API without Secrets: Introduction to Vulkan* Preface

Written by Pawel Lapinski

About the Author

I have been a software developer for over 9 years. My main area of interest is graphics programming, and most of my professional career has been involved in 3D graphics. I have a lot of experience in OpenGL* and shading languages (mainly GLSL and Cg), and for about 3 years I also worked with Unity* software. I have also had opportunities to work on some VR projects that involved working with head-mounted displays like Oculus Rift* or even CAVE-like systems.

Recently, with our team here at Intel, I was involved in preparing validation tools for our graphics driver's support for the emerging API called Vulkan. This graphics programming interface and the approach it represents is new to me. The idea came to me that while I'm learning about it I can, at the same time, prepare a tutorial for writing applications using Vulkan. I can share my thoughts and experiences as someone who knows OpenGL and would like to "migrate" to its successor.

About Vulkan

Vulkan is seen as an OpenGL's successor. It is a multiplatform API that allows developers to prepare high-performance graphics applications likes games, CAD tools, benchmarks, and so forth. It can be used on different operating systems like Windows*, Linux*, or Android*. The Khronos consortium created and maintains Vulkan. Vulkan also shares some other similarities with OpenGL, including graphics pipeline stages, GLSL shaders (sort of) or nomenclature.

But there are many differences that confirm the need for the new API. OpenGL was changing for over 20 years. Many things have changed in the computer industry since the early 90s, especially in graphics cards architecture. OpenGL is a good library, but not everything can be done by only adding new functionalities that match the abilities of new graphics cards. Sometimes a huge redesign has to be made. And that's why Vulkan was created.

Vulkan was based on Mantle*—the first in a series of new low-level graphics APIs. Mantle was developed by AMD and designed only for the architecture of Radeon cards. And despite it being the first publicly available API, games and benchmarks that used Mantle saw some impressive performance gains. Then other low-level APIs started appearing, such as Microsoft's DirectX* 12, Apple's Metal* and now Vulkan.

What is the difference between traditional graphics APIs and new low-level APIs? High-level APIs like OpenGL are quite easy to use. The developer declares what they want to do and how they want to do it, and the driver handles the rest. The driver checks whether the developer uses API calls in the proper way, whether the correct parameters are passed, and whether the state is adequately prepared. If problems occur, feedback is provided. For ease of use, many tasks have to be done "behind the scenes" by the driver.

In low-level APIs the developer is the one who must take care of most things. They are required to adhere to strict programming and usage rules and also must write much more code. But this approach is reasonable. The developer knows what they want to do and what they want to achieve. The driver does not, so with traditional APIs the driver has to make additional effort for the program to work properly. With APIs like Vulkan this additional effort can be avoided. That's why DirectX 12, Metal, or Vulkan are called thin-drivers/thin-APIs. Mostly they only communicate user requests to the hardware, providing only a thin abstraction layer of the hardware itself. The driver does as little as possible for the sake of much higher performance.

Low-level APIs require additional work on the application side. But this work can't be avoided. Someone or something has to do it. So it is much more reasonable for the developer to do it, as they know how to divide work into separate threads, when the image would be a render target (color attachment) or used as a texture/sampler, and so on. The developer knows what pipeline state or what vertex attributes changes more often. All that leads to far more effective use of the graphics card hardware. And the best part is that it works. An impressive performance boost can be observed.

But the word "can" is important. It requires additional effort but also a proper approach. There are scenarios in which no difference in performance between OpenGL and Vulkan will be observed. If someone doesn't need multithreading or if the application isn't CPU bound (rendered scenes aren't too complex), OpenGL is enough and using Vulkan will not give any performance boost (but it may lower power consumption, which is important on mobile devices). But if we want to squeeze every last bit from our graphics hardware, Vulkan is the way to go.

Sooner or later all major graphics engines will support some, if not all, of the new low-level APIs. So if we want to use Vulkan or other APIs, we won't have to write everything from scratch. But it is always good to know what is going on "under the hood", and that's the reason I have prepared this tutorial.

A Note about the Source Code

I'm a Windows developer. When given a choice I write applications for Windows. That's because I don't have experience with other operating systems. But Vulkan is a multiplatform API and I want to show that it can be used on different operating systems. That's why I've prepared a sample project that can be compiled and executed both on Windows and Linux.

Source code for this tutorial can be found here:

https://github.com/GameTechDev/IntroductionToVulkan

I have tried to write code samples that are as simple as possible and to not clutter the code with unnecessary "#ifdefs". Sometimes this can't be avoided (like in window creation and management) so I decided to divide the code into small parts:

- **Tutorial** files are the most important here. They are the ones where all the exciting Vulkan-related code is placed. Each lesson is placed in one header/source pair.
- **OperatingSystem** header and source files contain OS-dependent parts of code like window creation, message processing, and rendering loops. These files contain code for both Linux and Windows, but I tried to unify them as much as possible.
- **main.cpp** file is a starting point for each lesson. As it uses my custom Window class it doesn't contain any OS-specific code.
- **VulkanCommon** header/source files contain the base class for all tutorials starting from tutorial 3. This class basically replicates tutorials 1 and 2—creation of a Vulkan instance and all other resources necessary for the rendered image to appear on the screen. I've extracted this preparation code so the code of all the other chapters could focus on only the presented topics.
- **Tools** contain some additional utility functions and classes like a function that reads the contents of a binary file or a wrapper class for automatic object destruction.

The code for each chapter is placed in a separate folder. Sometimes it may contain an additional Data directory in which resources like shaders or textures for a given chapter are placed. This Data folder should be copied to the same directory in which executables will be held. By default executables are compiled into a build folder.

Right. Compilation and build folder. As the sample project should be easily maintained both on Windows and Linux I've decided to use CMakeLists.txt file and a CMake tool. On Windows there is a build.bat file that creates a Visual Studio* solution—Microsoft Visual Studio 2013 is required to compile the code on Windows (by default). On Linux I've provided a

build.sh script that compiles the code using make but CMakeLists.txt can also be easily opened with tools like Qt. CMake is of course also required.

Solution and project files are generated and executables are compiled into the build folder. This folder is also the default working directory, so the Data folders should be copied into it for the lessons to work properly. During execution, in case of any problems, additional information is "printed" in cmd/terminal. So if there is something wrong, run the lesson from the command line/terminal or look into the console/terminal window to see if any messages are displayed.

I hope these notes will help you understand and follow my Vulkan tutorial. Now let's focus on learning Vulkan itself!

Notices