Vulkan Abstraction Layer for Large Data Remote Rendering System

Primož Lavrič, Ciril Bohak, and Matija Marolt

Faculty of Computer and Information Science, University of Ljubljana,
Večna pot 113, 1000 Ljubljana, Slovenia,
pl9506@student.uni-lj.si,
{ciril.bohak, matija.marolt}@fri.uni-lj.si

Abstract. New graphics APIs require users to implement a lot of needed functionality, such as memory management, by themselves. In this paper we present an abstraction layer build on top of such API, in our case the Vulkan API, for purpose of off-screen rendering of large data. We also present a use case for such abstraction layer implementation – a remote rendering system for simple Path Tracing accessible through web-based client-side application. The preliminary evaluation results show that implementation of simple Path Tracer is significantly faster then comparable implementation in OpenCL. In conclusion we also present possible extension and improvements of the developed abstraction layer.

Keywords: Vulkan API, graphics library abstraction layer, real-time rendering

1 Introduction

Graphical pipeline for real-time rendering has gone through substantial changes throughout its development since early 1990s. Over the years, the developers have gained substantially more control over what is happening in individual stage of the pipeline with added support for programmable stages. Most recent graphical APIs (Vulkan [1], DirectX 12, Mantle and Metal) are providing the developers with even more control, but on the other hand also require of them to implement a lot of additional functionality (e.g. memory management), previously handled by the APIs (OpenGL and DirectX 1–11).

The remote rendering concept of large datasets is not a new concept. It was implemented for different domains such as large volumetric data [2] and was implemented on different platforms (e.g. a web-based approach [3]). The researchers have also proposed different approaches for transferring the information between client- and server-side. Authors of [4] propose to transfer the information in form of 3D video. An interesting approach on high-resolution remote rendering system with collaborative features for grid environments is presented in [5]. Our approach distinguishes from the above presented implementation in several ways: (1) developers can implement their own rendering solution (in our case we present shader-based path-tracing implementation), (2) the approach is very general and allows the extension for use with different kind of data and (3) our approach supports broad specter of hardware (e.g. not limited to specific manufacturer like CUDA) and does not necessarily need a GPU.

The rest of the paper is structured as follows: in section 2 we present the developed abstraction layer, in section 4 we present preliminary evaluation and results and in section 5 we give the pointers for future work and present the conclusions.

2 Vulkan API Abstraction Layer

Unlike OpenGL, Vulkan provides only basic functionality on top of the driver and requires the developer to implement the rest. The implemented abstraction layer is on a similar level as OpenGL, but unlike OpenGL, it is not implemented as a state machine, because that would hinder the multi-thread performance. Rather than implementing a state machine, we implemented high-level wrappers that employ RAII programming idiom, each wrapping one or more Vulkan objects. Figure 1 presents the diagram of the abstraction layer. Unlike Vulkan objects which are independent of each other, the presented objects form a hierarchy in which each child object is dependent of its parent. This way we enforce that the objects are created in the correct order and that the child objects and their resources are automatically disposed of upon its parent's disposal. Each object is retrievable from its parent in a form of a raw pointer and all operations that operate on some object take a raw pointer as input parameter (for example a Descriptor Set requires a pointer to Descriptor Set Layout). Raw pointers are used to emphasize that the objects are owned by the parent of the object and not by the developer, and also to prevent the developer from accidentally disposing of the object.

The abstraction layer provides a base-class for the renderer implementation. This class provides interface for managing Vulkan instance and more importantly Vulkan devices. Vulkan device objects are the core of the abstraction layer. Vulkan device allows developer to manage enabled features, extensions and to query the device properties, but more importantly, it provides the structures for compute and graphical operations. In the following sections we present four key components provided by Vulkan device: Program Manager, Allocation Manager, Descriptor Pool and Queue Families and their integration within the abstraction layer.

2.1 Program Manager

The task of the Program Manager is to load shaders and to construct pipeline layouts and render passes. The Program Manager requires that the developer specifies the directory that contains shaders and a shader configuration file. Shader configuration file contains relative paths to all shaders' source files and a list of pipeline configurations, each equipped with IDs of shaders that form the pipeline.

