at all, but if we want to use any feature (that is, geometry shaders) we must check whether it is supported and we must (explicitly) enable it later, during logical device creation. And this is

the reason why we need to create a logical device and not use physical device directly. A logical device represents a physical device and all the features and extensions we enabled for it.

The next part of checking physical device's capabilities—queues—requires additional explanation.

Queues, Queue Families, and Command Buffers

When we want to process any data (that is, draw a 3D scene from vertex data and vertex attributes) we call Vulkan functions that are passed to the driver. These functions are not passed directly, as sending each request separately down through a communication bus is inefficient. It is better to aggregate them and pass in groups. In OpenGL this was done automatically by the driver and was hidden from the user. OpenGL API calls were queued in a buffer and if this buffer was full (or if we requested to flush it) whole buffer was passed to hardware for processing. In Vulkan this mechanism is directly visible to the user and, more importantly, the user must specifically create and manage buffers for commands. These are called (conveniently) command buffers.

Command buffers (as whole objects) are passed to the hardware for execution through queues. However, these buffers may contain different types of operations, such as graphics commands (used for generating and displaying images like in typical 3D games) or compute commands (used for processing data). Specific types of commands may be processed by dedicated hardware, and that's why queues are also divided into different types. In Vulkan these queue types are called families. Each queue family may support different types of operations. That's why we also have to check if a given physical device supports the type of operations we want to perform. We can also perform one type of operation on one device and another type of operation on another device, but we have to check if we can. This check is done in the second half of CheckPhysicalDeviceProperties() function:

```
    uint32_t queue_families_count = 0;
    vkGetPhysicalDeviceQueueFamilyProperties( physical_device, &queue_families_count,
nullptr );
    if( queue_families_count == 0 ) {
      printf( "Physical device %p doesn't have any queue families!\n", physical_device );
      return false;
    }

    std::vector<VkQueueFamilyProperties> queue_family_properties( queue_families_count );
    vkGetPhysicalDeviceQueueFamilyProperties( physical_device, &queue_families_count,
&queue_family_properties[0] );
    for( uint32_t i = 0; i < queue_families_count; ++i ) {
      if( (queue_family_properties[i].queueCount > 0) &&
          (queue_family_properties[i].queueFlags & VK_QUEUE_GRAPHICS_BIT) ) {
        queue_family_index = i;
        return true;
      }
    }

    printf( "Could not find queue family with required properties on physical device
%p!\n", physical_device );
    return false;
```

14. Tutorial01.cpp, function CheckPhysicalDeviceProperties()

We must first check how many different queue families are available in a given physical device. This is done in a similar way to enumerating physical devices. First we call **vkGetPhysicalDeviceQueueFamilyProperties()** with the last parameter set to null. This way, in a "queue_count" a variable number of different queue families is stored. Next we can prepare a place for this number of queue families' properties (if we want to—the mechanism is similar to enumerating physical devices). Next we call the function again and the properties for each queue family are stored in a provided array.

The properties of each queue family contain queue flags, the number of available queues in this family, time stamp support, and image transfer granularity. Right now, the most important part is the number of queues in the family and

flags. Flags (which is a bitfield) define which types of operations are supported by a given queue family (more than one may be supported). It can be graphics, compute, transfer (memory operations like copying), and sparse binding (for sparse resources like mega-textures) operations. Other types may appear in the future.