After the shaders are loaded, we need to perform shader reflection in order to build the pipeline layouts. Shader reflection is a process in which we extract the types, sets, bindings, layouts and array counts of shader uniforms, attributes and sub-pass attachments from the shader source code. Unlike OpenGL, which has integrated shader reflection, Vulkan requires the developer to either implement shader reflection on his own or provide pipeline layout configuration using other means. In our implementation, we used library SPIRV-Cross [6, 7] to decompile SPIRV shaders and perform shader reflection. Because shader loading is intended to occur during the configuration time, the

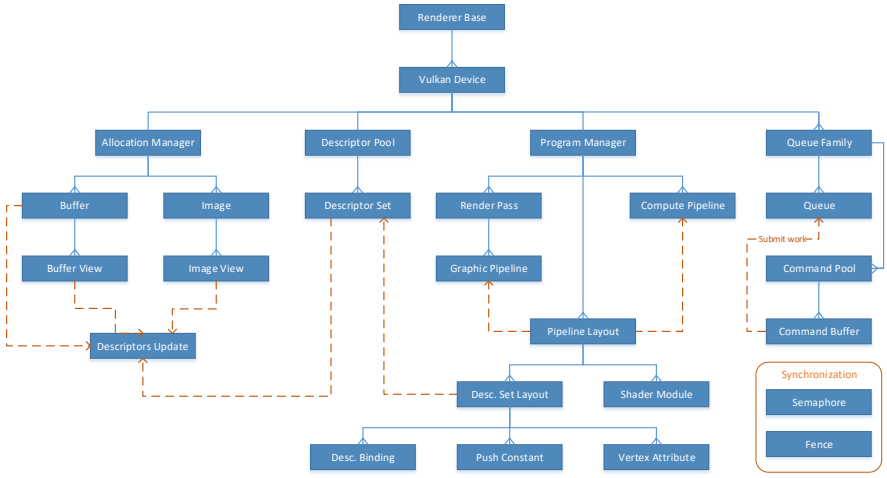

Fig. 1. Diagram showing the structure of our Vulkan API abstraction layer. Blue lines represent the composition association with either one-to-one or one-to-many relation. Orange arrows indicate that the object is used by the object the arrow is pointing to.

speed is not of crucial importance and we can afford to perform shader decompilation. We use the acquired data to populate pipeline layout descriptor sets, attributes and push constants and finally build the pipeline layout. Using the generated Pipeline Layout objects Program manager initializes the Compute Pipelines.

In the current implementation we only support Compute Pipelines, but intend to provide support for Graphical Pipelines in the future. Graphical Pipelines differ from Compute Pipelines as they are required to specify additional graphic related configuration, such as: vertex shader input, assembly configuration and depth and stencil state. In addition to that, Vulkan allows us to specify complex Render Passes, that consist of many Shader Programs given in a directed acyclic graph. This means that we need to specify the dependencies between the programs and connect the inputs and the outputs between the connected programs. In our implementation, we decided to let user provide additional graphic related configuration via a Pipeline State object and Render Pass configuration as additional data in shader configuration file. Pipeline State object encodes the configuration in a bit field using only 423 bits while maintaining a 32/64 bit hash of the bit field to speed up pipeline querying. As for the Render Pass configuration, it is better that we provide it in th configuration file, because a typical application usually uses a small number of different Render Passes and they do not change in runtime.

2.2 Allocation Manager

In terms of memory management, there is a huge difference between Vulkan and OpenGL. In OpenGL, the memory management was handled implicitly by the API. In Vulkan though, developer needs to allocate and manage the memory used by buffers and images.

For memory management, we decided to use Vulkan Memory Allocator library provided by AMD GPUOpen [8]. We decided to use this library, because it provides very efficient memory management, by allocating very large chunks of memory and implementing paging on top of it, thus minimizing the number of operating system allocation calls. It also provides all of the needed functionalities and allows developer to provide custom allocator implementation.

Vulkan Memory Allocator allocations are managed by the Allocation Manager and are completely hidden from the developer. The developer can request either Buffer or Image object from the Allocation manager simply by specifying its size and format. The user can then manipulate the Buffers and the Image by either writing/reading the data via the provided interface (CPU visible memory only) or by submitting Command Buffer object containing write/copy commands to the Queue.

Developer can also create View objects for both Images and Buffers (limited to texel buffers). A view is quite literally a view into an image/buffer. It describes how to access the image/buffer and which part of the image/buffer to access, for example, if it should be treated as a 3D texture, which mip-map levels should be used and so on. Created Views are used to bind the image/buffer to either frame buffer or descriptor set.

2.3 Descriptor Pool

We already presented the Descriptor Sets in subsection 2.2, more specifically Descriptor Set Layouts. But before Descriptor Sets can be used, they have to be allocated from the Descriptor Pool. In Vulkan, the Descriptor Pool requires that the maximum number of allocated Descriptor Sets and bindings of each type is specified in advance. One could set these to maximum possible value, but this would be very inefficient. Because of that, the abstraction layer allow the developer to query the number of Descriptor Sets and bindings from the Program Manager and provide it to the Descriptor Pool object. This way, the Descriptor Pool knows the exact number of Descriptor Bindings, and the developers are also still able to modify the count manually. For example, multiply it by two for implementation of double buffering.

After the Descriptor Pool is initialized one can start allocating Descriptor Sets. The Descriptor Sets can be filled with the data using Descriptor Update object that records write/copy operations and executes them once submitted to Vulkan Device object. After the data has been written to the Descriptor Set, one can bind it via a bind command submitted to the command buffer.

2.4 Queue Families

Each device can have up to three Queue Families: graphic, compute and transfer queues. Graphic Family supports all operations (including compute and transfer operations), Compute Family supports both compute and transfer operations and the Transfer Family supports only transfer operations. This means one could use graphic family to perform all operations, but it is usually better to perform transfer operations on the Transfer Family queue and compute operations on the Compute Family queue because of the driver optimization.

Because of that, we allow the developer to specify how many Queues of each family he wants to create during the Vulkan device initialization. This allows the distribution of the operations on different Queues to achieve better performance. Note that using too many Queues may hinder the performance due to the excessive synchronization.

Each Queue Family object also has a Command Pool object associated with it. Command Pool allows the allocation of the Command buffer objects for its Queue Family. One can use the Command buffer objects to record and submit the commands to the queue. In order to synchronize the command execution among multiple Command buffers and the host, we can use fences and semaphores wrapped in Fence and Semaphore objects.

3 Use Case: Remote Rendering

The developed Vulkan abstraction layer was used for implementation of remote rendering system. Using the Vulkan abstraction layer, we implemented a showcase example with a simple GPU Path Tracing renderer that renders and transmits the image to the user in real time, and listens for the user input.

3.1 System Architecture

The remote rendering system roughly consists of three parts: (1) NodeJS server, (2) web client and (3) remote renderer. NodeJS servers only task is to provide HTTP server that serves the web page to the client.

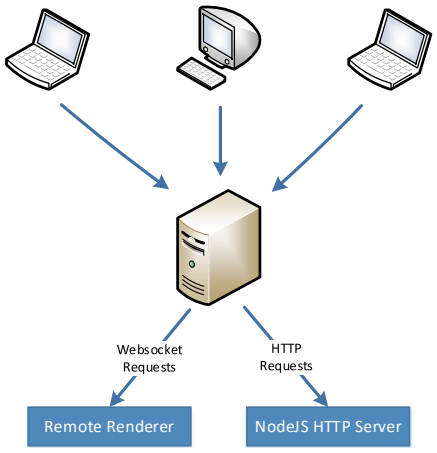

Fig. 2. Diagram showing the communication paths between clients and the server.