In our example we check for graphics operations support, and if we find it we can use the given physical device. Remember that we also have to remember the selected family index. After we chose the physical device we can create logical device that will represent it in the rest of our application, as shown in the example:

```
  if( selected_physical_device == VK_NULL_HANDLE ) {
    printf( "Could not select physical device based on the chosen properties!\n" );
    return false;
  }

  std::vector<float> queue_priorities = { 1.0f };

  VkDeviceQueueCreateInfo queue_create_info = {
    VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO,     // VkStructureType
sType
    nullptr,                                        // const void
*pNext
    0,                                              // VkDeviceQueueCreateFlags
flags
    selected_queue_family_index,                    // uint32_t
queueFamilyIndex
    static_cast<uint32_t>(queue_priorities.size()), // uint32_t
queueCount
    &queue_priorities[0]                            // const float
*pQueuePriorities
  };

  VkDeviceCreateInfo device_create_info = {
    VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO,           // VkStructureType
sType
    nullptr,                                        // const void
*pNext
    0,                                              // VkDeviceCreateFlags
flags
    1,                                              // uint32_t
queueCreateInfoCount
    &queue_create_info,                             // const VkDeviceQueueCreateInfo
*pQueueCreateInfos
    0,                                              // uint32_t
enabledLayerCount
    nullptr,                                        // const char * const
*ppEnabledLayerNames
    0,                                              // uint32_t
enabledExtensionCount
    nullptr,                                        // const char * const
*ppEnabledExtensionNames
    nullptr                                         // const VkPhysicalDeviceFeatures
*pEnabledFeatures
  };

  if( vkCreateDevice( selected_physical_device, &device_create_info, nullptr,
&Vulkan.Device ) != VK_SUCCESS ) {
    printf( "Could not create Vulkan device!\n" );
    return false;
  }

  Vulkan.QueueFamilyIndex = selected_queue_family_index;
  return true;
```

First we make sure that after we exited the device features loop, we have found the device that supports our needs. Next we can create a logical device, which is done by calling **vkCreateDevice()**. It takes the handle to a physical device and an address of a structure that contains the information necessary for device creation. This structure is of type VkDeviceCreateInfo and contains the following fields:

- sType – Standard type of a provided structure, VK_STRUCTURE_TYPE_DEVICE_CREATE_INFO here that means we are providing parameters for a device creation.
- pNext – Parameter pointing to an extension specific structure; here we set it to null.
- flags – Another parameter reserved for future use which must be zero.
- queueCreateInfoCount – Number of different queue families from which we create queues along with the device.
- pQueueCreateInfos – Pointer to an array of queueCreateInfoCount elements specifying queues we want to create.
- enabledLayerCount – Number of device-level validation layers to enable.
- ppEnabledLayerNames – Pointer to an array with enabledLayerCount names of device-level layers to enable.
- enabledExtensionCount – Number of extensions to enable for the device.
- ppEnabledExtensionNames – Pointer to an array with enabledExtensionCount elements; each element must contain the name of an extension that should be enabled.
- pEnabledFeatures – Pointer to a structure indicating additional features to enable for this device (see the "Device " section).

Features (as I have described earlier) are additional hardware capabilities that are disabled by default. If we want to enable all available features, we can't simply fill this structure with ones. If some feature is not supported, the device creation will fail. Instead, we should pass a structure that was filled when we called **vkGetPhysicalDeviceFeatures()**. This is the easiest way to enable all supported features. If we are interested only in some specific features, we query the driver for available features and clear all unwanted fields. If we don't want any of the additional features we can clear this structure (fill it with zeros) or pass a null pointer for this parameter (like in this example).

Queues are created automatically along with the device. To specify what types of queues we want to enable, we provide an array of additional VkDeviceQueueCreateInfo structures. This array must contain queueCreateInfoCount elements. Each element in this array must refer to a different queue family; we refer to a specific queue family only once.

The VkDeviceQueueCreateInfo structure contains the following fields:

- sType –Type of structure, here VK_STRUCTURE_TYPE_DEVICE_QUEUE_CREATE_INFO indicating it's queue creation information.
- pNext – Pointer reserved for extensions.
- flags – Value reserved for future use.
- queueFamilyIndex – Index of a queue family from which queues should be created.
- queueCount – Number of queues we want to enable in this specific queue family (number of queues we want to use from this family) and a number of elements in the pQueuePriorities array.
- pQueuePriorities – Array with floating point values describing priorities of operations performed in each queue from this family.