Web client is used as an interface to the remote renderer. Its job is to display the images that are broadcasted from the server and to capture the user input and forward it

to the server. To display images, we implemented a basic WebGL renderer that renders a given texture to the quad and displays it on HTML canvas. In the future we also intend to perform some lightweight rendering (2D, 3D annotations) on the client side and merge it with the remote renderer output. All the communication between the client and the remote renderer occurs via HTML Websockets, because of their very low overhead latency. The client is always listening for the remote renderer updates, which contain rendered images in the binary format. Because the remote renderer provides the image encoded in the format supported by WebGL, the client can simply forward the received image to its WebGL renderer and display it. In the showcase implementation, we also use mouse input information to move around the scene. We capture the mouse input on the client and transmit the data in JSON format to the remote renderer.

The remote renderer's primary task is to broadcast the rendered images and handle user input requests. First it opens up a socket and waits for the client to connect. When the first client connects it starts rendering the scene using renderer implemented on top of our Vulkan abstraction layer. Each rendering iteration computes one sample per pixel and send the updated image to all connected clients. When the remote renderer receives input update request (containing mouse position), it computes a new camera position, writes it to uniform buffer and triggers a redraw.

3.2 Rendering Example

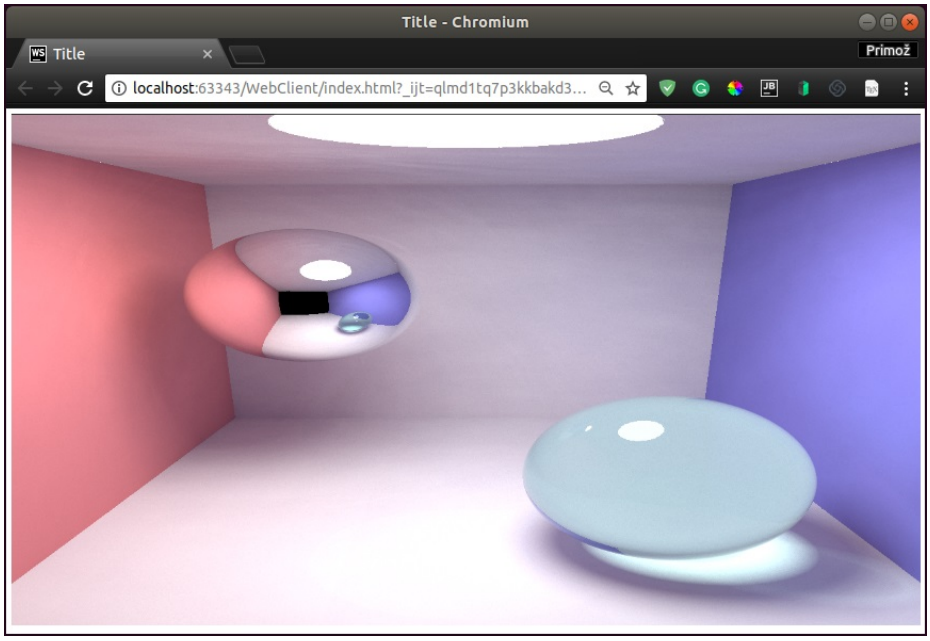

Fig. 3. Shows the image of the Cornell Box that was rendered by the remote renderer implemented on our Vulkan abstraction layer and the streamed to the web client.

Renderer computes the image by solving the rendering equation [9] using numerical integration. We implemented Monte Carlo path tracing with Russian roulette early path termination in GLSL compute shaders (compiled to SPIR-V). The implementation is loosely based on [10, 11]. Shader program inputs are scene data (spheres and planes), camera transformation and timestamp (updated each iteration), used for pseudo-random number generation. Upon execution, the shader program adds the radiance to the storage buffer that is used to accumulate radiance. When the shader program finishes, the data is read from the GPU and forwarded to the client.

4 Evaluation and Results

We conducted a preliminary performance evaluation by comparing three different implementations of the same path tracer algorithm: C# (CPU), OpenCL (GPU) and Vulkan (GPU) implementation. On both GPU implementations only one SPP was computed each kernel/shader program execution. On CPU implementation we parallelized the work and used 8 threads for the benchmark. We rendered a 1920x1080 image and measured the time required to reach 1000 SPP (GPU) and 100 SPP (CPU). The performance was evaluated on the following machine: CPU - Intel i7 6700k, RAM - 16 GB and GPU - NVIDIA GeForce GTX 1080Ti. We performed 10 time measurements for each implementation. Results of our testing are presented as average time required to reach 1000 SPP (100 SPP for CPU) and standard deviation in Table 1.

	CPU (100 SPP)	OpenCL (1000 SPP)	Vulkan (1000 SPP)
Average	4850s (∼80 min)	4.27s	3.91s
STD	157s	0.04s	0.06s

Table 1. Performance evaluation results of CPU, OpenCL and Vulkan path tracer implementations.

The results show that the OpenCL implementation and the Vulkan implementation on top of the Vulkan abstraction layer both perform very similarly, but the Vulkan implementation is on average 9% faster.

5 Conclusions and Future Work

In this work we present an abstraction layer built on top of the Vulkan API, which allows faster and easier development of graphics application. The abstraction layer was developed for purpose of implementing headless remote rendering application connected to web-based client-side end-user application. The preliminary evaluation, based on example use case application, shows that the developed system outperforms OpenCL implementation. Such rendering system could be used for visualizing large data on lightweight client devices (e.g. smart phones or tablets) by exploiting the processing power of dedicated server-side system.

There are still many possible extensions of the presented system, for example: implementation of swapchain abstraction that could be optionally used to display the content window of local application, implementation of streaming compression (GPU/CPU) for faster transfer of render results and adaptive streaming based on connection quality, structures that would enable seamless multi GPU support and buffer streamers that would enable rendering of large data that does not fit the memory of the GPU.

We are also planning to fully implement the remote rendering presented as use case and integrate it with an existing web-based medical data visualization framework.

References

1. Sellers, G., Kessenich, J.: Vulkan Programming Guide: The Official Guide to Learning Vulkan. Addison-Wesley Professional (2016)
2. Guthe, S., Wand, M., Gonser, J., Strasser, W.: Interactive rendering of large volume data sets (Nov 2002)
3. Yoon, I., Neumann, U.: Web-based remote renderingwith ibrac (image-based rendering acceleration and compression). Computer Graphics Forum **19**(3) (2000) 321–330
4. Shi, S., Jeon, W.J., Nahrstedt, K., Campbell, R.H.: Real-time remote rendering of 3d video for mobile devices. In: Proceedings of the 17th ACM International Conference on Multimedia. MM '09, New York, NY, USA, ACM (2009) 391–400
5. Karonis, N.T., Papka, M.E., Binns, J., Bresnahan, J., Insley, J.A., Jones, D., Link, J.M.: High-resolution remote rendering of large datasets in a collaborative environment. Future Generation Computer Systems **19**(6) (2003) 909 – 917 3rd biennial International Grid applications-driven testbed event, Amsterdam, The Netherlands, 23-26 September 2002.
6. Arntzen, H.K., et al.: KhronosGroup/SPIRV-Cross. `https://github.com/KhronosGroup/SPIRV-Cross` (Accessed on 15-02-2018).
7. Arntzen, H.K.: Using SPIR-V in practice with SPIRVcross
8. Sawicki, A., et al.: Vulkan Memory Allocator 1.0 - GPUOpen. `https://gpuopen.com/vulkan-memory-allocator-1-0/` (Accessed on 15-02-2018).
9. Kajiya, J.T.: The rendering equation. In: ACM Siggraph Computer Graphics. Volume 20., ACM (1986) 143–150
10. Beason, K.: smallpt: Global Illumination in 99 lines of C++. `http://www.kevinbeason.com/smallpt/` (2014) (Accessed on 15-02-2018).
11. Shirley, P., Morley, R.K.: Realistic Ray Tracing. 2 edn. A. K. Peters, Ltd., Natick, MA, USA (2003)