As I mentioned previously, each element in the array with VkDeviceQueueCreateInfo elements must describe a different queue family. Its index is a number that must be smaller than the value provided by the **vkGetPhysicalDeviceQueueFamilyProperties()** function (must be smaller than number of available queue families). In our example we are only interested in one queue from one queue family. And that's why we must remember the queue family index. It is used right here. If we want to prepare a more complicated scenario, we should also remember the

number of queues in each family as each family may support a different number of queues. And we can't create more queues than are available in a given family!

It is also worth noting that different queue families may have similar (or even identical properties) meaning they may support similar types of operations, that is, there may be more than one queue families that support graphics operations. And each family may contain different number of queues.

We must also assign a floating point value (from 0.0 to 1.0, both inclusive) to each queue. The higher the value we provide for a given queue (relative to values assigned to other queues) the more time the given queue may have for processing commands (relatively to other queues). But this relation is not guaranteed. Priorities also don't influence execution order. It is just a hint.

Priorities are relative only on a single device. If operations are performed on multiple devices, priorities may impact processing time in each of these devices but not between them. A queue with a given value may be more important only than queues with lower priorities on the same device. Queues from different devices are treated independently. Once we fill these structures and call **vkCreateDevice()**, upon success a created logical device is stored in a variable we provided an address of (in our example it is called VulkanDevice). If this function fails, it returns a value other than VK_SUCCESS.

Acquiring Pointers to Device-Level Functions

We have created a logical device. We can now use it to load functions from the device level. As I have mentioned earlier in real-life scenarios, there will be situations where more than one hardware vendor on a single computer will provide us with Vulkan implementation. With OpenGL it is happening now. Many computers have dedicated/discrete graphics card used mainly for gaming, but they also have Intel's graphics card built into the processor (which of course can also be used for games). So in the future there will be more than one device supporting Vulkan. And with Vulkan we can divide processing into whatever hardware we want. Remember when there were extension cards dedicated for physics processing? Or going farther into the past, a normal "2D" card with additional graphics "accelerator" (do you remember Voodoo cards)? Vulkan is ready for any such scenario.

So what should we do with device-level functions if there can be so many devices? We can load universal procedures. This is done with the **vkGetInstanceProcAddr()** function. It returns the addresses of dispatch functions that perform jumps to proper implementations based on a provided logical device handle. But we can avoid this overhead by loading functions for each logical device separately. With this method, we must remember that we can call the given function only with the device we loaded this function from. So if we are using more devices in our application we must load functions from each of these devices. It's not that difficult. And despite this leading to storing more functions (and grouping them based on a device they were loaded from), we can avoid one level of abstraction and save some processor time. We can load functions similarly to how we have loaded exported, global-, and instance-level functions:

```
#define VK_DEVICE_LEVEL_FUNCTION( fun )                                      \
if( !(fun = (PFN_##fun)vkGetDeviceProcAddr( Vulkan.Device, #fun )) ) {   \
  printf( "Could not load device level function: " #fun "!\n" );         \
  return false;                                                          \
}

#include "ListOfFunctions.inl"

return true;
```

16. Tutorial01.cpp, function LoadDeviceLevelEntryPoints()

This time we used the **vkGetDeviceProcAddr()** function along with a logical device handle. Functions from device level are placed in a shared header. This time they are wrapped in a VK_DEVICE_LEVEL_FUNCTION() macro like this:

```
#if !defined(VK_DEVICE_LEVEL_FUNCTION)
#define VK_DEVICE_LEVEL_FUNCTION( fun )
#endif
```

```
VK_DEVICE_LEVEL_FUNCTION( vkGetDeviceQueue )
VK_DEVICE_LEVEL_FUNCTION( vkDestroyDevice )
VK_DEVICE_LEVEL_FUNCTION( vkDeviceWaitIdle )

#undef VK_DEVICE_LEVEL_FUNCTION
```

17. ListOfFunctions.inl

All functions that are not from the exported, global or instance levels are from the device level. Another distinction can be made based on a first parameter: for device-level functions, the first parameter in the list may only be of type VkDevice, VkQueue, or VkCommandBuffer. In the rest of the tutorial if a new function appears it must be added to ListOfFunctions.inl and further added in the VK_DEVICE_LEVEL_FUNCTION portion (with a few noted exceptions like extensions).

Retrieving Queues

Now that we have created a device, we need a queue that we can submit some commands to for processing. Queues are automatically created with a logical device, but in order to use them we must specifically ask for a queue handle. This is done with **vkGetDeviceQueue()** like this:

```
vkGetDeviceQueue( Vulkan.Device, Vulkan.QueueFamilyIndex, 0, &Vulkan.Queue );
```

18. Tutorial01.cpp, function GetDeviceQueue()

To retrieve the queue handle we must provide the logical device we want to get the queue from. The queue family index is also needed and it must by one of the indices we've provided during logical device creation (we cannot create additional queues or use queues from families we didn't request). One last parameter is a queue index from within a given family; it must be smaller than the total number of queues we requested from a given family. For example if the device supports five queues in family number 3 and we want two queues from that family, the index of a queue must be smaller than two. For each queue we want to retrieve we have to call this function and make a separate query. If the function call succeeds, it will store a handle to a requested queue in a variable we have provided the address of in the final parameter. From now on, all the work we want to perform (using command buffers) can be submitted for processing to the acquired queue.

Tutorial01 Execution

As I have mentioned, the example provided with this tutorial doesn't display anything. But we have learned enough information for one lesson. So how do we know if everything went fine? If the normal application window appears and nothing is printed in the console/terminal, this means the Vulkan setup was successful. Starting with the next tutorial, the results of our operations will be displayed on the screen.

Cleaning Up

There is one more thing we need to remember: cleaning up and freeing resources. Cleanup must be done in a specific order that is (in general) a reversal of the order of creation.

After the application is closed, the OS should release memory and all other resources associated with it. This should include Vulkan; the driver usually cleans up unreferenced resources. Unfortunately, this cleaning may not be performed in a proper order, which might lead to application crash during the closing process. It is always good practice to do the cleaning ourselves. Here is the sample code required to release resources we have created during this first tutorial:

```
if( Vulkan.Device != VK_NULL_HANDLE ) {
  vkDeviceWaitIdle( Vulkan.Device );
  vkDestroyDevice( Vulkan.Device, nullptr );
}
```

```
if( Vulkan.Instance != VK_NULL_HANDLE ) {
  vkDestroyInstance( Vulkan.Instance, nullptr );
}

if( VulkanLibrary ) {
#if defined(VK_USE_PLATFORM_WIN32_KHR)
  FreeLibrary( VulkanLibrary );
#elif defined(VK_USE_PLATFORM_XCB_KHR) || defined(VK_USE_PLATFORM_XLIB_KHR)
  dlclose( VulkanLibrary );
#endif
}
```

19. Tutorial01.cpp, destructor

We should always check to see whether any given resource was created. Without a logical device there are no device-level function pointers so we are unable to call even proper resource cleaning functions. Similarly, without an instance we are unable to acquire pointer to a **vkDestroyInstance()** function. In general we should not release resources that weren't created.

We must ensure that before deleting any object, it is not being used by a device. That's why there is a wait function, which will block until all processing on all queues of a given device is finished. Next, we destroy the logical device using the **vkDestroyDevice()** function. All queues associated with it are destroyed automatically, then the instance is destroyed. After that we can free (unload or release) a Vulkan library from which all these functions were acquired.

Conclusion

This tutorial explained how to prepare to use Vulkan in our application. First we "connect" with the Vulkan Runtime library and load global level functions from it. Then we create a Vulkan instance and load instance-level functions. After that we can check what physical devices are available and what are their features, properties, and capabilities. Next we create a logical device and describe what and how many queues must be created along with the device. After that we can retrieve device-level functions using the newly created logical device handle. One additional thing to do is to retrieve queues through which we can submit work for execution.